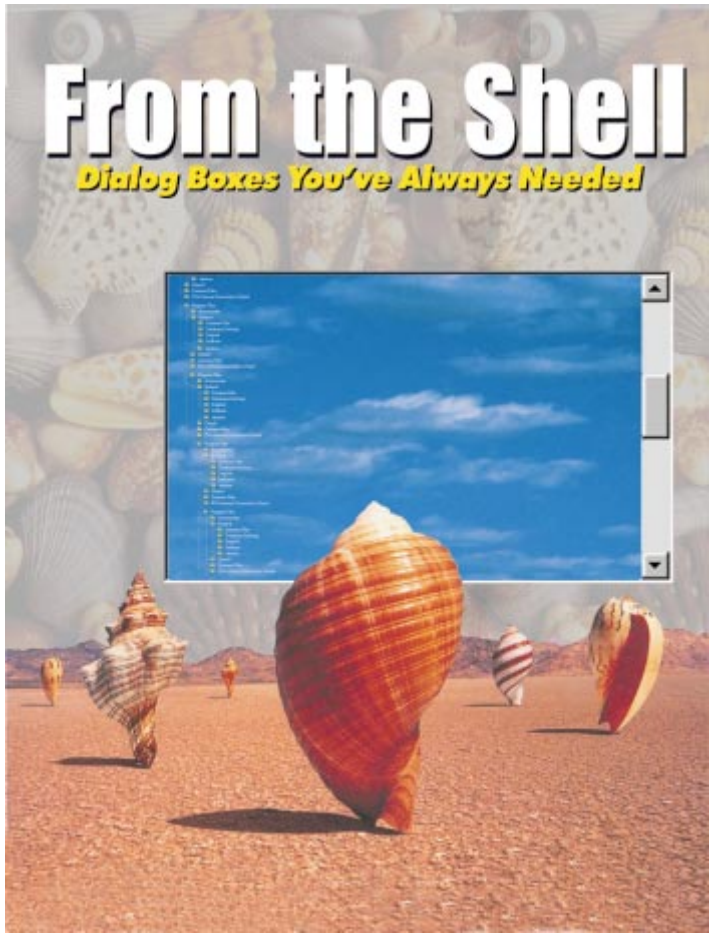
April 1999, Volume 5, Number 4




**Cover Art By:** *Darryl Dennis*

## ON THE COVER

**4 From the Shell** — Kevin J. Bluck and James Holderness
There are some standard Windows dialog boxes (e.g. Browse for Folder, Run, Format, etc.) that could prove handy in your Delphi applications. But how to get to them? Some are completely undocumented. Misters Bluck and Holderness show us how in this first part of their two-part series.

## FEATURES

**10 DBNavigator**
**Delphi Database Development: Part VII** — Cary Jensen, Ph.D.
There are three general ways to filter a dataset: a SQL SELECT statement, a range using an index, and the *DataSet.Filter* property. Dr Jensen explains the advantages and disadvantages of each, with examples.

**15 Algorithms**
**Three Searches** — Rod Stephens
Mr Stephens weighs the relative merits of three search techniques — Exhaustive, Binary, and Interpolation — and provides ready-to-run Delphi implementations of each.

**19 On Language**
**The Interface Advantage** — Eric Whipple
Despite their impressive features and acceptance, hierarchical objects can be inflexible — especially in distributed systems. Mr Whipple explains the relative advantages of the interface model for Delphi developers.

**25 Informant Spotlight**
**1999 Readers Choice Awards** — Chris Austria
It's that time of the year again. In fact it's that time of the millennium again, as our own Mr Austria relates your favorite third-party tools for the next one thousand years.

**29 OP Tech**
**Property Overriding** — Philip Brown
Using property overriding, as Mr Brown explains, it's possible to extend the standard functionality of *any* class using the built-in capabilities of Delphi — even if the accessor functions used to control the property are **private**.

## REVIEWS

**33 *Charlie Calvert's Delphi 4 Unleashed***
Book Review by Cary Jensen, Ph.D.

**34 *Mastering Delphi 4***
Book Review by Robert Vivrette

## DEPARTMENTS

**2 Delphi Tools**
**35 File | New** by Alan C. Moore, Ph.D.

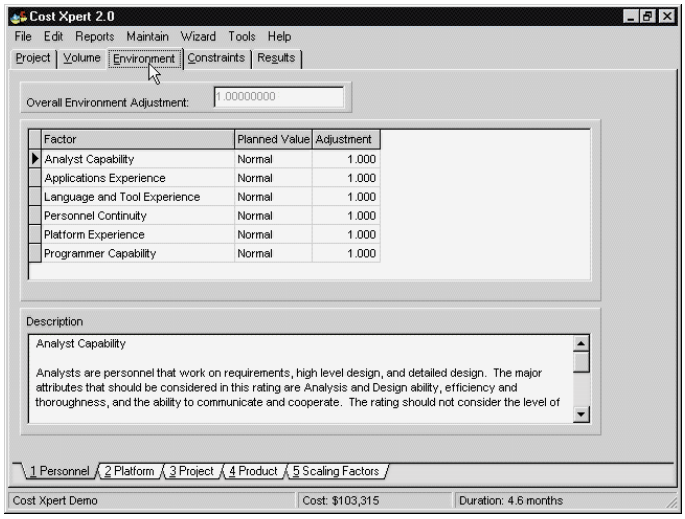
Delphi TOOLS

New Products and Solutions

## Marotz Releases Cost Xpert 2.0

**Marotz, Inc.** announced the release of ***Cost Xpert 2.0***, a new version of the company's software project estimating and planning tool.

Cost Xpert 2.0 helps developers estimate expected time and resources that a given software development project will consume, as well as provide a project baseline plan that can be imported into a management tool, such as Microsoft Project. Cost Xpert 2.0 can estimate projects developed in over 500 programming languages.




Cost Xpert 2.0 supports seven costing methodologies, including COCOMO II and Feature Points. The new version is also able to provide estimates for Year 2000-compliance development.

**Marotz, Inc.**
**Price:** US$1,995 for a single-user license; discounts are available for multi-seat licenses.
**Phone:** (800) 477-6168
**Web Site:** http://www.marotz.com

## Dart Announces PowerTCP Professional Edition

**Dart Communications** introduced ***PowerTCP Professional Edition***, an Internet toolkit with support for six development environments; PowerTCP Pro includes 16- and 32-bit controls, libraries, DLLs, VBXs, and sample applications for Delphi, Visual Basic, Visual C++, Borland C++, C++Builder, and PowerBuilder.

PowerTCP Pro includes TCP, TELNET, FTP, SMTP, POP3, UDP, SNMP, TFTP, VT320 emulation, and HTTP components.

PowerTCP Pro is version 4.1 of Dart's PowerTCP Internet toolkit line, and includes additional enhancements, such as apartment model threading and the capability to dynamically instantiate the controls without placing them on a form. The controls can then be used in non-blocking mode with events. These formless controls were used to build the FTP and SMTP server samples included in the Professional Edition.

**Dart Communications**
**Price:** US$598
**Phone:** (315) 431-1024
**Web Site:** http://www.dart.com

## AnyWare Announces AppTools VCL 2.01

**AnyWare Ltd.** released ***AppTools VCL 2.01***, a new version of the company's set of nine native components for Delphi 3 and 4 and C++Builder 3. In addition to improving the previous components, AppTools VCL 2.01 adds five new components that offer drop-in functionality.

The new components are *TATMostRecentlyUsedList*, which enables developers to add their own re-open menus, etc.; *TATControlEnabler*, which enables/disables groups of controls; *TATSettings*, which stores application settings; *TATShowURLButton*, which goes to a Web site when clicked; and *TATHyperLink*, which looks like a link in a browser.

The enhanced components are *TATWizard*, *TATTipOfTheDay*, *TATSplashScreen*, and *TATBrowseButton*.

AppTools includes detailed example applications with source and a comprehensive IDE-compatible help file.

**AnyWare Ltd.**
**Price:** US$30; current users of AppTools 1.x can upgrade for US$15.
**Fax:** (+44) 0 117 973 6888
**Web Site:** http://www.anyware.co.uk/anyware/apptools

## ZieglerSoft Announces ZieglerCollection one 1.50

**ZieglerSoft** announced the availability of ***ZieglerCollection one 1.50***, a set of over 70 components, functions, and routines for Delphi 1 through 4, as well as C++Builder 1 and 3.

Version 1.50 components include *TzMinMax*, *TzBigLabel*, *Tz3Dlabel*, *TzAngleLabel*, *TzTabListBox*, *TzBitmap*, *TzAnimated*, *TzBackground*, *TzBlendPaint*, *TzTileMap*, *TzLed*, *TzSegment*, *TzSegmentLabel*, *TzSegmentClock*, *TzGauge*, *TzSlideBar*, *TzFrame*, *TzDivider*, *TzMovePanel*, *TzTitleBar*, *TzHint*, *TzShowApp*, *TzVerSpilt*, *TzHorSplit*, *TzMouseSpot*, *TzCalc*, *TzShapeBtn*, *TzColorBtn*, *TzGradBtn*, *TzBitColBtn*, *TzIconColBtn*, *TzScope*, *TzPanelMeter*, *TzKnob*, *TzDblKnob*, *TzTripKnob*, *TzTrayIcon*, and others.

Functions and procedures are available for manipulating bitmaps, getting system information, and more.

**ZieglerSoft**
**Price:** US$52 (includes full source code).
**Phone:** (+45) 9811 3772
**Web Site:** http://www.zieglersoft.com

Delphi TOOLS

New Products and Solutions

## Cocolsoft Announces Cogencee

**Cocolsoft Computer Solutions** announced *Cogencee*, a compiler generator for Delphi written in Delphi. The compiler language (Cocol) behind Cogencee was used to generate itself.

Developers can use the compiler language (Cocol) and their Delphi embedded code to produce scanners, parsers, compilers, interpreters, language checkers, natural language processors, expert system shells, scripting languages, and calculators.

Cogencee is available for Delphi 1 (16-bit) for Windows 3.1 and Delphi 4 (32-bit), and comes with over a dozen examples, such as an expression calculator and MP (mini Pascal) compiler and interpreter.

**Cocolsoft Computer Solutions**
**Price:** Standard, US$300; Professional (includes source code), US$500.
**E-Mail:** info@cocolsoft.com.au
**Web Site:** http://www.cocolsoft.com.au

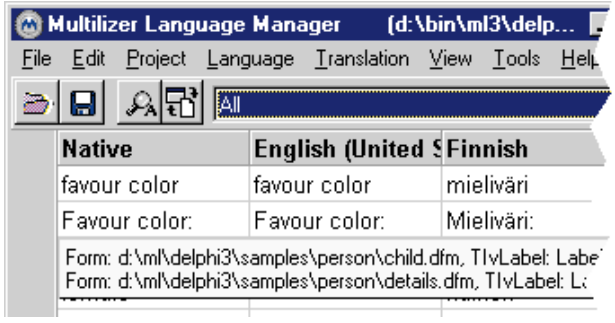

## Innoview Data Announces MULTILIZER VCL Edition 4.0

**Innoview Data Technologies** announced *MULTILIZER VCL Edition 4.0*, which improves work sharing between localization team members.

The MULTILIZER VCL is based on a platform-independent MULTILIZER Dictionary-Translator Architecture (MDTA), which keeps language-related tasks apart from programmer's tasks. The development is done with minimum impact on the source code.

MULTILIZER VCL Edition 4.0 includes enhanced VCL components and the Language Manager utility. Both provide new functionality, including support for Delphi 1 through 4 and C++Builder 1 or 3; the ability to translate resource strings (BDE, system, and error messages get translated); and year 2000 correction to *ShortDateFormat* and *LongDateFormat* variables.

MULTILIZER VCL Edition 4.0 also includes enhanced support for localization workgroups. The Language Manager now includes Deploy Wizard, which is used for sharing the dictionary data. Using it, developers can specify which languages they want to have translated by different persons. The Import Wizard re-integrates the information into a project.




**Innoview Data Technologies**
**Price:** Standard without source, US$290; Standard with source, US$580; Pro without source, US$790; Pro with source, US$1,580.
**Phone:** +358 9 4762 0550
**Web Site:** http://www.multilizer.com

## HREF Tools Ships WebHub VCL 1.67

**HREF Tools Corp.** announced *WebHub VCL 1.67*, which offers Delphi programmers a framework and set of components and tools for developing Web sites.

WebHub's file structure separates the EXE from HTML, JavaScript, Java, and multimedia files. When an application server program built with WebHub runs, it uses these separately stored and maintained files, and serves dynamic Web pages based on their content. Files may be changed and the application server may be refreshed at any time — without recompiling. This site maintenance process can be carried out at the machine or remotely.

Version 1.67 offers support for nested macros. WebHub macros are used in the "scripting" that occurs outside of Delphi by anyone running the WebHub application server on Windows NT. Using macro syntax, HTML specialists can re-use "chunks" of HTML and, from these external files, execute WebHub "Web action" components in the compiled application server. These "Web action" components are regular Delphi components whose behavior is customized by programming their *OnExecute* event. COM/DCOM is not required, but is supported.

**HREF Tools Corp.**
**Price:** From US$430
**Phone:** (707) 542-0821
**Web Site:** http://www.href.com




*By Kevin J. Bluck and James Holderness*

# From the Shell

## Part I: Dialog Boxes You've Always Needed

The Windows common dialog boxes provided in Comdlg32.dll can be immensely useful, but they don't cover every situation. There are plenty of other commonplace system dialog boxes you may need to use, but there are no clear instructions in the Windows documentation about how to access many of them. Trying to duplicate the same interface by building a dialog box manually can be extremely frustrating and time consuming. It's also unnecessary, because those dialog boxes are available for anybody who knows where to look. In this part of a two-part series on Windows shell dialog boxes, we'll be revealing a few of the well-known and undocumented dialog boxes provided by the shell that you can use in your own applications.

### Browsing for Folders

Most Delphi programmers know how to use the VCL's *TOpenDialog* to allow the user to browse for files to open. Sometimes, however, you want the user to browse for a folder, rather than a specific file. This is the purpose of the shell's Browse for Folder dialog box (see Figure 1). This dialog box is available via the documented function *SHBrowseForFolder*, shown here (this function is defined in the standard VCL *ShlObj* unit):

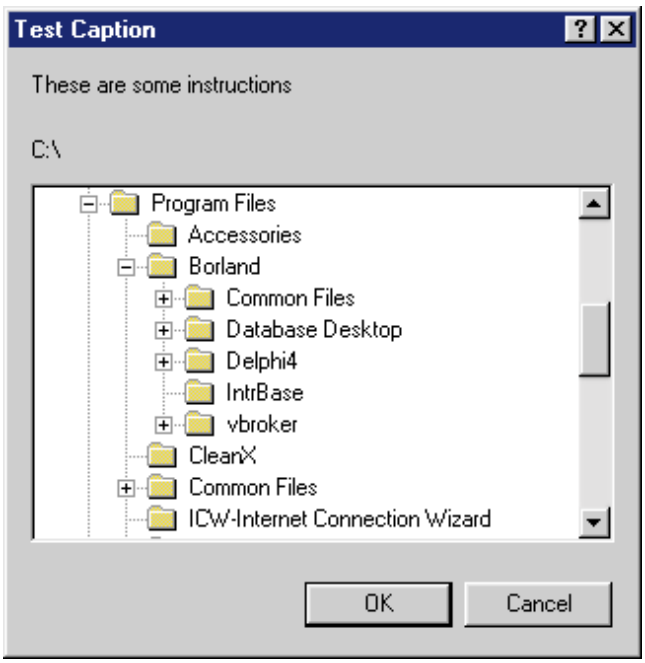



**Figure 1:** From the Windows shell: the Browse for Folder dialog box.

```
function SHBrowseForFolder(var BrowseInfo:
                           TBrowseInfo):
  PItemIDList; stdcall;
```

This function takes only a single parameter, but don't be deceived. This parameter is a very complicated record type: *TBrowseInfo*. The structure of *TBrowseInfo* is as follows:

```
TBrowseInfo = packed record
    hwndOwner:      HWND;
    pidlRoot:       PItemIDList;
    pszDisplayName: PChar;
    lpszTitle:      PChar;
    ulFlags:        UINT;
    lpfn:           TFNBFFCallBack;
    lParam:         LPARAM;
    iImage:         Integer;
  end;
```

The *hwndOwner* data member contains the handle of the window that will be the owner of the browser dialog box. You may set this to 0 if you don't wish to specify an owner window handle. The *pidlRoot* data member points to a PIDL specifying the location of the "root" folder to be shown in the dialog box. [For a description of PIDLs and their use, see "Shell Notifications" by Kevin J. Bluck and James Holderness in the March, 1999 *Delphi Informant*.] Only that root folder and its sub-

folders will appear in the dialog box. This member can be set to **nil**, in which case the root folder will be Desktop. The *pszDisplayName* data member points to a buffer that will receive the "display name" of the folder selected by the user. You must ensure that the buffer you pass is capable of accepting at least MAX_PATH number of characters. The *lpszTitle* data member points to a null-terminated string that will be displayed above the tree view of folders in the dialog box. This string can be anything you like, but is most often used to give instructions for the user. Be careful you don't make this string very long, or it will be truncated in a rather ugly fashion. The *ulFlags* data member allows you to specify flags that will govern the types of folders that will be shown in the dialog box, along with other dialog box options. This data member can include zero, or more, of the values shown here, combined with a logical **or** operator:

```
// Browsing for directory.
// Find file-system directory.
BIF_RETURNONLYFSDIRS   = $0001;
// Don't browse into net domains.
BIF_DONTGOBELOWDOMAIN  = $0002;
// Leave room for status text.
BIF_STATUSTEXT         = $0004;
// Find only file-system ancestors.
BIF_RETURNFSANCESTORS  = $0008;
BIF_BROWSEFORCOMPUTER  = $1000;
BIF_BROWSEFORPRINTER   = $2000;  // Find a printer.
BIF_BROWSEINCLUDEFILES = $4000;  // Find anything.
```

Note that if you want the dialog box to actually show the custom status text you specified in the *lpszTitle* data member, you must include the BIF_STATUSTEXT flag. The *lpfn* data member is a pointer to a function of type *TFNBFFCallBack*. This function type is shown in the following:

```
TFNBFFCallBack = function(DialogHandle: HWND;
  MessageID: UINT; PIDL: PItemIDList; Data: LPARAM):
  Integer; stdcall;
```

This function is a callback, and can be used to control and update the dialog box as the user interacts with it. It's up to you to implement this callback function in your code if you want to take advantage of this facility. If you don't wish to control the dialog box, you may set this data member to **nil**. The *lParam* data member allows you to specify some four-byte value, which will be sent to the callback function specified in the *lpfn* data member. This is most commonly some sort of pointer that will be meaningful to you at the time of the callback. You may set it to 0 if you wish. The *iImage* data member need not be set before the call to *SHBrowseForFolder*; it will receive the system image list index of the image associated with the folder the user decided to select. Initialize it to 0 if you like.

The *SHBrowseForFolder* function returns the PIDL that uniquely specifies the selected folder. You may convert this PIDL to a traditional path, assuming the folder is a file system object, using the documented shell function *SHGetPathFromIDList*, or the *GetPathFromPIDL* function provided in the *kbsdPIDL* unit included with this article's source code (see end of article for download details). You, the caller, are responsible for freeing the returned item identifier list, using either the *Free* method of the *IMalloc* COM interface, or the *PIDLFree* procedure provided in the *kbsdPIDL* unit. Don't try to free the PIDL using *FreeMem*, or any other method besides *IMalloc* or *PIDLFree*. If the user chose the **Cancel** button in the dialog box, the return value will be **nil**, and nothing needs to be freed (although you won't hurt anything by doing so).

Now that we've shown the dialog box, let's see how to control it in response to user actions. This is the purpose of the callback function we specify in the *TBrowseInfo* record. This callback is of type *TFNBFFCallBack*, shown previously.

You have to reverse your usual thinking with this function. Remember that you aren't calling this function; rather, you are implementing it, and the system will be calling it.

The *DialogHandle* parameter contains the window handle of the dialog box. You may use this handle for any normal Windows purpose, but it's typically used for sending messages to the dialog box. The *MessageID* parameter is not a *TMessage* record; rather, it is the identifier of the message the dialog box is sending you via the callback. It can be one of these two values:

```
BFFM_INITIALIZED = 1;  // The dialog is about to appear.
BFFM_SELCHANGED  = 2;  // The user picked something.
```

The PIDL parameter will contain whatever additional data is appropriate for the message. If *MessageID* is BFFM_INITIALIZED, this value will be **nil**. If *MessageID* is BFFM_SELCHANGED, this value will be a PIDL that identifies the folder the user has selected. The *Data* parameter contains whatever value you assigned to the *lParam* data member of the *TBrowseInfo* record. This can be useful for passing a pointer to a Delphi component (as you will see later), or any other four-byte value that interests you. An abbreviated example of an implementation of this callback function is shown in Figure 2.

From within this callback, you can send three custom messages back to the dialog box to change things in response to user actions. Here are the message IDs:

```
// Changes the dialog's status text.
BFFM_SETSTATUSTEXT = WM_USER + 100;
// Enables or disables OK button.
BFFM_ENABLEOK      = WM_USER + 101;
// Changes the selected folder.
BFFM_SETSELECTION  = WM_USER + 102;
```

Typically, these messages are sent to update the dialog box in response to user selections, signaled by the BFFM_SELCHANGED notification. You can also send any other message you would normally be able to send to a dialog box window. For example, you may send a WM_SETTEXT message to change the dialog box's title bar caption.

```
function BrowseForFolderCallback(DialogHandle: HWND;
  MessageID: UINT; PIDL: PItemIDList; Data: LPARAM):
  Integer;
begin
  // Respond to notification messages from the dialog.
  case (MessageID) of
    BFFM_INITIALIZED:
      DialogInitialized(DialogHandle, Data);
    BFFM_SELCHANGED:
      HandleNewSelection(DialogHandle, PIDL, Data);
  end;
  Result := 0;  // Always return 0.
end;
```

**Figure 2:** *TFNBFFCallBack* example.

Delphi doesn't define a special message record type for these messages, so simply use the standard *TMessage* type. This is a variant record type, but you should consider it to be defined as follows:

```
TMessage = record
  Msg:    UINT;
  WParam: LPARAM;
  LParam: LPARAM;
  Result: UINT);
end;
```

You send these messages using the standard API calls *SendMessage* or *PostMessage*, using the window handle to the dialog box provided by the *DialogHandle* parameter in the callback and a *TMessage* record, which you provide. Fill the message parameters according to each message ID, as shown in Figure 3. Note that the documentation in the Delphi Windows API Help file has the parameters for BFFM_ENABLEOK backwards. An example of updating the selection using a path is shown here:

| Message ID | WParam | LParam |
|---|---|---|
| BFFM_SETSTATUSTEXT | Not used | *PChar* pointing to new status text to set |
| BFFM_ENABLEOK | Not used | True to enable, False to disable |
| BFFM_SETSELECTION | True if *LParam* is a path, False if *LParam* is a PIDL | *PChar* pointing to the path to select or PIDL identifying the folder to select |

**Figure 3:** *SHBrowseForFolder* update message parameter values.

```
PostMessage(DialogHandle, BFFM_SETSELECTION, True,
  LPARAM(PChar(NewPath)));
```

## The About Windows Dialog Box

If you feel the need for some reason, you can show the About Windows dialog box (see Figure 4) in your applications. This dialog box is customizable to a small extent, but it's really only suitable for displaying the Windows logo and text. Frankly, we're surprised Microsoft allowed any customization at all. They've provided just enough leeway for practical jokers to cause mischief, but not enough to make it generically useful for other applications. This function is defined in the standard VCL *ShellAPI* unit:

```
function ShellAbout(Owner: HWND; ApplicationName: PChar;
  OtherText: PChar; IconHandle: HICON): Integer; stdcall;
```

The *Owner* parameter identifies the window that will own the dialog box. You may set this value to 0 if you wish to specify no owner. The *ApplicationName* parameter contains text that will display in the title bar of the dialog box, and on the first line of the dialog box after the mandatory "Microsoft" text. If the string includes the "#" character, this acts as a separator. In this case, the function will display the first part before the separator in the title bar, and the second part on the first line after the "Microsoft" text. The *OtherText* parameter contains text that will be shown in the dialog box after the Microsoft version and copyright text. The *IconHandle* parameter identifies an icon that will be displayed in the dialog box. If this parameter is 0, the function displays the default Microsoft Windows or Microsoft Windows NT icon. Note that these default Microsoft icons are quite large, so showing a typical 32 x 32 icon will look rather unattractive. If the function succeeds in displaying the dialog box, the return value is non-zero; otherwise, the return value is zero.

## Formatting Disks

The next function we'll discuss, *SHFormatDrive*, displays the Format dialog box (see Figure 5), and is semi-documented. Currently, the *SHFormatDrive* function is not in the Platform SDK documentation. However, Microsoft admits to its existence and exports it from Shell32.dll by name. The Delphi definition of the function is shown here:

```
function SHFormatDrive(Owner: HWND; Drive: UINT;
  FormatID: UINT; OptionFlags: UINT): DWORD; stdcall;
```

Microsoft's "official" documentation of this function, for now, consists solely of Microsoft Knowledge Base article ID Q173688, which, at press time, can be found on the Web at http://support.microsoft.com/support/kb/articles/q173/6/88.asp.


**Figure 4:** The About Windows dialog box.

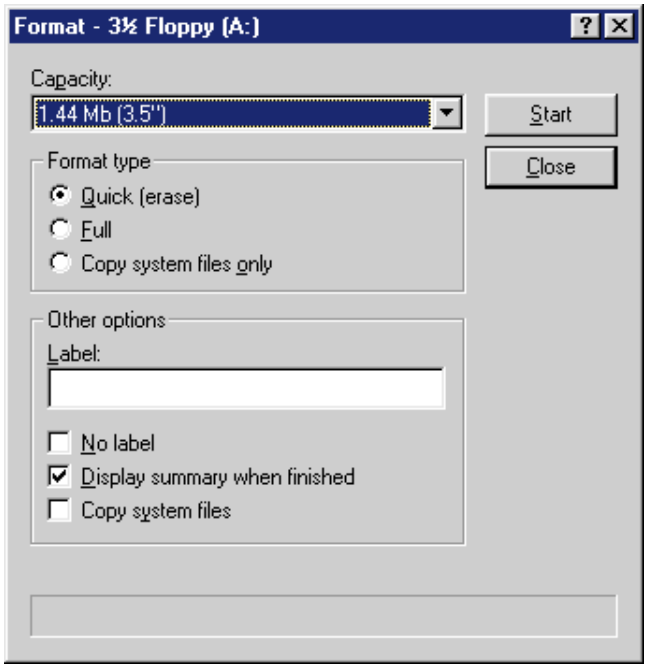



**Figure 5:** The Windows 95 Format dialog box.

The *Owner* parameter identifies the window that will own the dialog box. The "documentation" recommends you do not set this value to 0, but we notice no ill effects from doing so. The *Drive* parameter is where you specify a number that identifies the drive you wish to format. This is zero-based, starting at the A: drive, e.g. A: = 0, B: = 1, etc. You may only format one drive at a time. The selected drive identifier will appear in the dialog box's title bar caption. The *FormatID* parameter allows you to specify a "template" for formatting. Usually, you will simply pass the constant value SHFMT_ID_DEFAULT, which tells the dialog box to default to the default format scheme for the disk type in question. In Windows 95, you may also pass the low-order word from the return value of a prior call to this function to specify that the default scheme should be whatever format scheme was used by the prior call. The *OptionFlags* parameter takes bit-mask flags that allow you to set defaults. Currently, only two are officially defined. The following shows these flags and their values:

```
SHFMT_OPT_FULL     = $0001; // Check "Quick Format".
SHFMT_OPT_SYSONLY  = $0002; // Check "Sys Only".
```

Microsoft left the door open to define more, so you should ensure that all other bits in the mask are off to avoid unexpected side effects. The best way to do this is to set the bit-mask variable to 0 before setting desired flags.

Note that Windows 95 and NT implement different dialog boxes for this function. We have observed several variations from the documented standard for this function under NT 4.0. Setting the SHFMT_OPT_FULL is supposed to deselect the "Quick Format" checkbox, but it appears to actually do the opposite in NT. Apparently, somebody programmed the logic backwards. It works as expected in Windows 95. Also, setting the SHFMT_OPT_SYSONLY flag prevents the dialog box from appearing at all in NT, apparently because you can't SYS a disk under NT. You'd think NT would simply ignore the flag, but the dialog box is completely suppressed instead. These flags are mutually exclusive in the Windows 95 dialog box, because these choices are radio buttons. If you set both, SHFMT_OPT_SYSONLY wins. The final NT mutation is the return value. Apparently, it never returns anything except 0 for success, or SHFMT_ERROR on failure, regardless of the reason. It also returns 0 if you set the deadly SHFMT_OPT_SYSONLY flag. This eliminates some capabilities of the return value advertised in the Knowledge Base documentation, as will be explained next.

The *Return* value is a great example of how *not* to use a return value. If the function failed for some reason, it returns one of three error constants to identify the specific problem. These error constants are shown here:

```
SHFMT_NOFORMAT  = $FFFFFFFD;  // Drive is not formatable.
SHFMT_CANCEL    = $FFFFFFFE;  // Last format was canceled.
// Error, but drive may be formatable.
SHFMT_ERROR     = $FFFFFFFF;
```

If the function succeeded, under Windows 95, it returns the physical format ID of the last successful format. As mentioned, the low-order word of this value can be passed on subsequent calls to *SHFormatDrive* as the *FormatID* parameter to request a format of the same scheme. NT, however, always returns only 0 for success. Got all that? This means that to make any intelligent use of the return value, you must first test that it's not one of the three error constants, and, if not, then use the value for a completely different purpose. Apparently, the developer who came up with this function had never heard of reference parameters.

With this funky interface and the apparent problems in the NT version of this function, it's no wonder they don't want to publicize it.

## Windows NT and WideChar

Before we get involved in the completely undocumented functions, there is an important point of which you should be aware. Notice that the null-terminated string-type parameters for the undocumented functions are mostly declared as type *Pointer* instead of *PChar*. This is due to a little booby-trap, which is commonly the case for undocumented functions. All these string-type parameters declared as type *Pointer* take *PAnsiChar* on Windows 95, and take *PWideChar* on Windows NT. There is no choice of ANSI or UNICODE characters as would be expected for a documented function. Windows 95 uses the *PAnsiChar* version only, and Windows NT uses the *PWideChar* version only — take it or leave it. If you want your applications to function correctly on both platforms, you're going to have to check what operating system is in use at run time. If it's NT, you'll need to convert any string parameters to *PWideChar* before calling the function. When the function returns, you'll also need to convert any returned strings back to the *PAnsiChar* type again. It may be annoying, but that's the price you pay for using undocumented functions.

## Choosing Icons

The first completely undocumented function we'll be discussing is *PickIconDlg*. This function shows a dialog box with which the user can select an icon resource from a file (see Figure 6). It's used by the file type editor when selecting the icon to associate with a particular file type. It's also used in the shortcut properties dialog box when changing the icon. This function is exported from Shell32.dll by ordinal value 62, and the function declaration is as follows:

```
function PickIconDlg(Owner: HWND; FileName: Pointer;
  MaxFileNameChars: DWORD; var IconIndex: DWORD):
  LongBool; stdcall;
```

The *Owner* parameter identifies the window that owns the dialog box. The *FileName* parameter points to a buffer containing the

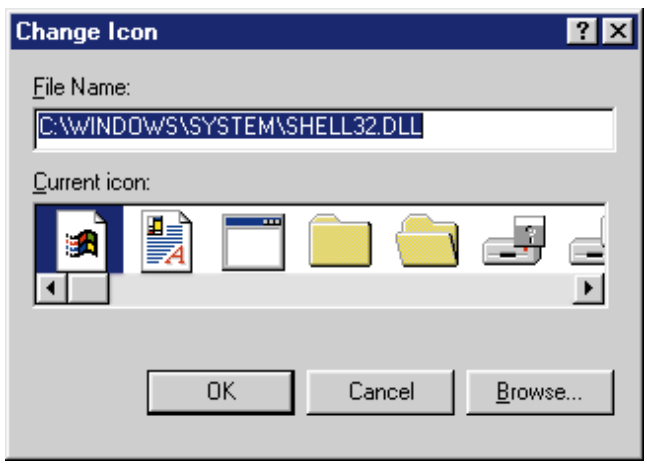



**Figure 6:** The Change Icon dialog box.

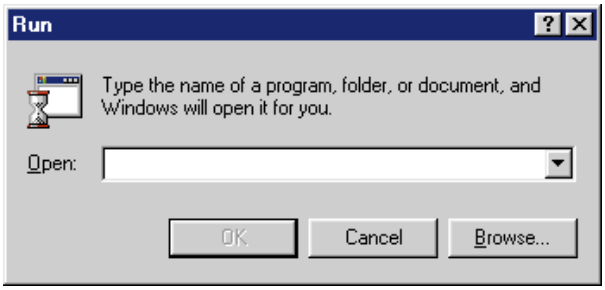



**Figure 7:** The Run dialog box.

name of the initial file that the dialog box will be browsing for icons. When the function returns, this buffer will contain the name of the file in which the user eventually found the desired icon. Because this is essentially a **var** parameter, and the new string could be longer than the original, it's best to provide a buffer capable of holding MAX_PATH characters, plus a null terminator, rather than directly casting a string-type variable. The *MaxFileNameChars* parameter specifies the size, in characters, of the *FileName* buffer. The *IconIndex* parameter takes the zero-based index of the icon that will be selected when the dialog box opens. When the function returns, this parameter will be set to the index of the icon last selected by the user.

If the user selects an icon, the return value is True. It's False if the user chooses the **Cancel** button, or the **Close** command on the **System** menu.

## Running Applications

The next function, *RunFileDlg*, is surprisingly flexible. It's the dialog box you see when launching applications from the **Run** selection of the **Start** button menu (see Figure 7). The function is exported from Shell32.dll by ordinal value 61; here's its function declaration:

```
procedure RunFileDlg(Owner: HWND; IconHandle: HICON;
  WorkPath: Pointer; Caption: Pointer;
  Description: Pointer; Flags: UINT); stdcall;
```

The *Owner* parameter identifies the window that owns the dialog box. The *IconHandle* parameter is the handle of the icon that will be displayed in the dialog box. If it's **nil**, the default icon will be used. The *WorkPath* parameter points to a string that specifies the working directory for the application that is run. The *Title* parameter points to a string to be placed in the title bar of the dialog box. If it's **nil**, the default title is used. The *Description* parameter points to a string displayed in the dialog box, briefly informing the user what to do. If it's **nil**, the default description is used. The *Flags* parameter is a set of bit flags that specify other properties of the dialog box. The following is the full list of flags:

```
RFF_NOBROWSE       = $01;  // Removes the browse button.
RFF_NODEFAULT      = $02;  // No default item selected.
// Determines the work directory from the file name.
RFF_CALCDIRECTORY  = $04;
RFF_NOLABEL        = $08;  // Removes the edit box label.
// Removes the Separate Memory check box (NT Only).
RFF_NOSEPARATEMEM  = $20;
```

A nice feature of this dialog box is that it allows you to control which applications the user may run. When the user selects the **OK** button, the dialog box's parent window is sent a notification message with details of the program about to be started. The notification is in the form of a WM_NOTIFY message with the notification code set to RFN_VALIDATE (-510) and the *lParam* pointing to a *TNM_RunFileDlg* record. This record definition can be seen in the following:

```
TNM_RunFileDlg = packed record
    hdr:         TNMHdr;
    lpFile:      Pointer;
    lpDirectory: Pointer;
    nShow:       LongBool;
  end;
```

The *hdr* data member is of type *TNMHdr*, a standard Windows data type included with every WM_NOTIFY message as the first data in the record pointed to by the pointer in the message's *LParam*. In other words, you can always assume the *LParam* will point to a *TNMHdr*, and, depending on the message type, there may be some additional data immediately following that record, as in this case. For the uninitiated, the *TNMHdr* data type is shown here:

```
TNMHdr = packed record
  hwndFrom: HWND;
  idFrom:   UINT;
  code:     UINT;
end;
```

The *hwndFrom* data member contains the window handle of the control sending the message. The *idFrom* data member contains the identifier of the control sending the message. The *code* data member contains the notification code that identifies the specific message being sent.

The "extra" data packaged after the *TNMHdr* record for this message consists of three additional data members. The *lpFile* data member points to a string containing the fully qualified path of the file to be run. The *lpDirectory* member points to a string specifying the working directory to be used by the application being run. Finally, the *nShow* data member specifies whether the application being run will be visible.

For this particular message, we're only interested in testing the *code* data member of the *TNMHdr* record to ensure we are receiving a Run File validation message, and that we will have access to the additional *TNM_RunFileDlg* data members. We know we have a *TNM_RunFileDlg* record available when the *TNMHdr* record's code data member is equal to RFN_VALIDATE (-510). An example of testing the notification message is shown here:

```
var
  FileToRun: String;
...
if TheMessage.Msg = WM_NOTIFY then
  if PNMHdr(TheMessage.LParam).code = RFN_VALIDATE then
    WideCharToStrVar(PNM_RUNFILEDLG(
      TheMessage.LParam).lpFile, FileToRun);
...
```

Notice how we cast the *LParam* parameter to type *PNM_RunFileDlg* only after we have verified the *TNMHdr* code is RFN_VALIDATE.

The value assigned to the notification message's return value determines whether the application will be run. The following is the list of possible values:

```
RF_OK     = $00;  // Allow the application to run.
RF_CANCEL = $01;  // Cancel operation; close the dialog.
RF_RETRY  = $02;  // Cancel operation; leave dialog open.
```

By default, this value is RF_OK, so unless you deliberately change the message's *Result* data member, the selected application will run.

## Conclusion

These standard shell dialog boxes provide an important ability to integrate your application with the Windows shell. You can use them as drop-in solutions to common problems, saving time and improving the polish of your applications. But we've only shown you a few of them. Next month, we'll show you several more dialog boxes that you may find very useful in your applications. Δ

*The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\APR\DI9904KB.*

Kevin J. Bluck is an independent contractor specializing in Delphi development. He lives in Sacramento, CA with his lovely wife, Natasha. He spends his spare time chasing weather balloons and rockets as a member of JP Aerospace (http://www.jpaerospace.com), a group striving to be the first amateur organization to send a rocket into space. Kevin can be reached via e-mail at kbluck@ix.netcom.com.

James Holderness is a software developer specializing in C/C++ Windows applications. He also runs a Web site on undocumented functions in Windows 95 (http://www.geocities.com/SiliconValley/4942). He is currently working for FerretSoft LLC, where he helps create the Ferret line of Internet search tools (http://www.ferretsoft.com). James can be reached via e-mail at james@ferretsoft.com or jholderness@geocities.com.



*By Cary Jensen, Ph.D.*

# Delphi Database Development

## Part VII: Filtering Datasets

**Over the past half year, this column has explored the basics of database development in Delphi. This month, this series continues with a look at techniques you can use to limit the display of records in a table to only those that contain certain data values. This technique is often referred to as filtering, and, in one form or another, it appears in almost every database application.**

Delphi provides three general ways to filter a DataSet. The most common technique, and the one that is generally preferred in a client/server environment, is to use a SELECT SQL query, employing a WHERE clause to specify which records to include in the view. The second technique is to use a range, which employs an index with range criteria to select records. The final technique is to use the *Filter* property. Each of these techniques have advantages and disadvantages, including major performance differences. As a result, it's not always obvious which technique you should use. The following sections discuss each of these techniques. At the end of this article, you'll find a prototype application you can use to help you select which technique is appropriate for your application.

### Filter Using Queries

Last month, we looked at record-searching techniques. One of the techniques we explored was the use of a SQL SELECT query to select a single record. This same technique can be used to select more than one record. The only difference between selecting a single record and selecting more than one is the WHERE clause of the SQL statement. A single record is selected if the WHERE clause tests for data that appears in only one record. However, if more than one record matches the WHERE clause, the query returns a subset of records.

For example, if you want to display a list of all customer contacts whose last names are Little and who live in the city of San Diego, you might use the following query string:

```
SELECT * FROM CUSTOMER
WHERE CONTACT_LAST = 'Little' AND
  CITY = 'San Diego'
```

In reality, however, it's unlikely that you would hard code such a query. Instead, you would use a parameterized query — one where parameters appear in place of hard-coded values for both the contact last name and the city. For example, the query may look like this:

```
SELECT * FROM CUSTOMER
WHERE CONTACT_LAST = :lname AND
  CITY = :cityname
```

At run time, values are assigned to the parameters before executing the query. In addition, if the query is prepared in advance, each subsequent execution of the query will be faster than if you let Delphi handle the preparation.

SQL SELECT queries that include WHERE clauses don't require the fields referenced in the WHERE clauses to be indexed. However, an index will nearly always improve the performance of these types of queries. Consequently, it's often a good idea to consider this when designing the tables you plan to query in this fashion.

Query preparation, and the use of query parameters, has been discussed in detail in earlier parts of

this series. However, it should be noted that the retrieval and manipulation of small sets of data is at the heart of fast-performing, client/server applications, and parameterized queries are an essential part of this task.

## Using Ranges

If you're using Tables instead of Queries to work with your data, you have two filtering options: use a range or a filter. In fact, there are some situations in which ranges and filters should be used together to arrive at the ideal subset of records.

Unlike SQL SELECT statements and filters, a range requires an index. In other words, if you want to apply a range, such as displaying all customers with a credit limit greater than US$1,000, there must be at least one index where the credit limit field is the first field in the index. Likewise, a city-based range requires at least one index where city is the first indexed field. Simply having an appropriate index isn't enough — a range can only be applied to the current index. If you want to set a range, and the index that the range applies to is not the current index, you must set the index before continuing.

You have two options when it comes to applying a range. The first, and easiest to use, is the *SetRange* method. This method has the following syntax:

```
procedure SetRange(const StartValues,
  EndValues: array of const)
```

Both arrays passed as parameters must have the same number of elements. The value in the first element of the first array corresponds to the beginning (or lowest) values for the range on the first field of the index. The value in the second element, if provided, identifies the lowest value for the range on the second field of the index, and so on. The elements of the second array identify the ending (or highest) values of the range for each indexed field, with the first element corresponding to the first field in the index; the second, if provided, for the second field in the index; and so on.

The arrays that you pass to the *SetRange* statement don't have to have the same number of elements as there are fields in the current index. For example, if the current index is based on the City, State, and Country fields, it's acceptable to set a range only on the City field, or both the City and State fields, if desired.

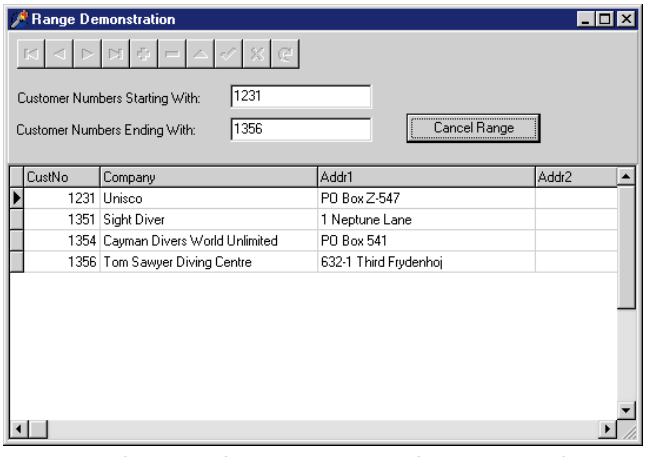



**Figure 1:** The example project RANGE demonstrates the use of *SetRange*.

The following code snippet demonstrates how *SetRange* can be used to limit the display of records in a table to those where the city field contains "New York." Assume that *Table1* is a component defined for a table named CLIENTS.DB. Furthermore, assume that this table has an index named CityIndex, which is a single field index on the City field of CLIENTS.DB. The following statement sets the *IndexName* property to CityIndex, then sets a range to display only those clients whose records contain New York in the City field:

```
Table1.IndexName := 'CityOrder';
Table1.SetRange(['New York'],['New York']);
```

To set a range based on a multi-field index, include more than one set of starting and ending values in the array parameters. For example, if you have a table named Invoices, and this table is using an index based on the fields CustNo and InvoiceDate, the following statement will display all records for customer C1573 for the dates 12/1/98 through 5/1/99:

```
Table1.SetRange(['C1573','12/1/98'],['C1573','5/1/99']);
```

## Using *ApplyRange*

An alternative to using *SetRange* is to use the methods *SetRangeStart*, *SetRangeEnd*, and *ApplyRange*. While these statements also require an index (either primary or secondary), it permits fields to be explicitly assigned their starting and ending values for the range without using an array. The following example defines the same range demonstrated in the preceding code snippet:

```
Table1.SetRangeStart;
Table1.FieldByName('CustNo').AsString := 'C1573';
Table1.FieldByName('InvoiceDate').AsString := '12/1/98';
Table1.SetRangeEnd;
Table1.FieldByName('CustNo').AsString := 'C1573';
Table1.FieldByName('InvoiceDate').AsString := '5/1/99';
Table1.ApplyRange;
```

Removing a range is much easier than applying one; use the *CancelRange* method. This method has the following syntax:

```
procedure CancelRange;
```

In general, you should issue a *Refresh* to a Table after calling *CancelRange*.

The use of *SetRange* is demonstrated in the RANGE project shown in Figure 1. (This and all other projects referenced in this article are available for download; see end of article for details.) The relevant code, found in the *OnClick* event handler for the button named *RangeButton*, is shown in Figure 2.

```
procedure TForm1.RangeButtonClick(Sender: TObject);
begin
  if RangeButton.Caption = 'Cancel Range' then
    begin
      RangeButton.Caption := 'Apply Range';
      Table1.CancelRange;
      Table1.Refresh;
    end
  else
    begin
      RangeButton.Caption := 'Cancel Range';
      Table1.SetRange([Edit1.Text],[Edit2.Text]);
    end;
end;
```

**Figure 2:** Code found in the *OnClick* event handler for the button named *RangeButton*.

## Using Filters

Filters, which were introduced in Delphi 2, make use of four properties of *TDataSet* descendants (*Table*, *Query*, *ClientDataSet*, and *StoredProc* objects): *Filter*, *Filtered*, *FilterOptions*, and *OnFilterRecord* (an event property). Using these properties, you can instruct a DataSet to display fewer than all of its records. What makes filtering special is that filters don't require an index. Furthermore, if an index is available, a filter will use it. This was not always the case. In Delphi 2, filters were inherently slower than ranges because they didn't use indexes. In Delphi 4, at least, performance tests indicate that filters make use of indexes when possible. That filters use indexes isn't discussed in the online documentation, and I haven't tested filters in Delphi 3. Nonetheless, filters are more widely applicable than the other record-limiting techniques.

Although the primary application of filters is to limit the records that are displayed in a DataSet, there is an additional capability offered by filters that is unmatched by ranges and SELECT queries. Specifically, filters permit you to display an entire database, yet still navigate between only the records that match the filter. For example, you can set a filter to match records based on the state of California, but still display all records in the DataSet. Then, using the four filter-specific navigation methods — *FindFirst*, *FindLast*, *FindNext*, and *FindPrior* — you can move to the first, last, next, and previous record where the Customer resides in California, skipping over any records in between.

As mentioned, some filters appear to use indexes. Specifically, simple filters based on a single field using the *Filter* property, as opposed to the *OnFilterRecord* event handler, will use an index if it's available. However, filters where the *Filter* property uses complex expressions (using AND and OR), as well as those that use an *OnFilterRecord* event handler, don't use indexes, and are inherently slower than index-based alternatives.

Because these types of filters don't make use of indexes, the number of records being filtered dramatically influences filter performance, with an increase in record number producing poorer performance. By comparison, SELECT queries that include WHERE clauses, filters that use indexes, and index-based ranges are influenced very little by the number of records in the DataSet. As a result, if you must use a filter that can't use an index, it's best to first use a range to limit the number of records in the current view before applying a filter to fine-tune the selection.

As implied in this section, there are three general ways to use filters. The first use employs only properties, while the second makes use of the *OnFilterRecord* event handler. The third use employs a filter, using either properties or events, but provides filtered navigation even though all records are displayed. Each of these is described in the following sections.

## Filtering with Properties

There are two properties that, when used together, produce a filtered DataSet. These properties are *Filter* and *Filtered*. *Filtered* is a Boolean property you use to turn the filter on and off. If you want to filter records, set *Filtered* to True; otherwise, set *Filtered* to False.

When *Filtered* is set to True, the DataSet uses the value of the *Filter* property to identify which records will be displayed. You assign to this property a string that contains at least one comparison operation involving at least one field in the DataSet. You can use any comparison operators, including =, >, <, >=, <=, and <>. As long as the field name doesn't include any spaces, you include the field name directly in the comparison without delimiters. For example, if your DataSet includes a field named Country, you can set the *Filter* property to the following value to filter only for customers in the US:

```
Country = 'US'
```

If your field name includes a space, you must enclose the field name in brackets. For example, if your DataSet is a Paradox table, and you want to display only those records where the customer's last name is Jones, and the field name in the table is "last name," assign the following value to the *Filtered* property:

```
[last name] = 'Jones'
```

These examples have demonstrated only simple expressions, which, as described earlier, will result in the filter using an index if one is available. However, complex expressions can be used. Specifically, you can combine two or more comparisons using the AND, OR, and NOT logical operators. Furthermore, more than one field can be involved in the comparison. For example, you can use the following *Filter* property value to limit records to those where the City field is San Francisco and the last name is Martinez:

```
City = 'San Francisco' and [last name] = 'Martinez'
```

Simply assigning a value to the *Filter* property doesn't automatically mean that records will be filtered. Only when the *Filtered* property is set to True does the *Filter* property actually produce a filtered DataSet. Furthermore, if no value appears in the *Filter* property, setting *Filtered* to True has no effect.

The use of the *Filter* and *Filtered* properties is demonstrated in the example project FILTER shown in Figure 3. This project contains an Edit control, in which the user can type a filter string. When the **Apply Filter** button is clicked, the event handler assigns the *Text* property of this Edit to the *Filter* property of the DataSet. Figure 3 shows this form after a *Filter* string has been entered. Notice that only those records that match what is in the **Filter** text box are displayed in the form's DBGrid.

The main form in the FILTER project also contains two checkbox components. These components control a third filter-related property of DataSets: the *FilterOptions* property. This property is a set property with two possible flags: *foCaseInsensitive* and *foNoPartialMatch*. When *foCaseInsensitive* is included in the set, the filter is not case sensitive.

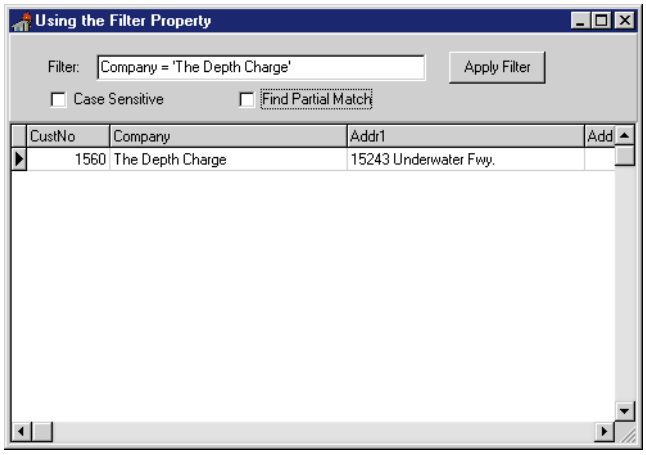



**Figure 3:** The example project FILTER demonstrates the use of *Filters*.

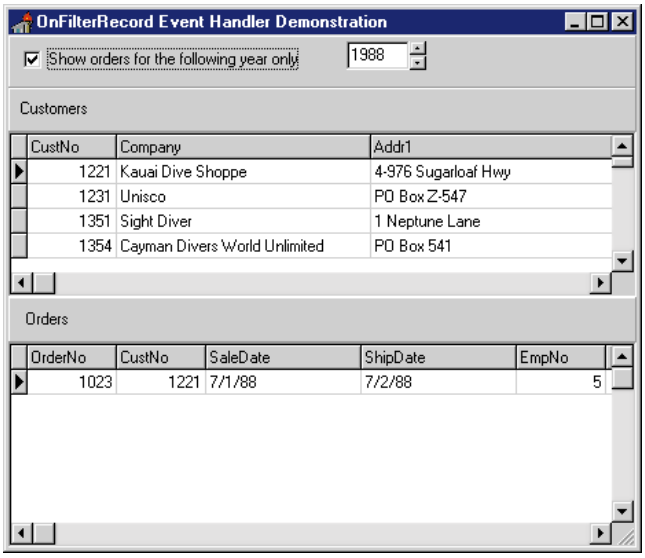



**Figure 4:** The example project ONFILT demonstrates the use of *OnFilterRecord*.

```
procedure TDataModule2.Table2FilterRecord(DataSet: TDataSet;
  var Accept: Boolean);
begin
  if (Table2.FieldByName('SaleDate').Value >= StrToDate(
      '1/1/'+IntToStr(Form1.UpDown1.Position))) and
     (Table2.FieldByName('SaleDate').Value <= StrToDate(
      '12/31/'+IntToStr(Form1.UpDown1.Position))) then
    Accept := True
  else
    Accept := False
end;
```

**Figure 5:** Code behind the *OnFilterRecord* event handler.

When *foNoPartialMatch* is included in the set, partial matches are excluded from the filtered DataSet. The *foNoPartialCompare* flag only applies for filters that include multiple conditions or multiple fields.

## Using the *OnFilterRecord* Event Handler

There is a second, somewhat more flexible way to define a filter. Instead of using the *Filter* property, you can attach code to the *OnFilterRecord* event handler for the DataSet. This event handler is passed a Boolean parameter by a reference that you use to indicate whether or not the current record should be included in the DataSet. From within this event handler, you can perform almost any test you can imagine. If, based on this test, you wish to exclude the current record from the DataSet, you set the value of the *Accept* actual parameter to False (this parameter is True by default). Note, however, that even though you can perform the exact same test using *Filter* and *OnFilterRecord*, the event-based technique never uses an index, and, therefore, should be avoided with large datasets.

The use of *OnFilterRecord* is demonstrated in the project ONFILT.DPR, shown in Figure 4. When your user checks the **Show orders for the following year only** checkbox, the *Filtered* property of the DataSet *(Table1* in *DataModule2)* is set to True. The year to be filtered on is controlled by an UpDown control. The actual filtering in this case is being performed in the *OnFilterRecord* event handler for the DataSet. Figure 5 shows the code attached to this event handler.

This code uses the *Position* property of the UpDown control (on *Form1*) to accept only those records within the specified year. If the current record passes this test, the value of *Accept* is set to False.

It's important to note that if you set the *Filter* property to a filter string and assign code to the *OnFilterRecord* property, both will be applied when *Filtered* is True. That is, only those records that match the filter string and those that are accepted by the event handler will appear in the DataSet.

As mentioned earlier, indexes are never used when you assign code to the *OnFilterRecord* event handler. Consequently, you should use this technique only when there is no other reasonable way to perform the filter. Furthermore, if the dataset that you are filtering contains a large number of records, you should first use a SQL SELECT statement or a Range to limit the size of the dataset.

## Navigating Using a Filter

Whether you have set *Filtered* to True or not, you can still use a *Filter* for the purpose of navigating selected records. For example, although you may want to view all records in a database, you may want to quickly move between records that meet specific criteria. For example, you may want to be able to quickly navigate between those records where an unpaid account balance exists.

The DataSet object surfaces four methods for navigating using a filter. These methods are *FindFirst*, *FindLast*, *FindNext*, and *FindPrior*. When you execute one of these methods, the DataSet will locate the requested record based on the current *Filter* property or *OnFilterRecord* event handler. This navigation, however, doesn't require that the *Filtered* property be set to True.

## Filters and Performance

There are some general rules of thumb you can follow when you're concerned about filtering performance. When using a remote database server, use prepared SELECT queries whenever possible. If it's not possible, use either ranges or filters that make use of indexes. The approach is definitely different when using a Table component. In those instances, use a range or a filter that uses an index. Furthermore, if you absolutely must use a filter that doesn't make use of an index, such as a filter that employs an *OnFilterRecord* event handler, do so only when the Table points to a dataset that contains few records (less than 500 or 1,000 records).

As mentioned in the last installment of this series, if you have any doubts, and particularly if fast performance is paramount, you should test your alternatives. I put together a small test application called TESTFILT, shown in Figure 6. This application permits you to test the various filtering methods against one another.

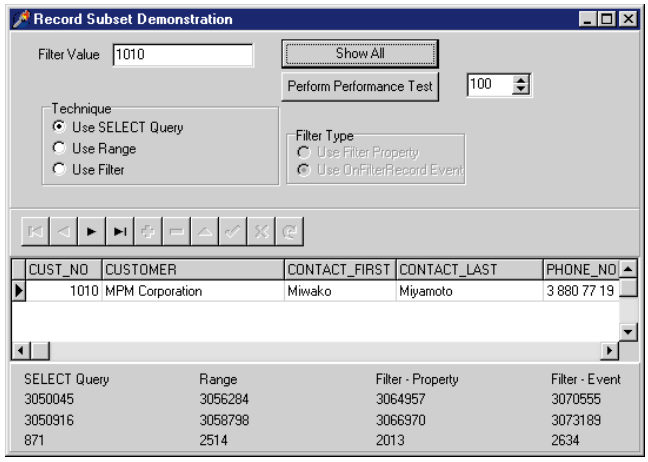



**Figure 6:** The example program TESTFILT can be used to test filtering techniques with your data.

## Using TESTFILT

While TESTFILT should be fairly easy to use as it is, you will likely want to make some changes to it before beginning your testing. The first thing you must do is adjust the constants defined in the interface section of the main.pas unit. These permit you to choose a database alias, a table, and a single field to be filtered. You must also define the parameterized query that will be used for the query-based filtering. In addition, if you're using a remote database that requires a user name and passwords, these also can be defined with these constants. The following is an example of these constant definitions for use with the InterBase CUSTOMER table that ships with Delphi:

```
const
  DatabaseAlias = 'IBLOCAL';
  DatabaseUserName = 'SYSDBA';
  DatabasePassword = 'masterkey';
  FilterFieldName = 'CUST_NO';
  SQLString =
    'SELECT * FROM CUSTOMER WHERE (CUST_NO = :cust)';
  Table1TableName = 'CUSTOMER';
```

Most of this information is used to initialize a Database, a Query, and a Table component used in the testing. This is shown in the *OnCreate* event handler from this project's data module (see Figure 7).

The only other adjustment that you might need to make involves the assignment of the query parameter. This occurs in two places: once in the *Button1 OnClick* event handler and again in the *Button2 OnClick* event handler. The following is how this line appears when testing the CUSTOMER table, in which the CUST_NO field is an integer field:

```
DataModule1.Query1.Params[0].Value := StrToInt(Edit1.Text);
```

If the field against which you're testing the filter isn't an integer, you must remove the *StrToInt* call and either replace it with an appropriate conversion function, or omit it altogether if your field is a String.

I ran this application to test filtering in four situations. Two of these involved small datasets of less than 100 records, and two of these involved large ones, containing approximately 100,000 records. In addition, one dataset of each size was a local Paradox table, and one each was an InterBase table. In all cases, I ran the test against an indexed integer field.

```
procedure TDataModule1.DataModule1Create(Sender: TObject);
begin
  Database1.AliasName := main.DatabaseAlias;
  Database1.Params.Clear;
  Database1.Params.Add('USER NAME=' + DatabaseUserName);
  Database1.Params.Add('PASSWORD=' + DatabasePassword);
  Query1.SQL.Add(main.SQLString);
  Query1.Prepare;
  Table1.TableName := main.Table1TableName;
  Table1.Open;
  FilterFieldObject := Table1.FieldByName(FilterFieldName);
end;
```

**Figure 7:** The *OnCreate* event handler from example program TESTFILT's data module.

| | Select | Range | Filter, Property | Filter, Event |
|---|---|---|---|---|
| **Small Local** | 4810 | 50 | 60 | 301 |
| **Large Local** | 4627 | 60 | 181902 | 602260 * |
| **Small Remote** | 871 | 2514 | 2013 | 2634 |
| **Large Remote** | 1080 | 3868 | 2813 | 4869400 * |

**Figure 8:** 100 subset selections measured in milliseconds (* estimated).

The results are shown in Figure 8. These reveal that the SELECT query was by far the best choice for filtering the InterBase table, with the performance largely unaffected by the size of dataset. This is what you might expect, and it's comforting to observe this effect. Also noteworthy was the fact that the similarity in query performance between small and large datasets was also observed with respect to local tables. However, unlike with InterBase tables, this query performance wasn't the best choice, being actually the worst option for filtering a small local table.

Another surprising result was the *Filter* property performance. The *Filter* property was excellent for all but the large local table, beating out the *Range* method. For a large local dataset, however, the performance of *Filter* was very poor. I can't explain this effect, other than assuming that the BDE generates a SQL query when you use a simple *Filter* on remote databases, but must sequentially scan local datasets. As you learned in the last installment of this series, sequential record access with local tables is extremely fast with small datasets. If this is correct, then the assumption that I made earlier in this column — that only filters can make use of indexes — is incorrect.

The final result to note is that when using remote tables, and especially when using large tables, the *OnFilterRecord* event handler was the slowest. In fact, its performance was so poor that I estimated its speed by running only 10 repetitions, then multiplying the result by 10. Running 10 repetitions took over eight minutes on the large remote table, and I estimated that 100 repetitions would have taken more than 81 minutes. This compares to just about one second for a SELECT query, and about three and two seconds for ranges and property-based, index-using filters, respectively.

## Conclusion

Delphi provides you with a number of solutions when it comes to filtering. Which technique you use will depend on the number of records in the dataset, and whether your data is remote or local. Also, if performance is important (and when dealing with a large database it is almost always important) you should create some realistic tests to determine which filtering technique is best. Δ

*The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\APR\DI9904CJ.*

Cary Jensen is president of Jensen Data Systems, Inc., a Houston-based database development company. He is co-author of 17 books, including *Oracle JDeveloper* [Oracle Press, 1998], *JBuilder Essentials* [Osborne/McGraw-Hill, 1998], and *Delphi in Depth* [Osborne/McGraw-Hill, 1996]. He is a Contributing Editor of *Delphi Informant*, and is an internationally respected trainer of Delphi and Java. For information about Jensen Data Systems consulting or training services, visit http://idt.net/~jdsi, or e-mail Cary at cjensen@compuserve.com.



*By Rod Stephens*

# Three Searches

## Delphi Implementations of Classic Techniques

Searching for an item in a list is a common programming task. For small lists, a program can simply look through the items until it finds its target, or reaches the end of the list. For larger lists, this is inefficient. This article describes three techniques for searching lists: exhaustive search, binary search, and interpolation search. Using these algorithms, you can quickly and easily search lists of any size.

### Exhaustive Search

To perform an exhaustive (or linear) search, a program starts at the beginning of the list and examines the items in turn until it finds either its target or the end of the list. This method is simple, so it's easy to implement and debug.

The *ExhaustiveSearch* function searches a resizable array for a target value (see Figure 1). It returns the index of the target in the array if it's present. It returns 0 if the target isn't in the array. This routine is reasonably fast for very small lists. If the list contains *N* items, the program will need to examine an average of *N* / 2 items to locate a specific target value. To conclude an item isn't present, however, the function must search every item in the list.

If the list is sorted, the exhaustive search function can do slightly better. As it searches the list, if the routine encounters a value larger than the target value, it can stop searching. Because the list is sorted, when the function finds a value greater than the target, it has passed the position where the target would have been if it were present.

```
type
  TLongArray = array[1..10000000] of Longint;
  PLongArray = ^TLongArray;

// Perform an exhaustive search of a list.
function ExhaustiveSearch(target: Longint; list: PLongArray;
  min, max: Longint): Longint;
var
  i: Longint;
begin
  for i := min to max do
    if (list^[i] = target) then begin
      Result := i;
      Exit;
    end;
  // We did not find the target; return 0.
  Result := 0;
end;
```

**Figure 1:** An exhaustive search function.

The code in Figure 2 shows the revised exhaustive search function. For a list containing *N* items, this version still takes an average of *N* / 2 steps to locate a target item that is in the list. When it searches for an item that is not present, however, it can conclude that the item is missing in an average of *N* / 2 steps instead of the *N* steps needed before.

### Binary Search

Exhaustive search is so simple that it's quite fast for small lists. It's also easy to implement and debug. For larger sorted lists, more sophisticated techniques, such as binary search, can be much faster. A binary search examines the value in the middle of the list and compares it to the target. If the target and middle items are the same, the search is over. If the target is larger than the middle item, the program continues by searching the larger items in the list. If the target is smaller, the program continues by searching the smaller items.

Figure 3 graphically shows a binary search for the target value 77. When the program starts, the target could lie in any position with index 1 through 10. It compares the target 77 to the item in the middle of the list. That item has value 71. Because 77 is greater than 71, the program continues to search for the target in the larger half of the list. This includes those items with indexes 7 through 10. With a single comparison, the program has eliminated all the other items from consideration.

Next, the program compares the target to the value in the middle of the items still under consideration. Because the program is now considering items with

indexes 7 through 10, the middle item has index 8 and value 83. The program compares that item to the target. Because 77 is less than 83, the search continues in the smaller half of the items still being considered. That includes only the item with index 7 and value 77.

```
// Perform an exhaustive search of a sorted list.
function ExhaustiveSearch(target: Longint;
  list: PLongArray; min, max: Longint): Longint;
var
  i: Longint;
begin
  for i := min to max do
    if (list^[i] >= target) then
      Break;

  if (i > max) then
    Result := 0   // Not found.
  else if (list^[i] = target) then
    Result := i   // Found.
  else
    Result := 0;  // Not found.
end;
```

**Figure 2:** An exhaustive search function for sorted lists.

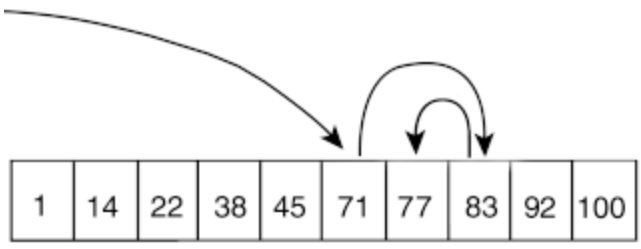



**Figure 3:** A binary search for the target value 77.

```
// Perform a binary search of a sorted list.
function TSearchForm.BinarySearch(target: Longint;
  list: PLongArray; min, max: Longint): Longint;
var
  mid: Longint;
begin
  // The target's index is always between min and max.
  while (min <= max) do begin
    mid := Round((max + min) / 2);
    if (target = list^[mid]) then // Here it is.
      begin
        Result := mid;
        Exit;
      end
    else if (target < list^[mid]) then
      // Search the left half.
      max := mid - 1
    else
      // Search the right half.
      min := mid + 1;
  end;
  // If we get here, the target is not present.
  Result := 0;
end;
```

**Figure 4:** A binary search function for sorted lists.

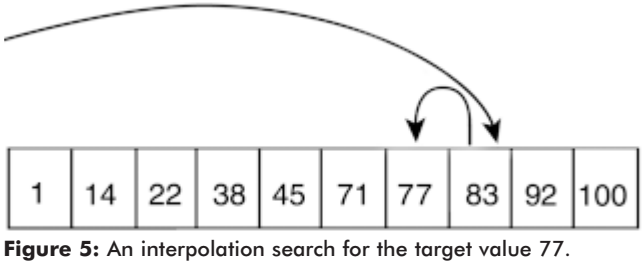



**Figure 5:** An interpolation search for the target value 77.

Now the program compares the target to the only item left. The two values match, so the program has found the target. In this example, binary search located the target after examining three items in the list. An exhaustive search would have needed to examine seven items to find the value 77. The difference here is small. In larger lists, a binary search can be much faster than an exhaustive search.

Every time the binary search examines an item in the list, it divides in half the number of items that must be considered. If the list contains $N$ items, the algorithm can divide the list in half at most $\log_2(N)$ times before there is only one item left to consider. At that point, the program has either found the target, or the target isn't present in the list. The time $\log_2(N)$ is much faster than the average of $N / 2$ steps required by exhaustive search. For example, if the list contains one million items, exhaustive search will take an average of 500,000 steps to locate an item. Binary search will need only $\log_2(1{,}000{,}000)$, or roughly 20 steps.

Figure 4 shows the code for a binary search function. The key to this function is that it always updates *min* and *max*, so *min* <= *target index* <= *max*. Each time the function compares the target to test items, it raises *min* or lowers *max*. Eventually, it either finds the target, or raises *min* and lowers *max* until *min* > *max*. At that point, because there are no values between *min* and *max*, the target must not be in the list.

## Interpolation Search

Binary search is fast because it eliminates many items from consideration without examining them. If the items in the list are numeric and reasonably evenly distributed, you can make the search eliminate even more items at each step using interpolation. Interpolation is the process of using known values to guess an unknown value. In this case, it means using known item positions to guess the target item's position.

For example, suppose you have a list of 50 values between 1 and 100, and you want to locate the target value 70. Because 70 is 70 percent of the way between the values 1 and 100, you would expect the target to lie 70 percent of the way through the list. In this example, that would be 70 percent of the way through a list of 50 items at position 35.

In structure, interpolation search is similar to binary search. It picks an item in the list and compares it to the target. If the target and test items have the same value, the search is over. If the target is larger than the test item, the algorithm continues by searching the larger items in the list. If the target is smaller, the algorithm continues by searching the smaller items.

The difference lies in how the two algorithms select the test item to compare with the target. Binary search selects the item in the middle of the indexes still being considered. Interpolation search uses interpolation to pick an item that is probably very close to the target's actual position.

This process is shown graphically in Figure 5. The program is searching for the target value 77 in a list of 10 items with values between 1 and 100. Because 77 is 77 percent of the way between the minimum and maximum values 1 and 100, the program looks at the item that is 77 percent of the way through the list. There are 10 items in the list, so that is at index 7.7, which rounds to index 8.

The program compares the value in position 8 with the target. Because 77 is less than 83, the program continues to search the items with indexes between 1 and 7. The program examines the items with indexes 1 and 7. They have values 1 and 77. Because 77 is 100 percent of the way from the value 1 to the value 77,

```
// Perform an interpolation search of a sorted list.
function TSearchForm.InterpolationSearch(target: Longint;
  list: PLongArray; min, max: Longint): Longint;
var
  mid: Longint;
begin
  while (min <= max) do begin
    // Watch for division by zero.
    if (list^[min] = list^[max]) then begin
      // If this isn't the item, it's not here.
      if (list^[min] = target) then
        Result := min
      else
        Result := 0;
      Exit;
    end;
    // Compute the dividing index.
    mid := Round(min + ((target - list^[min]) *
      ((max - min) / (list^[max] - list^[min]))));
    // Check that we are in bounds.
    if ((mid < min) or (mid > max)) then begin
      // The target isn't here.
      Result := 0;
      Exit;
    end;

    if (target = list^[mid]) then // The target is here.
      begin
        Result := mid;
        Exit;
      end
    else if (target < list^[mid]) then
      // Search the left half.
      max := mid - 1
    else
      // Search the right half.
      min := mid + 1;
  end; // End while (min <= max) do loop.

  // If we got to this point, the target isn't here.
  Result := 0;
end;
```

**Figure 6:** An interpolation search function for sorted lists.

the program examines the item with index 100 percent of the way from index 1 to index 7. That item has index 7 and matches the target value. The interpolation search finds its target value in its second step.

In this example, interpolation search located the target in two steps. Binary search took three steps to find the target in the earlier example, and exhaustive search would need seven steps. Interpolation search is the fastest, though the difference in speed is much smaller than the difference between binary and exhaustive search.

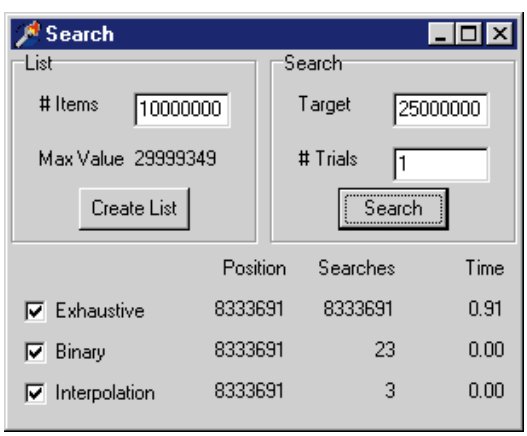



**Figure 7:** The example program, Search, in action.

For example, to locate a target in one million items, exhaustive search needs an average of 500,000 steps, binary search needs only 20 steps, and interpolation search might need as few as four or five steps. The difference between binary and interpolation search is significant, but less impressive than the difference between exhaustive and binary search.

Figure 6 shows an interpolation search function. If you compare it to the binary search routine, you will see how similar they are. The difference lies in how they calculate the value *mid* giving the index of the test item to compare with the target. Interpolation search calculates this middle index using the following statement:

```
mid := Round(min + ((target - list^[min]) *
  ((max - min) / (list^[max] - list^[min]))));
```

This code creates two special cases the program must consider. First, if list^[*min*] = list^[*max*], then this equation causes a divide-by-zero error. In particular, if the program narrows the list of items until *min* = *max*, these values will be the same. To prevent the divide-by-zero error, the program checks whether list^[*min*] = list^[*max*] before it calculates the new middle index.

Second, this code sometimes produces a test index that's smaller than *min*, or greater than *max*. Occasionally, this value can be far outside of this range. The new index may lie beyond the bounds of the list array, or it may even be negative. Fortunately, the only way the test index can be less than *min* or greater than *max* is if the target item's index is less than *min* or greater than *max*. Because the program always updates *min* and *max* so the target item lies between them, the target must not be present. After it calculates *mid*, the program compares it to *min* and *max* to see if it can stop searching.

## The Searches Compared

The example program Search demonstrates the search algorithms (Search is available for download; see end of article for details). Enter a number of items in the **# Items** text box, and click the **Create List** button. The program will create a list of the size you specified. It fills the list with random values, where each value is 1 to 5 greater than the one before it. The **Max Value** label shows the largest value in the list so you can pick values to locate.

Enter a value in the **Target** text box, and the number of times the program should repeat the searches in the **# Trials** text box. Then click the **Search** button to make the program search for the target using the methods you've checked. To get meaningful timing results for the binary and interpolation searches, you may need to use a large number of trials. Be sure the **Exhaustive** check box is checked only for small lists, and small numbers of repetitions.

## Conclusion

Figure 7 shows the Search program in action. In this case, the program found the target value 25 million in a list of 10 million items. Exhaustive search took more than 8 million steps in just under one second. Binary search needed only 23 steps and interpolation search needed only three. Binary and interpolation search were so fast, their times did not even register.

Exhaustive search is fast enough for searching short lists. It is also easy to implement and debug. That makes it a good choice for small lists, or when you just need to get some code running quickly. For larger lists, binary search is much faster, although it is more complex. Interpolation search is usually faster still,

though it has the disadvantage of working easily only with numeric lists. The improvement in speed is also not as great as the improvement of binary search over exhaustive search, so you should probably use binary search if you are working with a list of non-numeric values.

Evaluate your needs and pick the algorithm that's right for you. One of these three techniques can satisfy almost any program's searching needs. Δ

*The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\APR\DI9904RS.*

For more information on searching and other algorithms, take a look at Rod Stephens' book, *Ready-to-Run Delphi 3.0 Algorithms* [John Wiley & Sons, 1998]. The algorithms run in Delphi 3 or later. Learn more at http://www.delphi-helper.com/da.htm, or contact Rod at RodStephens@delphi-helper.com.



Interfaces / Class Hierarchies / OOP / Delphi 2, 3, 4

*By Eric Whipple*

# The Interface Advantage

## When It Comes to Flexibility, Interfaces Edge Out Hierarchies

**Just when you thought you'd mastered the object model, the paradigm shifts again. Often, developers now use the inheritance (or hierarchical) model to create fast and flexible "family trees" of classes for the objects in their systems. Inheritance, with its built-in code reuse, enables applications to be developed faster, and allows object methods to behave polymorphically.**

In spite of these impressive advantages, however, hierarchical systems can be alarmingly inflexible and proprietary. Seemingly trivial deviations in the structure of a class can produce costly difficulties in creating a practical and maintainable hierarchy. Therefore, as distributed objects come into the mainstream, the development community is beginning to embrace the interface model of object development. The interface model provides significant advantages in flexibility over the hierarchical model, yet continues to support the power of object-oriented techniques, such as polymorphism.

### An Interface Defined

An interface is essentially a set of method declarations that collectively defines some set of functionality. For example, the *IReports* interface might contain declarations for the following reporting functions:

```
function GetReport(ReportName: string):
  Boolean;
function CloseReport(ReportName: string):
  Boolean;
function PrintReport(ReportName: string):
  Boolean;
function DoDaysEndReports: Boolean;
function GetReportCount: Integer;
function ReportIsOpened(ReportName: string):
  Boolean;
```

The important difference between an interface and a class is that the interface only includes the method declarations. Classes that provide implementations for the methods in that interface are free to do so in whichever manner is appropriate. In addition, client-side variables can hold a reference to any server object that implements that interface.

A class, on the other hand, not only declares the methods, but also provides a specific implementation for each one. Object instances of the *TReports* class will handle method requests in exactly the same way. In addition, client-side variables are restricted to holding references to server objects instantiated from the *TReports* class and its descendants.

Interfaces allow server objects to concisely declare exactly which sets of functionality they can provide to clients (i.e. which interfaces they implement), without restricting the client to a particular implementation of those interfaces. If a server object is declared to implement a particular interface, it's essentially creating an unbreakable contract with its clients, i.e. that it can handle a specific set of requests, formatted in a certain way. The contract that's created, however, doesn't include information related to the implementation and handling of these requests. It simply guarantees a certain type of result, given a certain type of input. This contract provides the ultimate example of the "Black Box Theory."

### Hierarchies ...

The best way to discover the power of interfaces is to put them into practice. Consider the following scenario. A distributed system is being developed for a local hospital. The reporting server object is required to perform the reporting methods previously mentioned. The *DoDaysEndReports* method compiles and prints

```
module ReportsServer1
{
  interface IReports
  {
    boolean GetReport(in wstring ReportName);
    boolean PrintReport(in wstring ReportName);
    boolean CloseReport(in wstring ReportName);
    boolean DoDaysEndReports();
    long getReportCount();
    boolean ReportIsOpened(in wstring ReportName);
  };
 };
```

**Figure 1:** The *IReport* interface contains method declarations, but no implementations.

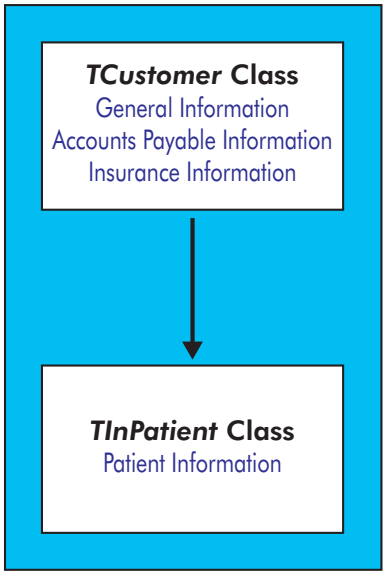



**Figure 2:** The patient-tracking system hierarchy.

a set of daily reports related to the productivity of the hospital. Each department of the hospital has a different set of reports that needs to be generated. In addition to day's-end reporting, some departments also require month's-end reporting to be done automatically on the last day of the month. To avoid a maintenance nightmare, the IT department decides to write one client-side reporting module to be used by all departments.

Under a traditional class-based system, the server object's *TReports* class resembles the code shown in Listing One on page 22 (this and all subsequent code is available for download; see end of article for details). It contains all of the previously mentioned methods, including specific implementations for each. The client module code is shown in Listing Two (beginning on page 22). Notice the declaration of the *ReportServer* variable. It's of type *TReport*; consequently, the *ReportServer* variable can only hold a reference to a server object instantiated from the *TReports* class, or one of its descendants.

The problem that arises in this situation is not in design or development, but in maintenance. Because all the reporting servers are required to belong to the same *TReport* family, architects and developers who maintain the reporting system as it changes or grows are required to have an intimate knowledge of the entire *TReport* genealogy. In a simple example such as ours, this is less evident. But in a more complex hierarchy, with many generations, finding the appropriate class from which to inherit can involve quite a bit of research. For example, the *TQuery* component descends from *TDBDataset*, which descends from *TBDEDataset*, which descends from *TDataset*. Deciding which of these classes from which to inherit a new class depends, among other things, on what functionality was declared at what level (generation).

Interfaces, on the other hand, do not require any knowledge of ancestral functionality. As long as the new class can perform the methods required by the interface, it is perfectly suitable for use by the client.

## ... or Interfaces?

Now let's consider the same example using interfaces. Instead of a *TReports* class, there's an *IReports* interface (see Figure 1). It contains the declarations for the noted methods, but no implementations. Each department has its own server object, instantiated from a class that implements the *IReports* interface. The departmental servers each implement the *IReports* methods in a way specific to a particular department. The classes that provide reporting to each department are unrelated through inheritance; that is, they have no common ancestors. The unifying thread between them is that each server provides reporting in a standardized way through the methods of the *IReports* interface.

The client module shown in Listing Three (beginning on page 23) is similar to the client code, with one important difference. The *ReportServer* variable is now declared to be of type *IReports*. In essence, this means that the *ReportServer* variable can hold a reference to any object that implements the *IReports* interface. It's this difference that enables the client object to not only be unaware of the logic involved in generating the correct set of reports, but also of the ancestral history of its server. It simply gives a request and gets a result.

The advantage to interfaces, then, is that they allow different objects to implement the same methods in different ways. But wait a minute! That's the definition of polymorphism — a technique that's certainly not particular to interfaces. As long as all the server objects belong to the same hierarchy of classes, they can behave polymorphically, and achieve the same results that interfaces do. While this is true, it's also true that the maintenance of a hierarchy of classes can produce another set of problems. In an expanding application, finding a place to insert a new entity in an existing hierarchy can be a difficult task. Let's consider another example.

## The Trouble with Hierarchies

The hospital is also creating a patient-tracking system. The class hierarchy shown in Figure 2 begins with a *TCustomer* class. The *TCustomer* class, designed to be used for general office visits (e.g. check-ups, physicals, etc.), contains properties and methods related to the following types of information:

- General
- Accounts Payable
- Insurance

The *TCustomer* class has a direct descendant: *TInPatient*. The *TInPatient* class inherits all the properties and methods of the *TCustomer* class. It also specifies a number of properties and methods specific to customers requiring more serious medical attention, as well as a more comprehensive system of medical documentation. For 90 percent of the hospital's business, this hierarchy works perfectly. Customer variables in the client application are defined as type *TCustomer*, and are, therefore, able to hold a reference to any object instantiated from the *TCustomer* or the *TInPatient* classes.

Problems begin when new entities are introduced into the system. For example, the hospital requires its doctors to spend one Saturday every three months at the hospital's free clinic for the homeless. For legal reasons, the doctor is required to keep detailed records of the treatment given to each person. In this case, the accounts payable and insurance information aren't applicable. In effect, all the people the doctor sees are InPatients, but none of them are Customers; many of the attributes that

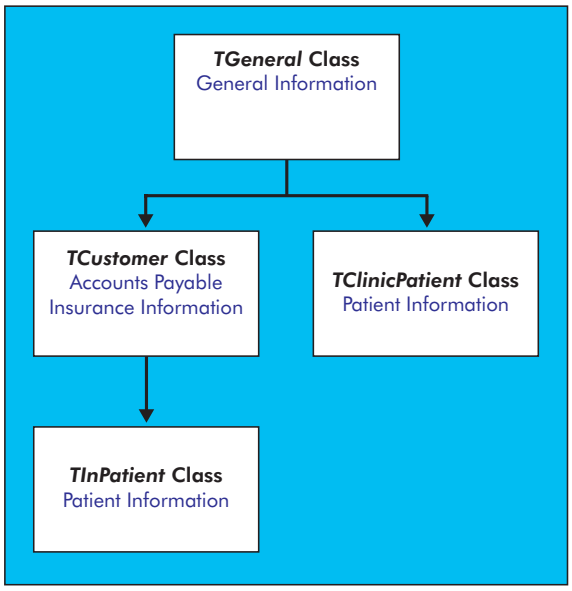



**Figure 3:** The new Med system hierarchy.

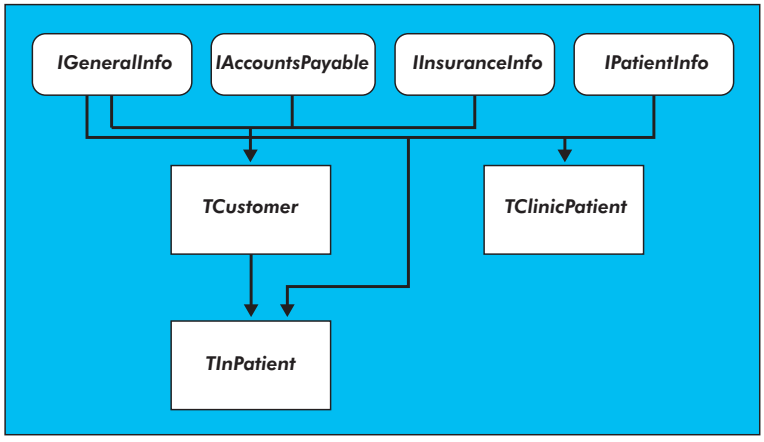



**Figure 4:** The Med system interface solution.

identify someone as a customer (e.g. Accounts Payable, Insurance, etc.) don't apply to clinic patients.

Under this scenario, the hospital application can become extremely inefficient. For each clinic patient, an *InPatient* object must be created so that medical information can be properly logged. Because the *TInPatient* class is a descendant of the *TCustomer* class, it inherits all the properties and methods of the *TCustomer* class. During the creation of the *InPatient* object, its inherited class properties will have memory allocated for them that will never be used.

What does this situation do to our class hierarchy? How can this new entity (the clinic patient) be inserted into our "family tree?" One answer involves creating an ancestor to the *TCustomer* class, and including in the class only the things common to the *TCustomer* and *TClinicPatient* classes (see Figure 3). *TClinicPatient* and *TCustomer*, then, descend from *TGeneralClass*. However, there are two main problems with this.

The first problem is that *TClinicPatient* is just one example of a class that requires the attributes of the descendant (*TInPatient*), but none or few of the attributes of the ancestor (*TCustomer*).

Another example might involve a patient who has no insurance and pays monthly installments toward his or her medical bills. For each new entity that's introduced into the system, the entire hierarchy must be rearranged so that all the classes can share common ancestors. This can be done in two ways:

- Use a language that supports multiple inheritance.
- For each new entity, create an ancestor (*TGeneralClass*) to the class with properties that need to be split up *(TCustomerClass),* and "factor out" the common properties; then create the new class (*TClinicPatient*) as a descendant of the ancestor *TGeneralClass*.

Using either of these solutions, the hierarchy can become a tangled web of countless generations of ancestors and descendants — an endless maintenance nightmare.

The second problem is that the client application's variable is currently defined as *TCustomer*. Under the new hierarchy, it can't hold a reference to a *ClinicPatient* object, because *TClinicPatient* isn't a descendant of *TCustomer*. But even if the variable's type was changed to *TGeneralClass*, there would still be a problem. For the methods common to *TInPatient* and *TClinicPatient* to behave polymorphically, they would have to be defined in the *TGeneralClass* class. This doesn't follow the rules of encapsulation, and is generally a bad idea.

It's important to note that it is the unpredictability of systems that causes these types of problems. It's possible to create an elegant hierarchy that serves a system's primary needs, but, as that system is maintained and business grows or changes, additions to the system require the reengineering of the entire hierarchy. As the hierarchy is twisted out of its original shape, it becomes more and more difficult to maintain.

## Interfaces to the Rescue

Now let's consider the same example using interfaces. We start by defining the *IGeneralInfo*, *IAccountsPayable*, *IInsuranceInfo,* and *IPatientInfo* interfaces. Each represents a group of methods related to a specific set of functionality. *TCustomer* is now a class that implements three interfaces: *IGeneralInfo*, *IAccountsPayable*, and *IInsuranceInfo*. The *TInPatient* class descends from *TCustomer*, and implements the *IPatientInfo* interface (see Figure 4).

The difference between these two systems may seem small at first, but we now have much greater flexibility. We can easily create a class named *TClinicPatient* that implements the *IGeneralInfo* and *IPatientInfo* interfaces. This frees us from the burden of finding — or creating — an appropriate place in the hierarchy for our new class. When the application needs to invoke a method related to the current object's InPatient information, it declares a variable of type *IPatientInfo*. This variable can hold a reference to any object that implements the *IPatientInfo* interface, regardless of its ancestral history. It's this element that allows the *IPatientInfo* methods to be called polymorphically.

## Conclusion

Interfaces give the developer a significant advantage in developing flexible, efficient, and maintainable systems. They allow the client to request a set of services from any server object that can provide them. In addition, the server objects that provide those services aren't required to be related in any way.

The use of interfaces not only eliminates the need for client-side variables to know the implementation details of the methods they're invoking, but also the ancestral history of the server object they're referencing. Interfaces also allow an object's functionality to be split into a set of "changeable parts," which greatly simplifies the adaptability of growing systems. Δ

The author would like to thank Ken Faw and Stephen Chirico for their help in preparing this article.

*The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\APR\DI9904EW.*

Eric Whipple is a Delphi trainer and mentor for Pillar Technology Group, Inc. of Detroit, a full-service consulting, training, and mentoring firm specializing in project management and in the analysis, design, and development of distributed, enterprise systems (http://www.knowledgeable.com). Eric is a Delphi 4 certified developer and trainer, and can be reached at ewhipple@kowledgeable.com or (317) 915-9031.

## Begin Listing One — *TReports* Class

```
unit ReportsServer1_IMPL;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  ComObj, StdVcl, CorbaObj, ReportsServer1_TLB, Dialogs;

type
  TReports = class(TCorbaImplementation, IReports)
  private
    ReportList : TStringList;
  public
    constructor Create(Controller: IObject;
      AFactory: TCorbaFactory); override;
    destructor Destroy; override;
  protected
    function CloseReport(const ReportName: WideString):
      WordBool; safecall;
    function DoDaysEndReports: WordBool; safecall;
    function GetReport(const ReportName: WideString):
      WordBool; safecall;
    function PrintReport(const ReportName: WideString):
      WordBool; safecall;
    function getReportCount: Integer; safecall;
    function ReportIsOpened(const ReportName: WideString):
      WordBool; safecall;
  end;

implementation

uses
  CorbInit, FormReportsServerMain;

constructor TReports.Create(Controller: IObject;
  AFactory: TCorbaFactory);
begin
  inherited Create(Controller,AFactory);
  ReportList := TStringList.Create;
end;

destructor TReports.Destroy;
begin
  ReportList.Free;
  inherited Destroy;
end;

{ The IReport method implementations. }
function TReports.CloseReport(
  const ReportName: WideString): WordBool;
begin
  with ReportList do
    if ReportIsOpened(ReportName) then
      begin
        Delete(IndexOf(ReportName));
        with frmReportsServerMain.lstbxCurrentReports do
          Items.Delete(Items.IndexOf(ReportName));
        Result := True;
      end
    else
      Result := False;
end;

function TReports.DoDaysEndReports: WordBool;
begin
  Result := True;
end;

function TReports.GetReport(const ReportName: WideString):
  WordBool;
begin
  if ReportList.IndexOf(ReportName) = -1 then
    begin
      ReportList.Add(ReportName);
      frmReportsServerMain.lstbxCurrentReports.Items.Add(
        ReportName);
      Result := True;
      frmReportsServerMain.stsbrStatus.SimpleText :=
        'There are currently ' +
        IntToStr(ReportList.Count) + ' opened.';
    end
  else
    Result := False;
end;

function TReports.PrintReport(
  const ReportName: WideString): WordBool;
begin
  Result := ReportIsOpened(ReportName);
end;

function TReports.getReportCount: Integer;
begin
  Result := ReportList.Count;
  frmReportsServerMain.stsbrStatus.SimpleText :=
    'There are currently ' + IntToStr(ReportList.Count) +
    ' opened.';
end;

function TReports.ReportIsOpened(
  const ReportName: WideString): WordBool;
begin
  Result := not(ReportList.IndexOf(ReportName) = -1);
end;

initialization
  TCorbaObjectFactory.Create('ReportsFactory', 'Reports',
    'IDL:ReportsServer1/ReportsFactory:1.0', IReports,
    TReports, iMultiInstance, tmSingleThread);
end.
```

## End Listing One

## Begin Listing Two — *ReportServer* of Type *TReports*

```
unit FormReportClientMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
```

```
  Forms, Dialogs, StdCtrls, ExtCtrls, Buttons,
  ReportsServer1_TLB;

type
  TfrmReportClientMain = class(TForm)
    Panel1: TPanel;
    pnlOpenedReports: TPanel;
    Label1: TLabel;
    lstbxOpenedReports: TListBox;
    rdgrpReportType: TRadioGroup;
    btnOpenReport: TButton;
    btnPrintReport: TButton;
    btnCloseReport: TButton;
    Panel2: TPanel;
    BitBtn1: TBitBtn;
    rdgrpReportScope: TRadioGroup;
    btbtnDaysEnd: TBitBtn;
    procedure btnOpenReportClick(Sender: TObject);
    procedure btnCloseReportClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnPrintReportClick(Sender: TObject);
    procedure btbtnDaysEndClick(Sender: TObject);
  end;

var
  frmReportClientMain: TfrmReportClientMain;

implementation

var
  // This variable can only hold a reference to objects
  // instantiated from the TReports class, or one of its
  // descendants.
  ReportsServer: TReports;

{$R *.DFM}

procedure TfrmReportClientMain.btnOpenReportClick(
  Sender: TObject);
var
  ReportName: string;
begin
  ReportName := rdgrpReportScope.Items[
    rdgrpReportScope.ItemIndex] + ' ' +
    rdgrpReportType.Items[rdgrpReportType.ItemIndex];

  { Invoking a server method. }
  if ReportsServer.GetReport(ReportName) then
    begin
      lstbxOpenedReports.Items.Add(ReportName);
      btnCloseReport.Enabled := True;
    end
  else
    MessageDlg('That report is already opened.',mtError,
               [mbOK],0);
end;

procedure TfrmReportClientMain.btnCloseReportClick(
  Sender: TObject);
begin
  with lstbxOpenedReports do begin
    if ItemIndex = -1 then
      MessageDlg('No report is currently selected.',
                 mtError,[mbOK],0)
    else begin
      ReportsServer.CloseReport(Items[ItemIndex]);
      Items.Delete(ItemIndex);
      if ReportsServer.getReportCount = 0 then
        btnCloseReport.Enabled := False;
    end;
  end;
end;

procedure TfrmReportClientMain.FormCreate(Sender: TObject);
begin
  ReportsServer :=
    TReportsCorbaFactory.CreateInstance('Reports1');
end;

procedure TfrmReportClientMain.btnPrintReportClick(
  Sender: TObject);
begin
  if ReportsServer.PrintReport(lstbxOpenedReports.Items[
                       lstbxOpenedReports.ItemIndex]) then
    ShowMessage('Printing...');
end;

procedure TfrmReportClientMain.btbtnDaysEndClick(
  Sender: TObject);
begin
  if ReportsServer.DoDaysEndReports then
    MessageDlg('Days end reports complete.',
               mtInformation,[mbOK],0);
end;

end.
```

## End Listing Two

## Begin Listing Three — *ReportServer* of Type *IReports*

```
unit FormReportClientMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, Buttons,
  ReportsServer1_TLB;

type
  TfrmReportClientMain = class(TForm)
    Panel1: TPanel;
    pnlOpenedReports: TPanel;
    Label1: TLabel;
    lstbxOpenedReports: TListBox;
    rdgrpReportType: TRadioGroup;
    btnOpenReport: TButton;
    btnPrintReport: TButton;
    btnCloseReport: TButton;
    Panel2: TPanel;
    BitBtn1: TBitBtn;
    rdgrpReportScope: TRadioGroup;
    btbtnDaysEnd: TBitBtn;
    procedure btnOpenReportClick(Sender: TObject);
    procedure btnCloseReportClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnPrintReportClick(Sender: TObject);
    procedure btbtnDaysEndClick(Sender: TObject);
  end;

var
  frmReportClientMain: TfrmReportClientMain;

implementation

uses
  ReportsServer1_TLB;

var
  // This variable can hold a reference to any object that
  // implements the IReports interface.
  ReportsServer: IReports;

{$R *.DFM}

procedure TfrmReportClientMain.btnOpenReportClick(
  Sender: TObject);
var
```

```
  ReportName: string;
begin
  ReportName := rdgrpReportScope.Items[
    rdgrpReportScope.ItemIndex] + ' ' +
    rdgrpReportType.Items[rdgrpReportType.ItemIndex];

  if ReportsServer.GetReport(ReportName) then
    begin
      lstbxOpenedReports.Items.Add(ReportName);
      btnCloseReport.Enabled := True;
    end
  else
    MessageDlg('That report is already opened.',
               mtError,[mbOK],0);
end;

procedure TfrmReportClientMain.btnCloseReportClick(
  Sender: TObject);
begin
  with lstbxOpenedReports do begin
    if ItemIndex = -1 then
      MessageDlg('No report is currently selected.',
                 mtError,[mbOK],0)
    else
      begin
        ReportsServer.CloseReport(Items[ItemIndex]);
        Items.Delete(ItemIndex);
         if ReportsServer.getReportCount = 0 then
           btnCloseReport.Enabled := False;
       end;
  end;
end;

procedure TfrmReportClientMain.FormCreate(Sender: TObject);
begin
  ReportsServer :=
    TReportsCorbaFactory.CreateInstance('Reports1');
end;

procedure TfrmReportClientMain.btnPrintReportClick(
  Sender: TObject);
begin
  if ReportsServer.PrintReport(lstbxOpenedReports.Items[
       lstbxOpenedReports.ItemIndex]) then
    ShowMessage('Printing...');
end;

procedure TfrmReportClientMain.btbtnDaysEndClick(
  Sender: TObject);
begin
  if ReportsServer.DoDaysEndReports then
    MessageDlg('Days end reports complete.',
               mtInformation,[mbOK],0);
end;

end.
```

## End Listing Three



*By Chris Austria*

# Tools for the New Millennium

## The 1999 *Delphi Informant* Readers Choice Awards

"The millennium is near! The world is ending!" the man on the street corner shouts at you at the top of his lungs. You smile, walk past him, and head toward the building that contains your modest cubicle in the sky. "Not quite yet," you think as you sit in your chair, boot up your PC, and crack open the latest issue of *Delphi Informant*.

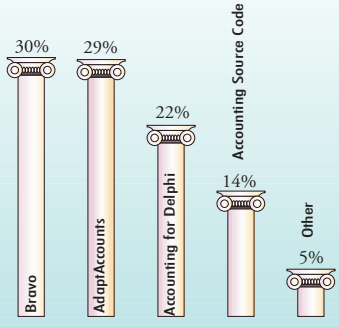
Indeed, the world is not ending; life is great, and busy as ever. Just ask the hundreds of vendors that busted their behinds during 1998 in an effort to provide you, loyal Delphi users, the best third-party products for your favorite development environment. To these vendors, every year is about moving on to bigger and better things, and this year is no exception.

129 products secured spots on the ballot for this, the fourth annual *Delphi Informant* Readers Choice Awards. While many of last year's 110 participating products remained, some vanished, and were quietly replaced. There were some changes in the ballot structure; while the Best ActiveX category was removed, new categories, including Best Accounting Package, Best Utility, and Best Charting/Imaging (a combination of last year's separate Best Charting and Best Imaging categories), were introduced.

### Best Accounting Package

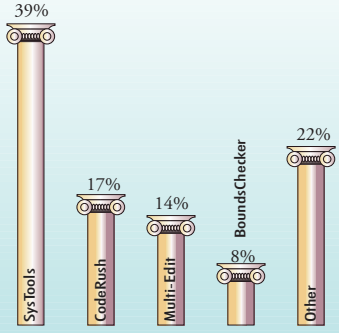
Seeing how close the tally was for Best Accounting Package, it's easy to see why we wanted to add this category. Barely passing AdaptAccounts by Adapta Software was Diversified Business Applications' Bravo, earning 30 percent to Adapta's 29. Bravo, the company's flagship product, is a complete solution capable of running with a variety of back-end databases, including Microsoft SQL Server, InterBase, Oracle, and Sybase.

Talk about close. If you like AdaptAccounts and didn't vote this year, maybe you'll think twice about skipping it next year!



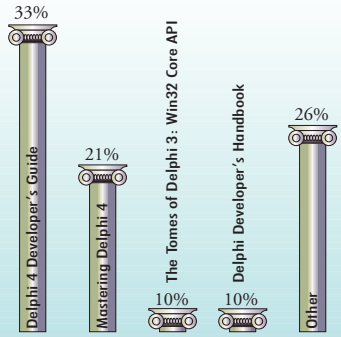

### Best Add-in/Library

This category has been popular every year, and this year is no exception. This year's winner is also no surprise; SysTools from TurboPower wins for the third year in a row. We have our first "three-peat" winner, garnering 39 percent. Coming in second — no easy achievement — is CodeRush from Eagle Software, rounding up 17 percent of the votes. This product wasn't on the ballot last year, and was able to debut near the top. If it keeps up this pace, who knows where it will be next year?



## Best Delphi Book

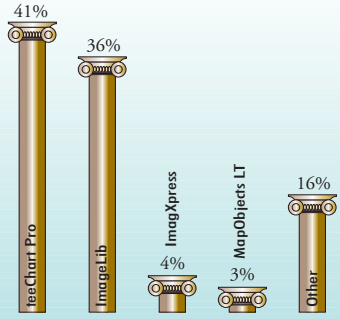
Important to every developer's collection are the books that help them get the most from their software, and this year's bunch offers many great titles. However, only two can be considered best sellers in our readers' eyes: *Delphi 4 Developer's Guide* [SAMS, 1998] by Steve Teixeira and Xavier Pacheco, and *Mastering Delphi 4* [SYBEX, 1998] by Marco Cantù. *Delphi 4 Developer's Guide* collected a hefty 33 percent of the votes, and *Mastering Delphi 4* amassed 21 percent.




## Best Charting/Imaging Tool

These numbers should be a cinch to show in a chart — using any of the great tools in this category, of course. This category brings another repeat winner: teeChart Pro from teeMach, which compiled 41 percent of the total votes in this category, compared to last year's whopping 50 percent. Obviously, most of you already know of teeChart's use in general purpose and specialized chart and graphing applications in engineering, finance, statistics, science, medicine, the Web, and business.

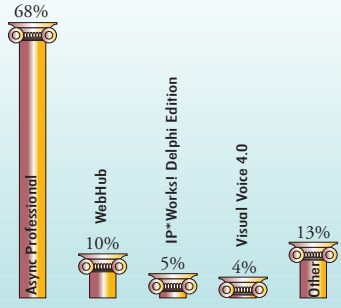
ImageLib from Skyline Tools made another great showing this year with 36 percentage points, compared to last year's 37. Skyline has never been one to ease up, and, chances are, they'll keep the competition on their toes this coming year.




## Best Connectivity Tool

This category's significance constantly grows as the world becomes more connected. The task of implementing connectivity functionality — at least for many Delphi users — lies in the hands of this year's winner: TurboPower Software's Async Professional, with a total of 68 percentage points! Hats off to TurboPower. Not only has the company dominated two categories so far, its products are three-peat performers in both of them.

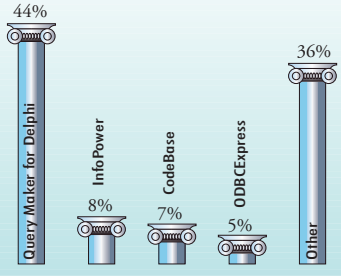
Worthy of mention is the runner-up, WebHub from HREF Tools, which slipped to 10 percent of the votes (compared to last year's 21), but I'm sure they're not slowing down a bit.




## Best Database Tool

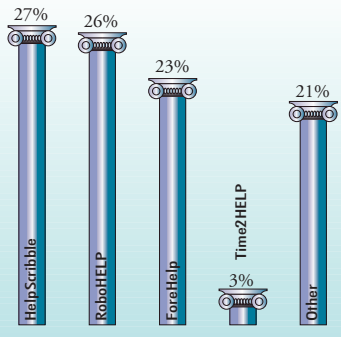
Another staple category, Best Database Tool is the second largest category on the ballot. With so many potential winners, this year's first-place bragging rights go to Strategic Edge's Query Maker for Delphi, a redistributable SQL query builder and report writer component. This new product accumulated an incredible 44 percentage points. Pretty good for its first time in our winner's circle!

Slipping a notch to second place is last year's winner, InfoPower from Woll2Woll Software. InfoPower amassed only 8 percent this year, but, knowing Woll2Woll, they'll be back to try and take back what was once theirs.



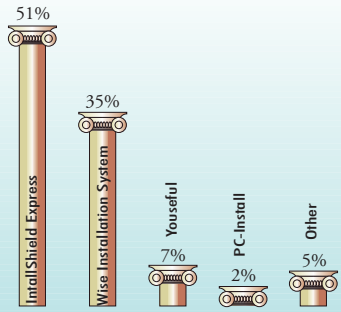

## Best Help Authoring Package

This category brings another underdog-turned-champion. Collecting a meager 7 percentage points last year, HelpScribble from Jan Goyvaerts gathered 27 percent of this year's votes — enough to edge out last year's winner, RoboHELP from Blue Sky Software, which garnered an uncomfortably close 26 percentage points. Apparently, HelpScribble needed no help, as its ability to parse source and generate an outline help file was enough to get you to vote in its favor.



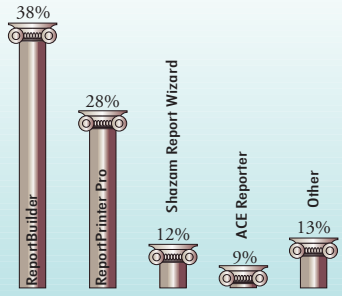
## Best Installation Package

There are only a few major players in this category, which makes the fight to the top that much more intense. Last year, InstallShield Express from InstallShield Software beat out Wise Installation System from Wise Solutions by only one percentile. With 51 percent of the votes, InstallShield gained some breathing room over Wise this year — 16 percent worth. Although InstallShield has managed three times to keep Wise out of pole position, it's always back and forth for these two competitors, so don't blink or you'll miss the next move.



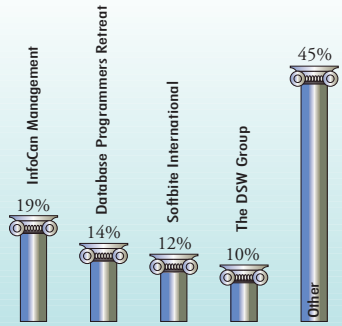

## Best Reporting Tool

It's always a pleasure to report (wink, wink) the results of this category. The winner in the two previous years, ReportPrinter Pro from Nevrona Designs, was hoping for a third first-place award this year. Alas, they must settle for a very respectable second, collecting 28 percent of the votes, as ReportBuilder from Digital Metaphors took first place with 38 percent of the votes. ReportBuilder 4.0 offers a new report wizard, drag-and-drop productivity tools, enhanced calculation components, mail/merge, and "keep together" support, to name a few.




## Best Training

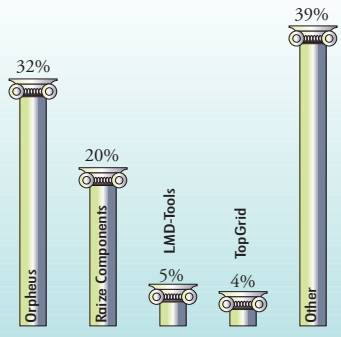
Let this year's winner pass their knowledge on to you, and you may learn how to be a winner — two times in a row. In a repeat performance, InfoCan Management took first place this year by gathering 19 percent of the votes. The runner-up, Database Programmers Retreat, went from 5 percent to 14 percent, narrowing the gap from behind. Softbite International came in third with 12 percent of the votes. With the margins among the top three finishers so close, who knows what next year will bring?




## Best VCL

This is the largest category on the ballot, consisting of 23 possible choices (not including "other") — almost twice the number of last year. This becomes more significant when you see that with all these great products to choose from, last year's winner held its spot at the top. Yes, TurboPower's Orpheus retained its title as Best VCL, garnering 32 percent of the votes for this category, a few points higher than they had last year.

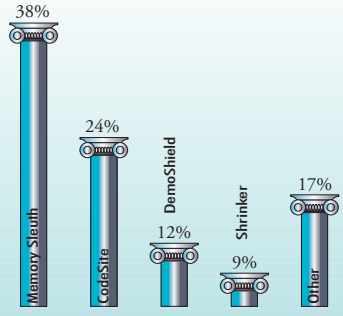
Second place honors go to Raize Software Solutions' Raize Components, which last year came in fourth — proof again that the competition is never too far behind, and one year can mean losing or gaining a lot of ground.




## Best Utility

This is another new category, but it already has a fairly large number of products to offer. The first to reach the top in this category is Memory Sleuth from — surprise, surprise — TurboPower Software, claiming 38 percent of your votes. Memory Sleuth is not something you easily forget, with its ability to help you track down dozens of memory leaks and resource bugs in your Delphi programs.

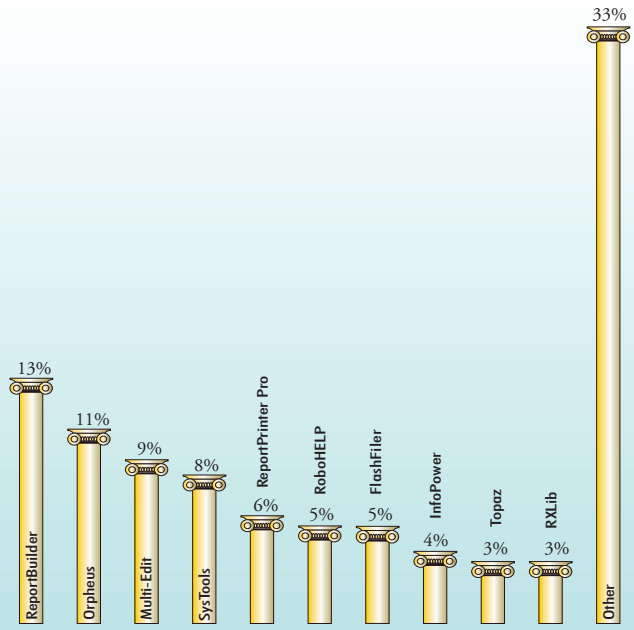
The folks at TurboPower simply don't leave anything to chance. That's because they know someone's always on their heels. In this case, it's CodeSite from Raize Software Solutions, which compiled a respectable 24 percent.



## Product of the Year

As always, this category instills the most excitement, because the winner claims the highest honors — the single-most coveted award. This year brings an upset, as first place is taken away from three-time Product of the Year winner InfoPower from Woll2Woll. This year, ReportBuilder from Digital Metaphors is the most popular Delphi product on the market, accumulating 13 percent of the votes.

To put the scope of the competition into better perspective, the second place winner, Orpheus from TurboPower, garnered a very close 11 percent of the votes, third place received 9 percent, and fourth place received 8. You can't get much closer than that. The gaps are getting narrower, making it that much more exciting to see who will come out on top next year.




## See You Next Year

This makes two Readers Choice Awards under my belt. I must say it's a challenging and exciting undertaking. It's an honor to not only witness the competition first-hand, but to present you, our readers, with the results. I thank everyone for their participation and contribution to this year's awards. Behind the printed results on these pages are vendors with the desire to stand tall, undaunted by the competition. Nothing short of perseverance, commitment, creativity, and good old-fashioned hard work enabled this year's winners to come out on top. Let's keep the passion alive until next year, or until the world ends — whichever comes first. Δ

Chris Austria is Products Editor at *Delphi Informant*, and can be reached via e-mail at caustria@informant.com.

## Contacting the Winners

**Best Accounting Package**
*Bravo*
**Diversified Business Applications, Inc.**
**Phone:** (510) 658-8535
**Web Site:** http://www.bravosoft.com

**Best Add-in/Library**
*SysTools*
**TurboPower Software**
**Phone:** (800) 333-4160 or (719) 260-9136
**Web Site:** http://www.turbopower.com

**Best Delphi Book**
*Delphi 4 Developer's Guide*
By Steve Teixeira and Xavier Pacheco
**SAMS**
**Phone:** (317) 228-4336
**Web Site:** http://www.mcp.com/publishers/sams

**Best Charting/Imaging Tool**
*teeChart Pro*
**teeMach, SL**
**Phone:** 34 972 59 71 61
**Web Site:** http://www.teemach.com

**Best Connectivity Tool**
*Async Professional*
**TurboPower Software**
**Phone:** (800) 333-4160 or (719) 260-9136
**Web Site:** http://www.turbopower.com

**Best Database Tool**
*Query Maker for Delphi*
**Strategic Edge**
**Phone:** (415) 563-3755
**Web Site:** http://www.strategicedge.com

**Best Help Authoring Package**
*HelpScribble*
**Jan Goyvaerts (JGsoft)**
**E-Mail:** jg@jgsoft.com
**Web Site:** http://www.jgsoft.com

**Best Installation Package**
*InstallShield Express*
**InstallShield Software Corp.**
**Phone:** (800) 374-4353 or (847) 240-9111
**Web Site:** http://www.installshield.com

**Best Reporting Tool**
*ReportBuilder*
**Digital Metaphors**
**Phone:** (972) 931-1941
**Web Site:** http://www.digital-metaphors.com

**Best Training**
**InfoCan Management**
**Phone:** (888) INFOCAN or (604) 736-5888
**Web Site:** http://www.infocan.com

**Best VCL**
*Orpheus*
**TurboPower Software**
**Phone:** (800) 333-4160 or (719) 260-9136
**Web Site:** http://www.turbopower.com

**Best Utility**
*Memory Sleuth*
**TurboPower Software**
**Phone:** (800) 333-4160 or (719) 260-9136
**Web Site:** http://www.turbopower.com

**Product of the Year**
*ReportBuilder*
**Digital Metaphors**
**Phone:** (972) 931-1941
**Web Site:** http://www.digital-metaphors.com

table of CONTENTS



*By Philip Brown*

# Property Overriding

## Extend the Functionality of *Any* VCL Class

The Visual Component Library (VCL) supplied with Delphi allows developers to extend the behavior of the existing controls by creating an **inherited** class, adding their own methods, properties, and events, and overriding methods that the VCL designers have supplied as **virtual**.

Sometimes, however, this is not enough. The VCL doesn't expose all methods in the **protected** section (by defining them as **private**), making them unavailable to descendant classes. Many key routines are defined statically, ruling out the possibility of overriding them and adding new functionality.

Fortunately, it's possible to extend the standard functionality of any class (including components) using the built-in capabilities of Delphi. One technique, buried deep within the Delphi documentation, is *property overriding*. This technique allows a class to redefine the behavior of a property, while still accessing the existing operations — even if the accessor functions used to control the property are **private**, and, therefore, typically unavailable.

```
type
  TAncestorClass = class
  protected
    procedure VirtualMethod; virtual;
  end;

  TDescendantClass = class(TAncestorClass)
  protected
    procedure VirtualMethod; override;
    procedure StaticMethod;
  end;

procedure TDescendantClass.VirtualMethod;
begin
  inherited;
  // New functionality added here.
end;
```

**Figure 1:** An example of using the **inherited** call.

Property overriding is a standard, "pure" technique, i.e. it doesn't break any of the rules of object-orientation or scope. Using a variation of property overriding, it's even possible to treat statically defined methods as **virtual**, providing new behavior in identically named methods in descendant classes, while retaining access to the original properties.

It should be remembered that a static method defined in a class can be used by descendants (as long as it's not declared in the **private** section), whereas a **dynamic** method can be overridden in descendant classes by declaring a method with the same name and parameter list, followed by the **dynamic** or **virtual** keyword. These overridden methods can, and nearly always do, call the functionality defined in the same method in ancestor classes through the **inherited** call (see Figure 1).

It's well known that the **inherited** keyword has an optional parameter as the name of the method. When defined, it's normally used to pass parameters to the same-named ancestor method (typically used in constructors). In fact, in this context, it's entirely unnecessary — the **inherited** keyword on its own means "call my ancestor method with the parameter list that was passed to me."

What's generally less known is that it's perfectly allowable to substitute a different method name as the parameter to the **inherited** call. This allows a descendant to call an ancestor method of a different name. A particular twist to this situation (which we will use to our advantage later) is bypassing functionality. Let's say the *TDescendantClass.StaticMethod*

wants to call *TAncestorClass.VirtualMethod*, but doesn't want to invoke the functionality defined in its own *VirtualMethod*. This can be achieved using the following code:

```
procedure TDescendantClass.StaticMethod;
begin
  inherited VirtualMethod;
end;
```

This illustrates calling a different **inherited** method, and also causes the *VirtualMethod* functionality defined in *TDescendantClass* to be bypassed.

## Changing the Behavior of a VCL Property

Let's say we want to extend the behavior of the *TEdit* VCL control — in particular, the behavior of the *ReadOnly* property. It would be nice if a *TEdit* control had a different background color when it was *ReadOnly*, and if it would disable its associated *TLabel*. This behavior would mimic that of Windows 95 and NT 4.0, and provide greater visual feedback to the user. Of course, we would also want to retain *TEdit*'s existing functionality. Looking at the VCL source code, this is the declaration for the *ReadOnly* property:

```
property ReadOnly: Boolean
  read FReadOnly write SetReadOnly default False;
```

Unfortunately, the *SetReadOnly* method is declared in the **private** section. Therefore, if we try to declare our own *ReadOnly* property with a new accessor function, there's no way we can call the *SetReadOnly* method defined in *TEdit* to retain existing functionality. What many people would do at this stage would be to provide a new property — called *Editable*, for example — that implements our own functionality but sets the *ReadOnly* property. This is unsatisfactory, as there are now two methods that perform similar functions.

If developers wanted to keep the same property name for the new method, they would be forced to implement their own *ReadOnly* properties with their new functionality and also mimic the behavior of the old *ReadOnly* property. This seems simple (to absorb any key presses); however, to provide a proper implementation, we must allow Clipboard copy operations (Ctrl C) and other special Windows key presses, e.g. Tab. Whichever way this is implemented (by copying the VCL source or coded from scratch), this approach is undesirable. What an abuse of object-oriented principles: copying code in an ancestor method simply because we have the source!

Of course, some of the blame for this lies with the Delphi developers that defined too many VCL methods as **private**. They've absolved themselves, of course, by providing mechanisms to work around this. They're not well documented, however. In fact, I stumbled across the technique simply by trying it out as a logical extension of the compiler's capabilities.

To extend the behavior of the *ReadOnly* property, we'll define our own *ReadOnly* property with our own *SetReadOnly* accessor function, then set the *ReadOnly* property defined in our ancestor (which will use our ancestor's *ReadOnly* property). Hang on a minute, what was that? Well, the **inherited** keyword allows us not only to call methods, but also to set properties. For example, the statement:

```
inherited ReadOnly := True;
```

allows us to set the *ReadOnly* property at the notional level of our ancestor, thereby calling whatever accessor functions were defined for the property at that level (even if they are **private** to our ancestor). Therefore, if we provide our own functionality to support our *ReadOnly* behavior, this mechanism lets us tap into that which the VCL provides — virtually for free.

```
interface

type
  TXEdit = class(TEdit)
  private
    FReadOnly: Boolean;
    FDesignColor: TColor;
    procedure SetReadOnly(Value: Boolean);
    procedure SetColor;
  protected;
    procedure Loaded; override;
  published
    property ReadOnly: Boolean
      read FReadOnly write SetReadOnly default False;
  end;

implementation

procedure TXEdit.Loaded;
begin
  inherited;
  FDesignColor := Color;
  SetColor;
end;

procedure TXEdit.SetColor;
begin
  if ReadOnly then
    Color := clBtnFace;
  else
    Color := FDesignColor;
end;

procedure TXEdit.SetReadOnly(Value: Boolean);
begin
  // Enabled/disable label that's linked to us. We can do
  // this by looking at our owner's Controls property for a
  // TLabel which has a FocusControl property linked to us.
  for ThisCtrl := 0 to Parent.ControlCount - 1 do
    if (Parent.Controls[ThisCtrl] is TLabel) and
       (TLabel(Parent.Controls[
         ThisCtrl]).FocusControl = Self) then
      Parent.Controls[ThisCtrl].Enabled := not Value;
  FReadOnly := Value;
  inherited ReadOnly := Value;
  // Change control background color.
  SetColor;
end;
```

**Figure 2:** The *TXEdit* class provides our new functionality.

The complete implementation of a new *TEdit* class (called *TXEdit*), which provides our new functionality, is shown in Figure 2. This technique — of using the **inherited** keyword to direct behavior to our ancestor — can be extremely useful in implementing new classes effectively, especially when used to set properties at the notional ancestor level.

Note that the *Loaded* **virtual** method is overridden so that after the component properties have been read in from the stream at run time, the original background color set at design time is remembered. Also, we need to override the *Create* constructor to set an initial value for this same original background color.

The behavior at design time is still a little less than ideal. For example, if we drop a *TXEdit* on a form, set its color to *clYellow*, and then toggle its *ReadOnly* property from False to True and

then back to False, the control color is set back to *clWindow*. This is simply because we are remembering the control's original color when it was created, not whenever it is changed. You could, of course, apply the same technique to the control's *Color* property. That way, when we change the control's *Color* property, we could save this color as its non-*ReadOnly* color. The behavior at run time, however, is fine and doesn't exhibit this behavior because at design time we defined what its default color will be.

## Extending Further: Changing Related Classes

There is one area of the VCL, however, where even these techniques don't allow us any latitude: related classes. Let's take as an example the *TListView* component, which is an excellent way of presenting a number of objects. The *TListView* class has a property, *Items*, which returns a *TListItems* object, which manipulates a set of *TListItem* objects — one for each element in the list view.

Because the *TListView* itself creates these classes, we can't change what class types are instantiated (and so cannot provide our own descendants of *TListItems* and *TListItem*). We can, of course, subclass the *TListView* class to produce our own *TXListView*, but we can't fundamentally change the fact that we will be dealing with *TListItems* and *TListItem* classes. (This isn't strictly true. We could completely hide the *Items* property, construct our own *TXListItems* class, and return this. However, there would be a lot of work and duplication of code involved, so it's not an elegant solution.)

*TListView* components are often used to allow the user to pick an element and manipulate it. In these cases, it's useful to attach an instance to the *Data* property of each *TListItem*, so the object represented by the list element is readily accessible. However, there's a snag to this approach. The standard way of clearing the list is to call *TListView.Items.Clear* (i.e. a method on *TListItems*), and, even if we have our own *TListView* descendant, we cannot be informed when this method is called. Therefore, if we attach instances to each *TListItem*, then call *TListView.Items.Clear*, those instances will not be freed; they will be left as "orphans," consuming system resources until the application exits. It's possible to tap into the *OnDeletion* event of the list view (which occurs whenever an item is deleted), but it's good class design to handle internal housekeeping invisibly, without requiring the user to remember to add event handlers for standard behavior. Of course, if we were to attach a routine to the event handler in code, then any event handler the user defined would eliminate the one we attached earlier.

It would be better if there were some way in which you could provide your own classes for related classes to VCL components, so we could override the *Clear* method of *TListItems*, for example, so we could handle the previous situation. Unfortunately, this isn't possible. However, using our property and method overrides, some intuition, and a little class-safe typecasting, it's possible to achieve the same effect.

We're going to provide three new classes — *TXListView*, *TXListItems,* and *TXListItem* — that allow us to extend their similarly named VCL counterparts. Behind the scenes, *TListView* will still be responsible for constructing the related objects (i.e. *TListItems* and *TListItem*), but we will "confuse" *TXListView* into thinking it's dealing with our own classes.

Figure 3 shows a few definitions to get us started. You can see from these definitions that our *TXListView* returns *TXListItems*, which are sets of *TXListItem* objects. In addition, we've given ourselves access

```
type
  TXListItem = class(TListItem)
  private
    function GetID: string;
    procedure SetID(Value: string);
  published
    property Identifier: string read GetID write SetID;
  end;

  TXListItems = class(TXListItems)
  private
    function GetItem(Index: Integer): TXListItem;
  public
    procedure Clear;
  published
    property Item[Index: Integer]: TXListItem
      read GetItem; default;
  end;

  TXListView = class(TListView)
  private
    function GetItems: TXListItems;
  published
    property Items: TXListItems read GetItems;
  end;
```

**Figure 3:** Defining our new classes.

to the *Clear* method of *TXListItems*, and we've added a new property to *TXListItem*, named *Identifier*. So how does it all hang together?

It starts with *TXListView*, which simply has this as its implementation of *GetItems*:

```
function TXListView.GetItems: TXListItems;
begin
  Result := TXListItems(inherited Items);
end;
```

Here, we're using our magical **inherited** property to return the *Items* property defined in our ancestor (the standard VCL *TListView*), but typecast as *TXListItems*. Note that we're still relying on our ancestor to handle the construction and destruction of the class; we just change the type that's returned. We do something similar for *TXListItems.GetItem*:

```
function TXListItems.GetItem (Index: Integer): TXListItem;
begin
  Result := TXListItem(inherited Item[Index]);
end;
```

We can now provide our own functionality in the *TXListItems.Clear* method:

```
procedure TXListItems.Clear;
begin
  // Use existing functionality.
  inherited Clear;
  // Our own functionality can be provided here.
  ShowMessage('List cleared!');
end;
```

Again, we're calling our ancestor's *Clear* method using the **inherited** keyword. If we only called *Clear*, we would initiate infinite recursion, causing stack overflow. The *Clear* method is static, not **dynamic**, but we're still able to reference it in a descendant and call it via **inherited**.

If we now drop a *TXListView* onto a form, it presents its *Items* property as *TXListItems*, and will, therefore, reference our *TXListItems* class

properties and methods. There is an important restriction in what we can and cannot do here; although we're typecasting the *Items* property to be a *TXListItems* object, it will still be constructed as a genuine standard VCL *TListItems*. We cannot lose sight of that, even if we treat it as *TXListItems*. In practice, this means we cannot add data elements to the class as **public** or **private** fields, or any **dynamic** methods. We can't do this because it would change the memory footprint of the class, confusing the compiler to the extent that it would reference memory incorrectly. The program has been compiled to expect a *TXListItems* class, but because this class is only constructed as a *TListItems*, memory addresses (of the field variables and Virtual Method Table) at run time will be for this class and not what the compiler expected. If these two vary, unpredictable (and usually terminal) side effects will result.

This is not, in practice, the straightjacket it seems to be. Genuinely useful functionality can be added simply by providing your own static methods (which can, of course, call ancestor static or **dynamic** methods), especially for these related classes, where typically all we want to do is provide an extra method or two and know when certain other **public** methods have been called.

You may have noticed the *Identifier* property on the *TXListItem* class. I've just said you can't add **private** fields, so why have I defined a new property on my phantom class? It's because you can make use of a particular property of *TListItem*, and store your own data in it. The *TListItem* represents an element presented in the *TListView*. If the list view's *ViewStyle* property is set to *vsReport*, it can have a number of columns. The first column is represented by the *Caption* property of *TListItem*, but all the subsequent columns are stored in the *SubItems* property, a standard *TStrings* class (a direct descendant of *TStringList*). Typically, a *TListItem* instance will have as many elements in its *SubItems* list as there are columns in the list view, less one (for the first column stored in the *Caption*).

There is no restriction on how many elements a *TStringList* can store, however, so we can store our own data in extra elements. This is what the *GetID* and *SetID* accessor functions do in *TXListItem*; they maintain the *Identifier* property in an extra element in the *SubItems* property. This is never displayed on screen, as there is no corresponding column in the list view. The *GetID* and *SetID* routines are slightly complicated because they must ensure the *SubItems* list is maintained correctly, but the basic technique is simple. (The routines are too long to be shown, but the full source code for the five classes introduced here can be downloaded; see below for details.)

## Conclusion

We've explored some of the powerful ways in which classes can interact using variations on the **inherited** keyword. Using it creatively has allowed us to elegantly solve some of the issues of extending components, and has opened areas of the VCL which have previously been inaccessible. Judicious use of the techniques presented here should allow you to extend existing components in new and exciting ways with minimum effort. And that's what object-oriented programming is all about. Δ

*The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\APR\DI9904PB.*

Philip Brown is a Senior Consultant with Informatica Consultancy & Development based in Surrey, UK. An OO advocate, he has been developing with Delphi since its release, and specializes in advising clients on the best way to leverage objects in a business environment. He can be contacted at phil@informatica.uk.com.

## *Charlie Calvert's Delphi 4 Unleashed*

I'm a huge Charlie Calvert fan. If you've ever attended one of his presentations at a conference (such as ICON, the Inprise Conference), or one of the many appearances he makes at user groups world-wide, you already know why. Charlie is one of the most entertaining speakers around. On top of that, he knows his stuff. One of the more amazing things about Charlie's presentations is that he makes any topic approachable, if not simple. Making a topic seem simple takes a great deal of time and effort. Charlie spends the time and makes the effort, and his audiences benefit.

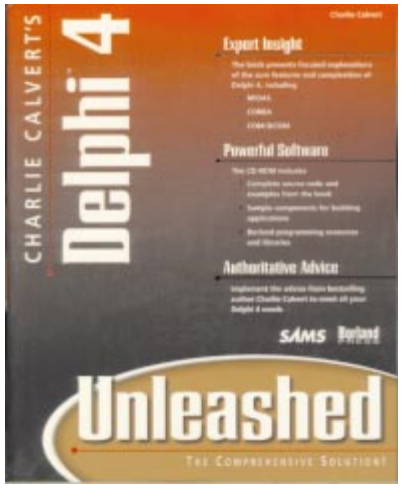
Which brings me to *Charlie Calvert's Delphi 4 Unleashed*. Charlie writes just as he speaks, i.e. he provides a lot of detail in an easy-to-digest package, liberally sprinkled with personal comments and insights. His writing style is casual, and he makes frequent use of the first person singular. Some readers may find this distracting, but for the most part, it works well. In fact, Charlie makes good use of this style to share his philosophy on programming (and just about anything else that occurs to him). For example, Charlie writes, "I believe that programming is among the most fascinating of all human endeavors. I think we are all wonderfully privileged to be alive at a time when it is possible to pursue this discipline." But not all of these side comments are so weighty. Some are simply delightful, such as, "There are various forms of madness in this world, but undoubtedly one of the most egregious is to call an OLE function and not check its return value." You get the idea.

*Unleashed* is extremely readable, but more importantly, it has excellent content. The section on creating components, while merely an overview, emphasizes a general philosophy about component design that should be required reading for every component developer. On the other hand, the section on the component object model (COM) is extensive, providing the reader with detailed discussions of interfaces, COM, and OLE automation.

If you want to get started with CORBA, you would be hard pressed to find a better book. The chapter on CORBA alone is worth the cover price, and is a perfect example of the clarity and simplicity that Charlie's hard work produces. *Unleashed* also has sections on Web development, including an excellent chapter written by Bob Swart on creating Web server extensions. An entire section on graphics programming includes extensive coverage of DirectX technologies (DirectDraw, DirectSound, etc.). In short, although *Unleashed* doesn't cover all aspects of Delphi 4, it does cover many of the more interesting topics very well.

Should you buy this book if you've purchased an earlier edition of Charlie's *Unleashed* series? The answer is an unqualified "Yes!" Although Charlie's previous editions (*Delphi Unleashed* and *Delphi 2 Unleashed* ) were excellent books, Charlie manages to pack a tremendous amount of new material in each edition, repeating a surprisingly small amount of information. I estimate that less than a fourth of this book is re-worked material. This is an amazing feat for an author — especially one with a day job.

I love this book, but I do have a couple of complaints. First, the production is downright sloppy; there are a surprising number of grammatical and typographical errors throughout the book that were obviously introduced during production. I hope SAMS reviews why this happened. Second, five of the book's chapters were not printed. Instead, they appear on the accompanying CD-ROM. I suspect it has




to do with the book's length. At 1,152 pages, it's a big book; including these chapters would have increased the book's size to more than 1,300 pages.

Nonetheless, this is an excellent book that I fully recommend. Most readers will delight in Charlie's clever observations and natural intellect, and everyone will enjoy the technical detail. Charlie works hard and makes his topics seem effortless, and the result is a book worthy of your shelf space. *Delphi 4 Unleashed* has something for everyone, and is a book that every Delphi developer should consider adding to their collection.

— *Cary Jensen, Ph.D.*

***Charlie Calvert's Delphi 4 Unleashed*** by Charlie Calvert, with contributions by Bob "Dr. Bob" Swart and Jeff Cottingham, SAMS Publishing, http://www.samspublishing.com.

**ISBN:** 0-672-31285-9
**Price:** US$49.99
(1,152 pages, CD-ROM)

## *Mastering Delphi 4*

Of all the Delphi books available, only a handful have achieved special status. When a new version of Delphi is released, developers immediately go to their book stores to find the one book explaining the best way to use it. *Mastering Delphi 4* is clearly among that handful of books that sets the standard. This latest edition of *Mastering* continues the tradition established by its previous three versions. The third edition was winner of a *Delphi Informant* Readers Choice Award for 1998 (the fourth edition is runner-up in this year's awards; see page 26).

*Mastering* has five sections: "Delphi and Object Pascal," "Using Components," "Writing Database Applications," "Components and Libraries," and "Real World Delphi Programming." The book starts with a thorough look at Delphi 4's primary enhancements, including its redesigned IDE and its extensions to Object Pascal. Chapter 2 includes excellent coverage of the new dynamic arrays, as well as one of the better discussions I've seen on Delphi's implementation of long strings and reference counting.

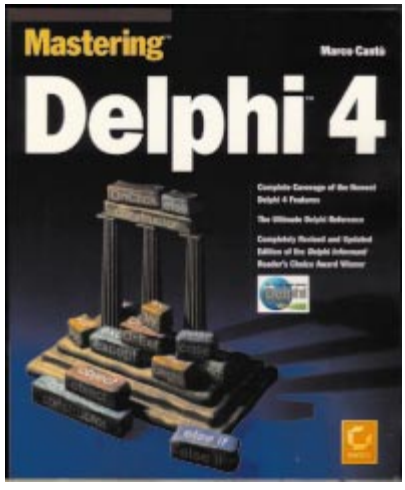
Throughout the book author Marco Cantù highlights the many new features found in Delphi 4, and does a remarkably good job of it. Naturally, with the addition of so many new features to Delphi over the years, some material must, by necessity, be covered rather briefly. One section I had hoped would have gotten a bit more attention was Delphi 4's new docking facilities, but the excellent coverage of other new features more than made up for it in my opinion.

As in previous editions, Marco provides strong coverage of database topics, as well as graphics and user-interface programming issues. The book is filled with excellent code examples illustrating many new and powerful techniques. The vast majority of these were small and elegant pieces of code. I have always said: "If it takes more than a few lines of code to do it, you're probably doing it wrong." Marco clearly applies this same thinking in his examples.

Out of curiosity, I dug up my original 1995 copy of *Mastering Delphi* (for Delphi 1) for comparison. Although both are of titanic physical proportions, I found little common material between them. Marco has gone to considerable effort to keep the material fresh and current. Subsequent editions of books typically contain few changes, but this is not the case with *Mastering Delphi 4*.

Another thing I enjoy about Marco's *Mastering* books is that they're not just technical reference books. I find them excellent simply for reading. And — believe me — that doesn't apply to many computer books! Marco has an excellent writing style; at times you feel he's standing over your shoulder saying "You're going to like this part ..." or "I bet you didn't know that ..." It's a personal, friendly style of writing that doesn't get bogged down in acronyms and tech-speak, yet gets to the meat of issues clearly and concisely.

There is definitely something here for everyone; *Mastering Delphi 4* is suitable




for beginning Delphi programmers as well as experts. If you're a Delphi beginner and want to learn what it can do, this book is for you. If you're an expert and want to dig into some of the deeper aspects of Delphi, there is plenty here for you as well. Overall, it achieves a satisfying balance between the two. I highly recommend it.

— *Robert Vivrette*

***Mastering Delphi 4*** by Marco Cantù, SYBEX, 1151 Marina Village Parkway, Alameda, CA 94501, http://www.sybex.com.

**ISBN:** 0-7821-2350-3
**Price:** US$49.99 (1,247 pages)
There's no CD-ROM. The code samples in the book can be obtained by visiting the SYBEX Web site.

table of CONTENTS

# Web Resources for Delphi Developers

In two previous columns, I discussed vendor-related Delphi sites and publisher sites. This month, I'll present two sites that provide information about file formats, two sites devoted to user interfaces, and five excellent Pascal sites.

**Learning about file formats.** Occasionally, I see questions posted to newsgroups or list servers about file formats. I recently learned about two sites that provide a wealth of information on this topic. The first, Wotsit's Format (http://www.wotsit.org) is a massive and well-organized site containing information about hundreds of file types. Its categories include graphics, movies and animation, spreadsheets, databases, fonts, games, word processing, and others. The site also includes additional programming information, such as algorithms and source code. It includes an excellent search engine, allowing you to easily find the information you need. It even includes an online chat forum where you can discuss programming issues.

Achim's File-Format Library (http://www.geocities.com/siliconvalley/horizon/3433/ff_.htm) is another site dedicated to this topic. Rather than including information on the site itself, it provides a wealth of links to other sites; there are over 900 links providing information on more than 600 file formats. Its main categories are 2D and 3D graphics, animation, sound, text, archives, database, programming, and miscellaneous. The miscellaneous category includes everything from game file formats to flowcharting application formats. As you can guess, there are many links to the Wotsit site, but there are links to many others as well.

**Best and worst user interfaces.** The Interface Hall of Shame (http://www.iarchitect.com/mshame.htm) is a unique site that should appeal to any developer writing Windows applications, regardless of language. Maintained by Isys Information Architects Inc., this site provides a wonderful collection of common UI design mistakes, e.g. faulty design of visual elements, improper use of error messages, etc. There is much wisdom in this quote from one of the examples: "Error messages are the antithesis of proper application design." You'll be surprised at some of the well-known applications from which the examples are taken. For examples of how to "do it right," take a look at its companion site, the Interface Hall of Fame at http://www.iarchitect.com/mfame.htm.

**Pascal the language.** As you know, Object Pascal is the language upon which Delphi is built. If you're new to Delphi, struggling to learn the language, or searching for special Pascal code, there are several sites that can help. The Pascal Language Tutorial (http://www.swcp.com/~dodrill/pasdoc/paslist.htm) covers the entire language. Like a text book, it consists of 16 chapters, structured to be studied in order, and you can download the source code for the example programs to study or modify.

Brian Brown and Peter Henry's Pascal Programming OnLine Notes (http://gatsby.tafe.tas.edu.au/pascal/pstart.htm) also provides a wealth of information. Topics span the entire language, from simple variables to arrays of records, from constants to pointers. If you wish to test your knowledge of Pascal, there is a cute "Test" section, written in JavaScript, that poses questions and gives you immediate feedback.

Need to search for particular Pascal code? Check out Pascal Central (http://www.pascal-central.com). In addition to technical information about Pascal and source code, this site includes a number of links to other Pascal sites and FTP sites from which you can download source code. I was particularly impressed with the page on Pascal Forums & Links.

One of the oldest Pascal sites I'm aware of is the SWAG Web site (http://www.gdsoft.com/swag/swag.html). This international site includes a large collection of source code and program examples, organized into 57 categories, covering the full range of programming topics. Included in these categories are data encryption, communications, and math routines. While many of these categories are specific to DOS programming with Pascal, there is one category devoted to Delphi.

Finally, the Coder's Knowledge Base (http://netalive.org/ckb) is similar to the previous site, featuring solutions to various programming problems. However, rather than being devoted strictly to Pascal, it includes language-independent algorithms. The emphasis is on Pascal code (Windows and general purpose) and it includes some Delphi code. The examples are code snippets rather than units or complete programs. The search facility makes it easy for you to find code examples in a variety of areas.

**In closing.** I would like to thank those readers who sent me sites to consider for this and future columns. I hope you find all of them beneficial. Next month, I'll present Delphi sites "off the beaten path." Δ

— Alan C. Moore, Ph.D.

*Alan Moore is a Professor of Music at Kentucky State University, specializing in music composition and music theory. He has been developing education-related applications with the Borland languages for more than 10 years. He has published a number of articles in various technical journals. Using Delphi, he specializes in writing custom components and implementing multimedia capabilities in applications, particularly sound and music. You can reach Alan on the Internet at acmdoc@aol.com.*