November 1996, Volume 2, Number 11

**Cover Art By:** *Doug Smith*

# From Database to Browser

*Generating HTML from Table Data*

## ON THE COVER



## FEATURES



## REVIEWS



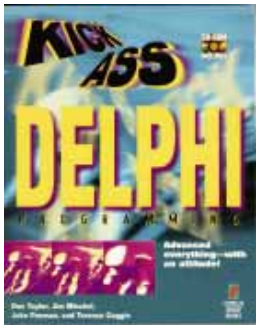
## DEPARTMENTS

# Delphi TOOLS

New Products and Solutions

## New Delphi Book

**Kick Ass Delphi Programming**
*Don Taylor, et al.*
Coriolis Group



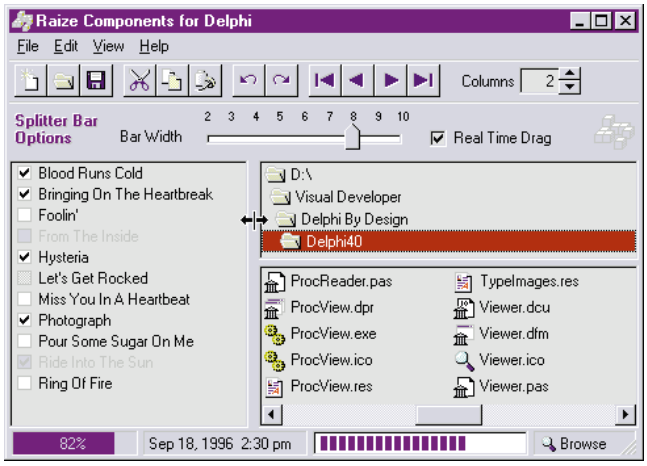
**ISBN:** 1-57610-044-8
**Price:** US$39.99
(523 pages, CD-ROM)
**Phone:** (602) 483-0192

## New Delphi Components

**Raize Software Solutions, Inc.** of Naperville, IL has announced ***Raize Components for Delphi***, a collection of more than 40 native Delphi components designed for both 16- and 32-bit development.

Featured components include: RzSplitter for creating Microsoft Explorer-style applications; RzTrackBar for adding standard thumb styles, owner-draw tick marks, and custom thumbs; and RzToolbar which adjusts its style according to the operating system, and its component editor adds buttons (including bitmaps) to the toolbar from a palette of 45 standard buttons.

Raize Components also features status components and RzSendMessage for creating Windows 95 logo-compliant applications that satisfy the e-mail requirement. This MAPI-compliant control sends e-mail by using string list properties to specify recipients and attached files.




It also includes more than 10 data-aware components. Data sets can be monitored using the RzDBStateStatus component.

These two components serve as the foundation for other list-oriented controls, such as RzCheckList and RzFontComboBox.

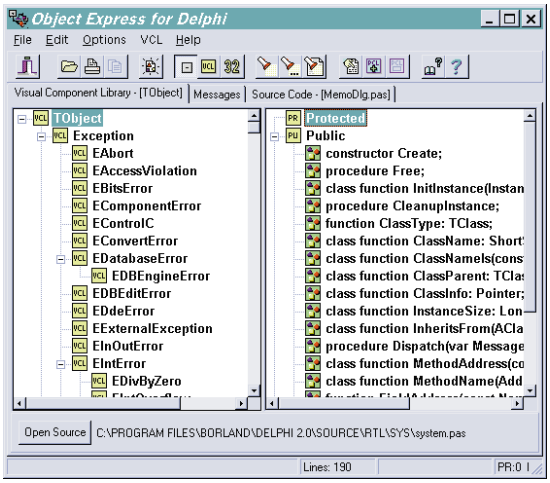
There are two demonstration programs available at the Raize Software Solutions Web site. Raize Components ship with complete source code, including all source code for the custom component and property editors.

**Price:** US$199.95
**Contact:** Raize Software Solutions, Inc., 2111 Templar Drive, Naperville, IL 60565
**Phone:** (630) 717-7217
**Fax:** (630) 717-7329
**E-Mail:** Internet: sales@raize.com
**Web Site:** http://www.raize.com

## OOPSoft Inc. Introduces Object Express for Delphi

**OOPSoft, Inc.** of Dallas, TX has introduced ***Object Express***, software that navigates the Delphi VCL and inherited classes. Object Express uses a tree-view format, and is currently available for Delphi 1 and 2 running on Windows 95 and Windows NT.

With Object Express, users can browse 16- and 32-bit VCL object trees, right-click to access a class' source code, directly inherit classes, add classes, and access Delphi's Help files.




Object Express' Quick Search feature provides direct access to more than 300 classes that comprise the VCL Tree. The user clicks on the Quick Search button, then enters the class name. Quick Search finds the class and displays its location on the tree. It also lists all properties and methods for the class. A right-click takes the user to the Object Pascal source where the class is defined.

Object Express includes drag-and-drop inheritance capabilities, enabling users to click on a particular class, and drag and drop it off the tree.

In addition, users can access Delphi's Help files. With the Extended Help button, the user can get in-depth information on a class in the VCL Tree, an object in the Class Object Tree, or a message in the Messages Tree.

With the ability to add custom classes to the VCL Tree, users can create a Personal Component Library.

Object Express makes it possible to create and manage multiple inherited classes simultaneously. This eliminates the need to close one file while working on another, and gives the user the ability to create inherited classes as fast as the PC allows.

**Price:** US$159
**Contact:** OOPSoft, Inc., 3422 Swan Lane, Irving, TX 75062
**Phone:** (888) 667-7638 or (972) 355-7401
**Fax:** (972) 255-4365
**E-Mail:** Internet: oopsoft@airmail.net

# Delphi TOOLS

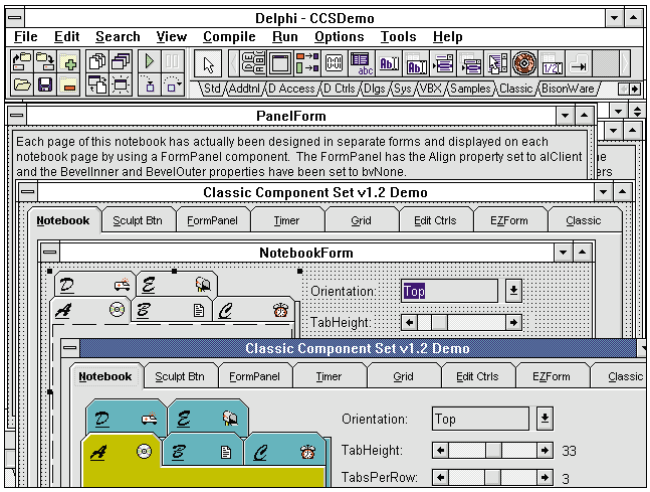
New Products and Solutions

## New Delphi Book

**Secrets of Delphi 2**
*Ray Lischner*
Waite Group Press


**ISBN:** 1-57619-026-3
**Price:** US$49.99
(831 pages, CD-ROM)
**Phone:** (415) 924-2575

## Classic Software Ships Classic Component Set Version 1.20




**Classic Software** of Inglewood, Australia released Version 1.20 of the ***Classic Component Set***. This version adds the *TcsDBDateEdit* and *TcsEZKeys* components, and *TcsEZForm* class.

*TcsDBDateEdit* is an enhanced *DBEdit* component which uses an Epoch setting to determine the century when inputting dates with no explicit century digits, much like SET EPOCH in CA-Clipper.

*TcsEZForm* is an abstract form class used to derive new forms that allow navigation between controls using the keyboard. The *TcsEZKeys* component can be used to change the default settings at design time.

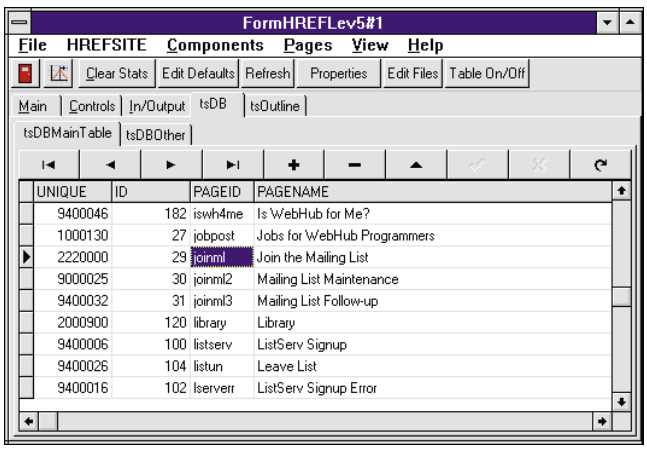
Adding enhanced navigation to new or existing forms is handled by adding the CSEZForm unit to the project, and adding CSEZForm to the new or existing form's **uses** clause, and changing the form's ancestor class from *TForm* to *TcsEZForm*. Enhanced navigation can be enabled or disabled on a form-by-form basis, or for all *TcsEZForm* forms.

Other components in the Classic Component Set include: *TcsNotebook*, *TcsFormPanel*, *TcsGrid*, *TcsSculptButton*, *TcsHiResTimer*, *TcsProperEdit*, *TcsDBProperEdit*, *TcsRankListBox,* and, in Delphi 1 only, *TcsAutoDefaults*.

A free trial version is available from Classic Software's Web site.

**Price:** Version 1.20, US$69, AUS$90, source code is included. There are no run-time royalties, and free technical support is available. The set ships with a 30-day money-back guarantee.
**Contact:** Classic Software, Unit 2, 19A Wood St., Inglewood, WA 6052, Australia
**Phone:** 61 9 271 5407
**Fax:** 61 9 271 5407
**E-Mail:** Internet: 100033.1230-@compuserve.com
**Web Site:** http://ourworld.compuserve.com/homepages/classicsoftware

## HREF Tools Announces the Release of WebHub EEP 8.9




**HREF Tools Corp.**, of Santa Rosa, CA, announced the release of ***WebHub Early Experience Package (EEP) 8.9***. Based on Delphi 2, WebHub EEP 8.9 is a Web application development tool that allows developers to put database information on the Web without CGI programming for intranets and the Internet. The new version integrates several components which perform various tasks, including database field lookup, full record display, VRML support, and e-mail generation. WebHub sites run on the Windows platform.

WebHub EEP 8.9 adds a *TWebDataForm* component which generates HTML for viewing or editing a database record. It connects to any Delphi-compliant database, including Oracle, Paradox, Access, dBASE, Sybase, Informix, and InterBase.

WebHub includes over 30 Web-specific components that can be combined to create an unlimited number of applications for interactive sites. The components can save state, track surfers individually, handle multiple simultaneous requests for high traffic sites, and separate HTML from programming code.

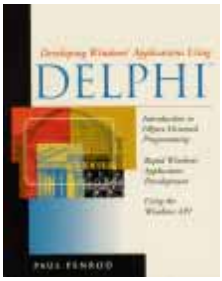
**Price:** Starts at US$575
**Contact:** HREF Tools Corp., 300 B St., Ste. 215, Santa Rosa, CA 95401
**Phone:** (800) 898-4733 or (707) 542-0844
**E-Mail:** Internet: martha@-mail.href.com
**Web Site:** http://www.href.com

# Delphi TOOLS

New Products and Solutions

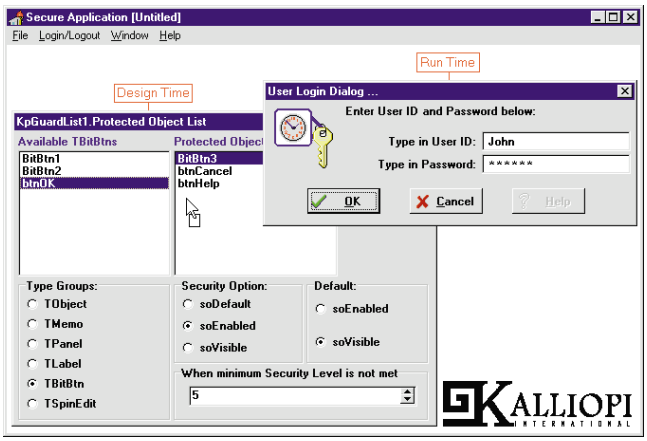
## New Delphi Book

Developing Windows Applications Using Delphi
*Paul Penrod*
John Wiley & Sons, Inc.



**ISBN:** 0-471-11017-5
**Price:** US$29.95 (353 pages)
**Phone:** (212) 850-6630

## Kalliopi International Announces Release of Security Components




**Kalliopi International** of Seattle, WA announced the release of its 32-bit compatible *Security VCL* components for Delphi. The two components, *TKpSecurityGuard* and *TKpGuardList*, work together to provide a complete Delphi Security Management System.

The GuardList component allows visual objects on the form to be protected. As users log in and out, the objects protected by the GuardList are enabled and disabled, based on the user's security level.

User login and logout, password verification, and functions to change passwords and edit user information are also provided.

The components are designed to be completely customizable.

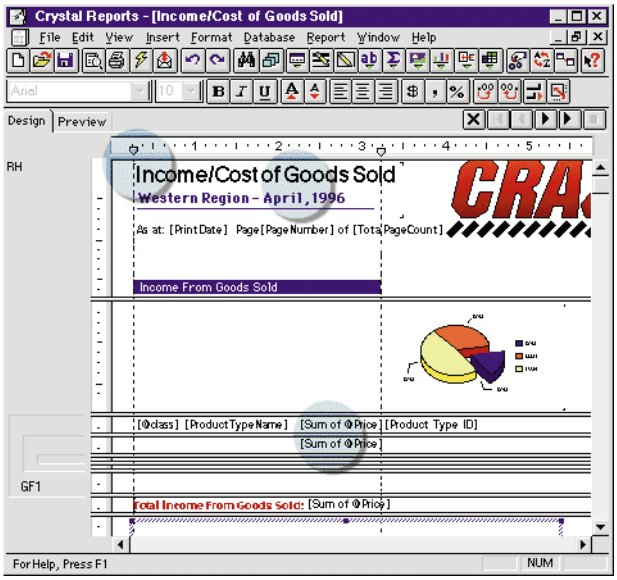
Every piece of text can be customized, including dialog captions and button labels. Properties are provided for access to important names such as user ID and password.

Security VCL features a one-way encryption algorithm, and supports a custom algorithm attached to the EncryptionEvent trigger.

The components are available through the CompuServe Shareware Registration Forum (ID 11704, and 11703 respectively, for the 16-bit version, and 13007 for the 32-bit compatible version).

**Price:** Version 1.0 (16-bit) and Version 1.02 (32-bit) with source code, US$49.95; Version 1.0, and 1.02, US$24.95.
**Contact:** Kalliopi International, 10423 40th Ave. NE, Seattle, WA 98125
**Phone:** (206) 522-7327
**Fax:** (206) 522-7327
**E-Mail:** CIS: 73523,342 or Internet: support@kalliopi.com
**CIS Forum:** GO DELPHI
**Web Site:** http://www.kalliopi.com

## Seagate's Information Management Group Ships Crystal Reports 5.0




**Seagate Software's Information Management Group** has launched *Crystal Reports 5.0*. This version adds new report types, database drivers, object-oriented report design control, additional developer features, and report distribution options, including the ability to publish reports on the Web.

With new native drivers for Oracle, Sybase, and Microsoft SQL Server, Crystal Reports 5.0 can read over 25 PC and SQL data sources, and access other data sources such as Microsoft Windows NT event logs, Microsoft Exchange data, and Web server activity logs.

Reports can be exported to more than 20 formats, including HTML, and distributed via e-mail, Lotus Notes, Microsoft Exchange, or the Web.

Crystal Reports can also be used as an application development tool. Rather then coding reports, developers can create and integrate reports into their database applications using various programming interfaces, including: the Delphi Visual Component Library (VCL), ActiveX Controls, Visual Basic Custom Controls (VBX), the MFC Class Library with AppWizard, or direct calls to the Crystal Report Engine.

**Price:** Crystal Reports Professional 5.0, US$395; upgrade for existing users, US$199. Crystal Reports Standard 5.0, US$195; upgrade for current users, US$79. The Crystal Report Engine can be distributed royalty free. The New Features Interactive Learning CD, US$99.
**Contact:** Seagate Software, Information Management Group, 1095 West Pender St., Vancouver, BC, Canada V6E 2M6
**Phone:** (800) 877-2340 or (604) 681-3435
**Fax:** (604) 681-2934
**E-Mail:** Internet: sales@-img.seagatesoftware.com
**Web Site:** http://www.img.seagate-software.com

Table of CONTENTS

# News LINE

November 1996

## Borland to Co-Sponsor Developers Conference

Desktop Associates Ltd. and Dunston Thomas Ltd., in conjunction with Borland, will host the Borland Developers Conference in London from April 20-24, 1997.

This year, the Borland Developers Conference will be divided into four major tracks, and multiple categories within those tracks. The tracks are: Delphi, Internet and intranet, client/server and databases, and a general track that includes sessions covering business solutions, operating systems, and the use of companion products to extend your Borland applications and development environments.

For more information, e-mail Chris Read at cread@dtuk.demon.co.uk, or call Borland at (408) 431-1000.

## Borland Announces Its Latest Version of InterBase

*Scotts Valley, CA* — Borland International Inc. has released InterBase 4.2, a platform-independent, SQL database server for Windows 95 and Windows NT. InterBase 4.2 features an improved version of the InterBase SuperServer Architecture. It also includes ODBC 2.5 drivers for Windows 95 and Windows NT, and thread-safe 32-bit client libraries.

The InterBase 4.2 advanced feature set includes: dramatic performance enhancements for large, multi-user systems; 32-bit GUI tools for interactive SQL, server management, and license management; and an identical code base and feature set across Windows 95 and Windows NT. It's also certified and optimized for Microsoft NT 4.0, and tuned for use with Borland's forthcoming all-Java JDBC driver for InterBase and InterClient.

Local InterBase, a single-user version of the server, is priced at US$249.95. The InterBase Server for Windows 95 is designed specifically for small teams that require no more than four concurrent users, and retails for US$599.95. InterBase Server for Windows NT is designed and optimized for larger client/server solutions, and is certified for use with SMP servers. It is priced at US$850. InterBase upgrades for 4.0 or 4.1 are priced at US$499.95. Additional license packs for Windows NT are available in single-, 10-, and 20-user configurations. For more information call Borland at (800) 233-2444, or visit their Web site at http://www.borland.com.

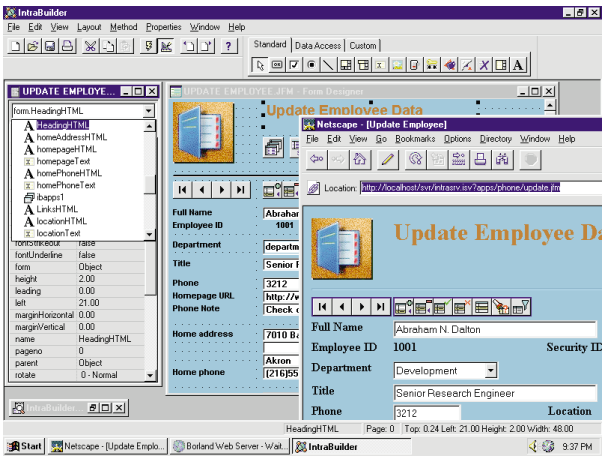
## Borland Names Paul W. Emery II as CFO

*Scotts Valley, CA* — Borland International Inc. has named Paul W. Emery vice president of finance and administration, and chief financial officer (CFO). Emery, who reports to Borland's Acting Chief Executive Officer and Chairman of the Board, William F. Miller, will be responsible for managing finance, information services, and operations.

Emery joins Borland after serving as CFO for Micromodule Systems. Previously, Emery held positions as vice president of finance and administration, and acting vice president of marketing for Megatest Corp.; CEO for System Industries; senior vice president of Santa Cruz Operation; CEO for Airmac Technology Systems; and CFO for Xynetics. Additionally, Emery served as a senior manager for IBM, and a finance manager for Ford Motor Co.

## Borland Announces IntraBuilder in Three Configurations

*Scotts Valley, CA* — Borland has announced product line and pricing strategies for its IntraBuilder line of live, data-driven intranet application development suites for Windows NT and Windows 95. Variously bundling Netscape FastTrack Web Server and Navigator Gold Internet client software, IntraBuilder is available in three configurations.




Designed for small workgroups and low-volume intranet applications, the Standard version includes Netscape Navigator Gold, and Borland's Personal Web Server with access to Paradox, FoxPro, dBASE, and Access data. The Standard version is priced at US$99.95.

IntraBuilder Professional version includes Netscape FastTrack Web server for Windows NT and the Netscape Navigator Gold browser, and is designed for higher-volume intranets. It includes support for desktop database formats, as well as remote data access to Microsoft SQL Server, and InterBase Windows NT Server. It also supports the leading Web server APIs, including NSAPI, ISAPI, and CGI. The Professional version retails for US$499.95.

Designed for high-volume, scalable, decentralized, and centralized IT environments, IntraBuilder Client/Server offers all the Professional version features, plus access to all corporate database servers, including native drivers for Oracle, Sybase, Informix, and DB2, and others through ODBC. IntraBuilder Client/Server allows developers to build multi-tier applications using Remote IntraBuilder Agents, which distribute incoming requests among several machines. IntraBuilder Client/Server is available for US$1,995.

For more information, contact Borland at (408) 431-1000, or visit their Web site at http://www.borland.com/-intrabuilder/.

table of CONTENTS

## On the Cover

Delphi 1 / Delphi 2 / Object Pascal / HTML

*By Keith Wood*

# From Database to Browser

## A Component to Write Data to HTML Tables

**Earlier this year, we developed an HTML-writing component that allowed us to control the generation of a Web page from a Delphi program with little knowledge of HTML. [Keith Wood introduced his *THTMLWriter* component in the May 1996 *Delphi Informant*.] This article builds on that work and describes another component that extracts data from any database available to Delphi and presents it in an HTML table for display on the Web.**

Dynamic Web pages are becoming more common every day. Organizations are placing data on the Web for clients and other interested people to access. Managing all this data is quite a task — one well-suited to a database. Combining the abilities of the *THTMLWriter* and *TDataSource* components makes it easier to create your Web site and keep it up-to-date.

### THTMLWriter

To recap, the *THTMLWriter* component provides numerous methods that either format text into HTML code as strings, or write HTML to a file for viewing. This article is about those methods related to HTML tables.

```
<table border=2 width=100% bgcolor=#FFFFFF>
<caption align=center valign=top>A sample HTML table
</caption>
<tr><th align=right>Part No</th><th>Description</th>
<th align=right>List Price</th></tr>
<tr><td align=right>900</td><td>Dive kayak</td>
<td align=right>3999.95</td></tr>
<tr bgcolor=#FF0000><td align=right>912</td>
<td>Underwater Diver Vehicle</td>
<td align=right>1680.00</td></tr>
<tr><td align=right>1313</td><td>Regulator System</td>
<td align=right>250.00</td></tr>
<tr><td align=right>1314</td>
<td>Second Stage Regulator</td>
<td align=right>365.00</td></tr>
</table>
```

**Figure 1:** A sample table description in HTML.

HTML tables are delimited by the `<table>` and `</table>` tags. Within these boundaries, tables comprise rows, denoted by the `<tr>` tag, which are composed of headings `<th>`, or cells `<td>`. Headings and cells contain text and other HTML tags. Additional table elements include borders that can be sized and made invisible (or colored in some browsers), and a caption that is applied to the entire table. All table elements can be aligned horizontally and vertically.

An example of HTML code for a table is shown in Figure 1. Figure 2 lists the *THTMLWriter* methods used to produce tables. Our new component invokes these methods at the appropriate times to present the information from the database in HTML.

### HTML Data Source

We want the ability to take data from any source available to Delphi and display it as an HTML table. Fortunately, Delphi is set up to easily interact with a large variety of data sources while hiding most of the operation's complexities. *TTable* provides an interface to database tables (desktop or server), while *TQuery* gives us access to the same data via SQL. These components are derived from *TDataset*, which encapsulates the common interface for any data source, allowing both to be treated the same when accessing the data (*polymorphism* in OO terminology).

| Procedures | Functions |
|---|---|
| *TableStartParams* | *FormatTableStartParams* |
| *TableStart* | *FormatTableStart* |
| *TableEnd* | *FormatTableEnd* |
| *TableRowStartParams* | *FormatTableRowStartParams* |
| *TableRowStart* | *FormatTableRowStart* |
| *TableRowEnd* | *FormatTableRowEnd* |
| *TableHeadingStartParams* | *FormatTableHeadingStartParams* |
| *TableHeadingStart* | *FormatTableHeadingStart* |
| *TableHeadingEnd* | *FormatTableHeadingEnd* |
| *TableHeadingParams* | *FormatTableHeadingParams* |
| *TableHeading* | *FormatTableHeading* |
| *TableCellStartParams* | *FormatTableCellStartParams* |
| *TableCellStart* | *FormatTableCellStart* |
| *TableCellEnd* | *FormatTableCellEnd* |
| *TableCellParams* | *FormatTableCellParams* |
| *TableCell* | *FormatTableCell* |

**Figure 2:** These *THTMLWriter* component methods deal with tables.

Delphi also provides the *TDataSource* as a buffer between *TTable* and *TQuery* and the display components we place on our forms. *TDataSource* can communicate in a standard way with any of the data sets. From this component, we derive our new *THTMLDataSource* component, which provides the *DataSet* property and our link to the data itself. Deriving the component, rather than simply including a *DataSet* property, retains all the functionality of the original, allowing the new component to be used just as the old one.

Next, we need a link to the *THTMLWriter* component that does all the work for us, so we add an *HTMLWriter* property to the component. To enable us to take full advantage of the abilities of this component, we must surface all the attributes we want to access. This involves creating a property from all the parameters we may want to alter when generating the table, including the alignment and color attributes.

The fields displayed in the table are provided by the *DataSet* in its *Fields* property, which can be set manually at design time, or generated automatically by Delphi. As with the *TDBGrid* component, we show only the visible fields, allowing additional fields required for the data access to be present without affecting the final output.

Also from the field definitions, we take the label to be displayed at the top of each column and the field's alignment. The column label is taken from the *DisplayName* property of the field. This is a pointer to the *DisplayLabel*, if entered, or *FieldName* property, and reflects what would appear in a DBGrid header for this field. To access the value, we need to de-reference the pointer by using the pointer symbol ( ^ ).

This de-referencing is not required in Delphi 2. To overcome this and have a single source for Delphi 1 and 2, we employ conditional compiler directives. This allows sections of code to be included or excluded based on various criteria. In this case, we require a conditional symbol defined only in Delphi 2, and achieved through the pre-defined WIN32 symbol:

```
{$IFDEF WIN32}
sCell := Fields[i].DisplayName;
{$ELSE}
sCell := Fields[i].DisplayName^;
{$ENDIF}
```

The column labels may not be required in all cases, so a Boolean property, *Headers*, is added to control their presence. Similarly, the default alignment within HTML may be desired over the field alignments as specified here, resulting in another Boolean property, *UseFieldAlign*.

## Displaying the Data

The data displayed for each field is taken from the *DisplayText* property of the fields. This is what Delphi uses in its data-aware components, incorporating any formatting specified in the *DisplayFormat* or *EditMask* properties. A consequence of this is that BLOB fields (such as graphics) aren't displayed in a meaningful way, but simply as field types in parentheses. Memo fields in the database are handled separately. Each line in the memo is extracted and written into the HTML table. To alter the values of the fields, we could also tap into their *OnGetText* event.

Another problem is that HTML doesn't include the graphic field's content on the page itself. Instead, a reference is made to an external image file. One way to show these images in an HTML table is to extract them from the database, write them to a temporary file, and include references to that file. The difficulty then becomes when to remove these temporary files. We can't tell when the user may request them again. An alternate, better solution is to set up a CGI program that extracts the graphic from the database and delivers it to the user directly on request, but that is not the topic of this article.

Calculated or lookup fields can be added as if we are displaying the results on a Delphi form. This allows more complicated formatting to be applied to the fields if necessary. Remember that we have the HTML generating resources of the *THTMLWriter* component at our disposal.

One of the more common reformatting tasks is to turn one of the columns into a link to other HTML documents. To expedite this, two additional properties are provided: the *LinkField* property, which specifies the field that supplies the text for the hot-spot within the table, and the *LinkTarget* property, which identifies the field to be used as the destination of the link.

To provide greater control over the appearance of the table, two events have been defined: *OnRowShow* and *OnCellShow*. The former is triggered at the start of every row, and the latter is triggered for each individual cell as it is formatted. Neither event is called for the header row if it's displayed. These events allow the alignment and colors for that row or cell to be altered as required. The type declaration for the *OnRowShow* property is shown here:

```
THTMLRowEvent = procedure(Sender: TObject;
  var ahAlignHoriz: THTMLAlignHoriz;
  var avAlignVert: THTMLAlignVert;
  var clrBackground, clrBorder, clrBorderLight,
      clrBorderDark: TColor) of object;
```

Note the use of the **var** directive to allow for changes to these values.

All of this is brought together in the one new method of the component, the *GenerateHTML* procedure. After the links to the HTML writer and data set have been made and the output parameters have been set, this method is invoked to produce the HTML table. It assumes the table is part of a larger HTML document, and doesn't perform any initialization or finalizing of that document. Thus the calling sequence should be:

```
MyHTMLWriter.Initialise;
MyHTMLDataSource.GenerateHTML;
MyHTMLWriter.Finalise;
```

Normally, additional code would provide a heading for the page and any other text related to the contents. The complete code for the *GenerateHTML* method is shown in Listing One beginning on page 9.

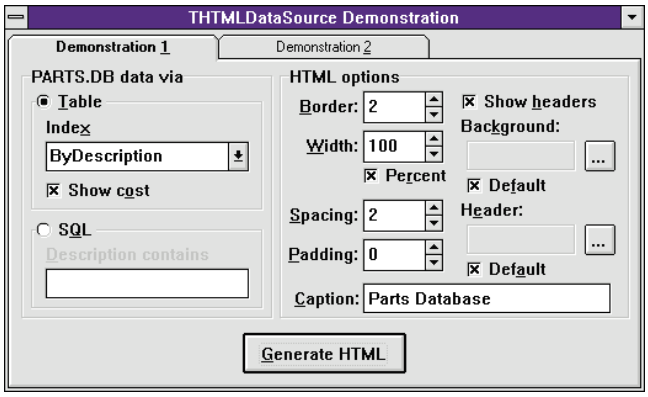
## Exceptions

Several error conditions can occur during the generation of the HTML table. These include having no *HTMLWriter* or *DataSet* assigned; specifying only one of the *LinkField* and *LinkTarget* fields; and the *LinkField* not being visible, having no visible fields, and having no records selected from the database. As usual, these are raised as exceptions for the calling routine to handle.

To simplify the processing of any errors, the exception used in this component is derived from the *EHTMLError* exception belonging to the *THTMLWriter* component. This means all HTML-related errors can be trapped and dealt with together by looking for the common ancestor.

## Demonstration Project

The demonstration project included with this article shows some of what the *THTMLDataSource* component can do. In each case, the HTML is generated into the file HTMLData.HTM, which can then be loaded into your browser. The data source is a Paradox table: WebSites.DB. The first page (see Figure 3) extracts data from the Parts.DB table that ships with Delphi. Access is provided through either a *TTable* or *TQuery* component, each



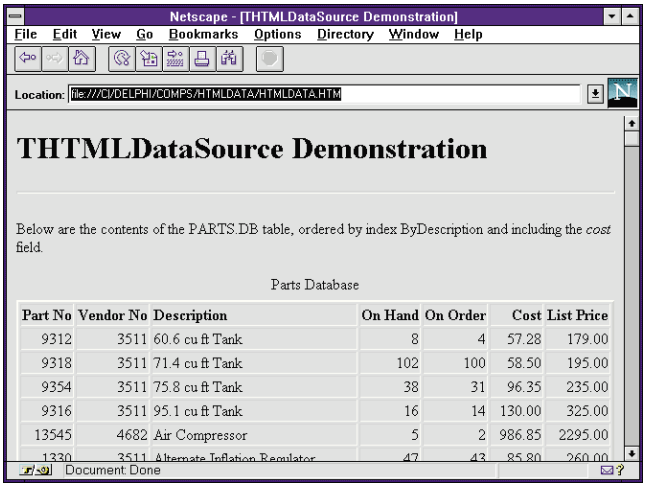



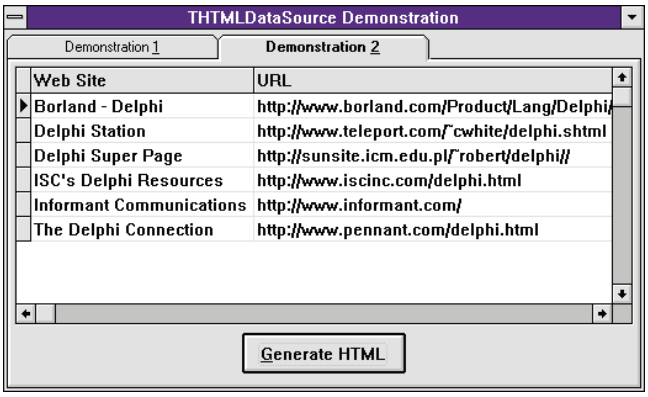

**Figure 3 (Top):** The first page of the demonstration project extracts data from a Paradox table (Parts.DB) that ships with Delphi.
**Figure 4 (Bottom):** The HTML page resulting from the query of Parts.DB.

with additional options. The table's order can be changed by selecting an index from the list, and the Cost column can be included or excluded. The latter is done by setting the field's *Visible* property to match the check box on the form. The query searches for text entered anywhere in the Description field. Try `Dive` or `Regulator` to get some results.

The right half of the page allows several of the properties of *THTMLDataSource* to be altered. The resulting HTML page can be seen in Figure 4. Try different combinations of options to see how they affect the output.

If you have a browser that supports colored cells in tables, you can see the colors selected for the table and header row. Also, the Cost column, when visible, is yellow, and rows with less than 20 items on hand are red. This is achieved through the *OnRowShow* and *OnCellShow* events. Check out the code to see how it works.

The second page of the demonstration project (see Figure 5) shows the component's ability to automatically generate links to other documents in the HTML table. A sample database with some URLs is provided and displayed on the screen. The *LinkField* and *LinkTarget* properties of the *THTMLDataSource* com-



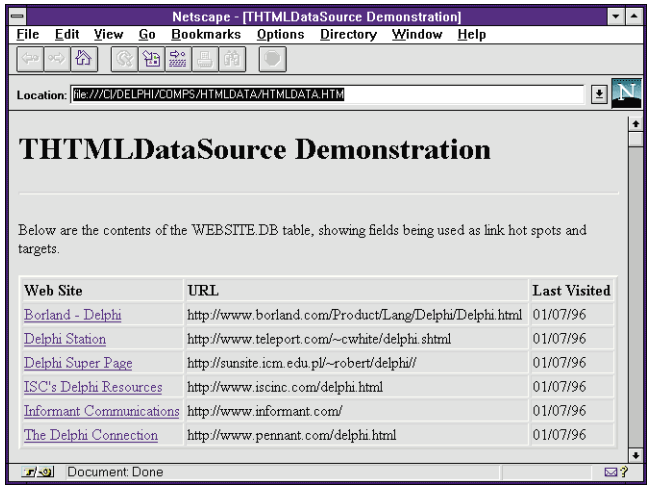




**Figure 5 (Top):** The second page of the demonstration program. **Figure 6 (Bottom):** When generated, the contents of the Web Site column become a link, with the destination coming from the URL field.

ponent are set to the Web Site and URL fields respectively. When generated (see Figure 6), the contents of the Web Site column become a link, with the destination coming from the URL field.

The demonstration project is best run outside of Delphi. Otherwise, exceptions that are trapped internally may appear (if "Break on Exceptions" is on) and disrupt the flow of the program.

## Conclusion

This extension to the *THTMLWriter* component enables us to display data as HTML from any data source available to Delphi. The quantity and appearance of the data is controlled in the same manner as showing it on a form in Delphi. Combine this with a CGI program to allow users to specify the data they are interested in, and you have a dynamic, up-to-date Web site.

Windows platforms are becoming more common as Web servers. By leveraging our knowledge and abilities in Delphi, we can provide more responsive applications to run on them, and move into new areas of endeavor. Δ

*The* THTMLWriter *component and demonstration project referenced in this article are available on the* Delphi Informant *Works CD located in INFORM\96\NOV\DI9611KW. Note: The* THTMLWriter *component featured in this download is an update to that provided with the previous article. The change involves moving the open HTML file from the* Create *method to* Initialise. *This allows multiple pages with the same name to be generated consecutively.*

Keith Wood is an analyst/programmer with CSC Australia, based in Canberra. He started using Borland's products with Turbo Pascal on a CP/M machine. Occasionally working with Delphi, he has enjoyed exploring it since it first appeared. You can reach him via e-mail at kwood@netinfo.com.au, or by phone at 6 291 8070.

### Begin Listing One — The Generated HTML Method

```
{ Generate HTML to display the table }
procedure THTMLDataSource.GenerateHTML;
const
  ahAlignments: array [TAlignment] of THTMLAlignHoriz =
    (ahLeft, ahRight, ahCentre);
var
  i, iCount: Integer;
  ahAlignHoriz: THTMLAlignHoriz;
  avAlignVert: THTMLAlignVert;
  clrBackground, clrBorder,
  clrBorderLight, clrBorderDark:  TColor;
  sCell: string;
  slMemo: TStringList;
  bmkSave: TBookmark;
begin
  { Check that HTMLWriter is set }
  if not Assigned(FHTMLWriter) then
    raise EHTMLDataSource.Create(tcTable,
            'No HTMLWriter assigned');
  { Check that DataSet is set }
  if DataSet = nil then
    raise EHTMLDataSource.Create(tcTable,
            'No DataSet assigned');
  { Check linked fields }
  if (LinkField <> nil)  or
     (LinkTarget <> nil) then
    begin
  if LinkField = nil then
    raise EHTMLDataSource.Create(tcTable,
            'Missing linked field name');
  if not LinkField.Visible then
    raise EHTMLDataSource.Create(tcTable,
            'Linked field is not visible');
  if LinkTarget = nil then
    raise EHTMLDataSource.Create(tcTable,
            'Missing link target field name');
end;

with DataSet do begin
  { Determine whether table has visible fields & records }
  iCount := 0;
  for i := 0 to FieldCount - 1 do
    if Fields[i].Visible then
      Inc(iCount);
  if iCount = 0 then
    raise EHTMLDataSource.Create(tcTable,
            'No fields in this dataset are visible');
  if RecordCount = 0 then
    raise EHTMLDataSource.Create(tcTable,
            'There are no records in this dataset');
```

```
{ Dump table to HTML }
with HTMLWriter do begin
  TableStartParams(Border, Width, CellSpacing,
    CellPadding, ColourBackground, ColourBorder,
    ColourBorderLight, ColourBorderDark, Caption,
    CaptionAlignHoriz, CaptionAlignVert);

     { Write headers }
  if Headers then
    begin
      TableRowStartParams(AlignHoriz, AlignVert,
        HeaderBackground,HeaderBorder,HeaderBorderLight,
        HeaderBorderDark);
      for i := 0 to FieldCount - 1 do
        if Fields[i].Visible then
          begin
            {$IFDEF WIN32}
            sCell := Fields[i].DisplayName;
            {$ELSE}

           sCell := Fields[i].DisplayName^;
            {$ENDIF}
            if UseFieldAlign then
             TableHeadingParams(FormatEscapeText(sCell),
               0,0,ahAlignments[Fields[i].Alignment],
               avDefault,clDefault,clDefault,clDefault,
               clDefault)
            else
             TableHeading(FormatEscapeText(sCell));
          end;
          TableRowEnd;
        end;

      { Don't update screen while processing }
      DisableControls;
      { Save data set position }
      bmkSave := GetBookmark;
      { Create temporary area for memo fields }
      slMemo := TStringList.Create;

      { Write contents of rows }
      try
        First;
        while not EOF do begin { Process all rows }
          { Check row alignment and colours }
          ahAlignHoriz    := ahDefault;
          avAlignVert     := avDefault;
          clrBackground   := clDefault;
          clrBorder       := clDefault;
          clrBorderLight  := clDefault;
          clrBorderDark   := clDefault;
          if Assigned(FOnRowShow) then
            OnRowShow(Self,ahAlignHoriz,avAlignVert,
              clrBackground,clrBorder,clrBorderLight,
              clrBorderDark);
          { And start the row }
          TableRowStartParams(ahAlignHoriz,avAlignVert,
            clrBackground,clrBorder,clrBorderLight,
            clrBorderDark);
          { Display each visible field }
          for i := 0 to FieldCount - 1 do
            if Fields[i].Visible then
              begin
                { Check column alignment and colours
                  - default to row values }
                if UseFieldAlign then
                  ahAlignHoriz :=
                    ahAlignments[Fields[i].Alignment]
                else
                ahAlignHoriz     := ahDefault;
                avAlignVert      := avDefault;
                clrBackground    := clDefault;
                clrBorder        := clDefault;
                clrBorderLight   := clDefault;
                clrBorderDark    := clDefault;

                if Assigned(FOnCellShow) then
                  OnCellShow(Self,Fields[i],ahAlignHoriz,
                    avAlignVert,clrBackground,clrBorder,
                    clrBorderLight, clrBorderDark);

                { And display the field }
                if Fields[i] is TMemoField then
                  { Add all the lines }
                  begin
                    slMemo.Assign(TMemoField(Fields[i]));
                    TableCellStartParams(0,0,0,
                    ahAlignHoriz, avAlignVert,
                    clrBackground,clrBorder,
                    clrBorderLight, clrBorderDark);
                    for iCount := 0 to slMemo.Count-1 do
                      EscapeText(slMemo[iCount] + ' ');
                    TableCellEnd;
                  end

                else { Add text representation of field }
                  begin
                    if Fields[i] = LinkField then
                      sCell :=
                        FormatLink(LinkTarget.AsString,
                        '',FormatEscapeText(
                        Fields[i].DisplayText))
                    else
                      sCell := FormatEscapeText(
                                Fields[i].DisplayText);
                    TableCellParams(sCell,0,0,0,
                    ahAlignHoriz,avAlignVert,
                    clrBackground,clrBorder,
                    clrBorderLight,clrBorderDark);
                  end;
              end;

            { Finish the row }
            TableRowEnd;
            Next;
          end;    { while }
        finally
          { Return to original position }
          GotoBookmark(bmkSave);
          FreeBookmark(bmkSave);
          { Update screen again }
          EnableControls;
          { Finish off the HTML table }
          TableEnd;
          { Release string list resources }
          slMemo.Free;
        end;    { try..finally }

    end;    { if Headers then... }
  end;    { with HTMLWriter do begin... }
end;    { with DataSet do begin... }
```

### End Listing One



*By Robert Vivrette*

# Fingerpainting

## Building a Custom Color Property Editor

**Remember fingerpainting as a child? Sticking your fingers in four or five jars of paint and then smearing your hands around to create countless new colors. Remember the feeling? It was cool!**

Well, it still feels cool, but the tools have changed. Instead of your fingers, you now have the Microsoft Windows color system. Granted, the system isn't perfectly designed, but it does allow you to fiddle with colors and come up with new ones.

In this article, we'll investigate how colors are represented in Delphi (and Windows), and how you can manipulate them to your advantage. At the end of the article, we'll cover how to integrate your custom colors directly into the Delphi IDE with the ColorEdt property editor.

### How Much Money Is $00FF80FF?

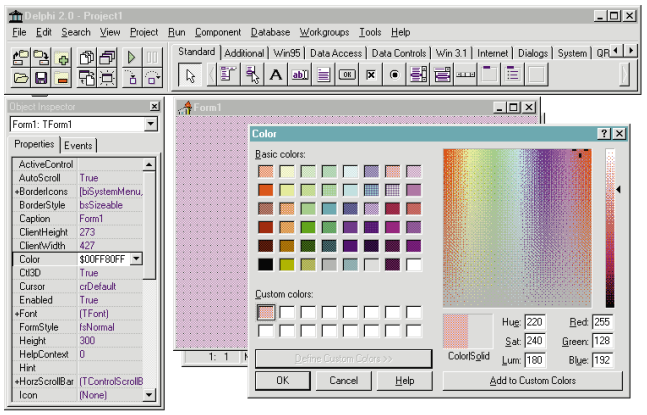
What happens when you select a color for a component, but the color is not one of the pre-defined ones? (By pre-defined, I mean of course, colors such as *clYellow*, *clSilver*, and *clLime*.) Let's say you want to change a form's color via the Object Inspector. You double-click on the value for the form's *Color* property to display the Color dialog box and see that some of the available colors don't match the standard set of colors in Windows. If you select one of these "non-standard" colors, you might expect the Object Inspector to display a descriptive word for your selection (e.g. *clPaleYellow* or *clUglyPink*). Instead, the Object Inspector displays something like `$00FF80FF` (see Figure 1). What the heck is this value? And what good is it to you?

To understand a bit more, we need to see the format of the type *TColor* (the base type hiding behind *Color* properties). Searching on "TColor | TColor Type" in Delphi's online Help reveals that *TColor* is declared as:

```
TColor = -(COLOR_ENDCOLORS + 1)..$02FFFFFF;
```

This type declaration defines a range of colors. The beginning (negative) values are used to represent the System colors (such as *clBtnFace* and *clWindow*). The positive values denote literal colors that the Windows GDI can represent.

If you continue reading online Help's description of the *TColor* type, you'll see that



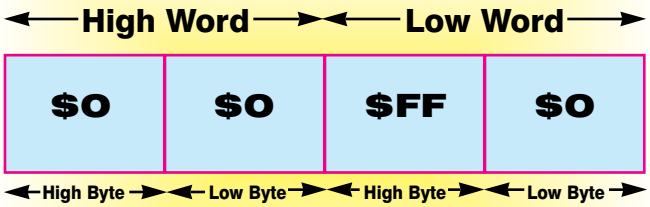



**Figure 1 (Top):** When a custom color is selected from the Windows Color dialog box, the Object Inspector displays a cryptic hexadecimal value.
**Figure 2 (Bottom):** The layout of the *TColor* type.

Delphi defines *TColor* as a four-byte hexadecimal number. This four-byte value is then disseminated into each of its component bytes. Again, from online Help:

> If you specify TColor as a specific 4-byte hexadecimal number instead of using the constants defined in the Graphics unit, the low three bytes represent RGB color intensities for blue, green, and red, respectively. The value $00FF0000 represents full-intensity, pure blue, $0000FF00 is pure green, and $000000FF is pure red. $00000000 is black and $00FFFFFF is white.

Now we have a little more information. According to this definition, the organization of *TColor* appears as shown in Figure 2. For each of these words, a high and low word (two bytes each) and high and low bytes (one byte) exist. The value diagrammed in Figure 2 would be $0000FF00, equating to solid green. The high byte of the high word is reserved for Delphi to specify the system-wide colors. For the purposes of this article, we're only interested in the low three bytes.

## Stop and Think

Let's think about this a bit. Suppose we don't want solid green, but something that's only about 80 percent green. It's simple to achieve. As mentioned, the low three bytes in *TColor* represent the amount of red, green, and blue in the color. Since each of these colors is represented by a single byte, we see that the allowable range for each of the red, green, and blue components of a *TColor* can be in the range of 0 to 255 (i.e. the range for a byte).

```
unit Coloradj;

interface

uses
  WinProcs,Graphics;

  function AdjustColor(A: TColor; Factor: Real): TColor;

implementation

function AdjustColor(A: TColor; Factor: Real): TColor;
var
  R,G,B : Byte;
begin
  Result := A;
  R := Round(GetRValue(ColorToRGB(A))*Factor);
  G := Round(GetGValue(ColorToRGB(A))*Factor);
  B := Round(GetBValue(ColorToRGB(A))*Factor);
  Result := RGB(R,G,B);
end;

end.
```

**Figure 3:** The *AdjustColor* function.

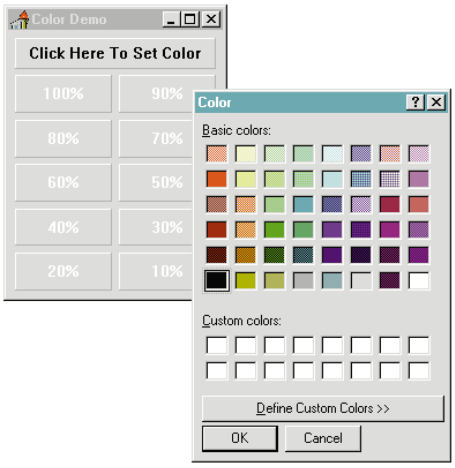



**Figure 4:** Clicking on Color Demo's top panel displays the Color dialog box.

If solid green was represented as 0 for red, 255 for green, and 0 for blue, then it should be easy to reduce the green value of 255 by 20 percent to get our 80 percent green. So, that's exactly what we'll do!

Figure 3 is the *AdjustColor* function. It accepts a color and a percentage, and returns a new color. The new color is formed by breaking the passed-in color into its component RGB values, which are then multiplied by the percentage supplied, and then recombined as a *TColor* for the function result.

The *ColorToRGB* function takes the passed-in *TColor* value and converts it to an RGB value. In most cases, the *TColor* is exactly equal to the RGB value. Yet when the *TColor* is holding a System color (such as *clBtnFace* or *clWindow*), it's first converted into the appropriate RGB value.

After obtaining an RGB value with *ColorToRGB*, the *GetRValue*, *GetGValue*, and *GetBValue* functions are used to break out the individual amounts of red, green, and blue. The results are then multiplied by the percentage passed in and assigned to the *R*, *G*, and *B* variables. Lastly,

these three variables are passed in to the Windows API function, *RGB*, that combines them back into a composite *RGB* (or *TColor*) value.

## Into Action

Let's put this function into action with two examples. *AdjustColor* has been saved as a unit named COLORADJ.PAS, and this unit has been placed into the **uses** clause for each of the sample programs we'll discuss.

Figure 4 shows the simple Color Demo program (DEMO1.DPR) which contains a number of panels. Clicking on the top panel (**Click Here To Set Color**) displays the common Color dialog box. After selecting a color, you can click the **OK** button to close the Color dialog box. The Color Demo application then "dyes" each of the 10 panels on the form with varying shades of the chosen color (see Figure 5).

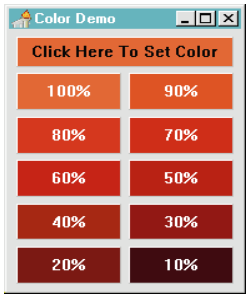



**Figure 5:** After you've selected a color, each panel in the Color Demo application displays a shade 10 percent lighter than the previous one.

The effect you see on-screen will be governed primarily by the color depth that your video card is configured to display. If you are viewing 16 or 256 colors, the panels will be displayed using dithered colors. This is Windows' way of simulating another color by using only a limited set of primary colors. If your system is displaying 16 million colors or higher, then you'll see perfectly solid color representations for each panel. The 16-million color depth can display these colors accurately because it's using exactly the specified values of red, green, and blue. The 16 and 256 color depths must map the selected RGB values to the closest one available in either the current Color palette or the System palette. Figure 6 shows the source code for the Color Demo application.

The second demonstration program (DEMO2.DPR) shows how you can put gradient fill patterns on your forms. Implementing a simple routine in a form's *FormPaint* event handler will allow you to control how any form is displayed. In this example, a gradient fill pattern is drawn using our *AdjustColor* function.

This technique is applied to a simple dialog box (see Figure 7). Figure 8 is the code behind this sample application. All that is done here is to repeatedly draw small rectangles across the panel's surface. Each rectangle is drawn in a slightly different shade than the last. The result is a simple, effective gradient fill across the form's surface.

## The ColorEdt Property Editor

Now that we know more about colors and how they're represented, we'll create a property editor to add custom colors to the Delphi IDE.

```
unit Demo1u;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, ColorAdj, ExtCtrls;

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    Panel2: TPanel;
    Panel3: TPanel;
    Panel4: TPanel;
    Panel5: TPanel;
    Panel6: TPanel;
    Panel7: TPanel;
    Panel8: TPanel;
    Panel9: TPanel;
    Panel10: TPanel;
    MainPanel: TPanel;
    ColorDialog1: TColorDialog;
    procedure MainPanelClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }

  end;
var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.MainPanelClick(Sender: TObject);
begin
  if ColorDialog1.Execute then
    with ColorDialog1 do begin
      MainPanel.Color := Color;
      Panel1.Color    := AdjustColor(Color,1.0);
      Panel2.Color    := AdjustColor(Color,0.9);
      Panel3.Color    := AdjustColor(Color,0.8);
      Panel4.Color    := AdjustColor(Color,0.7);
      Panel5.Color    := AdjustColor(Color,0.6);
      Panel6.Color    := AdjustColor(Color,0.5);
      Panel7.Color    := AdjustColor(Color,0.4);
      Panel8.Color    := AdjustColor(Color,0.3);
      Panel9.Color    := AdjustColor(Color,0.2);
      Panel10.Color   := AdjustColor(Color,0.1);
    end;
end;

end.
```

**Figure 6:** A sample program showing colors being modified with the *AdjustColor* function.


**Figure 7:** Using *AdjustColor* to do gradient backgrounds.

Whenever you display the *Color* property's drop-down list in the Object Inspector, Delphi uses a property editor that populates this list with its currently-defined *TColor* constants. To integrate custom colors into the Object Inspector, it's a simple matter of creating a new property editor for *TColor* values and extending it to handle custom values. This new property editor actually descends from the existing *Color* property editor so that we gain all the existing functionality Delphi provides.

```
unit Demo2u;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls,
  Buttons, ExtCtrls, ColorAdj;

type
  TForm1 = class(TForm)
    Label1: TLabel;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    procedure FormPaint(Sender: TObject);
    procedure BitBtn1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormPaint(Sender: TObject);
var
  a : Integer;
begin
  with Inherited Canvas do
    begin
      Pen.Style := psClear;
      for a := 0 to ClientHeight div 2 do
        begin
          Brush.Color :=
            AdjustColor(clLime,a*2/ClientHeight);
          Rectangle(0,ClientHeight-a*2,
                    ClientWidth+1,ClientHeight-a*2+3);
        end;
    end;
end;

procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  Close;
end;

end.
```

**Figure 8:** A sample program showing how to do gradient fills.

To build the color property editor, we must first create the interface that allows us to select the colors (see Figure 9). This is often the best approach for creating a dialog box-based property editor such as this. After creating a stand-alone application, you'll find that converting it to a property editor is fairly simple.

A name for the color the user has specified will appear in the edit field at the top of the form. The panel below it will show the currently-selected color, and the list box on the right will show the custom colors that have already been defined.

## Friendly, Friendly, Friendly

Some selected enhancements make the property editor as user-friendly as possible. The list box on the right is of an owner-

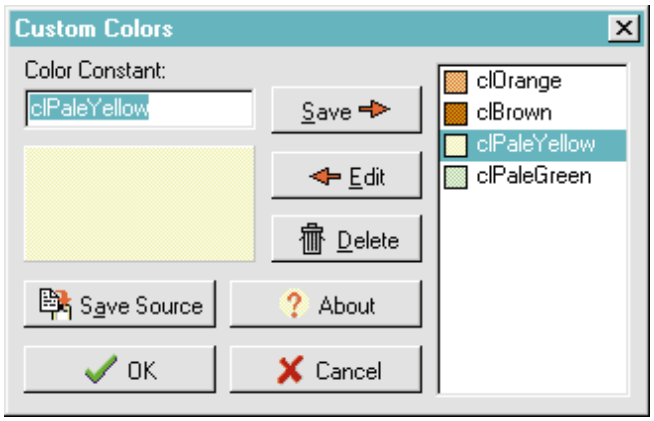



**Figure 9:** The interface for the ColorEdt property editor.

draw variety, allowing us to include small swatches of the colors. This will help a user visually associate the custom colors with their constants. Clicking on the color panel activates a *TColorDialog* (provided in Delphi) that allows the user to select and mix custom colors. After selecting a color in this way, a user can type in a color constant and click on the **Save** button to add the new color to the list of those currently defined.

Although the property editor completely integrates custom colors into the Delphi IDE, it must perform one additional task to be a complete solution. To allow a user to specify any of the constants within a piece of Delphi source code (e.g. a unit or project file) the compiler must know how these constants are defined. As a result, the property editor includes a button (named **Save Source**) that writes out the currently-defined custom color constants as a source code unit. Then, by adding this unit to any project, a user can continue to use the custom colors even outside the IDE.

The interface to the property editor is straightforward. Two principal ways of activating it are to enter a question mark ( `?` ) instead of a color name. The second is to type any color constant into the Object Inspector that has not already been defined. For the value of a form's *Color* property, for example, you could type in `clPeriwinkle`. This color is not one of the standard color constants, and it wasn't already specified as a custom color. Therefore, the property editor presents our dialog box to allow the user to pick the color. To enforce the naming scheme that Borland has established, a color constant is only recognized as such if the first two characters of the name are *cl*.

When the custom property editor is displayed, the user can simply click on the panel under the constant name. This displays the common Color dialog box provided by Windows. The user can then select one of the basic colors shown, or create a new color by clicking the **Define Custom Colors** button. After closing this dialog box, the panel that had been clicked will be the selected color. By clicking the **Save** button, the property editor adds that color constant definition into its internal array (which is then displayed by means of the list box). Clicking the

OK button closes the property editor and populates the original *Color* property with the selected color constant.

Now, if you return to Delphi's IDE and click on the *Color* property in the Object Inspector, the drop-down list displays the custom colors that were defined (see Figure 10). These custom colors are inserted at the top of the list above the already-defined color constants.

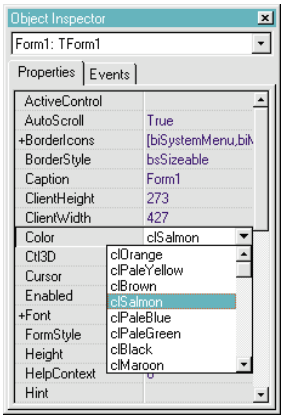



**Figure 10:** The Object Inspector displaying the custom colors in the *Color* property's drop-down list.

## Examining the ColorEdt Property Editor

Let's take a closer look at the ColorEdt property editor. The first place you can find information on creating property editors is in the DSGNINTF.PAS file that ships with Delphi. DSGNINTF.PAS is the principal unit of Object Pascal source code that controls all the behavior for property editors, from the simplest to the most complex. The organization of Delphi's property editors is much like the hierarchy of its VCL; very simple property editors exist, from which the more complex ones descend.

All we do for the ColorEdt property editor is descend from the already existing *TColorProperty*. Here is the declaration section of ColorEdt:

```
TRVColorProperty = class(TColorProperty)
  public
    procedure GetValues(Proc: TGetStrProc); override;
    function  GetValue: string; override;
    procedure SetValue(const Value: string); override;
  end;
```

You can see that the property class we are deriving is named *TRVColorProperty* and that it descends from *TColorProperty*. There are only three areas where we alter *TColorProperty*: the *GetValues* and *SetValue* procedures, and the *GetValue* function.

*GetValues* is a procedure that the Object Inspector uses whenever it must have a drop-down list property filled with values. That is exactly the type of property we have here. First, let's examine how we modify the behavior of *GetValues*:

```
procedure TRVColorProperty.GetValues(Proc: TGetStrProc);
var
  I : Integer;
begin
  for I := 0 to RVColorsList.Count-1 do
  Proc(FirstHalf(RVColorsList[I]));
  inherited GetValues(Proc);
end;
```

Within the ColorEdt property editor, a string list is configured to hold the custom color constants that have been defined. Each of the entries defines a constant name (e.g. *clPaleBlue*), followed by an equal sign ( = ) and the hexadecimal number assigned to that label. For example:

```
clPaleBlue=$00FFFF80
```

Notice that *GetValues* expects a passed-in parameter of *TGetStrProc*. The Object Inspector is passing in a pointer to a procedure that it's using to populate the list box. All that needs to be done is to call the passed-in procedure once for each of the custom colors in our list. The code simply runs through *RVColorsList* and passes the left side of the string (everything up to, but not including, the equal sign) to the passed-in procedure. To simplify the appearance of this code, the *FirstHalf* function was created. It simply accepts a color constant item as above, and strips the color name on the left side of the equal sign.

If that was all that was done, we would see only our custom colors. Because we have overridden the *GetValues* procedure, we need to duplicate the original work necessary to put in all of Delphi's standard color values. In this case, it's a simple matter of calling the inherited *GetValues* procedure from *TColorProperty*.

Next, the *GetValue* function was altered. This function, although named similarly to the procedure *GetValues*, serves a very different purpose. Whenever Delphi has a *TColor* value that it needs to convert to a color name, it uses *GetValues*. Because the *GetValue* function that came with *TColorProperty* is not aware of our custom color scheme, we must make some modifications here as well:

```
function TRVColorProperty.GetValue: string;
var
  I : Integer;
begin
  if not ColorToIdent(GetOrdValue, Result) then
    begin
      for I := 0 to RVColorsList.Count-1 do
        if StrToInt(SecondHalf(RVColorsList[I]))=
           GetOrdValue then
          begin
            Result := FirstHalf(RVColorsList[I]);
            Exit;
          end;
      FmtStr(Result, '$%.8x', [GetOrdValue]);
    end;
end;
```

When Delphi has a *TColor* value that must be converted, it calls this function. First, the program calls Delphi's *ColorToIdent* function that looks up *TColor* values that Borland already defined in the Graphics unit. The *GetOrdValue* function is declared in the DsgnIntf unit and returns the numeric representation of a property. If *ColorToIdent* finds the color constant (i.e. if it's one of the stock colors) then *GetValue* returns the string representation of the color in *Result*. If it's not found, however, we need to look through our string list (*RVColorsList*) to see if it's a custom color. If it's not found in that string list, then we need to take the numeric value of the color and format it into hexadecimal notation (just as Delphi always does with unrecognized colors). All that was changed here is the section that searches *RVColorsList*. The rest is the code in *TColorProperty*.

```
procedure TRVColorProperty.SetValue(const Value: string);
var
  NewValue      : Longint;
  CurrentValue  : string;
  RVColorDialog : TRVColorDialog;
begin
  if IdentToColor(Value, NewValue) then
    SetOrdValue(NewValue)
  else
    if Value <> '' then
      begin
        CurrentValue := RVColorsList.Values[Value];
        if CurrentValue <> '' then
          SetOrdValue(StrToInt(CurrentValue))
        else
          if (UpperCase(Copy(Value,1,2)) = 'CL') or
             (Value[1]='?') then
            begin
              RVColorDialog := TRVColorDialog.Create(Application);
              try
                with RVColorDialog do begin
                  if Value[1] <> '?' then
                    edtColorConstant.Text := Value
                  else
                    edtColorConstant.Text := '';
                  pnlColor.Color := clWhite;
                  if ShowModal = mrOK then
                    SetOrdValue(pnlColor.Color);
                end;
              finally
                RVColorDialog.Free;
              end;
            end
          else
            inherited SetValue(Value);
      end
    else
      inherited SetValue(Value);
end;
```

**Figure 11:** The *SetValue* procedure.

Now we'll discuss the last of the three modifications: the procedure *SetValue*, the exact opposite of *GetValue*. It takes a color label (such as *clOrange*) and converts it into the color constant that is defined in Figure 11.

*SetValue* is just a little more complicated than *GetValue*. First, we see if the color label is one that is already defined by Delphi. The *IdentToColor* function is used to make this determination. If so, *SetValue* ends and uses *SetOrdValue* to pass the numeric value of the color back to the Object Inspector.

If it's not a standard color, then we again need to look into the *RVColorsList* string list to see if the color label is defined as one of our custom colors. Again, if it's found, the procedure exits and returns the selected value back to the Object Inspector by means of the *SetOrdValue* procedure. However, if it isn't, we need to check whether the user is typing in a new custom color, or wants to see the ColorEdt dialog box.

As mentioned before, we had established that if a string value was entered and wasn't recognized, but began with the letters *cl*, then the property editor dialog box is displayed. In addition, if the string value entered was a question mark, the dialog box is also displayed. If the user

```
procedure LoadCustomColorsFromIni;
begin
  RVColorsList.Clear;
  ColorINI := TIniFile.Create('RVCOLORS.INI');
  ColorINI.ReadSectionValues('Custom Colors',RVColorsList);
  ColorINI.Free;
end;

procedure SaveCustomColorsToIni;
var
  I : Integer;
begin
  ColorINI := TIniFile.Create('RVCOLORS.INI');
  ColorINI.EraseSection('Custom Colors');
  for I := 0 to RVColorsList.Count-1 do
    ColorINI.WriteString('Custom Colors',
                         FirstHalf(RVColorsList[I]),
                         SecondHalf(RVColorsList[I]));
  ColorINI.Free;
end;

initialization
  RVColorsList := TStringList.Create;
  LoadCustomColorsFromIni;

finalization
  SaveCustomColorsToIni;
  RVColorsList.Free;
```

**Figure 12:** The final pieces of code in the COLOREDT.PAS unit.

closes the property editor dialog box by clicking **OK**, then the color that was selected is used by *SetValue* and is passed back to the Object Inspector.

Another interesting behavior of ColorEdt is that it does not interfere with the normal behavior of the *Color* property. If you double-click on the *Color* property, you still get Delphi's Color dialog box.

## Finishing Touches

Finally, we had to instruct the property editor to save these custom colors somewhere, and to restore them each time Delphi was restarted. The simplest way was to use an .INI file. At the bottom of the COLOREDT.PAS file we see the code shown in Figure 12.

The **initialization** and **finalization** sections of the code may be new to some of you. The code in the **initialization** section of a unit is executed once when the application starts. The code in the **finalization** section of a unit is likewise executed once, when the application closes.

These two sections are used to load and save the custom colors to the .INI file. When the program starts, the string list *RVColorsList* is created. Then the *LoadCustomColorsFromIni* procedure is called to populate this string list. When the application shuts down, the *SaveCustomColorsToIni* procedure writes out the custom colors and disposes of the *RVColorsList* string list.

## Conclusion

Because of space limitations, a discussion of how the ColorEdt property editor dialog box functions is beyond the scope of this article. However, all of its features are fairly

straightforward and are covered in the accompanying source code. You may want to examine some of the more interesting aspects of this dialog box, e.g. how the owner-draw list box is built.

The simple function *AdjustColor* shows an easy way of scaling color brightness, but the principles involved can easily be extended into additional types of color manipulation. For example, you could provide a function that decreases the blue component, while increasing the green component.

These techniques of manipulating Delphi's color constants demonstrate more of Delphi's easy-to-implement graphics capabilities. Just like fingerpainting, the possibilities are limited only by your artistic flair. Δ

*The demonstration files referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\NOV\DI9611RV.*

Robert Vivrette is a Senior Programmer/Analyst for Pacific Gas & Electric and Technical Editor for *Delphi Informant*. He is also author of a free, online journal, The Unofficial Newsletter of Delphi Users, that can be found at http://www.informant.com/undu. He can be reached on CompuServe at 76416,1373 (on the Internet, use 76416.1373@compuserve.com).



*By Ray Lischner*

# Subproperty Editors

## Undocumented Tricks for Creating Surrogate Components

**You're finishing a nifty new calendar component. All you need to complete your finest work is a property editor that lets the user choose the date the calendar displays. You consult the Component Writer's Guide and find the *paSubProperties* flag. Great! It's just what you want. The user can double-click the *Date* property to expose the *Day*, *Month*, and *Year* properties.**

But, how do you create the subproperty editors? The short answer is ... you can't. The only property editors that can have subproperties are *TClassProperty* and *TSetProperty* (and their subclasses). If you want to define subproperties for any other type of property, such as *TDateTime*, you're out of luck.

Or are you? Through a little clever programming, you can create fake subproperties. This article instructs you how to create subproperty editors for any type of property.

### Subproperty Review

Let's quickly review properties and subproperties. The Object Inspector displays the published properties of a component. When you write a custom component, you can also supply property editors to make it easier to use your component at design time. You can also write property editors for any of the standard components, and you can replace standard property editors with your own.

Every property has a type — specifically, an integer, floating point (except *Real*), enumerated, set, character, class, method, or string. In Delphi 2, you can also have *Variant* or *WideChar* properties. Each property has a property editor, which is an instance of the class *TPropertyEditor*, or one of its subclasses. Delphi defines a property editor class for every type, and you can register a property editor class for a specific property, or any property of a particular type.

Properties can in turn have their own properties, just as components have properties. A property of a property is called a *subproperty*, as illustrated in Figure 1. In the pre-defined property editors, the only uses for subproperties are in set and class properties. A set's subproperties are the set elements, which use the *TSetElementProperty* editor. The subproperties of a class property are the published properties of the class.

Figure 2 illustrates class and set subproperties for the *TFont* class and the *TFontStyles* set type. There are other uses for subproperties, though. For example, your new calendar component would be easier to use if the *Date* property had subproperties for its constituent *Day*, *Month*, and *Year*. This is a natural use for subproperties. Unfortunately, Delphi hides the pieces necessary to define your own subproperties.

### Why Subproperties Are Hard to Define

It's instructive to see how Delphi hides the

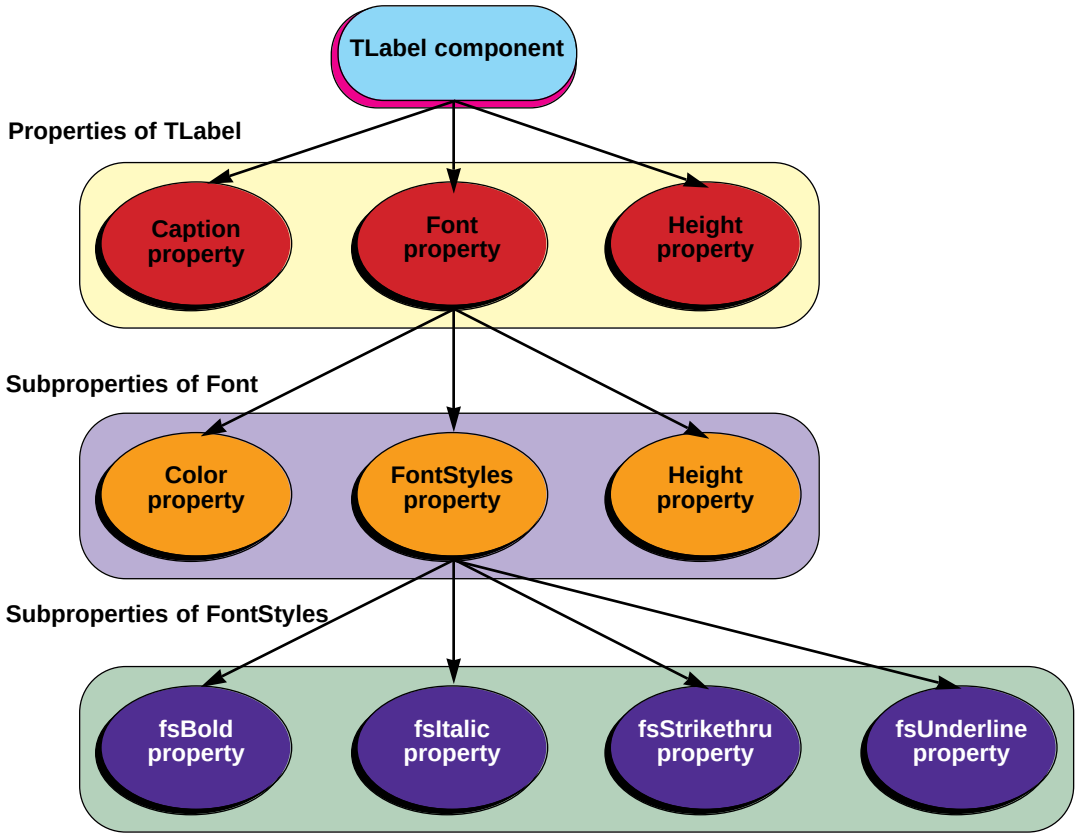



**Figure 1:** Properties with subproperties.

subproperty feature, if only as a lesson in how *not* to design object-oriented software. When the *paSubProperty* attribute is set, the Object Inspector calls the *GetProperty* routine. The property editor overrides this method to create a property editor for each subproperty. There's the rub. You can't create an instance of a property editor.

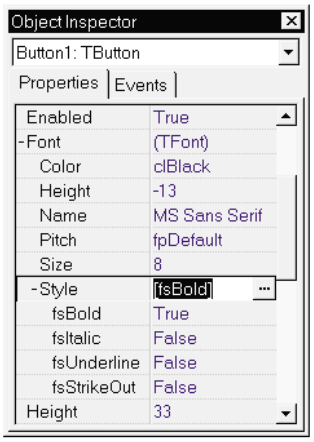



**Figure 2:** The Object Inspector displaying subproperties.

It turns out the base class *TPropertyEditor* declares its *Create* constructor in its **private** section, not **protected**. This means you can never create a valid instance of any class derived from *TPropertyEditor*. Sure, you could call *TObject.Create* to create an instance, but it would not be valid. Only *TPropertyEditor* can set the *Designer* property, which it does in its constructor.

So how does the Object Inspector create its property editors? The same way you do — by calling *GetComponentProperties*. This routine, shown in Figure 3, is the only way to create a property editor object. The only problem is that it creates a property editor for every property of a component. This is fine for the Object Inspector, but a subproperty is different. It is not a real property of a component, so *GetComponentProperties* cannot create a property editor for a subproperty.

Object Pascal honors **private** declarations in units that use the unit where the class is declared. Any code in the same unit as *TPropertyEditor* (such as *TSetProperty* and *TClassProperty*) can refer to the **private** fields of *TPropertyEditor*. This allows developers to create mutually cooperating classes, such as the property editors, but it also allows sloppy programming, such as making the constructor **private**, not **protected**.

## Use a Surrogate Component

Now you know what you can't do. So what *can* you do? You can derive a class from *TSetProperty* or *TClassProperty*, and let the parent class handle the subproperties for you. If you want to define subproperties for a type other than a set or class type, then you need to fake out Delphi. You can create a property editor that looks like it has subproperties to the user. The way to accomplish this trick is to define a hidden component whose published properties are the desired subproperties.

Let's look at how to define a surrogate component for *TDateTime* that would allow users to set the parts of a date and time in the Object Inspector, while still allowing for convenient use in the component's code. This example creates a subproperty for all the constituent parts of the *TDateTime* class. It's a simple exercise to create a property editor for just

```
procedure GetComponentProperties(
  Components: TComponentList; Filter: TTypeKinds;
  Designer: TFormDesigner; Proc: TGetPropEditProc);
```

```
type
  { Define a unique type identifier for each constituent
    part. The default property editors limit the input to
    the specified range. You can also supply custom
    property editors, like TS_PropMonthProperty. }
  TS_PropDateTimeDay      = 0..31;
  TS_PropDateTimeMonth    = 0..12;
  TS_PropDateTimeYear     = Integer;
  TS_PropDateTimeHour     = 0..23;
  TS_PropDateTimeMinute   = 0..59;
  TS_PropDateTimeSecond   = 0..59;
  TS_PropDateTimeMilliSec = 0..999;

  { Pseudocomponent that is used to create subproperty
    editors for the constituent parts of a TDateTime. }
  TS_PropDateTime = class(TComponent)
  private
    { Cache the Date/Time property value. }
    fDateTime: TDateTime;
    { Pointer back to the property editor. }
    fEditor: TS_DateTimeProperty;
    function GetDay: TS_PropDateTimeDay;
    function GetMonth: TS_PropDateTimeMonth;
    function GetYear: TS_PropDateTimeYear;
    function GetHour: TS_PropDateTimeHour;
    function GetMinute: TS_PropDateTimeMinute;
    function GetSecond: TS_PropDateTimeSecond;
    function GetMilliSec: TS_PropDateTimeMilliSec;
    procedure SetDay(Value: TS_PropDateTimeDay);
    procedure SetMonth(Value: TS_PropDateTimeMonth);
    procedure SetYear(Value: TS_PropDateTimeYear);
    procedure SetHour(Value: TS_PropDateTimeHour);
    procedure SetMinute(Value: TS_PropDateTimeMinute);
    procedure SetSecond(Value: TS_PropDateTimeSecond);
    procedure SetMilliSec(Value: TS_PropDateTimeMilliSec);
    procedure SetDateTime(Value: TDateTime);
  public
    property DateTime: TDateTime read fDateTime
                                 write SetDateTime;
    property Editor: TS_DateTimeProperty read fEditor
                                         write fEditor;
  published
    { Declare the properties in the order they should be
      shown to the user. TS_DateTimeProperty preserves the
      declaration order. }
  property Year: TS_PropDateTimeYear read GetYear
                                     write SetYear;
  property Month: TS_PropDateTimeMonth read GetMonth
                                       write SetMonth;
  property Day: TS_PropDateTimeDay read GetDay
                                   write SetDay;
  property Hour: TS_PropDateTimeHour read GetHour
                                     write SetHour;
  property Minute: TS_PropDateTimeMinute read GetMinute
                                         write SetMinute;
  property Second: TS_PropDateTimeSecond read GetSecond
                                         write SetSecond;
  property MilliSec: TS_PropDateTimeMilliSec
              read GetMilliSec write SetMilliSec;
end;
```

**Figure 3 (Top):** Declaration of *GetComponentProperties*.
**Figure 4 (Bottom):** Declaration for the surrogate date and time component.

the date or time part of *TDateTime*. This is left as an exercise for the reader.

```
{ Return the year part of the DateTime. }
function TS_PropDateTime.GetYear: TS_PropDateTimeYear;
var
  Year, Month, Day: Word;
begin
  DecodeDate(DateTime, Year, Month, Day);
  Result := Year;
end;

{ Return the hour part of the DateTime. }
function TS_PropDateTime.GetHour: TS_PropDateTimeHour;
var
  Hour, Minute, Second, MilliSec: Word;
begin
  DecodeTime(DateTime, Hour, Minute, Second, MilliSec);
  Result := Hour;
end;

{ Set the year part of the DateTime. }
procedure TS_PropDateTime.SetYear(
  Value: TS_PropDateTimeYear);
var
  Year, Month, Day: Word;
begin
  DecodeDate(DateTime, Year, Month, Day);
  DateTime := EncodeDate(Value, Month, Day) +
              Frac(DateTime);
end;

{ Set the hour part of the DateTime. }
procedure TS_PropDateTime.SetHour(
  Value: TS_PropDateTimeHour);
var
  Hour, Minute, Second, MilliSec: Word;
begin
  DecodeTime(DateTime, Hour, Minute, Second, MilliSec);
  DateTime := EncodeTime(Value, Minute, Second, MilliSec) +
              Int(DateTime);
end;
```

**Figure 5:** Getting and setting part of a date and time.

First, you need to define a component as shown in Figure 4. The component has published properties for the constituent parts of a date and time. Only the property editor creates an instance of the component class, so no one ever places this component on a form. The *TS_PropDateTime* class is a trick class, useful only when implementing a property editor.

You must declare the *TS_PropDateTime* types, because a property's type must be declared with an identifier. General type specifications are not allowed by Object Pascal. In this situation, it's helpful to use unwieldy names that won't clash with names in other units.

The definition of these methods is straightforward, using Delphi's encode and decode routines. Just be sure that when you set part of the date and time, you don't accidentally alter the other parts. Because all the routines are similar, only some are shown in Figure 5. The source code that accompanies this article contains the full class definition. When rebuilding the date and time, remember that the date is the integer part, so the encoded time is added to `Int(DateTime)`, and the encoded date is added to `Frac(DateTime)`.

When the date and time are set, the property editor must be informed of the new value. This happens in the *SetDateTime*

```
{ When the user changes a constituent part of the DateTime,
  notify the master property editor, so it can update the
  property value string. As per standard property editor
  usage, changing a property value sets the value for all
  selected property editors. }
procedure TS_PropDateTime.SetDateTime(Value: TDateTime);
begin
  if fDateTime <> Value then
    begin
      fDateTime := Value;
      if Editor <> nil then
        Editor.SetDateTime(Value);
    end;
end;
```

**Figure 6:** Setting the date and time and updating the property editor.

```
type
{ The property editor for TDateTime. It creates a hidden
  object and property editor just to access the subproperty
  editors of the hidden property editors, pretending that
  they are subproperties of TS_DateTimeProperty. }
  TS_DateTimeProperty = class(TFloatProperty)
  private
    SubProps: TList;    { Sorted list of subproperties. }
    ChildList: TComponentList;  { List of hidden objects. }
    procedure GetSubProps(PropEdit: TPropertyEditor);
    procedure SetChildValues;
  public
    destructor Destroy; override;
    function GetAttributes: TPropertyAttributes; override;
    function GetValue: string; override;
    procedure SetValue(const Value: string); override;
    procedure GetProperties(
      Proc: TGetPropEditProc); override;
    procedure SetDateTime(Value: TDateTime);
  end;
```

**Figure 7:** Declaration of *TS_DateTimeProperty*.

procedure (see Figure 6). The property editor creates an instance of the surrogate component, and calls *GetComponentProperties* to retrieve the component's property editors as though they were subproperty editors. Because the user can select multiple components, the property editor must create a *TS_PropDateTime* component for each component selected by the user.

Figure 7 shows the *TS_DateTimeProperty* editor. The list of surrogate components is stored in the *ChildList* field.

## Get Set

Implementing the new *GetValue* and *SetValue* functions is easy, as shown in Figure 8. A zero date time is displayed as an empty string, because it's not a valid *TDateTime* value. Similarly, an empty string is stored as a zero *TDateTime* value. You must also override *GetAttributes* to set the *paSubProperties* attribute.

When the *TDateTime* value changes in the property editor, the *SetValue* method calls *SetChildValues*. This notifies the surrogate components of the new value. When the property editor is destroyed, the list of child components must also be freed. You have already seen that when any child component's value changes, the component notifies

```
{ Return the property editor attributes, including
  paSubProperties. }
function TS_DateTimeProperty.GetAttributes :
  TPropertyAttributes;
begin
  Result := inherited GetAttributes + [paSubProperties];
end;

{ The value 0.0 is not a valid TDateTime (month=0, day=0),
  so just use an empty string. }
function TS_DateTimeProperty.GetValue: string;
begin
  if GetFloatValue = 0.0 then
    Result := ''
  else
    Result := DateTimeToStr(GetFloatValue)
end;

{ Set the date/time value from the string Value. }
procedure TS_DateTimeProperty.SetValue(
  const Value: string);
begin
  if Value = '' then
    SetFloatValue(0.0)
  else
    SetFloatValue(StrToDateTime(Value));
  SetChildValues;
end;
```

**Figure 8:** Setting and getting the value of *TS_DateTimeProperty*.

```
{ Destroy the property editor and free its list
  of hidden components. }
destructor TS_DateTimeProperty.Destroy;
begin
  ChildList.Free;
  inherited Destroy;
end;

{ Update all the hidden, child property editors, if any. }
procedure TS_DateTimeProperty.SetChildValues;
var
  I: Integer;
begin
  if ChildList <> nil then
    for I := 0 to ChildList.Count-1 do
      TS_PropDateTime(ChildList[I]).DateTime :=
        GetFloatValueAt(I);
end;

{ When Date/Time changes, notify all hidden components. }
procedure TS_DateTimeProperty.SetDateTime(Value: TDateTime);
begin
  SetFloatValue(Value);
  SetChildValues;
end;
```

**Figure 9:** Using the list of child components for *TS_DateTimeProperty*.

the property editor by calling the *SetDateTime* method. This method, in turn, propagates the change to the other child components. These methods are shown in Figure 9.

The hard part is having the Object Inspector request the subproperties by calling the *GetProperties* method. If the user has edited the subproperties earlier, the list of surrogate components already exists. If it doesn't exist, it's created. Because the user can select multiple components, each with a different *TDateTime* value, every *TS_PropDateTime*

```
{ When the Object Inspector requests the subproperties, it
  is time for TS_DateTimeProperty to do its thing. Create a
  hidden TS_PropDateTime object to parallel each component
  that is currently selected. Then request the property
  editors for the hidden TS_PropDateTime objects. Unfortu-
  nately, GetComponentProperties sorts the properties
  alphabetically, which makes the subproperties more dif-
  ficult to use. Instead, use the property's run-time type
  info (TPropInfo) to get its NameIndex. The NameIndex
  gives the order in which properties are declared, which
  is the order in which these particular properties should
  be shown to the user. }
procedure TS_DateTimeProperty.GetProperties(Proc:
TGetPropEditProc);
var
  PropDateTime: TS_PropDateTime;
  I: Integer;
begin
  if ChildList <> nil then
    SetChildValues
  else
    begin
      ChildList := TComponentList.Create;
      for I := 0 to PropCount-1 do begin
        PropDateTime := TS_PropDateTime.Create;
        PropDateTime.DateTime := GetFloatValueAt(I);
        PropDateTime.Editor := Self;
        ChildList.Add(PropDateTime);
      end;
    end;

  SubProps := TList.Create;
  try
    { Build a list of subproperty editors,
      in declaration order. }
    GetComponentProperties(ChildList, [tkInteger],
                           Designer, GetSubProps);
    for I := 0 to SubProps.Count-1 do
      { The subproperty editor list has holes for
        Name and Tag. Skip them. }
      if SubProps[I] <> nil then
        Proc(TPropertyEditor(SubProps[I]));
  finally
    SubProps.Free;
    SubProps := nil;
  end;
end;

{ Discard property editor for Tag, so only the ones that
  are specific to date and time are shown to the user. }
procedure TS_DateTimeProperty.GetSubProps(
  PropEdit: TPropertyEditor);
var
  Index: Integer;
begin
  if CompareText(PropEdit.GetName, 'Tag') = 0 then
    PropEdit.Free
  else
    begin
      { Keep the sub property editors in
        declaration order. }
      Index := TExposePropertyEditor(PropEdit).NameIndex;
      if Index >= SubProps.Count then
        SubProps.Count := Index+1;
      SubProps[Index] := PropEdit;
    end;
end;
```

**Figure 10:** Getting the subproperties for *TS_DateTimeProperty*.

component is initialized with the *TDateTime* value for its respective component. The surrogate components also point back to the property editor, so they can propagate

```
{ Typecast any property editor to TExposePropertyEditor to
  expose the property's NameIndex fields from the PropInfo.
  This is just a clever hack to gain access to an otherwise
  protected field of TPropertyEditor. }
type
  TExposePropertyEditor = class(TPropertyEditor)
  public
    function NameIndex: Integer;
end;

function TExposePropertyEditor.NameIndex: Integer;
begin
  Result := GetPropInfo^.NameIndex
end;
```

**Figure 11:** Exposing a protected method with *TExposePropertyEditor*.

value changes. The *ChildList* field holds the list of surrogate components.

The *GetProperties* method must issue a callback procedure for each subproperty editor. Start by calling *GetComponentProperties*, which creates and initializes the property editors for the subproperties. Set the filter to *tkInteger*, so *GetComponentProperties* returns property editors only for the integer-type properties. All the constituent date and time properties are integers, but so is the *Tag* property. Remove the *Tag* property by looking at its name and freeing the property editor if it is *Tag*.

## All in Order

By default, these property editors are sorted in alphabetical order (*Day*, *Hour*, *MilliSec*, *Minute*, *Month*, *Second*, *Year*). They are most useful, however, when they are ordered by magnitude (i.e. *Year*, *Month*, *Day*, *Hour*, *Minute*, *Second*, *MilliSec*), so you should rearrange them. Thus, *GetSubProps* stores the property editors in a list, *SubProps*. The index of each property editor in the list is given by the editor's *NameIndex*, which is its index in declaration order. Remember that in *TS_PropDateTime*, the order of the published properties is the same order the Object Inspector should use. This leaves holes in the list (e.g. for *Tag*), so skip over any **nil** items in the list.

After the list is built, issue the callback procedure *Proc* for each subproperty. The Object Inspector can then display the property values for the subproperties. Figure 10 shows the *GetProperties* and *GetSubProps* methods.

## Into the RTTI

The NameIndex field is part of the Run-Time Type Information (RTTI) for a property, in the *TPropInfo* record. (This information is not documented, but you can learn about it in the TypInfo.int or TypInfo.pas file. Look in Delphi's \Source or \Doc directory.) The *GetPropInfo* method returns a pointer to the *TPropInfo* record for the property.

Unfortunately, the *GetPropInfo* method is protected in *TPropertyEditor*. For *TS_DateTimeProperty* to get the property information, it must be able to call a protected method. Figure 11 shows the trick that exposes the

```
{ A special property editor for the month. }
type
  TS_MonthProperty = class(TIntegerProperty)
  public
    function GetAttributes: TPropertyAttributes; override;
    function GetValue: string; override;
    procedure SetValue(const Value: string); override;
    procedure GetValues(Proc: TGetStrProc); override;
  end;

{ Add a value list, with all the month names. }
function TS_MonthProperty.GetAttributes:
  TPropertyAttributes;
begin
  Result := inherited GetAttributes + [paValueList]
end;

{ Get the value string by looking up the month number. }
function TS_MonthProperty.GetValue: string;
begin
  Result := LongMonthNames[GetOrdValue]
end;

{ Set a new value by looking up the month name. If it's not
  found, call the inherited SetValue to convert the integer
  value. Check the long and short month names. }
procedure TS_MonthProperty.SetValue(const Value: string);
var
  Month: Integer;
begin
  for Month := Low(LongMonthNames)
  to High(LongMonthNames) do
    if CompareText(Value, LongMonthNames[Month]) = 0 then
      begin
        SetOrdValue(Month);
        Exit;
      end;

  for Month := Low(ShortMonthNames)
  to High(ShortMonthNames) do
    if CompareText(Value, ShortMonthNames[Month]) = 0 then
      begin
        SetOrdValue(Month);
        Exit;
      end;

  inherited SetValue(Value);
end;

{ Retrieve all the month names, for the value list. }
procedure TS_MonthProperty.GetValues(Proc: TGetStrProc);
var
  I: Integer;
begin
  for I := Low(LongMonthNames) to High(LongMonthNames) do
    Proc(LongMonthNames[I]);
end;
```

**Figure 12:** Editing a month by name with *TS_MonthProperty*.

NameIndex field. By casting any property editor to *TExposePropertyEditor*, you can call the *NameIndex* method. Because *TExposePropertyEditor* derives from *TPropertyEditor*, it can call a protected method, such as *GetPropInfo*. Because the *NameIndex* method is not virtual, there is no problem with the fact that the property editor object does not derive from *TExposePropertyEditor*.

## Not without Risk

However, a warning is in order. If you were to use the **as** operator to cast the property editor to *TExposePropertyEditor*, Delphi would raise a run-time exception. Thus, this is a dangerous trick, and you should use it only in specific circumstances. In the right situation, however, it is a powerful technique, and one that makes the *TS_DateTimeProperty* class feasible.

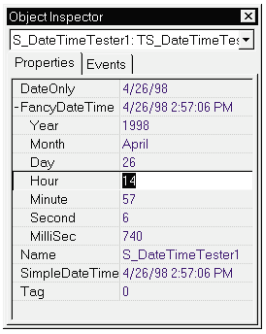



**Figure 13:** An example of the *TS_DateTimeProperty* editor in use.

## Putting It to Use

Remember that the purpose of adding subproperty editors to the date/time property editor is to make it easier for the user to set a date and time. It is simplest to choose a month by name, not by number. Thus, you can define the *TS_MonthProperty* editor, as shown in Figure 12.

Now that the *TS_DateTimeProperty* editor is complete, how is it used? Again, quite simply! Remember that this property editor will apply to any property of type *TDateTime*. Therefore, all you need to do is add a new property for a component that is of type *TDateTime*, and the *TS_DateTimeProperty* will automatically provide the property editor for it.

Figure 13 shows the new *TS_DateTimeProperty* editor in use. Notice how easy it is to enter a specific date or time by entering the constituent parts. Figure 14 shows the unit S_Test.pas used to test the *TS_DateTimeProperty* editor.

```
unit S_Test;

{ Test components for demonstrating property editors. }

interface

uses
  SysUtils, Classes, DsgnIntf;

type
  TS_DateTimeTester = class(TComponent)
  private
    fDateTime: TDateTime;
  public
    constructor Create(Owner: TComponent); override;
  published
    property FancyDateTime: TDateTime read fDateTime
                                       write fDateTime;
    property DateOnly: TDateTime read fDateTime
                                    write fDateTime;
  end;

implementation

{ An object is initialized to zero, but that is not a valid
  TDateTime value, so initialize the TDateTime to something
  that makes sense, such as the current date and time. }
constructor TS_DateTimeTester.Create(Owner: TComponent);
begin
  inherited Create(Owner);
  SimpleDateTime := Now;
end;

end.
```

**Figure 14:** The unit S_Test.pas used to test the *TDateTime* property editor.

## Conclusion

Delphi makes it difficult to create subproperties for arbitrary property types, but it is possible. You can create a surrogate component that defines the desired subproperties. The parent property editor creates an instance of the surrogate component, and pretends that the component's properties are its own subproperties. With a little extra work to ensure the parent property editor and the subproperty editors communicate their changes, you can complete the illusion that your property has subproperties. And users never know about the magic that takes place behind the scenes. Δ

This article is based on material for Ray Lischner's *Secrets of Delphi 2* [Waite Group Press, 1996]. The book is available for US$49.99 from your local bookstore, or by calling (800) 428-5331.

*The demonstration source referenced in this article is available on the Delphi Informant Works CD located in INFORM\96\NOV\DI9611RL.*

Ray Lischner is a software author and consultant for Tempest Software (http://www.tempest-sw.com). His current focus is object technology, especially in Delphi, Java, and Smalltalk. You can reach him at delphi@tempest-sw.com.



*By Cary Jensen Ph.D.*

# Cross-Platform Delphi

## or, IFDEFing for Fun and Profit

**It seems we are forever writing applications for two platforms: the one that dominates the marketplace (currently Windows 3.1), and the one that will (Windows 95). Specifically, Delphi programmers must be able to move easily between the 16-bit, segmented-memory environment of Delphi 1, and the 32-bit, flat-memory world of Delphi 2 — often with the same application.**

However, compiling 16- and 32-bit executables from one set of source requires planning, and sometimes, compromise. This month's column discusses why this is increasingly so. It also reviews some of the issues and techniques you need to keep in mind when building cross-platform applications.

### Background

Delphi 2 has been available for almost nine months. Unfortunately for Borland, the acceptance of Microsoft Windows 95 is less widespread than expected, especially among large corporations. Likewise, the number of end-users running Windows NT is relatively small. Consequently, not many Delphi developers have been developing applications exclusively for the 32-bit platform, which is the only platform that can run Delphi 2. In fact, based on my interactions with other Delphi developers, most continue to program exclusively in Delphi 1.

This situation won't last forever; recent developments pave the way for an increase in 32-bit applications. The first is the cost of RAM and hard disk space. Because Windows 95, and particularly Windows NT, require a lot of both, the cost of these resources was one obstacle preventing companies from upgrading their operating systems. Now, however, price is not nearly as big a factor.

The second development is the release of Windows NT 4.0. Many companies have been waiting to make their operating system decision based on this new release of NT. Now that it's available, some corporations will upgrade to NT 4.0, while others will decide to migrate to Windows 95.

As a result of these developments, it's likely most independent consultants, as well as many corporate developers, will find themselves building Delphi applications that must run on both 16- and 32-bit operating systems. Unfortunately, unlike Borland's C/C++ product line, Delphi 2 doesn't permit you to compile both. Instead, a developer must compile an application using Delphi 1 to create a 16-bit .EXE, and again in Delphi 2 to create the 32-bit version.

Using two versions of Delphi to compile an application in 16 and 32 bits is often more than inconvenient. It can involve major compromises and time-consuming coordination. The reason is that Delphi 2's feature set is not

only more extensive than Delphi 1's, but there are also incompatibilities between the two products. While the original Delphi 1 VCL (Visual Component Library), and RTL (run-time library) were designed to be compilable on 16- or 32-bit operating systems, the same is not entirely true of Delphi 2's VCL.

## What to Do?

The following is a list of considerations, potential problems, and possible solutions that you can use when you need to build a cross-platform application.

**Use Delphi 1 to design and maintain the application.** Start by using only those components shared by Delphi 1 and 2. One way to ensure this is to do all your design work in Delphi 1; every component available in Delphi 1 is available in Delphi 2. While this may result in a 32-bit application that looks more like a Windows 3.x application, it prevents you from maintaining two sets of source files.

Another reason to do all your design work in Delphi 1 is that it prevents Delphi 2 from storing properties in the DFM file that aren't in the Delphi 1 version. If you accidentally modify a project under Delphi 2 and it adds a property to the DFM file, you'll see an error message when you attempt to load that application in Delphi 1. (You can usually get around this by instructing Delphi 1 to ignore the error. As long as you do not need that property — in which case the application cannot be compiled with Delphi 1 — the next time you compile the application in Delphi 1, it will remove the unnecessary property from the DFM file.)

**Avoid non-Delphi objects.** Avoid using objects that aren't native Delphi components. Specifically, do not use VBXes, OCXes, or ActiveX controls. VBXes are only supported under Windows 3.x, while OCXes and ActiveX controls are supported only under Windows 95 and Windows NT.

**Be aware of data type ambiguities.** Avoid code that is sensitive to the size and range of platform-dependent data types. For example, don't write code that assumes an integer will be either 16- or 32-bit. (This includes strings, which we'll discuss shortly.)

**Use conditional compiles.** Use compiler directives to conditionally compile code that can only be compiled under one platform or the other. For example, the following statement checks the pre-defined conditional symbol, WIN32, and then declares the variable *InitStorage* as a *TIniFile* type in Delphi 1, or a *TRegIniFile* type in Delphi 2, accordingly:

```
{$IFDEF WIN32}
var
  InitStorage: TRegIniFile;
{$ELSE}
var
  InitStorage: TIniFile;
{$ENDIF}
```

Another location where conditional compiler directives can be useful regards the inclusion of units in **uses** clauses. For example, while the *TRegIniFile* class is defined in the Registry unit, the *TIniFile* class is defined in the IniFiles unit. Sometimes, however, you may find that including a conditional compiler directive in the middle of a **uses** clause will generate a compiler error. Specifically, sometimes a statement like the following will not compile:

```
uses
  SysUtils, WinTypes, WinProcs, Messages,
  {$IFDEF WIN32}
  Registry,
  {$ELSE}
  IniFiles,
  {$ENDIF}
  Classes, Graphics, Controls, Forms, Dialogs;
```

In those cases, make the entire **uses** clause conditional:

```
{$IFDEF WIN32}
uses
  SysUtils, WinTypes, WinProcs, Messages, Registry,
  Classes, Graphics, Controls, Forms, Dialogs;
{$ELSE}
uses
  SysUtils, WinTypes, WinProcs, Messages, IniFiles,
  Classes, Graphics, Controls, Forms, Dialogs;
{$ENDIF}
```

**Be careful with strings.** The default string type in Delphi 2 is an ANSIString, while the default string type in Delphi 1 is comparable to Delphi 2's ShortString. These string types are not compatible. There are two ways to get around this problem.

One is to explicitly declare the length of a string to be 255 characters or less using the `string[nnn]` syntax. For example, even in Delphi 2, the following statement will declare the string variable *MyString* to be of type ShortString:

```
var
  MyString: string[30];
```

The second technique is to conditionally include the `{$H-}` compiler directive under Delphi 2. The `{$H-}` compiler directive instructs Delphi 2 to use ShortString by default. You must execute this statement conditionally, because `{$H-}` will produce a compile error in Delphi 1:

```
{$IFDEF WIN32}
{$H-}
{$ENDIF}
```

**Watch those DLLs.** 16- and 32-bit DLLs are not compatible, so you'll need to create and compile them separately. DLLs created for Windows 95 should use the **stdcall** directive for compatibility with other 32-bit DLLs. This directive takes the place of the **export** directive used in Delphi 1. [For an in-depth discussion of this topic (and many others), see Ray Lischner's article "Classy DLLs" in the October 1996 *Delphi Informant*.]

**Watch unit size.** Remember that Delphi 1 can't use units over 64KB. Delphi 2 has no such limit. This is why Delphi 1 uses two units to provide the Windows 3.x interface:

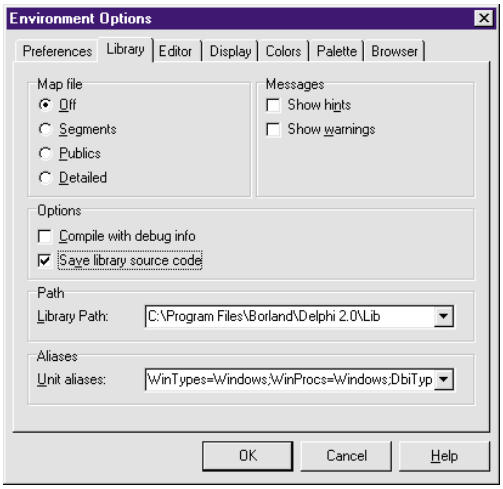



**Figure 1:** The Library page of the Environment Options dialog box.

WinProcs and WinTypes. Delphi 2 uses a single unit, Windows, for this purpose. This is one more reason you shouldn't create your cross-platform applications in Delphi 2; Delphi 2 will add the Windows unit to the interface **uses** clause, and Windows is not defined for Delphi 1.

However, Delphi 2 still recognizes WinProcs and WinTypes. It does this through the Unit aliases definition found on the Library page of the Environment Options dialog box (see Figure 1). Notice that this field includes the statement `WinTypes=Windows`. This instructs Delphi 2 to use the Windows unit any time it sees a reference to WinTypes in a **uses** clause.

**Watch those resource files.** 32-bit resources are not compatible with 16-bit resources. You will need to compile these resource files separately for each platform. You can compile a resource script into a 16-bit resource file with the BRCC.EXE version of the Borland Resource Compiler. Use Delphi 2's BRCC32.EXE to compile resource scripts into 32-bit resources.

**Beware of Windows API calls.** Some of the Windows 3.1 API calls have been changed or made obsolete in Windows 95. If you need to make explicit API calls, make sure they are compatible. Again, this is a place where conditional compiler directives can be helpful.

**Provide a uniform look.** Delphi 2 applications use a smaller, lighter-weight font than Delphi 1 applications. If you want to change a Delphi 1 application to have a look similar to that of Delphi 2, use the Object Inspector to set the *Font* property of each form to MS San Serif, Regular style, 8 point. All objects on that form whose *ParentFont* property are set to *True* will then use that font.


**Figure 2:** You've probably seen this one. It's the error message often displayed when, from the Delphi 1 IDE, you open a project compiled in Delphi 2.

**Respond to "Duplicate resource" errors.** When opening a project in Delphi 1 that was compiled in Delphi 2, you'll often get the Error dialog box shown in Figure 2. It's a common error that simply means you must rebuild the project. Click on **OK**, then select **Compile | Build All** from the menu. This will cause all units to recompile to 16-bit DCUs (Delphi Compiled Units).

## Conclusion

While creating a single set of source files that can be compiled using Delphi 1 and Delphi 2 requires some coordination, it is not particularly difficult. Furthermore, the benefit of maintaining a single source base will usually far outweigh the burden of making the source compatible with both Delphi 1 and 2.

Interestingly, the two most common requests from Delphi developers at the recent Borland Developer's Conference were for a single version of Delphi that can compile 16- or 32-bit applications, and an updated version of Delphi for Windows 3.x. If either request is granted, building cross-platform applications with Delphi would be even easier. Δ

Cary Jensen is President of Jensen Data Systems, Inc., a Houston-based database development company. He is author of more than a dozen books, including *Delphi In Depth* [Osborne/McGraw-Hill, 1996]. He is also Contributing Editor to *Delphi Informant*. You can reach Jensen Data Systems at (713) 359-3311, or via CompuServe at 76307,1533.



*By James Callan*

# Data of Many Shapes

## or, Building Mighty Morphing Power Grids

**Polymorphism can simplify your life. This barely-pronounceable OOP technique curbs complexity and imposes order on an otherwise chaotic world. By shrouding diversity, polymorphism helps us build generic, enduring systems. Widely believed to be a specialized technique useful only for graphics applications or programming books, the P-word has in fact long been used in database applications. Surprised?**

This article explains how you can use inheritance, polymorphism, and specialization in your client/server applications. We'll begin with concrete examples of these techniques in action, and provide a firm conceptual foundation. Moving to database modeling, we'll introduce subtypes. We'll end by creating an Extra Sensory Perception (ESP) game that builds on all the ideas introduced in this article, and shows you how to add Mighty Morphing Power Grids to your next application.

What, you may ask, are Mighty Morphing Power Grids? They're those super-snazzy dual-row grids in Quicken and other "killer" applications that visibly change, depending on what kind of data you add. These applications use one simple form for dozens of different kinds of transactions. Figure 1 illustrates one such transaction in a commercial product named QuickXpense.

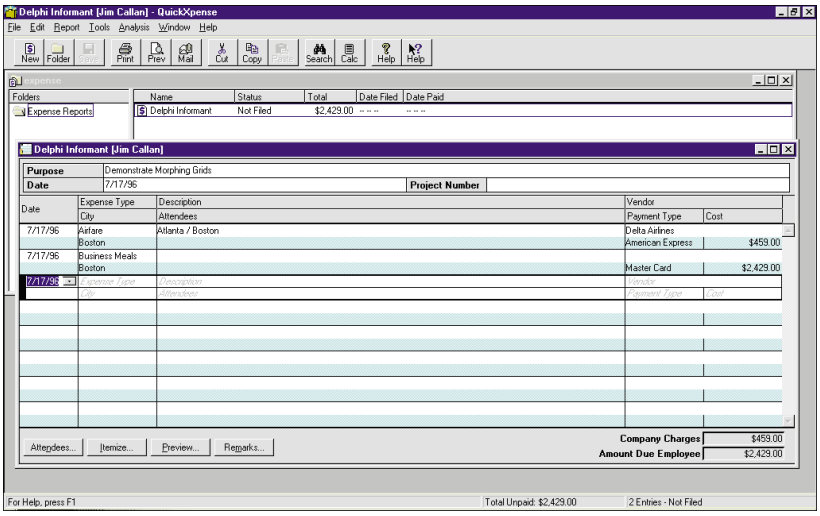



**Figure 1:** QuickXpense is a good example of an application with morphing capabilities.

If you're like me, you're under a constant barrage to add the latest clever features to your projects, but what's a software designer to do? Lose more sleep? Drink more Jolt Cola? There's a better way, using — yes — polymorphism, and Delphi makes it easier than you think.

### Polly Who?

I'm amazed how academics can obfuscate. Let's define *polymorphism* simply and directly. Literally, the term means having many forms. In the world of objects, it means you can ask two different objects to do the same thing, and both will respond sanely.

Consider the following example: In the first, big-screen "Batman" movie, Bruce Wayne invites Vicki Vale to his mansion for a romantic evening. Once they're asleep, Batman slumbers hanging upside-down, but Vicki dozes in a bed. Here, you have two "people" objects, each receiving a "sleep" message. One sleeps in bed, the other on a chin-up bar.

Strange, yet polymorphic. Batman and Vicki Vale are both people. All people sleep, so both are capable of sleeping. However, there are some differences between bats and people. Hence, the method by which a Batman implements the "sleep" function is different from how

an ordinary person implements "sleep." This difference is how each person specializes their way of sleeping.

The power of polymorphism gives client objects a common way of dealing with server objects that share a common ancestry. For example, all shapes have a *Draw* method, and all windows have a *Paint* method. Polymorphism eliminates diversity in areas where objects share similar experiences. *Specialization*, on the other hand, is what permits a square to implement the *Draw* method differently than the way a circle implements *Draw*.

Perhaps one of the best illustrations of the relationships between polymorphism, inheritance, and specialization is to remind you where you first encountered these very old concepts.

## From Sets ...

It's funny how every generation learns "new math." Math is ancient, yet we seem to constantly reinvent it. In one of the latest reincarnations of math, grade school teachers introduced *Venn Diagrams* to teach the concept of sets.

In a Venn Diagram, sets are represented by different, colored ovals. The set of everything is called the *universal* set, designated by the letter "u". Because everything belongs in the universe, the universal set frames a Venn Diagram.

When we have a set of ducks and a set of geese and put them together, we call the combined set a *union*. The union results in a set of ducks and geese. Similarly, let's say we take all books I have read as one set, and as another set, take all the books you have read. If we create an *intersection* between the two sets, we end up with a third set that contains only the books we *both* have read.

The notion of set intersections inevitably leads to the concept of set *subtraction*. If there are books we both have read, and books that each of us has read, then there must be books that neither of us have read. These books are what remains when we subtract our books from the set of all books. Figure 2 contains a series of Venn Diagrams that illustrate these points using the book example.

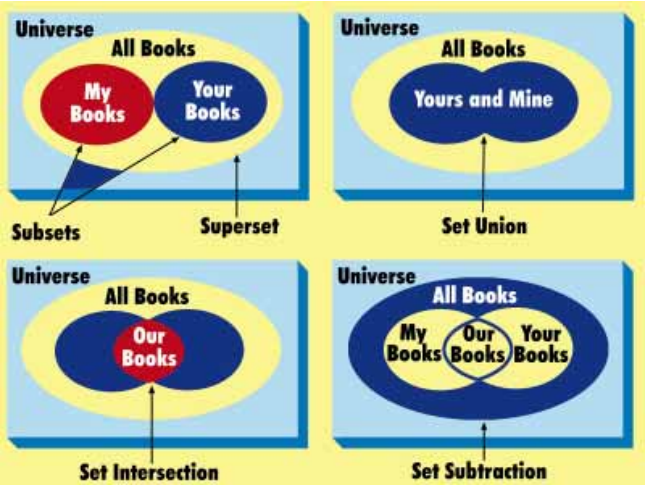



**Figure 2:** Venn Diagrams are used to describe sets.

Some of these sets are nested within other sets. These nested sets are called *subsets*. The set of all books is a *superset* to the set of all books you've read. Similarly, the books you've read are a subset of all books. Both "My Books" and "Your Books" share a common ancestry. We could say that both subsets inherit characteristics from their superset. Let's examine this inheritance more closely.

Books can be read. I read with a pen in hand and tend to make notes in the margins. When you read, however, you may leave your books in pristine condition. Both sets of books can respond to being read, yet the fact that their "state" is different after having been read can be an indication of polymorphism. Thus, the mere act of reading could specialize a book, and be used to distinguish one book from another.

The subject matter covered in each book is another way of distinguishing them — there may be many such ways. When a characteristic (or characteristics) of an item (a.k.a. an element) determines its membership in a set, it's called the *characteristic function* for the set. The characteristic function for the set of books you have read is the test to determine whether you have read the book.

Intuitively, we see that sets form the theoretical basis underlying inheritance, polymorphism, and specialization. There is much more to object-orientation than set theory, but consider the Venn Diagrams in Figure 2 as a foundation for what follows.

## ... to Databases

A relational database is a collection of tables that interact based on the principles of relational algebra (which is itself used to manipulate sets). A well-designed database provides rich information about the real world entities represented by the data stored in the tables. A database is a set of tables. A table is a collection of interesting characteristics about similar entities — such as people, places, or things. Tables are represented by columns and rows. Each row represents one entity (or object). A table is thus a set of characteristics of entities in the world we are modeling.

Understanding sets is imperative to developing useful databases. For example, you create an intersection between two sets when you execute a SQL SELECT statement that *joins* two tables. As we'll see, sets are also critical to building powerful data grids.

## The Power behind the Grids

Staring at Figure 1 probably won't tell you a lot about the database design that lies behind the spiffy screen. Figure 1 is only one view of a grid that changes (or *morphs*) based on the type of expense being entered. For example, entering a meal expense requires different information from the user than entering a mileage expense or that of a plane ticket. The grid gains its power from its ability to accommodate different types of expenses in a single, compact, consistent, and convenient manner. Let's consider the problem of how we might design a database to support time and expenses like that shown in Figure 1.

The first thing to notice about this screen is that the master source of data seems to be the employee. Employees enter their expenses, and expenses are tied to a specific employee. This gives us one set of relationships (they are bi-directional) between expenses and employees. Based on this relationship, we can construct the very simple *entity relationship diagram* (ERD) shown in Figure 3. An ERD is a standard way of analyzing data relationships. In an ERD, boxes represent entities and lines represent the relationship of the entities. The ERD in Figure 3 can be read as:

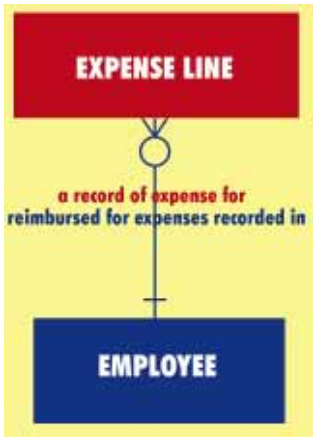



**Figure 3:** An entity relationship diagram (ERD).

- *Each Employee may be reimbursed for expenses recorded in one or more Expense Items*, and
- *Each Expense Item must be a record of expense for only one Employee*.

The formalisms of the diagram nicely capture the requirements of the system. ERDs per se are outside the scope of this article, but here are a few of the rules:

- A single line (like that connecting the EMPLOYEE entity) indicates that only one record can exist in the relationship.
- Conversely, the line with the "crows foot" (like that connecting the EXPENSE LINE entity) indicates there can be multiple records.
- The line perpendicular to the line connecting the EMPLOYEE entity indicates there must be an EMPLOYEE entity.
- The open circle on the line connecting the EXPENSE LINE entity indicates there may be no EXPENSE LINE.

The next thing to notice about Figure 1 is that the Expense Line morphs on the basis of the type of expense entered. When the line morphs, it displays different fields for the user to enter. Some fields have drop-down lists, while others are simple edit fields. From this observation we can conclude that certain fields (known as *attributes* in data modeling parlance) are only applicable for certain types of expenses. We can also conclude (based on drop-down combo boxes) that certain expenses have relationships to other entities in the system.

These two conclusions lead to extending the ERD in Figure 3 to one that resembles Figure 4. Each type of expense has been broken out into its own entity. Also note that the new expense entities are contained inside the larger Expense Line entity. These new entities are called *subtypes*, and the larger "containing entities," such as Expense Item, are termed *supertypes*. Subtypes and supertypes are the database equivalents of subsets and supersets.

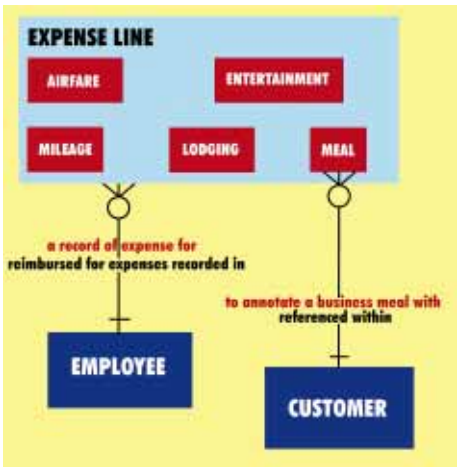



**Figure 4:** A more complex ERD.

The new and improved ERD in Figure 4 may be read the same way as the ERD in Figure 3, but with the additional provisions that:

- *There are five types of Expense Lines: Airfare, Entertainment, Mileage, Lodging, and Meal*,
- *Each Meal Expense Line must be to annotate a business meal with only one Customer*, and
- *Each Customer may be referenced within one or more Meal Expense Lines*.

Clearly, we are simplifying a detailed and improved model to illustrate a point. Figure 4 captures more of the requirements than Figure 3. In many ways, the version in Figure 4 is also more powerful.

Subtypes are interesting because they inherit all the relationships and attributes present in their supertype. Thus, from inheritance we know that:

- *Each Mileage Expense Line must be a record of expense for only one Employee*.

Subtypes in ERDs correspond precisely with persistent inherited classes (when relational databases are used for the persistent store) in the class diagrams of all leading object-oriented design (OOD) methodologies. Figure 4 illustrates this technique of data inheritance. Later, we'll develop a program that demonstrates behavioral and data inheritance working together.

The ERD in Figure 4 can be implemented in many ways. You could store all the expenses in one, two, or five tables. Although the rationale to select one method over another is beyond the scope of this article, the analysis techniques that lead to this type of ERD are relevant to creating powerful data grids.

## Making the Static Dynamic

ERDs create only a static view of the relationships between the entities of our systems. ERDs also have limitations in the way they move us toward presenting our data to users. Alternative design techniques must be used. OOD techniques were introduced to permit us to capture both the static relationships and the dynamic interaction between

objects. OOD techniques can also help us model the way we present data to users.

Ideally, we would like a common manner of dealing with our data so that access to it can be controlled and well managed. For now, relational databases are that common access method. (However, good commercial object-oriented databases are just over the horizon, e.g. Oracle8.) In addition to data access, we also want a consistent way of presenting our data, yet in a manner that supports diversity among our objects.

The expense entry grid of Figure 1 presents data consistently (row per expense), yet it permits high variability in how data is displayed and entered (row-to-row morphing). The power grid is an example of what we want in our applications. And do we need to resort to C++ to create Mighty Morphing Power Grids? Of course not; Delphi has the power.

## Subtle Power: The Best Kind

The best kind of power is the kind that goes unnoticed. I drive a Jaguar XJS, V-12. Many people think the Jaguar is wimpy compared to the BMW 7xx series, until they drive one and feel its subtle power. Delphi is like this.

Delphi 2 introduced the DBCtrlGrid component as a way of displaying multi-column data. The DBCtrlGrid provides a data-aware, multi-column grid that allows you to place certain data-aware controls, such as DBEdit fields, into the first grid cell. It then dynamically replicates the controls in the other cells as needed. It's cool, but doesn't permit you to place just any kind of control into the cells.

The DBCtrlGrid surfaces an *OnPaintPanel* event that allows you to control the appearance of how data is displayed. Unfortunately, this event is not very helpful in overcoming the component-imposed limitations on what controls are allowed on panels during data entry and editing. We need a better way to edit in DBCtrlGrids. We need freedom in our choice of controls, as well as a way to maintain control programmatically. We need a way to morph while editing.

## Hover Editing

Because the DBCtrlGrid is tied to a DataSource component, the Grid's state is governed by the DataSource's state. Therefore, when the DataSource is in edit mode, the Grid is in edit mode; and when the DataSource is read-only, the Grid is read-only. What if we used a component that gives us more flexibility when we are editing, and used the DBCtrlGrid to display our data? Furthermore, because the *OnPaintPanel* event provides the coordinates of a rectangular region in which we can display the current record, we can position our "editing" component directly over this region (think of it as "hover editing"). We'll simply stack the components and have them collaborate.

We'll use the ubiquitous Panel component as an editing surface. Because you can place any control on it, using a Panel brings control choice freedom. If we need to morph our panel based on the type of data entered, we can alter the visibility of our controls, or switch panels altogether. Intuitively, this scheme works; however, how will it look in a real application?

## The Personal Psychic Network

Imagine you are the Director of Information Technology for the world famous Personal Psychic Network (PPN). People from all over the globe call the psychics at PPN for personal readings (credit cards accepted). PPN's popularity is growing, but they have a shortage of psychics. The human resources manager approaches you, explaining that she wants an employment exam to help her separate the good psychics from the "also-rans." You propose a simple Extrasensory Perception (ESP) exam to test for psychic aptitude.

After researching ESP, you determine that a simple program using the ESP symbols (developed by Dr Rhine of Duke University) will serve nicely. In his experiments, he used a deck of cards depicting triangles, crosses, circles, waves, stars, and squares. You'll use an electronic deck of cards, the computer will select a card, and the psychic will attempt to name the card.

You decide to avoid text, keeping the program symbolic and intuitive, yet colorful and inviting. Your program has seven basic functions that allow a user to:

1) read about the program,
2) create new tests,
3) delete tests,
4) take a test by making ESP symbol selections,
5) retrieve previous tests,
6) graph their test results, and
7) exit the program.

To select a symbol, a user simply presses the corresponding button depicting the desired symbol. After the user makes a selection, the choice is committed. The computer immediately reveals its choice and the user's selection, and signals a match. The test is fair because the computer commits to all its choices when the test begins.

## Data Model

Figure 5 illustrates the data model for PPN's Psychic Diviner. Each ESP Test must comprise precisely 24 ESP Trials. Each ESP Trial must be for only one ESP Test. Furthermore, each ESP Trial is either Non-selected, Correct, or Incorrect. *Non-selected* means the computer has selected a symbol, but the psychic has not. *Correct*

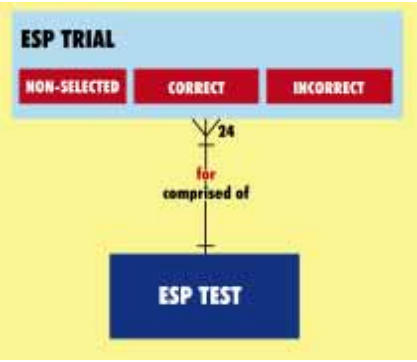



**Figure 5:** The data model for PPN's Psychic Diviner.

and *Incorrect* correspond to whether the psychic and computer match. Notice that subtypes are mutually exclusive. Figure 6 illustrates the database tables that correspond to the ERD in Figure 5.

TESTS.DB

| Field Name | Type | Size | Key | Required | Default |
|---|---|---|---|---|---|
| Name | A | 10 | * | Y | |
| Description | A | 45 | | Y | |
| TestID | + | | | Y | |

TRIALS.DB

| Field Name | Type | Size | Key | Required | Default |
|---|---|---|---|---|---|
| TestID | I | | * | Y | |
| TrialNo | S | | * | Y | |
| Target | A | 1 | | Y | |
| Choice | A | 1 | | Y | N |

**Figure 6:** The structures of the tables for Psychic Diviner.

Beyond the table design, we must define a characteristic function that will help us place each trial record into either the Non-selected, Correct, or Incorrect sets. Therefore, if the Choice field contains an "N" (default during creation), the record is Non-selected. If the Choice field matches the Choice field, then the record is Correct. If the Choice field is not an "N" and does not match the Choice field, the record is Incorrect.

Although our ERD is simpler than the ERD depicted in Figure 4, and our table design is much less complex than one needed to represent corporate expenses, our use of subtypes is exactly the same. Instead of using an "expense type" field, we'll use a combination of two fields on which to build a characteristic function. A grid display for expenses would look different than one for ESP symbols. However, the methods are similar for painting the grids, editing within cells, and morphing. An expense grid, or any other business grid, could include lookup combo boxes that appear and disappear, or other, similar morphing features.

To build your Psychic Diviner application, create the tables shown in Figure 6 using the Database Desktop. Under **Tools | Alias Manager** create a new public alias named `SubTypes`. Set its path to the directory where you have stored the Tests and Trials tables. Use the Image Editor (see Figure 7) to create 32x32 pixel bitmaps of a red triangle, a blue cross, a lime circle, three purple wavy lines, a yellow star, and a magenta square. (The simplest way to create the star is to draw a pentacle and erase the interior lines.) We'll use the values `T`, `C`, `E`, `W`, `S`, and `R` in our Trials table to represent each of these symbols, respectively.

## Forming the Form

Start a new project. On the default form, drop two Panel components, aligning one to the top of the form and the other to the bottom. On the top Panel, place a DBLookupComboBox, a DBEdit, and two Labels. Using the Object Inspector, name the DBLookupComboBox `TestNameLKUFld` and the DBEdit control `TestDescriptionFld`. Set the *Caption* property of the first Label to `Test &Name:`, and the second to `&Description:`. Then change their focus controls to the DBLookupComboBox and DBEdit controls, respectively. Make the value for the Panel's *Caption* property blank and then set its *BevelOuter* property to *bvNone*. Resize and position the controls on the Panel to resemble Figure 8.

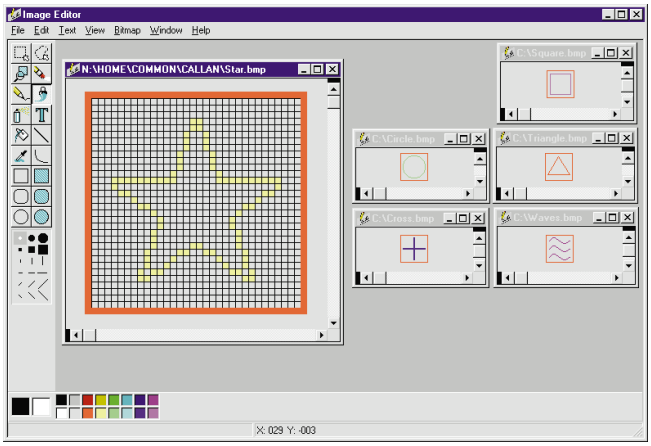



**Figure 7:** Using the Image Editor to modify the shapes used in the Psychic Diviner's test.

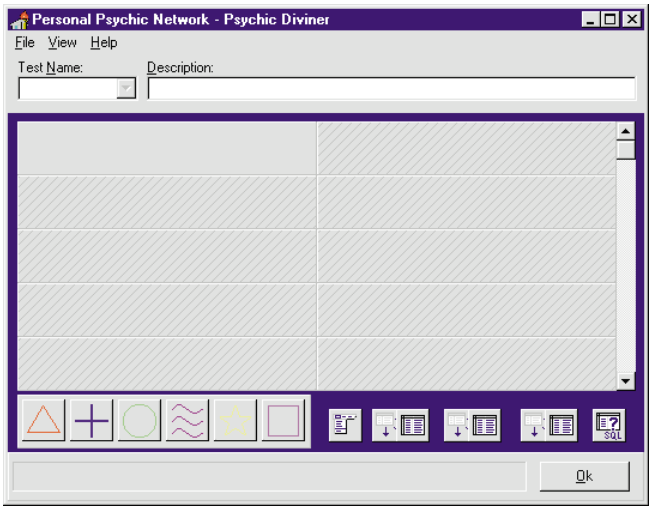



**Figure 8:** The Psychic Diviner at design time.

Next, add a MainMenu component to the form, then double-click on it to create the form's main menu. Using the Menu Designer, add `&File`, `&View`, and `&Help` as top-level menu options. Under **File** add `&New` and `&Delete`, followed by a separator (insert a dash character for its *Caption*), and `E&xit`. Add `&Accuracy` under **View** and set its shortcut key to F1 via the drop-down list of keys. Lastly, add an `&About` option under **Help**, and close the Menu Designer.

On *Panel2*, place a third Panel and a Button. Set *Panel3*'s *Alignment* property to *taLeftJustify*, its *BevelOuter* property to *bvLowered*, and its *Font.Color* property to *clBlue*. Name the Button `OkBtn` and set its *Caption* to `&OK`. Next, make sure the *Caption* properties for *Panel2* and *Panel3* are blank. The lower panel should now match the one in Figure 8.

On the main form, just above *Panel2*, drop an additional Panel and name it `EditPanel`. On `EditPanel`, place six BitBtn components and space them equally. Set their *Height* and *Width* properties to `38` and make their *Caption* properties blank; then set their *Glyph* properties to the bitmap files that you created earlier in the Image Editor. Set the *Tag* properties for the BitBtn components to 1 through 6 (we'll use the *Tag*s later) and name them

TriangleBtn, CrossBtn, etc. Change the EditPanel's *Visible* property to *False* (we'll programmatically control the Panel), and make its *Caption* blank.

Next, position a DBCtrlGrid component in the center of the form and change its *Name* property to TrialsGrid. Set its *RowCount* property to 5, its *ColCount* property to 2, and its *Color* property to *clBtnFace*. Right-click on EditPanel, and from the SpeedMenu, select **Bring To Front**. This raises the Z-order for EditPanel above that of TrialsGrid. You can check this by overlapping the edge of TrialsGrid with EditPanel. EditPanel should obscure TrialsGrid. Now set the form's *Caption* property to: Personal Psychic Network - Psychic Diviner and set its *Color* property to *clActiveCaption*. After proper sizing, your form should look exactly (less the non-visible components of the next section) like the form shown in Figure 8. Now save the form as PSYCHIC and the project as PPNESP.

## Becoming Data Aware

Next, you'll add links to the database tables you created in the Database Desktop. Begin by adding three DataSource components and three Table components to the form. Set all the Table *DatabaseName* properties to SubTypes. Name one DataSource TestsDS, and set its *DataSet* property to the Table you've named TestsTB, and for which you have set its *TableName* property to TESTS. Name another DataSource TestsLKUDS, and set its *DataSet* property to a Table that you have named TestsLKUTB, and for which you have set its *TableName* property to TESTS. Similarly, set up a DataSource (TrialsDS) and Table (TrialsTB) for the Trials table. Point the TrialsTB's *MasterSource* property to TestsDS, and use the Field Links Designer to link the TestID fields for both tables. This sets the *MasterFields* property to *TestID*.

Set the *DataSource* and *DataField* properties for TestNameLKUFld to TestsDS and Name, respectively. Next, specify TestNameLKUFld's lookup by setting its *ListSource* property to TestsLKUDS, its *ListField* property to Name;Description, and its *KeyField* to Name. Now change TestNameLKUFld's *DropDownWidth* property to 300 so the Description field will appear in the drop-down list. After this, set the *DataSource* and *DataField* properties for TestDescriptionFld to TestsDS and Descrip, respectively. Finally, set TrialsGrid's *DataSource* to TrialsDS.

Our last step is to create a cascaded delete for tests. Whenever we delete a test, we must delete all of its related trials. To illustrate the manual method, add a Query component to the form (named DeleteTrialsSQL), which contains this DELETE statement in its *SQL* property:

```
DELETE FROM Trials WHERE TestID = :TestID
```

Set the *TESTID* parameter's *DataType* property to *Integer*, and point its *DataSource* to *TestsDS*. Now we can cascade delete all the trials for a test by executing DeleteTrialsSQL's *ExecSQL* method.

## The Power Plan

The Psychic Diviner will be an unusual Delphi database application for a variety of reasons. Notice that the application lacks a DBNavigator component. This control would clutter our form, so we won't use one. The application also differentiates itself in the use of its drop-down combo box. The combo box serves as a browser rather than as a tool to allow the user to select new values. It replaces the DBNavigator's Prior, Next, First, and Last buttons. The most obvious difference is that our DBCtrlGrid has no fields. It's a bit crazy to use a component that replicates fields without fields, but it does highlight the powerful things you can accomplish through component collaboration. We don't need fields because the *OnPaintPanel* event handler will paint the fields as we need them.

After selecting **File | New**, the user will be prompted to enter the name for a new test. After validating the name, the program will add a new Test record and randomly select 24 ESP symbols, adding a Trial record for each. The program will then generate a default Test Description and pre-position the cursor for data entry.

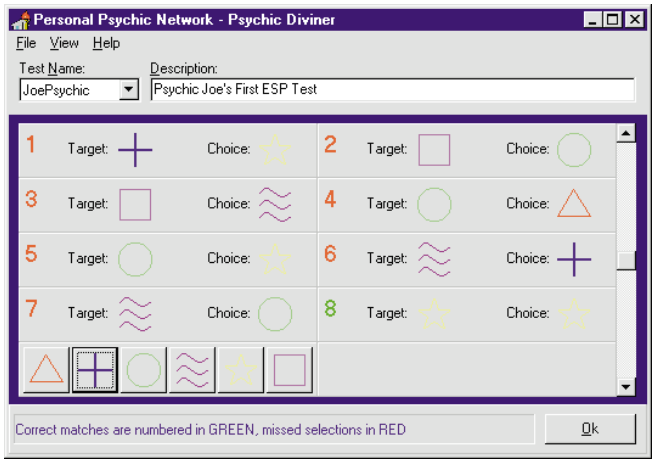
After generating the new trials, the TrialsGrid will call the *OnPaintPanel* event handler for each record. Because the user has not made a choice yet, the program will position the EditPanel over the active record for user selection. All remaining records are Non-selected, so they remain blank.

When the user makes a selection, it's saved. After the selection is saved, three things occur:

- The cursor advances to the next record. Changing records causes the *OnPaintPanel* event to be called twice. *OnPaintPanel* is called once for the previous panel and once for the panel containing the new record.
- The new panel's *Paint* call repositions the EditPanel for user selection. The old panel will paint the panel using the bitmaps on the EditPanel's buttons.
- Additionally, the old panel will number the trial, highlighting the trial number in red or green, depending on the user's accuracy. If you find the application colors too loud, you can always build a monochrome version, or one using more muted colors.

Before leaving the subject of program design, we should also discuss our program's use of Windows resources. The user sees many ESP symbols in the program; however, a single copy of each of the six bitmaps will be used. When the user requests an accuracy display, a pie chart is created and displayed on the EditPanel. The EditPanel morphs into a custom dialog box. This is done to save system resources, demonstrate an alternative dialog box method, and provide an example of how the Panel can be morphed.

Of the remaining program elements, some perform various data translations between our bitmap names and the characters we've chosen to store in our database. Others communicate with the user through various system message boxes.



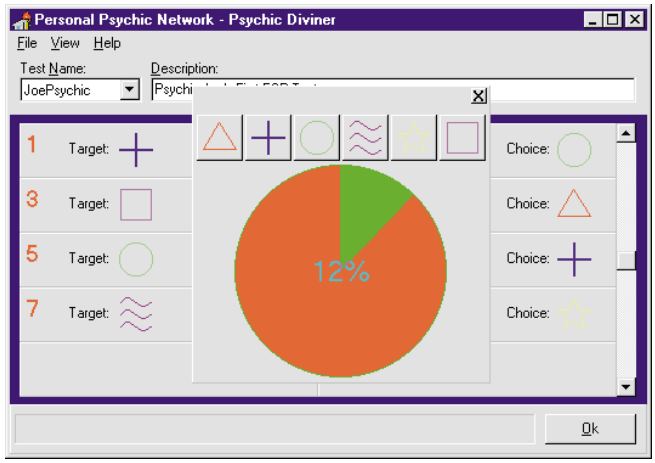




**Figure 9 (Top):** A view of the run-time version of the ESP test. **Figure 10 (Bottom):** The percentage meter shows the accuracy of the current ESP test participant.

## Adding the Power

To add the power to your mighty morphing power grid, add the code contained in Listing Two (beginning on page 35). Until you add an About box to the project, you may want to forego adding the *About1.OnClick* event handler. Figure 9 shows the Psychic Diviner at run time. Figure 10 shows the pie chart that reports the accuracy of the psychic.

If you add hints to each component, the hints will be displayed in *Panel3*. You will need to manually set the *OnClick* event handler for each BitBtn on the `EditPanel` to the *ESPSymbolClick* procedure that you manually create. By declaring the *ESPSymbolClick* method in the class definition for the form, Delphi automatically picks it up in the Event Inspector's drop-down list for the buttons.

By setting the drop-down list's *DataSource* to **nil**, we allow it to activate and the current record to change. Because the tests are WORM (write once, read many), we won't need to use the drop-down list for setting a name. You could embellish the combo box by adding *OnKeyDown* handlers, but most users find the way this additional handler blanks the combo box field a bit distracting.

Using the **mod** and **div** operators in conjunction with the `TrialsGrid`'s *Panel* properties moves the `EditPanel` from cell to cell. The array-indexing equations, provided in the `TrialsDS` *OnDataChange* event handler, should work for any size grid. The only caveat is to ensure your controls fit on the `EditPanel` when it's sized to fit into the cell.

Pay particular attention to the `TrialsGrid`'s *OnPaintPanel* event handler. The *varEmpty* and *varNull* conditions are checked using a Variant variable. This checking was not required for Delphi 1, but Delphi 2 introduced Variants for OLE and OCX compliance. Unfortunately, some of the Variant data conversions raise exceptions when DataSets encounter null records. This test eliminates the nasty conversions.

The *DrawChoice* method paints the cells for Correct and Incorrect trials by drawing on the `TrialsGrid` canvas. The DBCtrlGrid provides a separate canvas for each cell that can be used in the *OnPaintPanel* event handler. We reduce memory by reusing the same bitmaps that are on our `EditPanel`'s BitBtn components. Memory can also be reduced further by dynamically loading the bitmaps from files, but this adds overhead.

The *IndexSymbol* method converts a random number from 1 to 6 into a single character representation for an ESP symbol. *ExpandSymbol* converts from the single character representation into the full word name that corresponds to the component names of the `EditPanel`'s BitBtns.

The *MorphPanel* method changes the `EditPanel` into a decorated modal dialog box. This is a nifty technique that is helpful for eliminating additional forms from projects. The Close SpeedButton's *OnClick* event is dynamically set to the *CloseGraph* method. It restores the `EditPanel` and returns it to its proper position. The *CalcAccuracy* method uses some embedded dynamic SQL to determine the percentage of correct answers as the test proceeds.

## Conclusion

We've discussed how and why subtypes arise in data models. We have also demonstrated that using subtypes in our data models gives us new ways of adding inheritance and specialization to our database applications.

By using polymorphism in data-aware grids, we can alter the appearance of data, *and* change the way the data is entered. Through polymorphism, we can shroud diversity and divine differences. Perhaps the subtle power of Delphi is now more apparent? Δ

*All source code, bitmaps, and database files for the sample Psychic Diviner program are available on the Delphi Informant Works CD located in INFORM\96\NOV\DI9611JC.*

James Callan, an 18-year computing veteran and former consulting director for Oracle Corporation, is currently president of Gordian Solutions, Inc., an information technology consulting provider in Cary, NC. A frequent writer and speaker on information technology and client/server computing, Jim specializes in product design. He can be reached at (919) 460-0555, or by e-mail at 102533.2247@compuserve.com.

### Begin Listing Two — The Psychic Unit

```
unit Psychic;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, DBCGrids, Buttons, StdCtrls, Menus, Mask,
  DBCtrls, ExtCtrls, DBTables, DB, Gauges { Add to list };

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    Panel2: TPanel;
    TestNameLKUFld: TDBLookupComboBox;
    TestDescriptionFld: TDBEdit;
    Label1: TLabel;
    Label2: TLabel;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    View1: TMenuItem;
    Help1: TMenuItem;
    New1: TMenuItem;
    Delete1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    Accuracy1: TMenuItem;
    About1: TMenuItem;
    Panel3: TPanel;
    OKBtn: TButton;
    EditPanel: TPanel;
    TriangleBtn: TBitBtn;
    CrossBtn: TBitBtn;
    CircleBtn: TBitBtn;
    WavesBtn: TBitBtn;
    StarBtn: TBitBtn;
    SquareBtn: TBitBtn;
    TrialsGrid: TDBCtrlGrid;
    TestsDS: TDataSource;
    TestsTB: TTable;
    TestsLKUDS: TDataSource;
    TestsLKUTB: TTable;
    TrialsDS: TDataSource;
    TrialsTB: TTable;
    DeleteTrialsSQL: TQuery;
    procedure FormCreate(Sender: TObject);
    procedure OKBtnClick(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
    procedure TestNameLKUFldDropDown(Sender: TObject);
    procedure TestNameLKUFldCloseUp(Sender: TObject);
    procedure Panel1Exit(Sender: TObject);
    procedure New1Click(Sender: TObject);
    procedure Delete1Click(Sender: TObject);
    procedure TrialsDSDataChange(Sender: TObject;
      Field: TField);
    procedure TrialsGridPaintPanel(DBCtrlGrid: TDBCtrlGrid;
      Index: Integer);
    procedure ESPSymbolClick(Sender: TObject);
    procedure Accuracy1Click(Sender: TObject);
    // Manually add this handler
    procedure About1Click(Sender: TObject);
  private
    Closer: TSpeedButton;   // To close accuracy display
    Graph: TGauge;          // To display accuracy graph
    Editing: Boolean;       // Are we editing?
    OldLeft: Integer;       // Where were we?
    OldTop: Integer;
    Morphed: Boolean;       // Are we morphed?
    function IndexSymbol(Index: Integer): string;
    function ExpandSymbol(Symbol: string): string;
    procedure DrawChoice(Index: Integer;
      Target, Choice: string);
    // See events for TApplication
    procedure DisplayHint(Sender: TObject);
    // Morphs EditPanel to accuracy graph
    procedure MorphPanel;
    // Closes accuracy graph
    procedure CloseGraph(Sender: TObject);
    // Calculates the accuracy rating
    function CalcAccuracy: Integer;
  public
    function GetBitMap(Symbol: string): TBitMap;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

uses AboutDlg;

{ IndexSymbol - Translates an integer into a one character
  symbol that represents a Rhine ESP Symbol

    T = Triangle          W = Waves
    C = Cross             S = Star
    E = Circle (Ellipse)  R = Square (Rectangle) }

function TForm1.IndexSymbol(Index: Integer): string;
const
  Trans = 'TCEWSR';
begin
  Result := Trans[Index];
end;

{ ExpandSymbol - Translates a single character ESP symbol
  into the full ESP symbol name }
function TForm1.ExpandSymbol(Symbol: string): string;
begin
  case Symbol[1] of
    'T': Result := 'Triangle';
    'C': Result := 'Cross';
    'E': Result := 'Circle';
    'W': Result := 'Waves';
    'S': Result := 'Star';
    'R': Result := 'Square';
  end;
end;

{ GetBitMap - Searches the BitButtons on the form by name
  looking for a full ESP symbol name in the component name
  and returns the bitmap contained on the bit button }
function TForm1.GetBitMap(Symbol: string): TBitMap;
var
  I : Integer;
begin
  // Use Tag property to speed up the loop
  for I:= 0 to ComponentCount-1 do
    if (Components[I].Tag <> 0)              and
       (Pos(Symbol,Components[I].Name) > 0) then
      begin
        Result := TBitBtn(Components[I]).Glyph;
        Exit;
      end;
end;

{ DrawChoice - Paints the grid cell based on whether the
  choice is Correct or Incorrect }
procedure TForm1.DrawChoice(Index: Integer;
  Target, Choice: string);
var
  Glyph: TBitMap;
begin
  with TrialsGrid.Canvas do begin // Paint on Cell's Canvas
    // Characteristic Function for SubType
    if Target = Choice then
      Font.Color := clGreen
    else
      Font.Color := clRed;
      Font.Size  := 14;
      TextOut(5, 5, IntToStr(Index));
      Font.Size  := 8;
      Font.Color := clBlack;
      TextOut(40, 14, 'Target:');
      Draw(80, 6, GetBitMap(Target));
      TextOut(155, 14, 'Choice:');
      Draw(195, 6, GetBitMap(Choice));
  end;
end;
```

```
{ ESPSymbolClick - All bit buttons come here
  to be serviced }
procedure TForm1.ESPSymbolClick(Sender: TObject);
var
  Choice: string;

begin
  if not Morphed then
    begin
      Choice := IndexSymbol(TComponent(Sender).Tag);
      TrialsTB.Edit;
      // Set choice
      TrialsTB['Choice'] := Choice;
      // Save the change
      TrialsTB.Post;
      // Hide the panel
      EditPanel.Visible := False;
      // Auto-advance to next record
      TrialsTB.Next;

      // Move back to top when done
      if TrialsTB.EOF then
        TrialsTB.First;
    end
end;

{ CalcAccuracy - Calculates the accuracy rating from all
  selected trials by counting the number or correct and the
  number of incorrect and determining the percentage
  correct out of the total }
function TForm1.CalcAccuracy: Integer;
var
  // Dynamic SQL query used to count answers
  Counter: TQuery;
  NumRight: Integer;
  NumWrong: Integer;
begin
  try
    Counter := TQuery.Create(EditPanel);
    with Counter do begin
      DatabaseName := 'SubTypes'; // Count right answers
      SQL.Add('SELECT COUNT(TRIALNO) FROM TRIALS');
      SQL.Add('WHERE (TESTID = ' +
              TestsTB.FieldByName('TestID').AsString +
              ') AND (TARGET = CHOICE)');
      Open;
      NumRight := Fields[0].AsInteger;
      Close;
      SQL.Clear;    // Count wrong answers
      SQL.Add('SELECT COUNT(TRIALNO) FROM TRIALS');
      SQL.Add('WHERE (TESTID = ' +
              TestsTB.FieldByName('TestID').AsString +
              ') AND (TARGET <> CHOICE) AND (CHOICE <> ' +
              '''' + 'N' + '''' + ')');
      Open;
      NumWrong := Fields[0].AsInteger;
      Close;
    end;
  finally
    Counter.Free;
  end;
  Result := Trunc(NumRight/(NumRight+NumWrong)*100);
end;

{ MorphPanel - Changes EditPanel into an accuracy
  graph modal window }
procedure TForm1.MorphPanel;
var
  I: Integer;
begin
  Morphed := True;
  Editing := EditPanel.Visible;
  TrialsGrid.Enabled := False;  // Disable grid
  with EditPanel do begin
    Visible  := False;
    OldLeft  := Left;
    OldTop   := Top; // Morph the panel
    Left     := 150;
    Top      := 25;
    Height   := Height+200;
  end;
  for I:= 0 to ComponentCount-1 do
    if Components[I].Tag <> 0 then
      with TBitBtn(Components[I]) do Top := Top+20;
        Closer := TSpeedButton.Create(EditPanel);
      with Closer do begin
        Parent      := EditPanel;
        Font.Style  := [fsBold];
        Height      := 13;
        Width       := 13;       // Add some new components
        Caption     := 'X';
        Left        := EditPanel.Width-16;
        Top         := 2;
        OnClick     := CloseGraph; // Map the click event
        Visible     := True;
      end;
      Graph := TGauge.Create(EditPanel);

      with Graph do begin
        Parent       := EditPanel;
        BorderStyle  := bsNone;
        BackColor    := clRed;
        ForeColor    := clGreen;
        Height       := 175;
        Top          := 65;
        Left         := 35;
        Width        := 175;
        Kind         := gkPie;
        Font.Size    := 18;  // Provide a hint
        Visible      := True;
        Hint := 'The percentage indicates the ' +
                'ESP rating for this exam';
      end;
      try
        Graph.Progress := CalcAccuracy;
        EditPanel.Visible := True;
      except  // Empty table creates exception
        CloseGraph(Self);
        messageDlg('Must have test to calculate accuracy.',
                   mtInformation, [mbOK], 0);

      end;
end;


{ CloseGraph - Changes the accuracy graph back into
  the edit panel }
procedure TForm1.CloseGraph(Sender: TObject);
var
  I: Integer;
begin
  EditPanel.Visible := False;
  Graph.Free;
  Closer.Free;
  for I:= 0 to ComponentCount-1 do
    if Components[I].Tag <> 0 then
      with TBitBtn(Components[I]) do Top := Top-20;
  with EditPanel do begin

    Left    := OldLeft;
    Top     := OldTop;  // Restore panel
    Height  := Height-200;
    if Editing then
      Visible := True;
  end;
  TrialsGrid.Enabled := True;
  Morphed := False;
end;

procedure TForm1.DisplayHint(Sender: TObject);
begin
  // Displays hints at form bottom
  Panel3.Caption := Application.Hint;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  // See on-line help on TApplication
  Application.OnHint := DisplayHint;

  // Hides space below grid
  Height := Height-45;
```

```
 // Open the tables
  TestsTB.Open;
  TestsLKUTB.Open;
  TrialsTB.Open;
  Morphed := False;
end;

procedure TForm1.OKBtnClick(Sender: TObject);
begin
  Application.Terminate;
end;

procedure TForm1.Exit1Click(Sender: TObject);

begin
  Application.Terminate;
end;

procedure TForm1.TestNameLKUFldDropDown(Sender: TObject);
begin
  // Make the field editable
  TestNameLKUFld.DataSource := nil;
end;

procedure TForm1.TestNameLKUFldCloseUp(Sender: TObject);
begin
  // Move to selected record
  TestsTB.Locate('Name', TestNameLKUFld.Text, []);
  TestNameLKUFld.DataSource := TestsDS
end;

procedure TForm1.Panel1Exit(Sender: TObject);
begin
  // Implicit posting
  if TestsTB.State in [dsEdit] then
    TestsTB.Post;
end;

{ File:New - Creates a new test and randomly generates
  the 24 trials that comprise the test }
procedure TForm1.New1Click(Sender: TObject);
var
  NewTest: string;
  TestID: Integer;
  I: Integer;
begin
  NewTest := InputBox('Enter New Test Name',
                      'Test Name:', '');
  if NewTest <> '' then
    begin
      if TestsTB.Locate('Name', NewTest, []) then
        begin
          messageDlg('A test named ' + NewTest +
                     ' already has been saved. ' +
                     'Try another test name.', mtError,
                     [mbOK], 0);
          Exit
        end;
      TestsTB.Insert;
      TestsTB['Name'] := NewTest;
      TestsTB['Description'] :=
        'Describe the new test here.';
      TestsTB.Post;
      TestID := TestsTB['TestID'];
      Randomize; // Set up the random generator
      TrialsDS.Enabled := False; // Avoid screen flicker
      for I:= 1 to 24 do begin
        TrialsTB.Insert;
        TrialsTB['Target'] :=
          IndexSymbol(Trunc(Random(6)+1));
        TrialsTB['TrialNo'] := I;
        // Explicitly set Non-selected value

        TrialsTB['Choice']  := 'N';
        TrialsTB.Post
      end;
      TrialsDS.Enabled := True;
      // Position on description field
      TrialsTB.First;
      TestDescriptionFld.SetFocus;
      // Resynch test name field
      TestsLKUTB.GotoCurrent(TestsTB);
      // Make sure lookup refreshes
      TestNameLKUFld.DataSource := nil;
      TestNameLKUFld.DataSource := TestsDS;
    end
  else
    messageDlg('New Test Generation Cancelled',
               mtInformation, [mbOK], 0)
end;

procedure TForm1.Delete1Click(Sender: TObject);
begin
  if messageDlg(
    'Are you sure you want to delete this test?',
     mtConfirmation, [mbYes, mbNo], 0) = mrYes then

    begin
      if EditPanel.Visible then
        EditPanel.Visible := False;
      DeleteTrialsSQL.ParamByName('TESTID').AsInteger :=
        TestsTB['TestID'];
      DeleteTrialsSQL.ExecSQL;
      TestsTB.Delete
    end
end;

{ This handler is responsible for causing the EditPanel to
  hover over the active cell in the TrialsGrid. You may
  want to experiment with different size grids. }
procedure TForm1.TrialsDSDataChange(Sender: TObject;
  Field: TField);
begin
  if TrialsTB['Choice'] = 'N' then
    with TrialsGrid do begin
      EditPanel.Left :=
        (PanelIndex mod ColCount)*PanelWidth+Left;
      EditPanel.Top  :=
        (PanelIndex div ColCount)*PanelHeight+Top;
      EditPanel.Width   := PanelWidth;
      EditPanel.Height  := PanelHeight;
      EditPanel.Visible := True;
      TriangleBtn.SetFocus;
    end
end;

procedure TForm1.TrialsGridPaintPanel(
  DBCtrlGrid: TDBCtrlGrid; Index: Integer);
var
  Choice: string;
  // Workaround for OLE variant NULL field assignment
  V: Variant;
begin
  V := TrialsTB.FieldByName('Choice').AsString;
  if (VarType(V) = varEmpty) or
     (VarType(V) = varNull)  then
    Choice := ''
  else
    Choice := V;
  if (Choice <> '')  and
     (Choice <> 'N') then
    DrawChoice(TrialsTB['TrialNo'],
      ExpandSymbol(TrialsTB['Target']),
      ExpandSymbol(Choice));
end;

procedure TForm1.Accuracy1Click(Sender: TObject);
begin
  MorphPanel;
end;
```

```
procedure TForm1.About1Click(Sender: TObject);
begin
  AboutBox := TAboutBox.Create(Application);
  AboutBox.ShowModal;
  AboutBox.Free;
end;

end.
```

**End Listing Two**

# At Your Fingertips

Delphi / Object Pascal

*By David Rippy*

*Nothing is particularly hard if you divide it into small jobs.*

— Henry Ford

## How do I create a table at run time?

One of the most important aspects of programming in Object Pascal is gaining a firm understanding of creating and freeing components at run time. A classic example is to programmatically create a Table object that is freed once it isn't needed. This is more efficient than adding a Table component to the form. In addition, the form is less cluttered and easier to work with at design time.

The purpose of this example is to create a temporary table (`TblTemp`) to populate *Form1*'s ListBox with the contents of the Name field from table FRUIT.DB (see Figure 1). Because the table is only needed for a short time — to add items to the ListBox — there is no need to use a Table component. Instead, the example creates the `TblTemp` component at run time, then frees it once *ListBox1* has been populated.

The first step is to add the DB and DBTables units to the **uses** clause. These units must be included to use the Table component created by the code. (When you add a Table component at design time, these units are added automatically.)

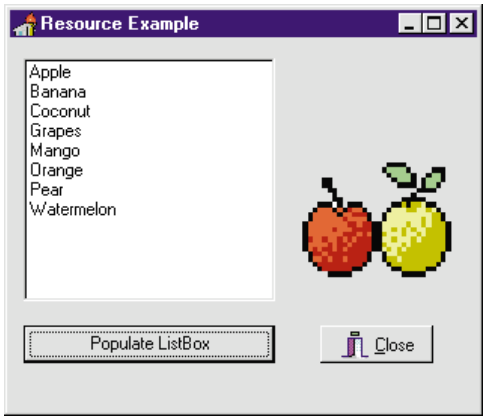



**Figure 1:** The list box is populated by a temporary table.

The code in Figure 2 is attached to the *OnClick* event handler of the **Populate ListBox** button. This code creates the temporary `TblTemp` Table and assigns the necessary properties, iterates through the FRUIT.DB table to populate *ListBox1*, and finally closes and frees the table. The first key statement:

```
TblTemp := TTable.Create(Self);
```

creates an instance of `TblTemp`. Note that because we don't want to assign an owner to `TblTemp`, *Self* is passed to the *Create* method.

When you create a component (or any object) explicitly in this fashion, it's vitally important to free the resources it's using by calling the *Free* method. In this code example, the **with** construct makes the statement a bit hard to discern. Outside a **with** block, the statement would appear like this:

```
TblTemp.Free;
```

Once you become familiar with this technique, it can be applied to essentially any Delphi component, such as *TQuery*, *TIniFile*, and *TMediaPlayer*. (Please note that in an actual application you would never refer to a table in the hard-wire fashion shown here. You would instead assign and use an alias.) — *D.R.*

## How can I quickly view the SQL assigned to a *TQuery* object at run time?

Debugging forms that contain Query components can be tricky at times, particularly if several SQL statements can potentially be assigned

```
procedure TForm1.Button1Click(Sender: TObject);
var
  TblTemp: TTable; { Declare TblTemp }
begin
  { Create instance of TblTemp }
  TblTemp := TTable.Create(Self);
  TblTemp.DatabaseName := '';
  with TblTemp do begin
    try
      TableName := 'FRUIT.DB';
      Open;
      First;
      while not EOF do begin
        ListBox1.Items.Add(FieldByName('Name').AsString);
        Next;
      end;
    finally
      Close;
      Free;
    end;
  end;
end;
```

**Figure 2:** This code is attached to the *OnClick* event handler of *Button1*.

to a single Query object. A technique often used by client/server developers to help the debugging process is to save the Query's SQL code to a text file. The SQL statement can then be easily viewed with a text editor to ensure it's what's expected and is correct.

This technique is easy to implement, requiring only a few lines of code. Figure 3 shows the *OnClick* method of the Save SQL button (see Figure 4). The key to this example is the *SaveToFile* method for the *SQL* property of the Query object. When the button is pressed, *SaveToFile* is invoked, and the *SQL* property value is written to the file SQL.TXT in the current directory. The contents of SQL.TXT can then be viewed in any text editor (see Figure 4).

In a production environment, you would place this code just before the query is executed, instead of attaching it to a button. Again, you would not hard-wire any directory or table name. — *Mike Leftwich, Ensemble Corporation*

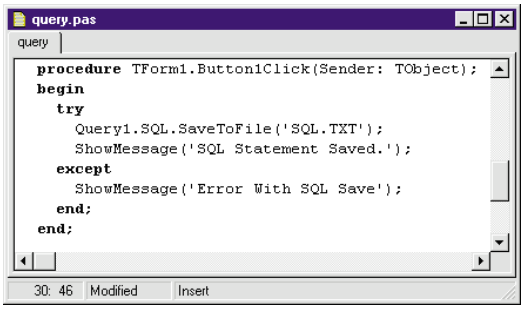



**Figure 3:** *SaveToFile* allows you to save the *SQL* property value to a text file.

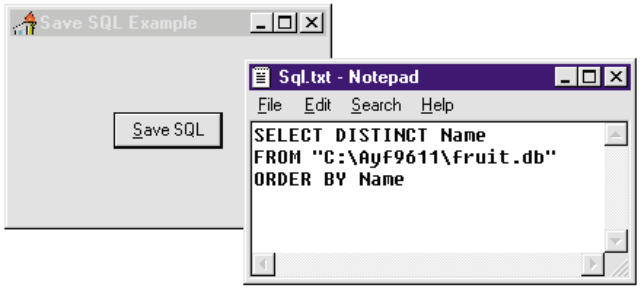



**Figure 4:** You can use any text editor to view the SQL statement once it is saved.

## How can I add blinking text to a form?

Adding a blinking text Label to a form is a great way to attract attention or display an application's status during a lengthy process. For example, the blinking message "Working…" shown in Figure 5 will deter an impatient user from rebooting the machine while the National Budget is being balanced.

Although Delphi's Label component doesn't have a "blink" property, the same effect is obtained by toggling the *Visible* property of the component. The form in Figure 5 contains a Timer, Button, and Label. The Timer's *OnTimer* event handler contains the code that creates the blinking effect for *Label1*. With this code in place, the timer's *Enabled* property is set to *True* in the *OnClick* event handler of *Button1*, causing the Label to start blinking (see Figure 6).

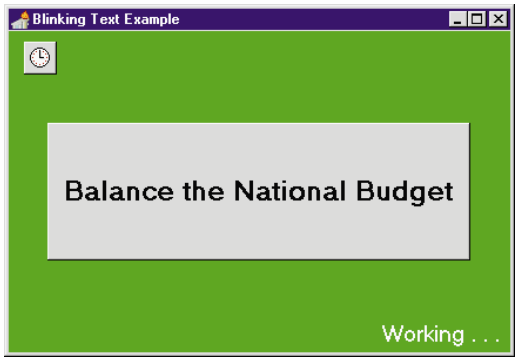



**Figure 5:** The "Working…" Label blinks in this example.

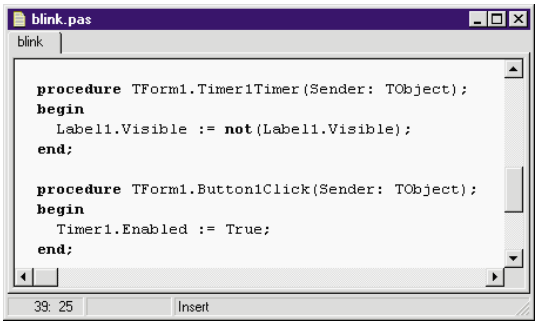



**Figure 6:** *OnTimer* and *OnClick* events for the Timer and Button components.

You can adjust the blink rate by setting the *Interval* property of *Timer1*. The lower the value, the faster the Label will blink. As you might have guessed, this technique will work on any object with a *Visible* property — not just Label components. — *D.R.* Δ

*The demonstration projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\NOV\DI9611DR.*

David Rippy is a Senior Consultant with Ensemble Corporation, specializing in the design and deployment of client/server database applications. He has contributed to several books published by QUE. David can be reached on CompuServe at 74444,415.

CASE STUDY

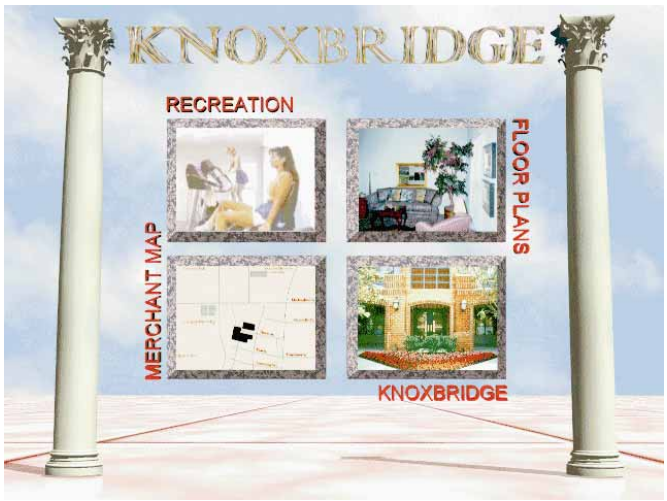
By *David Rippy*

# Inquire Within

## Engaging Kiosk System Reels in Apartment Shoppers

**With over 97,000 apartments in nearly 200 cities, Lincoln Property Company (LPC) is one of the largest property management companies in the United States. To maintain this status, LPC is constantly looking for innovative ways to attract and retain new renters; companies like LPC lose business when a potential renter leaves the rental office of an apartment complex before a leasing agent can offer assistance. Another area of concern is the high cost of printed materials, specifically the floor plans and related pamphlets taken by prospective renters who often do not return.**

With these pitfalls in mind, LPC partnered with Ensemble Corporation's Multimedia Division to implement a high-profile, state-of-the-art kiosk system for the reception area of its Knoxbridge rental office. A kiosk system combats the "walkout" problem by entertaining and informing the customer when the leasing agent is busy. Also, a kiosk system can dramatically reduce off-site printing costs by supplying printouts of floor plans and other literature as needed. Equally important is that the system allows modifications to the literature as often as necessary, for a fraction of what a print shop would charge.



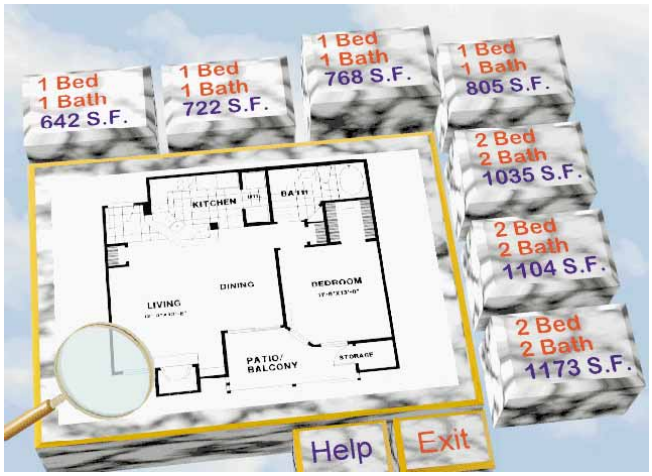

The application has a stately appearance on a 21-inch touch-screen monitor.

These were the goals of the Knoxbridge project:

- reduce printing costs by at least 75 percent,
- reduce the number of walkouts by 50 percent,
- provide a system that reflects the corporate image and quality standards of LPC,
- provide a modular system that can be expanded as needed,
- provide professional-quality 2D and 3D graphics throughout the system, and
- provide a high-quality musical soundtrack throughout the system.

### Implementation

Several development environments were available for creating interactive applications, but only Delphi offered both the multimedia aspects and the solid database




Potential renters can preview any of Knoxbridge's seven floor plans.


The interactive map helps sell the neighborhood.


The rotating, 3D floor plans fuel prospective renters' imaginations.


Still shots of each room round out the visual presentation.


A slide show takes visitors on a virtual tour of the Knoxbridge complex.

functionality needed for a project of this nature. In fact, Delphi provided all the necessary development features "out of the box."

The multiple 2D and 3D animations found throughout the application were created with Autodesk's 3D Studio. Featuring highly detailed renderings of each room, the animations give the user a completely unique and memorable way to shop for an apartment. Unlike other marketing materials that force the shopper to imagine a floor plan's appearance, the Knoxbridge kiosk application provides an accurate view of every available option. These animations, combined with the speed and power of Delphi, give the application a highly professional look and feel.

The four-person project team included a senior consultant from Ensemble, a seasoned Delphi programmer, and a pair of artists specializing in 3D graphics. Working jointly with LPC's Marketing Director and Vice President of Technology, Ensemble created a flexible system that can accommodate any of LPC's apartment complexes across the United States.

A critical factor to the success of the system was Ensemble's ability to "bring to life" the vision of LPC's Marketing Director. Using Ensemble's iterative prototyping methodology, the team could continually test and refine the system until it achieved a perfect combination of excitement and ease of use.

## Results

The kiosk application was an instant success, with many potential tenants using the system each day. Since implementation, the kiosk system has substantially reduced

walkouts, increased apartment rentals, and saved thousands of dollars annually in printing and administrative costs. With the business concept now proven, the system will be rolled out to other LPC properties, each with its own set of custom floor plans, amenities, and area attractions.

The kiosk system is truly a testament to Delphi's ideal balance of compiler technology and rock-solid database features — a perfect match for interactive applications such as this. More importantly to the client, Delphi's Rapid Application Design approach kept development time (and cost) to a minimum. Δ

**APPLICATION PROFILE**

Ensemble Corporation of Dallas, TX recently implemented a state-of-the-art, multimedia kiosk system for the Knoxbridge apartment complex, managed by Lincoln Property Company. The interactive application was designed to entertain and inform potential apartment renters when a leasing agent is not available. Users can review and print out apartment floor plans, examine amenities, and learn about local merchants and institutions.

**Target Audience:** Property management companies and real-estate agents.

**Users:** Potential apartment renters and apartment leasing agents.

**Third-Party Tools:** Autodesk 3D Studio.

**Ensemble Corporation**
12655 N. Central Expressway, Suite 700
Dallas, TX 75243
**Phone:** (214) 960-2700
**Fax:** (214) 960-2704

**Autodesk**
642 Harrison St., San Francisco, CA 94107
**Phone:** (415) 547-2000
**Fax:** (415) 547-2222
**Web Site:** http://www.autodesk.com
**E-mail:** Internet: ktxwebmaster@ktx.com

David Rippy is a Senior Consultant with Ensemble Corporation, specializing in the design and deployment of client/server database applications. He has contributed to several books published by QUE. David can be reached on CompuServe at 74444,415.



*By Shamiq Cader*

# A Matter of Time

## Exploring the TDateTime Object

**Have you ever tried to add five minutes and 30 seconds to the current time? At first this may seem like a trivial computation. If you look closer, however, the complexity of this logical operation becomes clear. For example, what do you do if the time is 11:55 PM on the 31st of December? Suddenly, this doesn't look so trivial.**

Wouldn't it be nice if a programming language exists that allows you to easily perform date and time manipulation? Actually there is. The language, of course, is Object Pascal, and it's the *TDateTime* type that makes date and time manipulation a breeze.

### The *TDateTime* Type

The *TDateTime* type stores date and/or time values in one object. *TDateTime* is of type Float, and stores date and time information in an X.Y format, where X is the date information and Y is the time information. The date is stored as the number of days since year 1 (i.e. 1/1/1); the time is stored as a decimal fraction of the day.

To clarify, consider the following example. Let's say Delphi needs to store 9:00 AM in the *TDateTime* format. At this time, nine hours has elapsed (that is, 9/24 of the day has expired). Therefore, the fraction of the day that has elapsed (since midnight) is 0.375.

This scenario is more complex when minutes are introduced. For instance, what if 9:15 AM must be stored? Fifteen minutes is 15/60 of an hour, or 0.25 hours. This is 0.25/24 of a day, which equates to 0.0104167. Therefore, *TDateTime* stores 9:15 AM as 0.37500 + 0.0104167, or simply 0.3854167.

### *TTimeDate*'s Advantages

Before proceeding, let's discuss the advantages of storing date and time in this format. First, as already described, one variable stores date and time, making date/time manipulation much easier.

Second, handling date/time values as a Float makes date and/or time math easy as well. For example, assume we must determine the correct time after 10 minutes has elapsed. If the beginning time is 11:55 PM, 10 minutes will take us into the next day. With the *TDateTime* format, everything takes care of itself.

For example, when 10 minutes is added to the fraction part, and if the result exceeds 12:00 AM, *TDateTime* will automatically "carry a one" into the integer part, i.e. the date will be increased by one. An example shows this idea more clearly. Let's say a *TDateTime* object is storing 728902.996527778 (8/31/96 11:55 PM). If we add 10 minutes, 0.006944444, to this value, the result is 728903.003472222 (or 9/1/96 12:05 AM).

Examining this, you may wonder how much logic is needed to convert this *TDateTime* type into a readable date and time format. After all, what good is 693500.44543 if it

cannot be understood as a valid date and time? Fortunately, Delphi provides several functions to convert this cryptic number.

### *Time*, *Date*, and *Now*

Delphi provides three simple ways to get the system date and/or time via the *TDateTime* object. To illustrate, Figure 1 shows the results of the following *OnFormShow* routine. (Note that this example assumes six appropriately-named Edit components.)

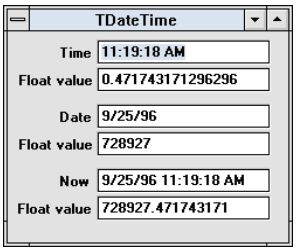



**Figure 1:** The *TDateTime* object stores date/time information as a Float value.

```
procedure TForm1.FormShow(Sender: TObject);
begin
  EditTime.Text      := TimeToStr(Time);
  EditTimeValue.Text := FloatToStr(Time);

  EditDate.Text      := DateToStr(Date);
  EditDateValue.Text := FloatToStr(Date);

  EditNow.Text       := DateTimeToStr(Now);
  EditNowValue.Text  := FloatToStr(Now);
end;
```

The Object Pascal *Time* function returns the system time as a *TDateTime* type which allows us to format it as we see fit. Figure 1 shows the results in two formats. First, the *TimeToStr* function is used to display the time in the standard fashion for the United States (more about this later). Below it, the *FloatToStr* function shows us the value behind the scenes, 0.471743171296296, where the integer (date) portion is set to zero.

The *Date* function returns the system date. The Float value is 728927 and there is no fractional (time) value.

The *Now* function returns the entire date/time value: 728927.471743171.

### Displaying the System Date and Time

System date and time are often displayed in an edit box, label, or memo field. In all cases, the information must be converted to a string format. Delphi provides several functions to display the system date and time.

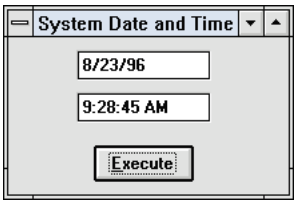



**Figure 2:** Extracting and displaying the system date and time.

The program in Figure 2 uses the *DateToStr* function to convert the date part of the *TDateTime* format into a readable string that can be displayed in an edit box. Similarly, the *TimeToStr* function converts the time part of the *TDateTime* format into a readable string. The string's format (e.g. 23:00:00, or 11:00:00 PM) is determined by Windows settings (which are outside the scope of this article).

More importantly, *DateToStr* and *TimeToStr* convert the *TDateTime* format into readable strings. The .PAS unit for the System Date and Time program is shown in Listing Three on page 49.

### More Date and Time functions

Figure 3, the Add 15 Minutes program, is an implementation of the scenario we discussed earlier. This program adds 15 minutes (instead of 10) to a date and time entered by the user, and displays the resulting date and time in separate edit boxes. Here we are introduced to the *StrToDate* and *StrToTime* functions that try to convert strings to the *TDateTime* format. If the conversion fails, an *EConvertError* exception is raised. This can be used to validate the date or time the user has entered.

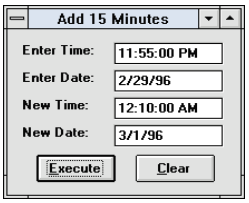



**Figure 3:** A sample program to add 15 minutes to the current time.

Note that this program uses the *MyDate* and *MyTime* variables to hold the date and time values the user has entered. Let's say the same variable is used:

```
{ X.0 where X is an integer }
MyDate := StrToDate(Edit2.Text);
{ 0.Y where Y is a fraction }
MyDate := StrToTime(Edit1.Text);
```

This code causes a problem, because the second call to the *StrToTime* function will overwrite the date portion of *MyDate*. However, using two variables to construct a third works just fine:

```
MyDate   := StrToDate(Edit2.Text);  { MyDate   = X.0 }
MyTime   := StrToTime(Edit1.Text);  { MyTime   = 0.Y }
DateTime := MyDate + MyTime;        { DateTime = X.Y }
```

Because 15 minutes equates to 0.0104166 (or 15/60/24) days, adding 0.0104166 to *DateTime* adds 15 minutes to the date and time entered by the user. The *DateToStr* and *TimeToStr* functions will display the result in the appropriate edit boxes.

If the user enters `11:55:00 PM` into **Enter Time** and `2/28/96` into **Enter Date**, the result displays `12:10:00 AM` and `2/29/96` in **New Time** and **New Date**, respectively. Similarly, if the user enters `11:55:00 PM` as the time and `2/28/95` as the date, the result is displayed as `12:10:00 AM` and `3/1/95`. Thus, we can see Delphi handles every detail, including leap years. (Error checking was not implemented in the program referenced here. We'll discuss exception handling later.) The code for the Add 15 Minutes program is shown in Listing Four on page 49.

### Formatting the Date and Time String

Although several methods exist for formatting the date and time string, a convenient way to do this is to use the *FormatDateTime* function. *FormatDateTime* can be used in place of *DateToStr* or *TimeToStr* to format the display string to any desired form (see Figure 4). For a complete code listing of the associated .PAS file, see Listing Five on page 50.

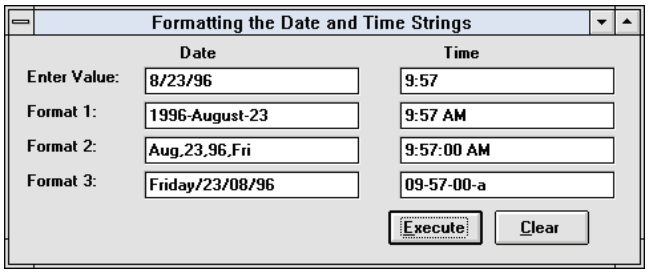



**Figure 4:** A program that displays the date and time in different formats.

The *FormatDateTime* function takes two parameters: the format string indicating the desired format and *TDateTime*. Several formats can be obtained by using *FormatDateTime*. Delphi's online Help features a complete list of format specifiers; just search using "FormatDateTime".

Let's look at an example of *FormatDateTime*. Assuming *MyDate* has 01/08/96, using:

```
FormatDateTime('yyyy mmmm dd',MyDate)
```

yields `1996 January 08` in the display.

The `yyyy` indicates a four-digit, complete-year format (1996).

If `yy` was used:

```
FormatDateTime('yy mmmm dd',MyDate)
```

the short-year format displays `96 January 08`.

The sidebar "Date and Time Combos" on page 47 lists all the possible combinations for the date January 8, 1996 and the time 2:05:08 (assigned to the *MyDate* variable).

## More Date and Time Functions

The program in Figure 5 introduces the use of the *EncodeDate* and *DayOfWeek* functions. The *EncodeDate* function takes three parameters: *Year*, *Month*, and *Day*. *EncodeDate* then tries to put them together and return a valid date as a *TDateTime* format:

```
try
  DT := EncodeDate(Year, Month, Day);
except
  on EConvertError do
    { something }
end;
```

If the conversion fails, an *EConvertError* is raised. This could be used to inform the user that an invalid date was entered. Any error message can be generated in place of the `{ something }` comment. In this program, a flag is set and a message dialog box is displayed. A flag was used simply because an **else** clause was preferred.

Note that all three parameters for *EncodeDate* are of type Word. This is important when dealing with the year parameter. A valid year falls between 1 and 9999. Therefore, `1996` and `96` indicate very different things, i.e. `96` equates to 96 AD, not 1996 AD.

Several other functions belong to the same family as *EncodeDate*. These are *EncodeTime*, *DecodeDate*, and *DecodeTime*.

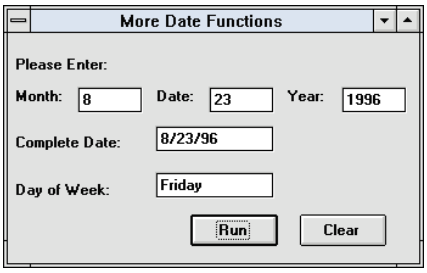



**Figure 5:** A program demonstrating the *EncodeDate* and *DayOfWeek* functions and the exception that's raised when converting to and from the *TDateTime* format.

The *EncodeTime* function works similarly to *EncodeDate*:

```
DT := EncodeTime(Hour, Min, Sec, Msec);
```

where *DT* is of type *TDateTime*, and *Hour*, *Min*, *Sec*, and *Msec* are all of type Word.

The functionality of *DecodeDate* is just the opposite of *EncodeDate*. *DecodeDate* is a procedure that takes four parameters:

```
DecodeDate(DT, Year, Month, Day)
```

*DT* is of type *TDateTime*, and *Year*, *Month*, and *Day* are of type Word. *DecodeDate* is defined as:

```
DecodeDate(DT: TDateTime, var Year, Month, Day: Word);
```

As you've probably guessed, this function will disseminate a *TDateTime* structure and extract the year, month, and day as separate variables.

The *DecodeTime* function is similar to *DecodeDate*. *DecodeTime* extracts hours, minutes, seconds, and milliseconds from a *TDateTime* object and returns the values as four variables.

The program in Figure 5 also introduces the *DayOfWeek* function (see Listing Six, beginning on page 50). It takes a *TDateTime* and returns an integer that corresponds to the day of the week. It returns 1 through 7 in place of Sunday through Saturday.

This program obtains three values from the user (year, month, and day) and tries to convert that information into a *TDateTime* format. If it succeeds, the program displays the complete date using the *DateToStr* function, and the day of the week using the *DayOfWeek* function. The value returned by *DayOfWeek* is passed into the **case** statement and the appropriate string is displayed. If the conversion fails, an *EConvertError* exception is raised, and a message dialog box is generated.

## Converting to and from the DOS Date and Time Format

DOS often stores its date and time format as a Longint. Two bytes are used as the date and two bytes are used as the time. When interfacing with DOS, converting from one format to the other is often necessary. Delphi provides two

## Date and Time Combos

Delphi provides a number of variables for date and time formatting. Delphi automatically manages the variables *ShortDateFormat* and *LongDateFormat* to contain a date or time formatting string that is appropriate for your locale. For example, if Windows is configured for use in a European country, the date and time formats may be different than those for the United States. It's recommended that you use these pre-defined variables whenever possible because they will allow your program to automatically adapt for use in regions with different date and time formats.

In all cases, the format strings used should be enclosed within quotes.

**Date.** The date consists of the date, month, year, and day of the week. Figure A assumes the date is January 8, 1996.

| Specifier | Example | Description |
|---|---|---|
| d | 8 | Displays day without leading zero |
| dd | 08 | Displays day with leading zero |
| ddd | Mon | Displays day as abbreviation |
| dddd | Monday | Displays full day |
| ddddd | 1/8/96 | Displays date using short-date format |
| dddddd | Monday January 8, 1996 | Displays date using long-date format |
| m | 1 | Displays month without leading zero |
| mm | 01 | Displays month with leading zero |
| mmm | Jan | Displays month as abbreviation |
| mmmm | January | Displays month in full |
| yy | 96 | Displays year in two digits |
| yyyy | 1996 | Displays year in full |

**Figure A:** Formatting dates.

If the date must be displayed as Monday/08/01/96, the format statement is:

```
FormatDateTime('dddd/dd/mm/yy',MyDate)
```

The separator used ( / ) could be changed to ( - ) by changing the format string to:

```
'dddd-dd-mm-yy'
```

instead of:

```
'dddd/dd/mm/yy'
```

**Time.** Time consists of hours, minutes, and seconds. The time could also be formatted in several different ways. Figure B assumes the time in *MyTime* is 2:05:08.

| Specifier | Example | Description |
|---|---|---|
| h | 2 | Displays hour without leading zero |
| hh | 02 | Displays hour with leading zero |
| n | 5 | Displays minutes without leading zero |
| nn | 05 | Displays minutes with leading zero |
| s | 8 | Displays seconds without leading zero |
| ss | 08 | Displays seconds with leading zero |
| t | 2:05 AM | Displays time using short time format |
| tt | 02:05:08 AM | Displays time using long time format |
| am/pm | 02:05:08am | Uses 12-hour clock and inserts am/pm immediately after time string |
| a/p | 02:05:08a | Same as above but inserts a or p |

**Figure B:** Formatting time.

If the time must be displayed as (10-50-00-p), the format statement is:

```
FormatDateTime('hh-nn-ss-a/p',MyTime)
```

Notice that the separator can be changed by inserting any value between the format codes.

Delphi's online Help has a list of the available format strings. To reference them, search on "formatting data".

— *Shamiq Cader*

functions to achieve this. These are *FileDateToDateTime:*

```
FileDateToDateTime(DosVar : Longint) :
TDateTime
```

and *DateTimeToFileDate*:

```
DateTimeToFileDate(DT : TDateTime) :
Longint
```

*FileDateToDateTime* takes a Longint and returns the corresponding *TDateTime* conversion. This value can then be used within Delphi as the familiar *TDateTime* type.

*DateTimeToFileDate* takes a *TDateTime* type and returns a Longint. This Longint can then be used from within any DOS application as a regular DOS date/time type.

(Note that this article does not intend to show you how DOS stores date and time. This information is available in most DOS-related materials. The point is that Delphi provides two functions to convert the DOS date and time into a format Delphi recognizes, and vice versa.)

## Object Pascal Routines

Before concluding, let's discuss several other date/time routines provided by Object Pascal. These routines were left in Delphi to provide backward compatibility with previous versions of Pascal, and are still valuable.

The *GetDate* function corresponds to using the *Now* and *DecodeDate* functions. *DayofWeek* takes any value between 0 to 6 where 0 corresponds to Sunday:

```
procedure GetDate(var Year, Month, Day,
DayofWeek : Word);
```

The *GetTime* function corresponds to using the *Now* function followed by the *DecodeTime* function. Here *Sec100* corresponds to 100th of a second:

```
procedure GetTime(var Hour, Minute,
Second, Sec100 : Word);
```

| Function(s) | Description |
|---|---|
| *Now, Date, Time* | Retrieves system date/time or both |
| *DateToStr, TimeToStr* | Converts *TDateTime* to string |
| *StrToDate, StrToTime* | Converts string to *TDateTime* |
| *DecodeDate, DecodeTime* | Separates date/time into individual components |
| *EncodeDate, EncodeTime* | Assembles the *TDateTime* from the individual components |
| *DayOfWeek* | Returns day of week for given date |
| *FormatDateTime* | Formats date/time for output |
| *FileDateToDateTime, DateTimeToFileDate* | Converts to and from DOS date/time format |
| *GetTime, GetDate, SetTime, SetDate* | Object Pascal routines |

**Figure 6:** A summary of functions used in the demonstration programs.

The *SetDate* and *SetTime* functions can be used to set the operating system date and time:

```
procedure SetDate(Year, Month, Day : Word);

procedure SetTime(Hour, Minute, Second, Sec100 : Word);
```

The valid parameter ranges are:

- `Year1980-2099`
- `Month1-12`
- `Day1-31`
- `Hour0-23`
- `Minute0-59`
- `Sec0-59`
- `Sec1000-99`

Unlike the Delphi routines, if a parameter is invalid, no exception is raised — the request is simply ignored. For a summary of all the functions we've covered, see Figure 6.

## Conclusion

Confronting date and time issues in your Windows applications is not easy. (And this programmer's task will be *monstrous* as the year 2000 approaches. For more information on managing end-of-the-century software issues, see the sidebar "On December 31, 1999, at 11:59:59 PM ..." on page 48).

Fortunately, Delphi provides the tools necessary for your applications to handle date/time topics with ease. Δ

*The demonstration projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\NOV\DI9611SC.*

Shamiq Cader is a Software Engineer at Texas Engineering & Mechanical Co. in Galveston, TX. He has an MS in Computer Science from the University of Houston at Clear Lake and worked on designing and building FaxLink Enterprise, a Windows-based fax server. Released in May 1996, FaxLink Enterprise is written totally in Delphi. You can contact Shamiq at shameek@neosoft.com or shameek@juno.com.

## On December 31, 1999, at 11:59:59 PM ...

... you won't be kissing your sweetheart when the Big Apple drops in New York's Times Square. Instead, your palms will sweat, your brows will twitch, and you won't rest the weekend of January 1, 2000.

Why?

Because on Monday, January 3, 2000, several of your best software clients will complain about the incorrect dates in the programs you've built. (Consider this: if you have created a financial, numbers-crunching application based on a fiscal-year calendar, your clients should start becoming frustrated, oh, sometime around September 30, 1999.)

Most developers never thought their programs would live to see the end of the 20th century. As a result, many applications will behave badly when the date switches from 12/31/99 to 1/1/00. For example, some programmers may have only reserved two digits in a database field to hold the year number (e.g. 92). The programmer would then hard-code "19" in front of the date to display or print it accurately. However, when the year turns over to "00", we are suddenly thrown back to 1900 from the software's point of view.

One estimate states this problem will easily cost billions of dollars in software maintenance and lost revenue. Other reports add that it's quite likely consumers will receive telephone or cable bills for 100 years plus 30 days of service. Think of the irate customers you'd have to face. And all because you wanted to save two bytes per date field! Fortunately, this isn't an issue for Delphi applications, because the date/time fields and routines encapsulate a complete date in them.

For more information about issues facing developers as the 20th century ends, check out the Year2000 Web site at http://www/year2000.com/cgi-bin/clock.cgi (see Figure C). It contains links to articles about software problems and the year 2000, discussion groups, press releases, and 21st century FAQs. You can also find information on 1999/2000 solutions conferences and seminars for developers.

— *Robert Vivrette & Myrna Dingle-Gold*

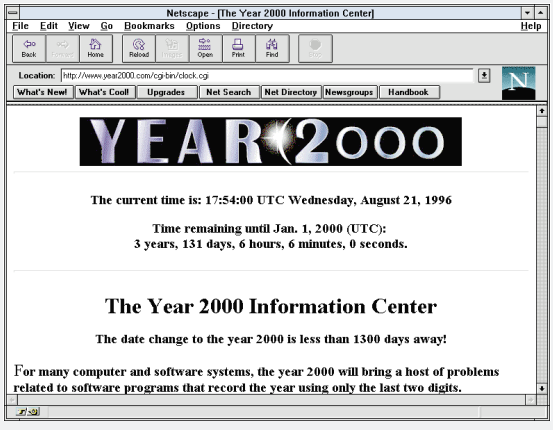



**Figure C:** The Year2000 Web site.

## Begin Listing Three — System Date and Time

```
{ Demo program to extract system date and time }
unit Fig1;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
var
  DateTime : TDateTime;
begin
  { Extracts the date and time }
  DateTime := Now;
  { Converts the date to a string }
  Edit1.Text := DateToStr(DateTime);
  { Converts the time to a string }
  Edit2.Text := TimeToStr(DateTime);
end;

end.
```

## End Listing Three

## Begin Listing Four — Add 15 Minutes Program

```
{ This program adds 15 minutes to the time entered by the
  user. It will also take care of the date if the time goes
  beyond midnight. Leap years are also handled. }
unit Fig2;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Edit4: TEdit;
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
var
  DateTime        : TDateTime;
  MyDate, MyTime  : TDateTime;
begin
  { Get the date entered by the user }
  MyDate := StrToDate(Edit2.Text);
  { Get the time entered by the user }
  MyTime := StrToTime(Edit1.Text);
  { Add them to get one TDateTime type }
  DateTime := MyDate + MyTime;
  { Adds 15 minutes to the time }
  DateTime := DateTime + (15/60/24);
  { Converts back to a string for displaying }
  Edit3.Text := TimeToStr(DateTime);
  Edit4.Text := DateToStr(DateTime);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  { Clears all edit boxes }
  Edit1.Text := '';
  Edit2.Text := '';
  Edit3.Text := '';
  Edit4.Text := '';
end;

end.
```

## End Listing Four

## Begin Listing Five — Formatting Date and Time Strings

```
{ Demonstation program that displays date and time in
  specific formats }
unit Fig3;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Edit4: TEdit;
    Label4: TLabel;
    Edit5: TEdit;
    Edit6: TEdit;
    Edit7: TEdit;
    Edit8: TEdit;
    Label5: TLabel;
    Label6: TLabel;
    Button1: TButton;
    Button2: TButton;
    procedure Button2Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);

  private
    { Private declarations }
  public

    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button2Click(Sender: TObject);
begin
  { Clear Edit boxes }
  Edit1.Text := '';
  Edit2.Text := '';
  Edit3.Text := '';
  Edit4.Text := '';
  Edit5.Text := '';
  Edit6.Text := '';
  Edit7.Text := '';
  Edit8.Text := '';
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  MyDate, MyTime : TDateTime;
begin
  MyDate := StrToDate(Edit1.Text);    { e.g. 01/08/96 }
  MyTime := StrToTime(Edit5.Text);    { e.g. 22:50 }
  { e.g. 1996-January-08 }
  Edit2.Text := FormatDateTime('yyyy-mmmm-dd',MyDate);
  { e.g. Jan,8,96,Mon }
  Edit3.Text := FormatDateTime('mmm,d,yy,ddd',MyDate);
  { e.g. Monday/08/01/96 }
  Edit4.Text := FormatDateTime('dddd/dd/mm/yy',MyDate);
  { e.g. 10:50 PM }
  Edit6.Text := FormatDateTime('t',MyTime);
  { e.g. 10:50:00 PM }
  Edit7.Text := FormatDateTime('tt',MyTime);
  { e.g. 10-50-00-p }
  Edit8.Text := FormatDateTime('hh-nn-ss-a/p',MyTime);
end;

end.
```

## End Listing Five

## Begin Listing Six — Day of Week

```
{ Demonstration program to demonstrate EncodeDate,
  DayOfWeek, and the exceptions that can be raised when
  converting to and from the TDateTime format }
unit Fig4;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)

    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Button1: TButton;
    Button2: TButton;
    Label5: TLabel;
    Edit4: TEdit;
    Label6: TLabel;
    Edit5: TEdit;
    procedure Button2Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public

    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button2Click(Sender: TObject);
begin
  { Clears the Edit Boxes }
  Edit1.Text := '';
```

```
  Edit2.Text := '';
  Edit3.Text := '';
  Edit4.Text := '';
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  DT        : TDateTime;
  ErrorFlag  : Boolean;
  DOW       : Integer;
begin
  ErrorFlag := FALSE;
  { Try to encode the year, month and day given by
    the user }
  try
    DT := EncodeDate(StrToInt(Edit3.Text),
                     StrToInt(Edit1.Text),
                     StrToInt(Edit2.Text));
    { Exception clause }
  except
    { If an error occurs set the flag }
    on EConvertError do ErrorFlag := True;
  end;

  if ErrorFlag then
    { Error flag was needed to catch the else part }
    ShowMessage('Invalid Date')
  else
    { If valid display date }
    Edit4.Text := DateToStr(DT);
  { Get DayOfWeek integer and pass it into
    the case statement }

  DOW := DayOfWeek(DT);
  { Match the string }
  case DOW of
    1 : Edit5.Text := 'Sunday';
    2 : Edit5.Text := 'Monday';
    3 : Edit5.Text := 'Tuesday';
    4 : Edit5.Text := 'Wednesday';
    5 : Edit5.Text := 'Thursday';
    6 : Edit5.Text := 'Friday';
    7 : Edit5.Text := 'Saturday';
  end;

end;

end.
```

**End Listing Six**

## *Teach Yourself Delphi 2 in 21 Days:* A Worthwhile Challenge

If you have programming experience and need a fast way to learn Delphi 2, you'll find what you need in *Teach Yourself Delphi 2 in 21 Days* by Dan Osier, Steve Grobman, and Steve Batson [SAMS, 1996]. Anything worth having is worth working for, and the authors intend to make you work. Written for the commercial developer who needs practical how-to knowledge, *Teach Yourself* will get you up to speed in no time (well, 21 days).

More than a simple update of *Teach Yourself Delphi in 21 Days* by Andrew Wozniewicz and Namir Shammas [SAMS, 1995], *Teach Yourself Delphi 2 in 21 Days* was completely rewritten, using new examples, covering more material, and moving at a faster pace than the first edition; the only similarity is the book's format.

As with other "21 Days" titles by SAMS, this release is well organized; it contains daily reviews and exercises, weekly overviews, and weekending retrospectives. The daily exercises are arranged in logical order, and can be easily managed in your spare time. *Teach Yourself* is a "build-it-as-you-go" book, as it should be. Don't expect a CD-ROM, you won't find one. You don't need one, either; the authors teach you Delphi 2 with short and snappy exercises. For the best results, work through all the examples. However, if you need a quick fix, you can always download the source from the MCP Forum on CompuServe (GO MACMILLAN).

Day 1 kicks off your three-week odyssey with an overview of Delphi, including discussions about RAD programming, the differences between Delphi 1 and 2, VCLs, variables, constants, procedures, functions, events ... you get the idea.

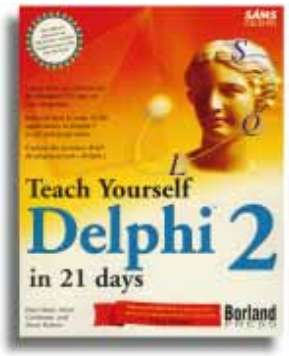
Day 2 covers the IDE, including the basics (Component Palette, Object Inspector, etc.), the Delphi menu structure, and customization. Days 3 and 4 introduce and address Object Pascal. Before you know it, you'll have a grasp of constants, variables, data types and structures, operators, etc. On Day 5 you learn what constitutes a Delphi application: projects, forms, units, and more. You'll enjoy learning the fundamentals of GUI design on Day 6. The week ends with lessons in object-oriented programming. Topics include the software life-cycle, software engineering, object-oriented design, and more. The exercises here could be better, but the authors' intent to foster good Windows applications is commendable.

The concise Week 1 review and Week 2 preview offer a good time to reflect on what you've learned, as well as prepare for the coming exercises.

*Teach Yourself* begins the second week with exceptions and events, and Day 9 provides a good primer on the Visual Component Library, possibly the most concise VCL introduction I've read. The next section introduces topics necessary to spice up your applications. Day 10 offers instruction about graphics; you'll sample the color palette, draw shapes, and learn to use PaintBoxes and Bitmaps. On Day 11 you're introduced to multimedia and animation. By day's end you'll know double-buffered animation techniques, and will have built a 3D spinning cube applet. Day 12 addresses the important and well-covered subject of file and directory management. This section teaches about text and binary files, as well as binary arithmetic, file attributes, and block reads.

No Delphi book would be complete without addressing databases. Day 13 covers databases, the Database Desktop, data-aware controls and grids, and accessing field values using Object Pascal. Calculated fields and searching by range are also addressed. You'll end the week with an introduction to SQL. By this time you should be starting to get serious about Delphi 2.

The final week begins with instruction on InterBase, and the use of transaction management, subqueries, joins, and stored procedures. Day 16 introduces the fundamentals of ReportSmith, and Day 17 teaches you how to print the reports you've just created. After all, what good




is a database if you can't format and print the reports?

The final days in this three-week tutorial include instruction on OLE 2, using and writing DLLs, writing your own visual components, and developing your own installation program. Your final exercise teaches you how to manipulate the Registry and qualify your applications for the Windows 95 logo. A review of Week 3 completes the 21-day training.

Every day of instruction ends with a summary, Q&A section, quiz, and set of appropriate exercises. All in all, the format lends itself to quickly teaching the basics of working in and with Delphi 2. You'll find *Teach Yourself Delphi 2 in 21 Days* a challenge, but well worth the effort.

— *James Callan*

*Teach Yourself Delphi 2 in 21 Days* by Dan Osier, Steve Grobman, and Steve Batson, SAMS Publishing, 1996, 201 West 103rd Street, Indianapolis, IN 46290, (800) 428-5331.

**ISBN:** 0-672-30863-0
**Price:** US$35.00
706 pages

# A Lexus with Hardwood Seats?

Developing a successful software application is not unlike engineering a quality automobile. The most respected automobile makers in the world — such as Lexus, BMW, and Volvo — are successful because of their ability to combine performance, reliability, and ergodynamics into a marketable package. So too, the most popular software applications combine these three qualities.

Unfortunately, this is far from typical of most software applications built today. While most developers focus on speed and stability, there is a tendency to overlook the final element of this triad: usability. This propensity to dismiss user interface (UI) issues as trivial is prevalent among developers and IT managers alike. And even if Lexus had a superior engine and unrivaled reliability, not many people would buy one if it had wooden seats and an AM radio. The same principle holds true for your application: no matter what is under the hood, the application will only be accepted if its UI is successful. This month we'll look at the Windows interface, focusing on emerging trends in interface design to keep in mind when creating Delphi applications.

**Similarity.** UIs are becoming increasingly similar. Windows applications look far more alike today than they did five years ago. At one time, it seemed that every major Windows software vendor had their own distinct look and feel, as did Borland with BWCC.DLL custom controls. Windows 95's new interface — along with Microsoft's rigid Win95 logo requirements — helped curb this trend. While different vendors still put their own twists on an application's UI, Windows 95 applications are remarkably similar.

The result is consistency across applications. As Charles Petzold once said, "Even a bad interface is good — so long as it's consistent among applications. I don't care what I have to do to invoke a File Open dialog box, or to navigate among the fields of the dialog, as long as once I learn how to do it, I don't ever have to learn something else." (*PC Magazine*)

**Organization.** UIs are becoming better organized in terms of controls, commands, and online Help. Windows controls certainly have evolved since Windows 3.0, which had little to offer. Windows 3.1 made several advancements, particularly with the tabbed dialog box. Windows 95, however, goes far beyond this interface metaphor with several new controls, the most significant of which is the Treeview. The Treeview control is significant because most applications deal with managing organized collections of data, and this data is often assembled hierarchically in the real world. Many types of database applications, for example, could use a Treeview control rather than relying on the ubiquitous table grid.

Commands are also better organized. In Windows 3.0, most commands were accessed either through a top-level menu or through keyboard shortcuts. By Windows 3.1, most applications employed toolbars and some even provided context-sensitive, pull-down menus. In Windows 95, users have the most flexible interface environment ever; for example, applications typically have multiple, dockable toolbars, enabling you to mix and match depending on your current tasks.

Furthermore, Help is more accessible. In the past, online Help was always held in a separate application window. Applications provided some degree of context sensitivity, but Windows 95 transformed this to bring Help into the application itself. For more information on how to use "What's This?"-style Help in your applications, I recommend an excellent resource named WHTHLP.TXT in the Delphi 2 forum (GO BDELPHI32) on CompuServe.

**Innovation vs. Consistency.** Creating a successful UI involves balancing innovation with consistency. A developer's goal should be to produce an innovative interface, but one that respects the boundaries of Windows 95 standards. Don't get uppity: the same truth applies to automobile makers. You may see Lexus come out with state-of-the-art stereo technology next year, but you won't see them replace the steering wheel or brake pedal with controls they believe work better.

— Richard Wagner

*Richard Wagner is Contributing Editor to* Delphi Informant *and Chief Technology Officer of Acadia Software in the Boston, MA area. He welcomes your comments at rwagner@acadians.com, or on the File | New home page at http://www.acadians.com/filenew/-filenew.htm.*