# 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 5
## Building applications, components, and libraries 5-1

## Chapter 6
## Developing the application user interface 6-1

## Chapter 8
## Working with graphics and multimedia 8-1



## Chapter 9
## Writing multi-threaded applications 9-1

## 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

## Chapter 26 **Using XML in database applications** **26-1**

# 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 34 Working with type libraries 34-1



## Chapter 35 Creating COM clients 35-1

# Part V
# Creating custom components

## Chapter 40
## Overview of component creation 40-1



## Chapter 41
## Object-oriented programming for component writers 41-1



## Chapter 42
## Creating properties 42-1

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*.

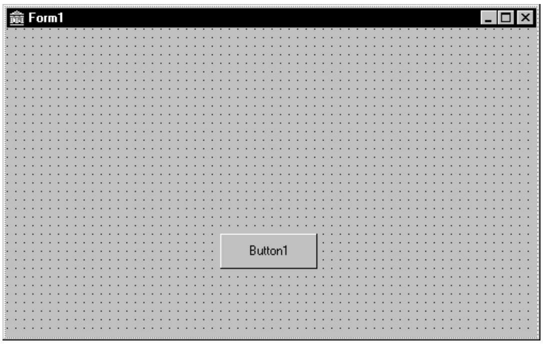
```
var
  Form1: TForm1;
```

*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.

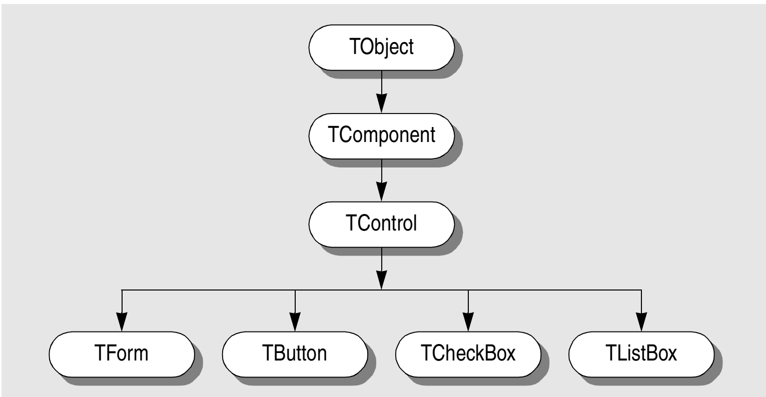
Ownership applies only to *TComponent* and its descendants. If you create, for example, a *TStringList* or *TCollection* object (even if it is associated with a form), you are responsible for freeing the object.

**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.

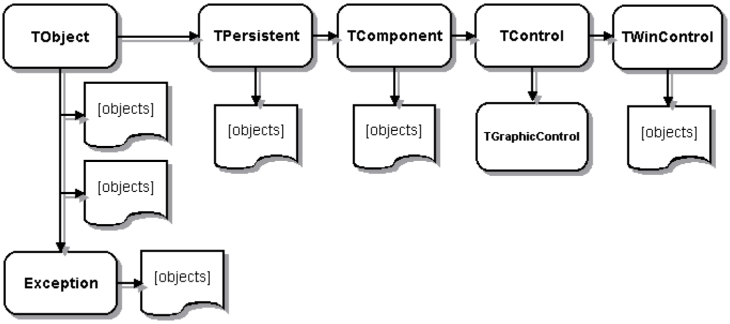
**Figure 3.2** Objects, components, and controls




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.

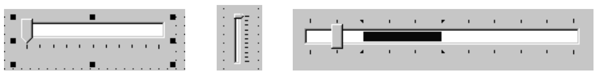
The scroll bar component is not used very often, because many visual components include scroll bars of their own and thus don’t require additional coding. For example, *TForm* has *VertScrollBar* and *HorzScrollBar* properties that automatically configure scroll bars on the form. To create a scrollable 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.

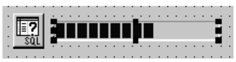
If you only want to display one text string at a time in the status bar, set its *SimplePanel* property to *True* and use the *SimpleText* property to control the text displayed in the status bar.

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.

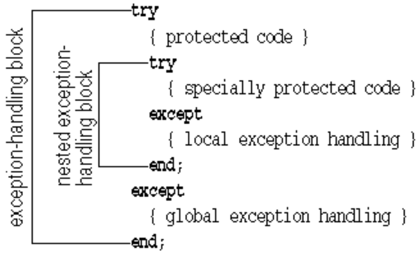
In the simplest case, for example, you can protect a resource allocation, and within that protected block, define blocks that allocate and protect other resources. Conceptually, that might look something like this:

protected block

nested protected block

```
{ 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:




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.<br>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.<br>In the {$V-} (relaxed) state, any short string type variable is allowed as an actual parameter, even if the declared maximum length is not the same as that of the formal parameter. Be aware that this could lead to memory corruption. For example:<br>`var S: string[3];`<br>`procedure Test(var T: string);`<br>`begin`<br>`  T := ‘1234’;`<br>`end;`<br>`begin`<br>`  Test(S);`<br>`end.` |
| {$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).<br>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.<br>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.<br>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.<br>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.

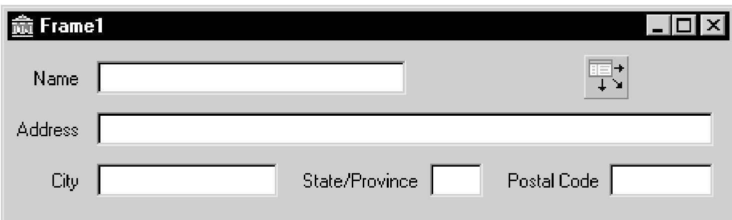
- Select *Frames* from the Standard page of the component palette and click on a form or another frame. A dialog appears with a list of frames that are already included in your project; select one and click OK.

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:

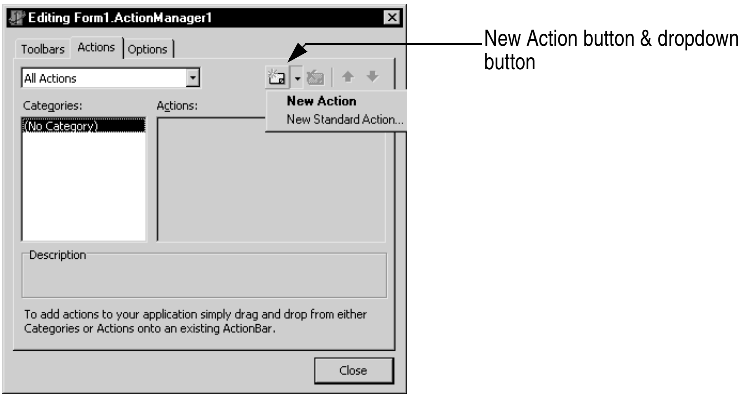
- Drop an Action Manager onto a form.
- Add actions to the Action Manager, which organizes them into appropriate action lists.
- Create the action bands (that is, the menu or the toolbar) for the user interface.
- Drag and drop the actions into the application interface.

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.




**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.<br>*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

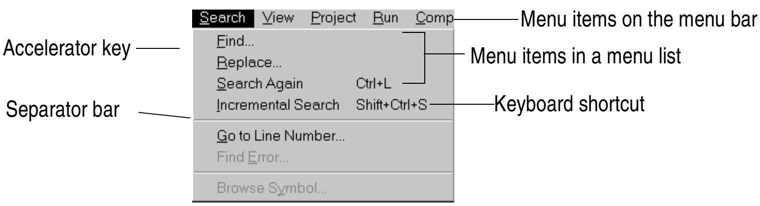
Menus provide an easy way for your users to execute logically grouped commands. The Menu Designer enables you to easily add a menu—either predesigned or custom tailored—to your form. You add a menu component to the form, open the Menu Designer, and type menu items directly into the Menu Designer window. 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

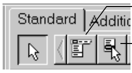



For information about hooking up menu items to the code that executes when they are selected, see "Associating menu events with event handlers" on page 3-28.

## 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




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.

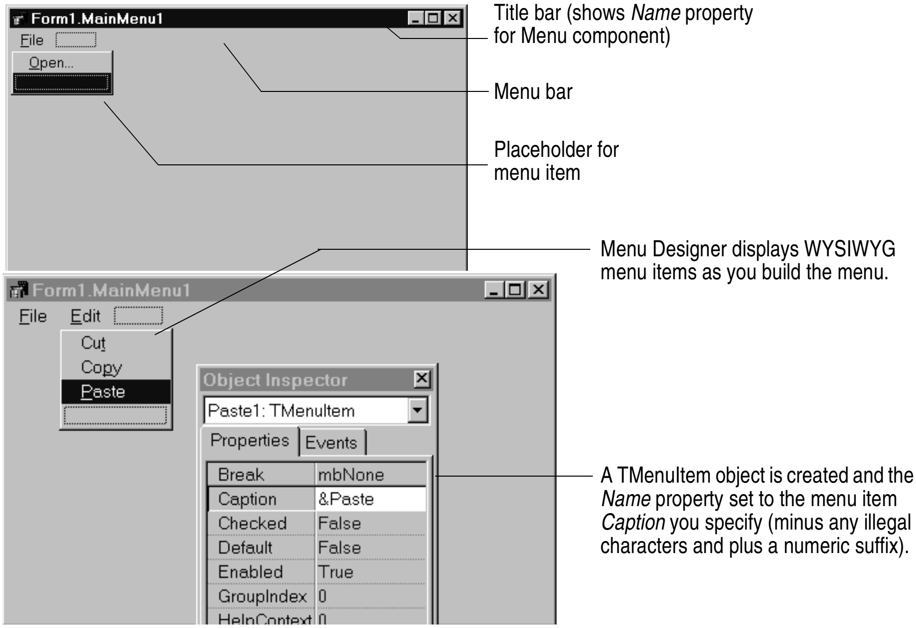
  or

- 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


**Figure 6.6** Menu Designer for a main menu




## 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.

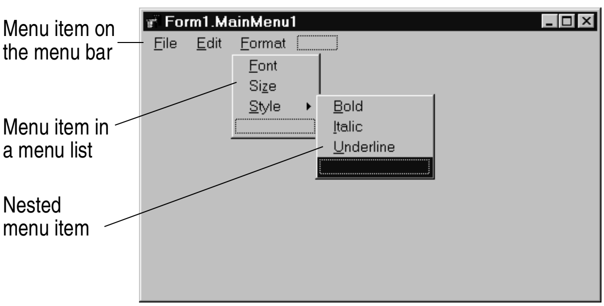
**Caution** Keyboard shortcuts, unlike accelerator keys, are not checked automatically for duplicates. You must ensure uniqueness yourself.

## 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

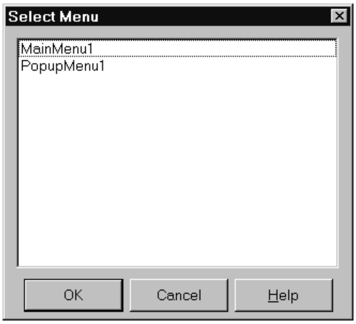
If you're designing several menus for your form, you can use the Menu Designer context menu or the Object Inspector to easily select and move among them.

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.

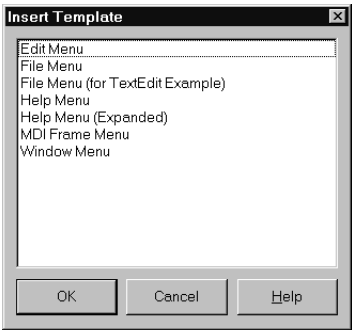
To add a menu template to your application,

**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.

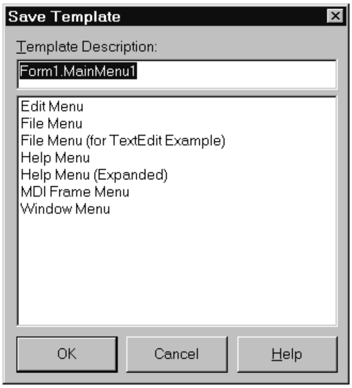



**Figure 6.10** Save Template dialog box for menus

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)<br>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;
```

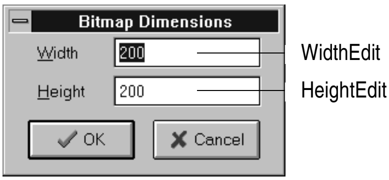
## Replacing the picture

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.




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.<br>On Linux, configuration files are created as hidden files starting with a dot (.). Many are placed in the /etc directory and your home directory.<br>Linux also uses environment variables such as LD_LIBRARY_PATH (search path for libraries). Other important environment variables:<br>HOME Your home directory (/home/sam)<br>TERM Terminal type (xterm, vt100, console)<br>SHELL Path to your shell (/bin/bash)<br>USER Your login name (sfuller)<br>PATH List to search for programs<br>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:<br>`-` 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.<br>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.<br>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.<br>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 | AnsiStrLower | ExtractFileDir |
| AnsiCompareFileName | AnsiStrPos | ExtractFileExt |
| AnsiExtractQuotedStr | AnsiStrRScan | ExtractFileName |
| AnsiLastChar | AnsiStrScan | ExtractFilePath |
| AnsiLowerCase | AnsiStrUpper | ExtractRelativePath |
| AnsiLowerCaseFileName | AnsiUpperCase | FileSearch |
| AnsiPos | AnsiUpperCaseFileName | IsDelimiter |
| AnsiQuotedStr | ByteToCharIndex | IsPathDelimiter |
| AnsiStrComp | ByteToCharLen | LastDelimiter |
| AnsiStrIComp | ByteType | StrByteType |
| AnsiStrLastChar | ChangeFileExt | StringReplace |
| AnsiStrLComp | CharToByteIndex | WrapText |
| AnsiStrLIComp | CharToByteLen | |

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.

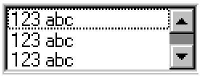
## BiDiMode property

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.

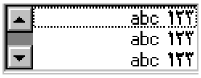
**Figure 12.1** TListBox set to bdLeftToRight




### 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.

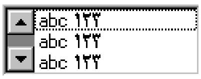
**Figure 12.2** TListBox set to bdRightToLeft




### bdRightToLeftNoAlign

*bdRightToLeftNoAlign* draws text using right to left reading order, the alignment is not changed, and the scroll bar is moved.

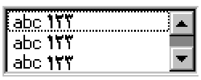
**Figure 12.3** TListBox set to bdRightToLeftNoAlign




### 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




## 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.

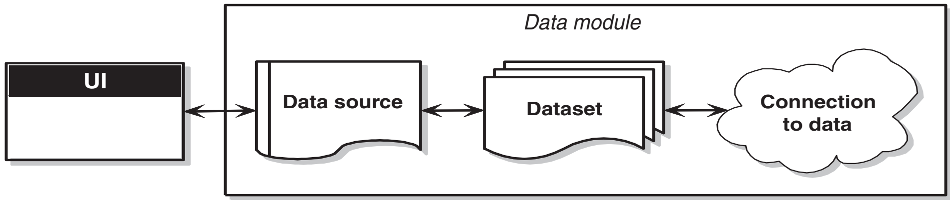
# Database architecture

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:

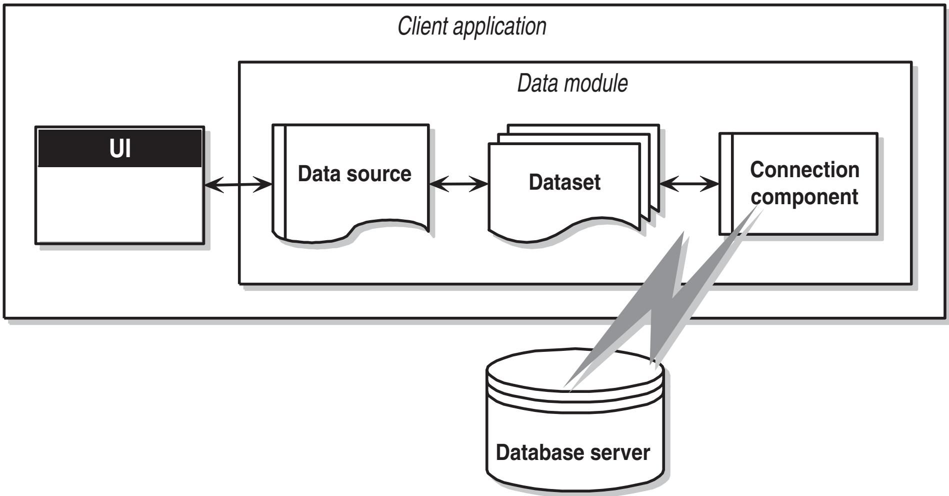
- Connecting directly to a database server. Most datasets use a descendant of *TCustomConnection* to represent the connection to a database server.
- Using a dedicated file on disk. Client datasets support the ability to work with a dedicated file on disk. No separate connection component is needed when 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”.

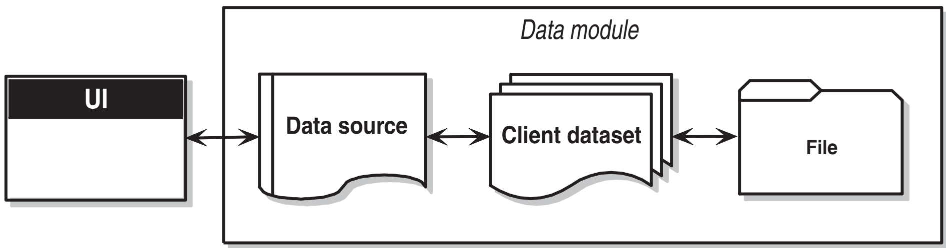
This architecture represents either a single-tiered or two-tiered application, depending on whether the database server is a local database such or a remote database server. The logic that manipulates database information is in the same application that implements the user interface, although isolated 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.)

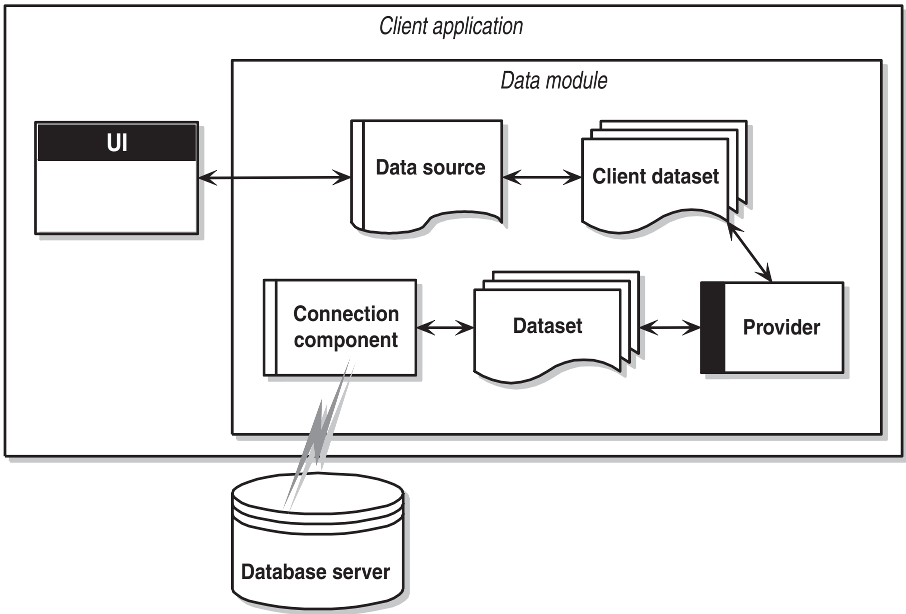
There are two versions of the architecture that connects a client dataset to an external dataset:

- 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.

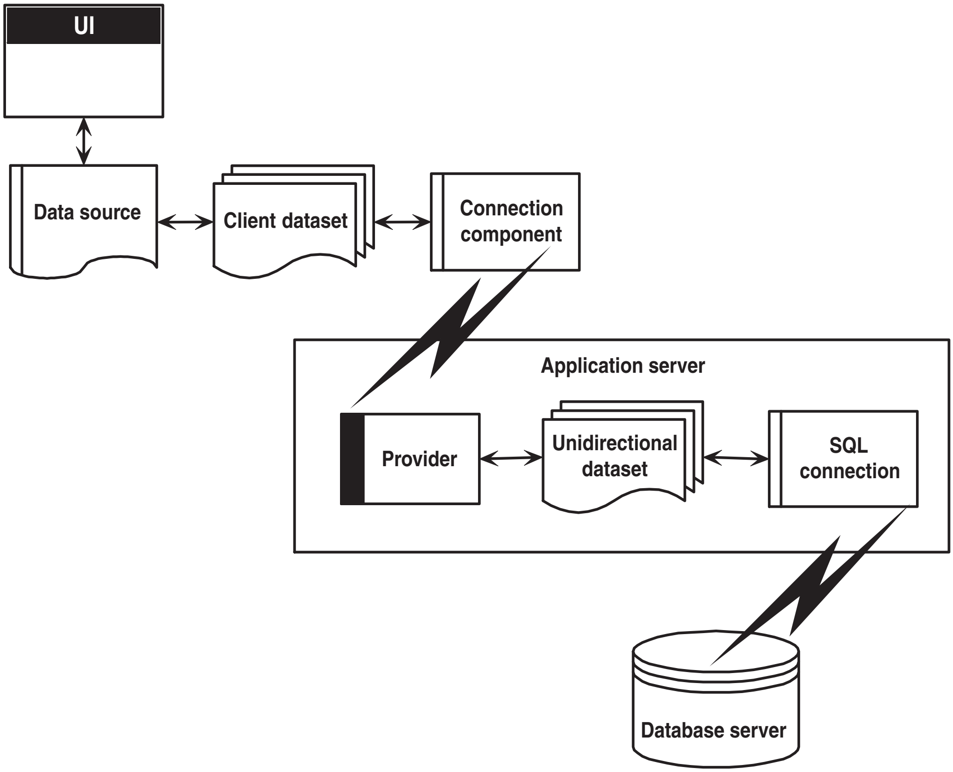
Once the client dataset is linked to the provider and the provider is linked to the source dataset, these components automatically handle all the details necessary for fetching, displaying, and navigating through the database records (assuming the source dataset is connected to a database). To apply 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:

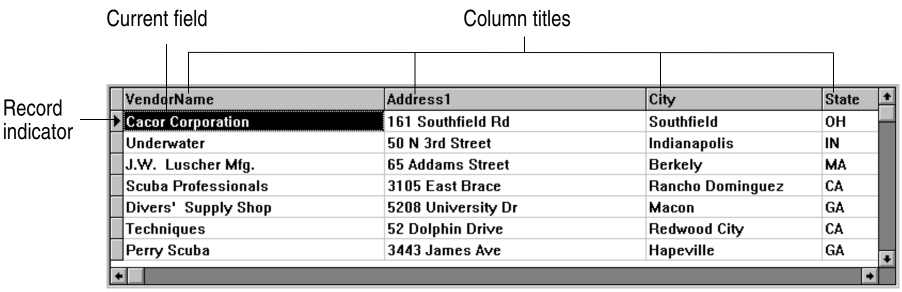
- **Master-detail forms:** You can represent information from both a master table and a detail table by including both controls that display a single field and grid controls. For example, you could display information about a single customer with a detail grid that displays the orders for that customer. 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




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.<br>*cbsEllipsis*: Displays an ellipsis (...) button to the right of the cell. Clicking on the button fires the grid's *OnEditButtonClick* event.<br>*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.<br>*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

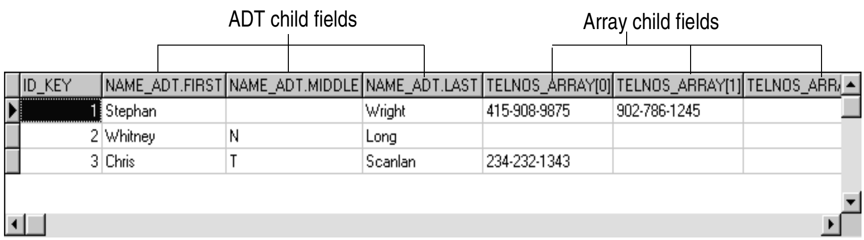



Figure 15.3 and 15.4 show the grid with an ADT field and an array field. Figure 15.3 shows the fields collapsed. In this state they cannot be edited. Figure 15.4 shows the fields expanded. The fields are expanded and collapsed by clicking on the arrow in the fields title bar.

**Figure 15.3** TDBGrid control with Expanded set to False

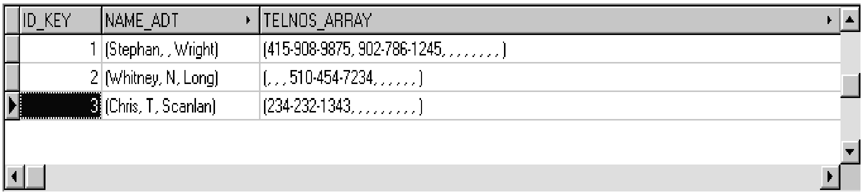
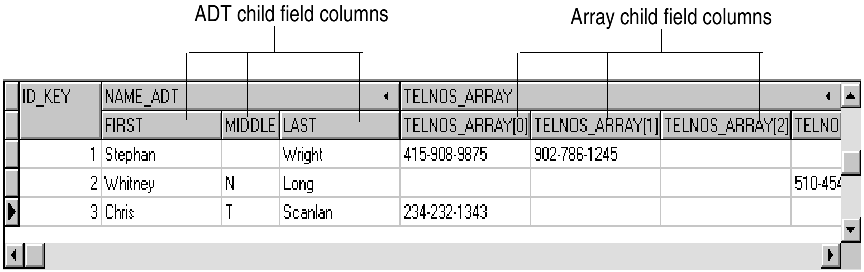



**Figure 15.4** TDBGrid control with Expanded set to True




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.

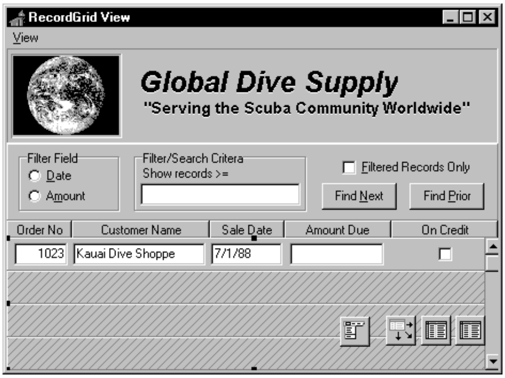
3 Place individual data controls within the design cell for the grid. The design cell for the grid is the top or leftmost cell in the grid, and is the only cell into which you can place other controls.

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.<br>*False*: Prevents record deletion. |
| AllowInsert | *True* (default): Permits record insertion.<br>*False*: Prevents record insertion. |
| ColCount | Sets the number of columns in the grid. Default = 1. |
| Orientation | *goVertical* (default): Display records from top to bottom.<br>*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.<br>*False*: Does not display a focus rectangle. |

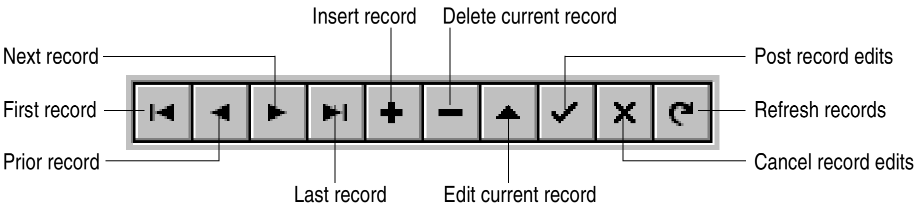
For more information about database control grid properties and methods, see the online *VCL Reference*.

# 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.

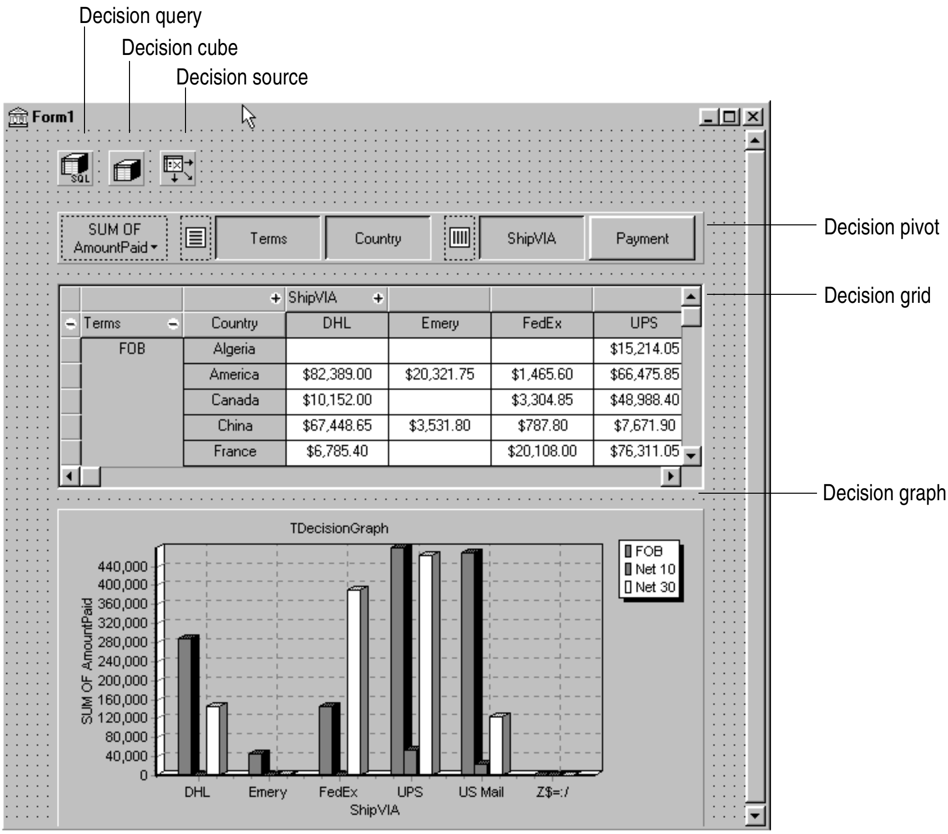
## Overview

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




# 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.

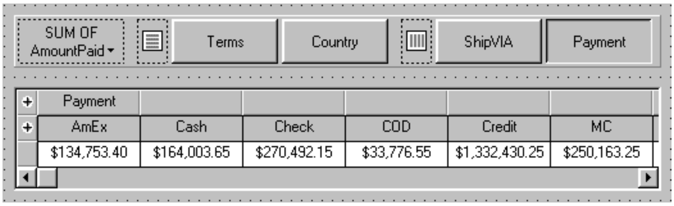
Crosstabs can be one-dimensional or multidimensional.

## 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

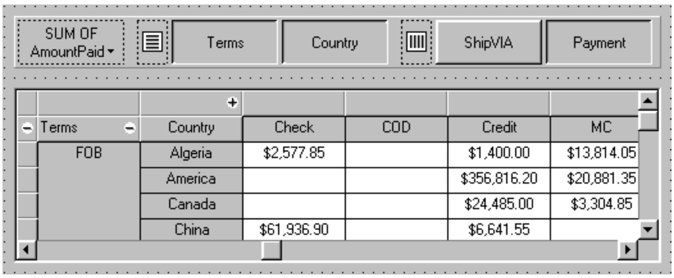



## Multidimensional crosstabs

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




# 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*).

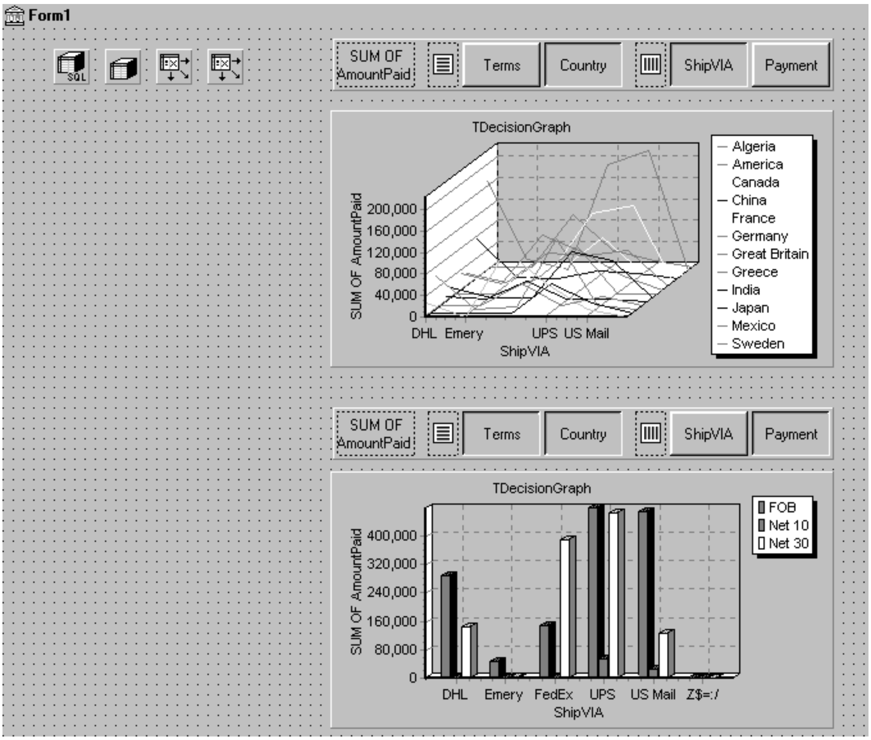
Graphed data comes from a specially formatted dataset such as *TDecisionQuery.* For an overview of how the decision support components handle and arrange this data, see page 16-1.

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.<br>*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.<br>*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.<br>*False*: Locale translation does not occur. |
| *Visible* | *True* (the default): Permits display of field in a data-aware grid.<br>*False*: Prevents display of field in a data-aware grid component.<br>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 dateset'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.

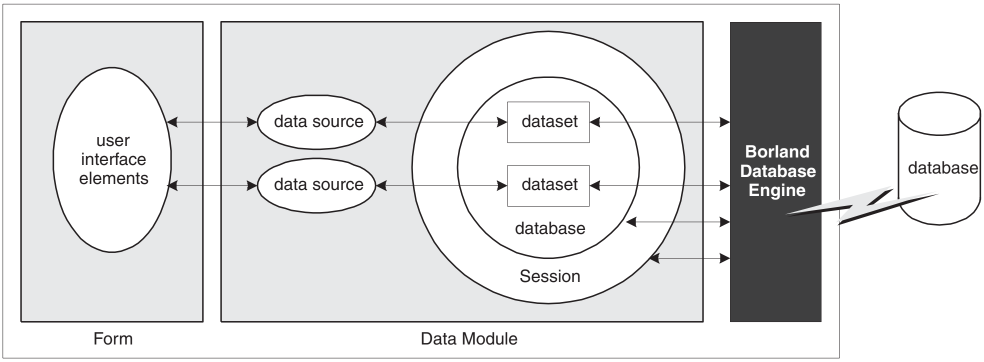
## BDE-based architecture

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*<br>*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:<br>1: Table<br>2: View<br>4: System table<br>8: Synonym<br>16: Temporary table<br>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:<br>1: Procedure<br>2: Function<br>4: Package<br>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:<br>1: Row ID<br>2: Row Version<br>4: Auto increment field<br>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:<br>1: Non-unique<br>2: Unique<br>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'<br>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)<br>Trim(Field1, '-') | No | Removes third argument from front and back. If no third argument, trims spaces. |
| TrimLeft | TrimLeft(StringField)<br>TrimLeft(Field1, '$') <> '' | No | See Trim. |
| TrimRight | TrimRight(StringField)<br>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.

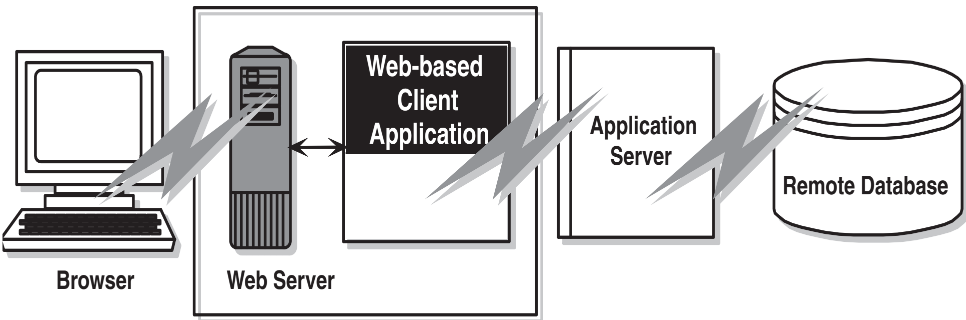
1 For the connection to the main remote data module, add and set up a connection component as described in “Connecting to the application server” on page 25-23. The only limitation is that you can’t use a CORBA or SOAP connection.
2 For each child remote data module, use a *TSharedConnection* component.
   - 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:

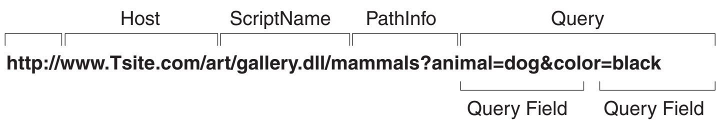
- RFC822, “Standard for the format of ARPA Internet text messages,” describes the structure and content of message headers.
- RFC1521, “MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies,” describes the method used to 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




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.

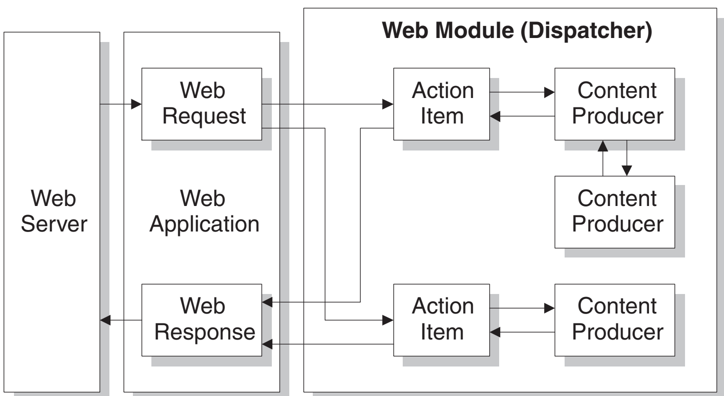
**Warning** Do not include the forms unit in the project **uses** clause after the CGIApp or ISAPIApp unit. Forms also declares a global variable named *Application*, and if it appears after the CGIApp or ISAPIApp unit, *Application* will be initialized to an object of the wrong type.

## 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:


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":

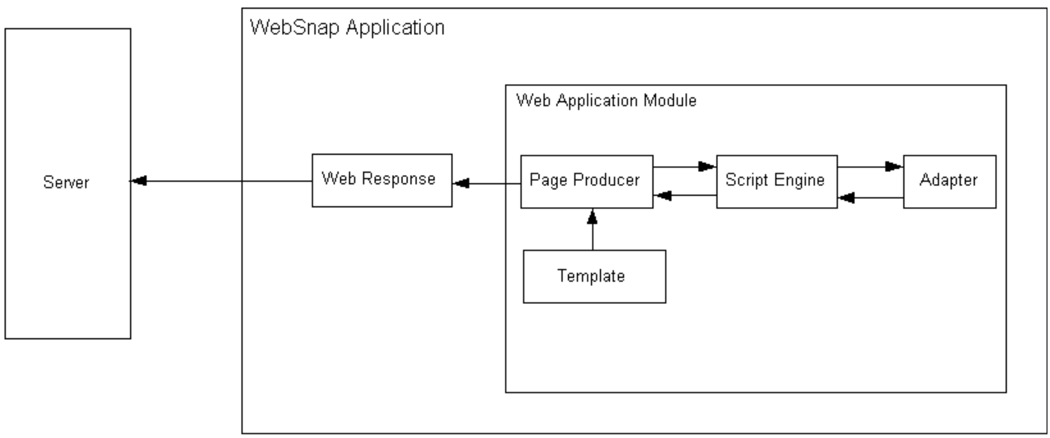
```
<a href="<%=Adapter1.EditRow.LinkToPage("Details", Page.Name).AsHREF%>">Edit...</a>
```

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.

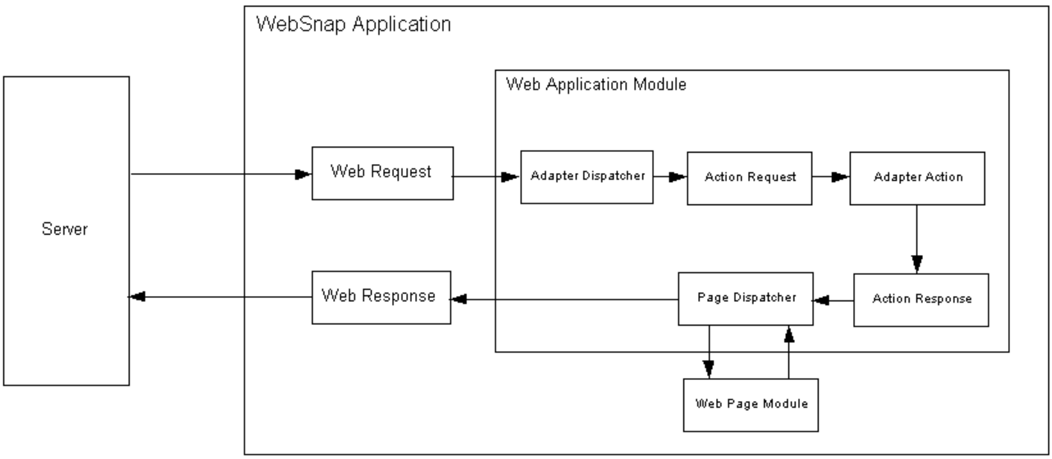
The Action response methods include:

- *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.

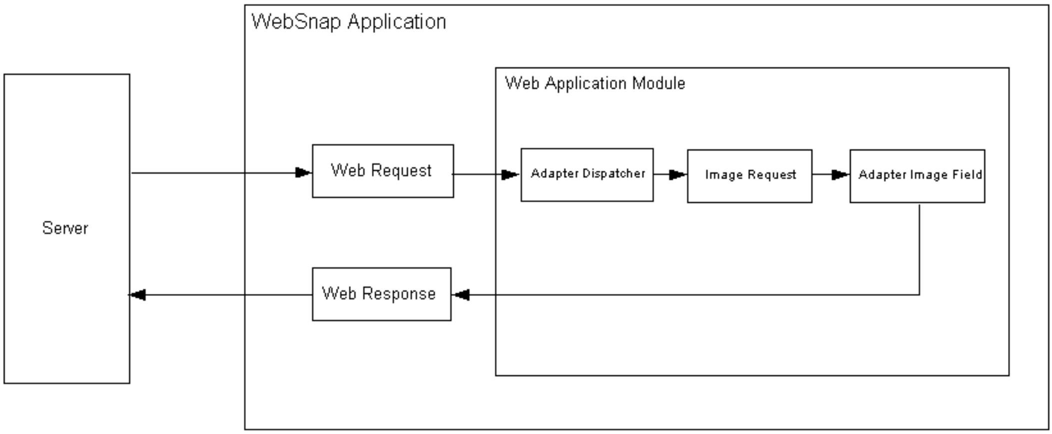
- Image Request Parameters - Identifies the parameters needed by the adapter image. For example, the *TDataSetAdapterImageField* object needs key values to identify the record that contains the image.

### 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

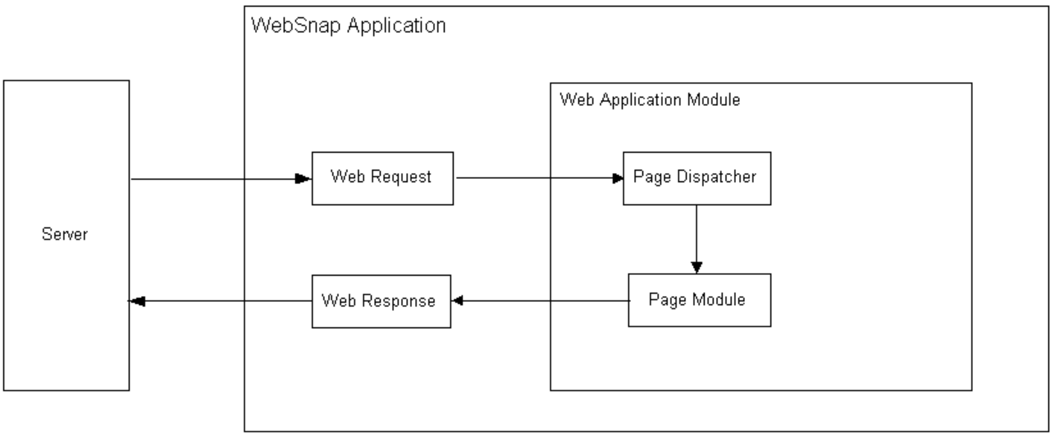
When the page dispatcher receives a client request, it determines the page name by checking the Pathinfo portion of the target URL's request message. If the pathinfo portion is not blank, the page dispatcher uses the ending word of pathinfo as the page name. If the pathinfo portion is blank, the page 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.

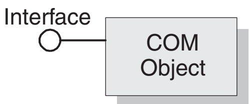
**COM server** A module, either an EXE, DLL, or OCX, that contains the code for a COM object. Object implementations reside in servers. A COM object implements one or more interfaces.

**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.

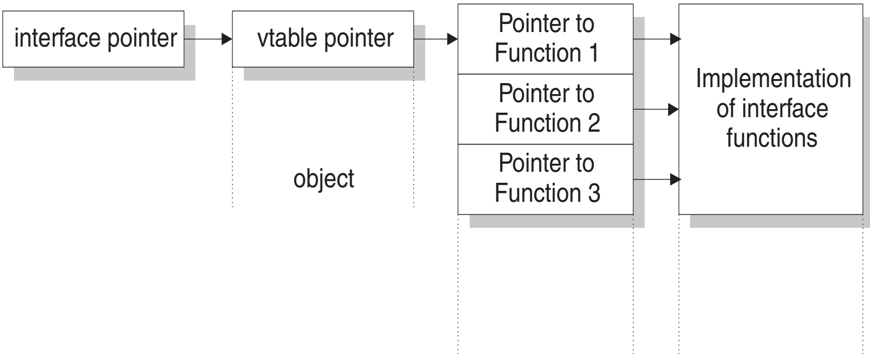
Objects track their own lifetime through the *IUnknown* methods, *AddRef* and *Release*, which are simple reference counting methods. As long as an object's reference count is nonzero, the object remains in memory. Once the reference count reaches zero, the interface implementation can safely dispose 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:

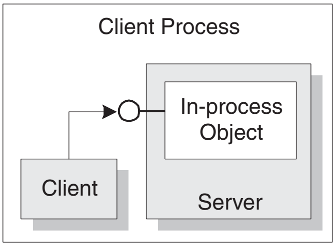
**In-process server** A library (DLL) running in the *same process space* as the client, for example, an ActiveX control embedded in a Web page viewed under Internet Explorer or Netscape. Here, the ActiveX control is downloaded to the client machine and invoked within the same process as the Web browser.

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.<br><br>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.<br><br>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.

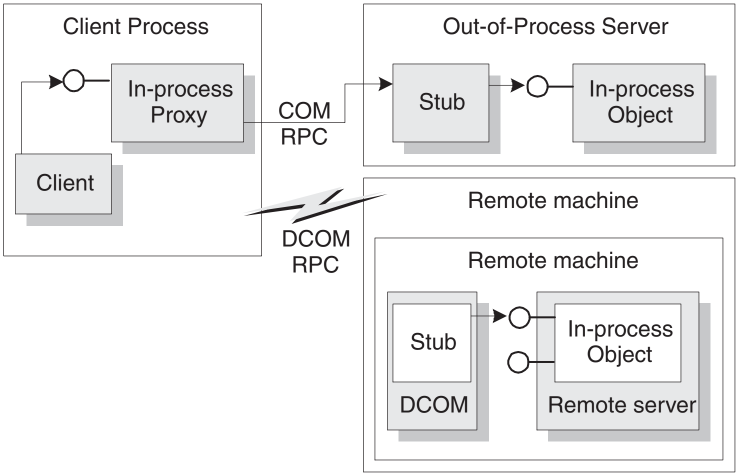
**Figure 33.3** In-process server




**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

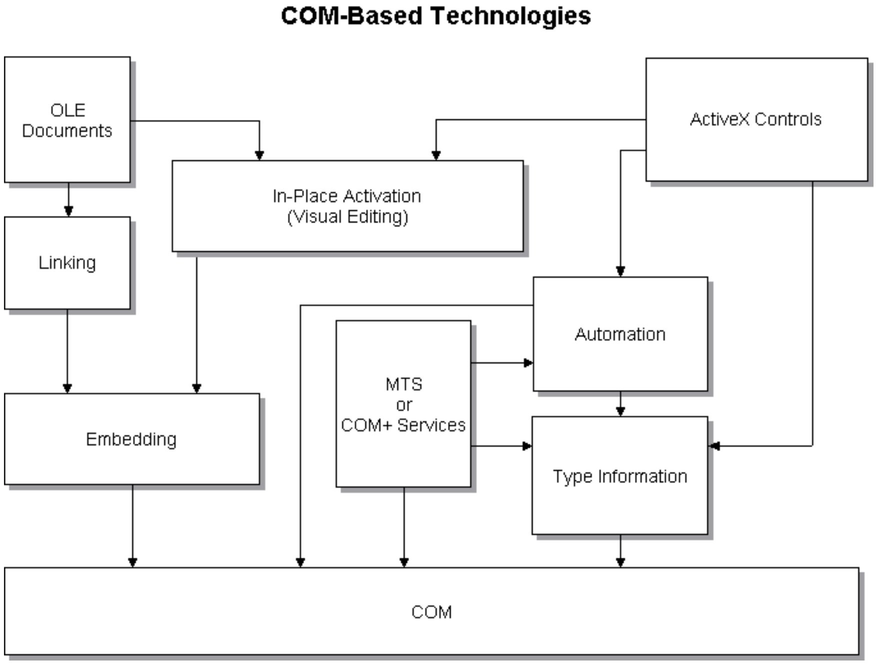
COM was originally designed to provide core communication functionality and to enable the broadening of this functionality through extensions. COM itself has extended its core functionality by defining specialized sets of interfaces for specific purposes.

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.

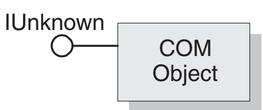
## Implementing COM objects with wizards

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.

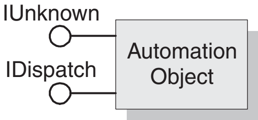
**Figure 33.6** Simple COM object interface




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

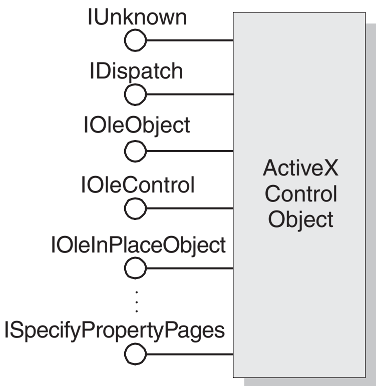



Table 33.2 lists the various wizards and the interfaces they implement:

**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:<br>Implements the interface that you specify, either dual or dispatch. Provides server-side support for generating events, if requested.<br>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.<br>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:<br>Generates a CoClass that corresponds to the VCL control on which the ActiveX control is based and which implements all the ActiveX interfaces.<br>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:<br>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)

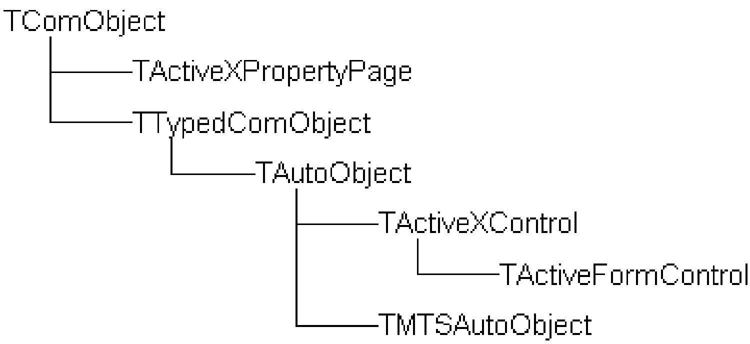
| Wizard | Implemented interfaces | What the wizard does |
|---|---|---|
| Type Library | None, by default | Creates a new type library and associates it with the active project. |
| ActiveX library | None, by default | Creates a new ActiveX or Com server DLL and exposes the necessary export functions. |

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.<br>Support for aggregation, OLE exception handling, and safecall calling convention on dual interfaces.<br>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:<br>Support for the *IDispatch* interface.<br>Auto-marshaling support. |
| ActiveX Control | TActiveXControl | Everything provided by *TAutoObject*, plus:<br>Support for embedding in a container.<br>Support for in-place activation.<br>Support for properties and property pages.<br>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:<br>Support for the *IObjectControl* interface. |
| Property Page | TPropertyPage (uses TActiveXPropertyPage internally) | Support for *IUnknown* and *ISupportErrorInfo* interfaces.<br>Support for aggregation, OLE exception handling, and safecall calling convention on dual interfaces.<br>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.<br>**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.<br>**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

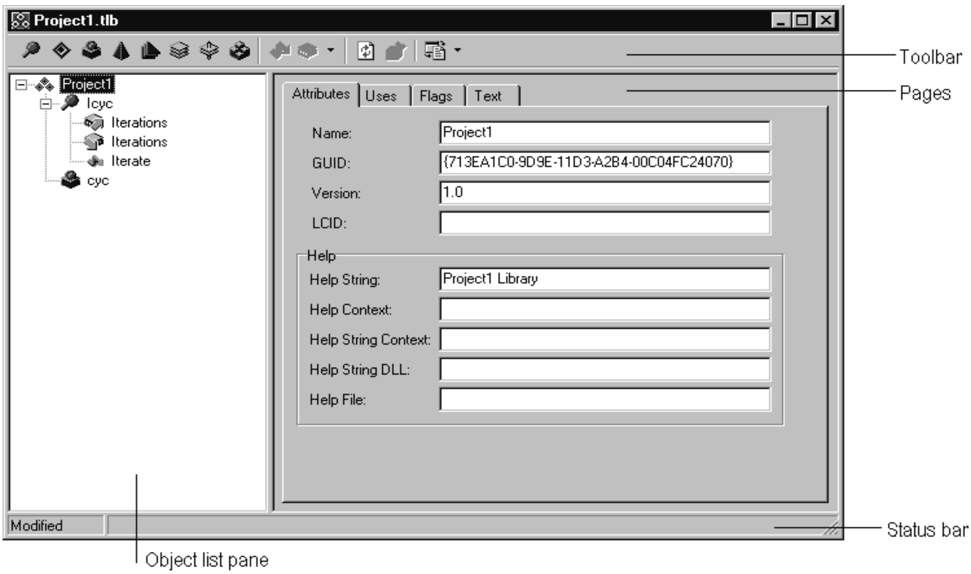
The main elements of the Type Library editor are described in Table 34.2:

**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




## 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. |

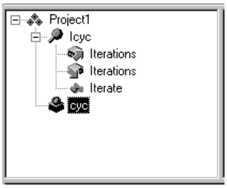
In the third box, you can choose to refresh, register, or export your type library (save it as an IDL file), as described in "Saving and registering type library information" on page 34-24.

## 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.<br>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.<br>The thread's local variables are reliable across multiple calls.<br>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.<br>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.<br>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.<br>This model should not be used with objects that have a user interface (visual controls).<br>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

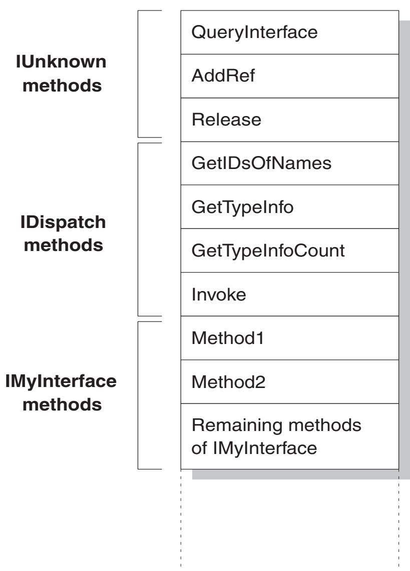
A dual interface is a custom interface and a dispinterface at the same time. It is implemented as a COM VTable interface that derives from *IDispatch*. For those controllers that can access the object only at runtime, the dispinterface is available. For objects that can take advantage of compile-time 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.

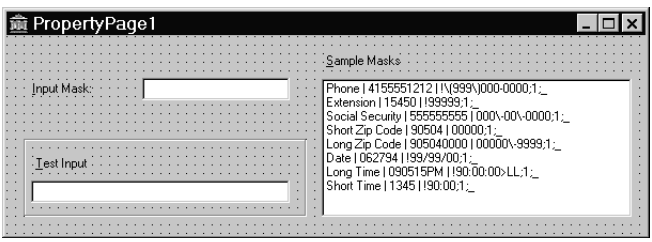
The wizard creates a new form and implementation unit for the property page. The form is a descendant of *TPropertyPage*, which lets you associate the form with the ActiveX control whose properties it edits.

## 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




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.<br>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.<br>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.<br>All objects of a single-threaded component execute on the main thread.<br>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.<br>Very limited scalability.<br>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.<br>Two objects can execute concurrently as long as they are not in the same activity.<br>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

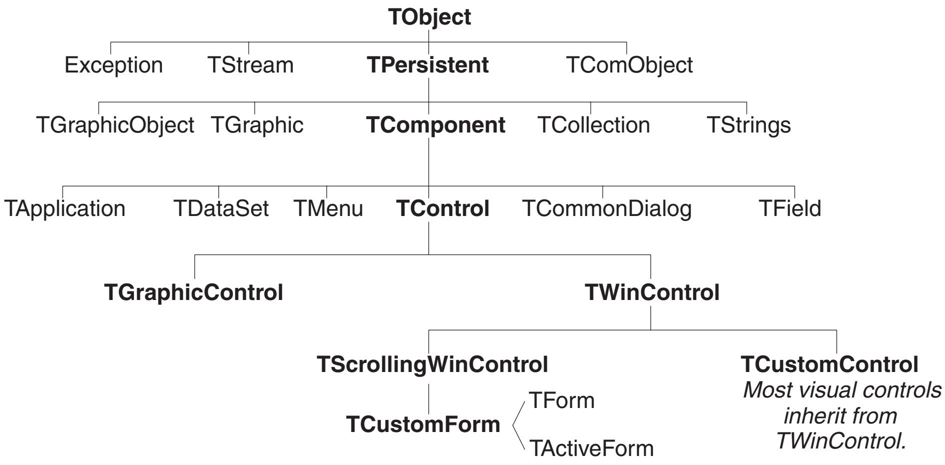
For information on installing new components, see “Installing component packages” on page 11-5.

## 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:

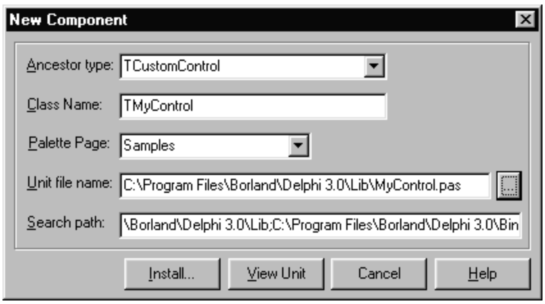
- Creating a unit
- Deriving the component
- Registering the component

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




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

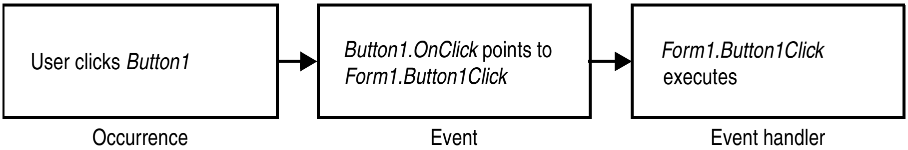
Events are implemented as properties, so you should already be familiar with the material in Chapter 42, “Creating properties,” before you attempt to create or change a component’s events.

## 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:




- 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* | *OnDragDrop* | *OnEndDrag* | *OnMouseMove* |
| *OnDblClick* | *OnDragOver* | *OnMouseDown* | *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:

Event → MainWndProc → WndProc → Dispatch → Handler

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

## C

# D

# E

# F

## G

## H

# I

## J



## K



## L

## M

# N

# O

## P

# Q



# R

# S

## T

## U

## X



## Y



## Z