# What's Bred in the Bone

## Delphi and Object-Oriented Programming

**Cover Art By:** *Victor Kongkadee*

## ON THE COVER


8 **OOP for the Uninitiated** — Mark Ostroff
For many developers, Delphi is their first object-oriented programming environment. Mr Ostroff gets them on the right path with his introduction to OOP — the principles, the terminology, and the myths.

17 **The Triumph of Objects** — Zack Urlocker
Objects are now the *lingua franca* of programming. But it wasn't always this way! Mr Urlocker offers his perspective as Delphi Group Product Manager and reminisces about his first experience with OOP.


## FEATURES


19 **Informant Spotlight** — Brian Johnson
It's no secret that many programmers are leaving Visual Basic to begin new development projects in Delphi. Mr Johnson offers ten ideas — from declaring variables to getting help — to make the migration easier.

23 **The Way of Delphi** — Gary Entsminger
Mr Entsminger introduces Delphi exception handling, explains the **try...except** and **try...finally** Object Pascal constructs, and offers us a gem of a component for trapping run-time errors gracefully.

29 **DBNavigator** — Cary Jensen, Ph.D.
Whether you're working with Table or Query components, the *TField* object and its children are an integral part of Delphi database programming. As Dr Jensen points out, it's not always what you *see* that counts!

33 **From the Palette** — Jim Allen & Steve Teixeira
Misters Allen and Teixeira offer up a 3-D Label component that's sure to come in handy. Along the way, they just happen to demonstrate overriding a constructor, inheriting properties, the *Paint* method, and much more.

40 **At Your Fingertips** — David Rippy
It's more slick info from master-tipster Rippy! This month's bunch includes a one-statement incremental search, controlling DBGrid columns, and keeping your Windows color palettes from hurting each other.


## REVIEWS


42 **Conversion Assistant**
Product review by Gary Entsminger

46 ***Delphi Programming for Dummies***
Book review by Jerry Coffey

47 ***Design Patterns***
Book review by Richard Curzon


## DEPARTMENTS


2 **Editorial**

3 **Delphi Tools**

5 **Newsline**


magazine MENU

# Editorial

## *What's bred in the bone will not out of the flesh.*

English proverb from the Latin — 1290

**Okay, I should have done this last issue. We've received many comments about our lack of an editorial page, so here goes. Frankly, I didn't want to burn a page with a-word-from-the-editor, but it's great that you're interested in our plans for the magazine. This month's cover topic is object-oriented programming. A real stretch hunh? Which brings me to the "What's bred ..." quotation. Besides being a paean to good (and bad) breeding, the proverb applies remarkably well to Delphi's incorporation of OO principles. Delphi is object-oriented to the bone, so all Delphi programming is object-oriented programming.**

We begin with an introduction to OOP by Mark Ostroff. Mark does a great job of putting a new spin on the topic with some novel hardware analogies — and of dispelling some widely held OOP myths. Next, Delphi product manager Zack Urlocker shares some of the thinking that went into the development of Borland's new product and how it's intended to be used.

If I had a dollar for every time I've heard "I'm a Visual Basic developer, but ..." I'd probably be in Bangkok or Madras or Delphi instead of writing this. For those of you making the VB-to-Delphi hop, Brian Johnson offers 10 pointers for making the transition easier. And speaking of VB, renowned VB master and OOP specialist Gary Entsminger has come up with the cool idea of an exception-handling component. It's all in this month's "The Way of Delphi" which you can expect to see regularly in these pages. Paradox master Cary Jensen's "DBNavigator" column follows, where each month Cary will share his database skills.

Borland's own Jim Allen and Steve Teixeira then uncork a custom 3-D Label component in their "From the Palette" column. Besides being a useful component in its own right, it's a great demonstration of how to pick an ancestor component and then selectively accept, override, or add to the inherited behavior. It's a perfect fit for our OOP issue. David Rippy rounds out the featured articles with his bi-monthly column, "At Your Fingertips". David has a penchant for illustrating useful tips with brevity and humor, and this month's offering is no exception.

Probably the "hottest" third-party product right now is EarthTrek's Conversion Assistant which helps convert VB projects into Delphi projects. Gary Entsminger has structured his review in a way I think most programmers will like. Rather than run through a list of features, he presents us with a test conversion that demonstrates exactly what the Conversion Assistant does and does not convert.

So there you have it. Each issue will have a cover topic supported by two or three articles, and a sub-topic that usually takes the form of the "Informant Spotlight". With any luck we'll be able to create some synergy between articles — Brian Johnson's VB article and the Conversion Assistant review provide a good example. Each month we'll also feature the latest add-in products in our Delphi Tools section, and keep you informed regarding any Delphi-related news in our Newsline section. Finally, most issues will end with book reviews. We'll review books on a wide variety of topics, but make a special effort to review Delphi-specific titles.

*Delphi Informant* is a young magazine and we want to get it off on the right foot. Please let us know what you think. The response so far has been overwhelmingly positive, but you had many questions as well. Will the magazine be for beginning programmers? Advanced programmers? Will you have product reviews? Book reviews? Will you cover database and client/server topics?

The answer to all these questions is "Yes!" To use the vernacular, we want the decision to buy *Delphi Informant* to be a "no brainer." That is, if you're a Delphi developer, you'll want the magazine. That's the beauty of a product-specific magazine. *Delphi Informant* can serve the spectrum of Delphi users — from hobbyists to consultants to corporate developers to academia.

So how are we doing so far? Are we covering the topics you need to know about? Do you want more articles that address Object Pascal language basics? More or fewer database-related articles? I want to hear from you and the best way to reach me is via e-mail at CompuServe address 70304,3633. Article topics and comments about the magazine — positive and negative are welcome. And please let me know if you want an editorial page (i.e. *this* page). Hey! It could have been far worse. I could have run my picture and told you all about the last conference I attended.

Thanks for reading and I hope you enjoy *Delphi Informant*.

— *Jerry Coffey,* Editor-in-Chief

table of CONTENTS

Delphi TOOLS

New Products and Solutions

### Delphi Training

**The Data Guidance Company** is offering *Delphi training* in Minneapolis, MN this month. *Beginning Delphi* runs June 5 & 6, 1995 and costs US$495. *Advanced Delphi* is scheduled for June 7 & 8, 1995 for US$695. A discount price of US$1095 is available for those enrolled in both courses. For more information, or to enroll, call (612) 544-4219.

**The Coriolis Group** is offering a *Delphi Starter Kit* that contains the *Delphi Programming Explorer*, a 500+ page Delphi programming book by Jeff Duntemann; **EarthTrek's** *Conversion Assistant*, a tool that reads and converts Visual Basic project and program files into Delphi; and several custom controls. The kit sells for US$99.99, and includes a CD-ROM. For information, or to order, call (800) 410-0192 or (602) 483-0192

## Woll2Woll Software Announces Info*Power* 1.0

**Woll2Woll Software** of San Jose, CA has announced the release of **Info*Power* 1.0**, a set of data-aware Delphi VCL components for developers creating database front-ends with Delphi.

Info*Power* has several components that allow database application end-users to sort, filter, view, select, and edit information. According to the company, Info*Power* is composed of native Delphi components automatically linked into the compiled .EXE file, allowing developers to distribute and install a group of separate controls into the user's Windows directory without additional constraints.

Info*Power*'s features include: a Super Database Grid for creating a table grid that includes check boxes, combo boxes, a Super Database Grid for displaying multiple related tables; a Lookup Combobox for displaying one or more columns of data in a single, drop-down listbox; and an Advanced Filtering component for viewing specific data.

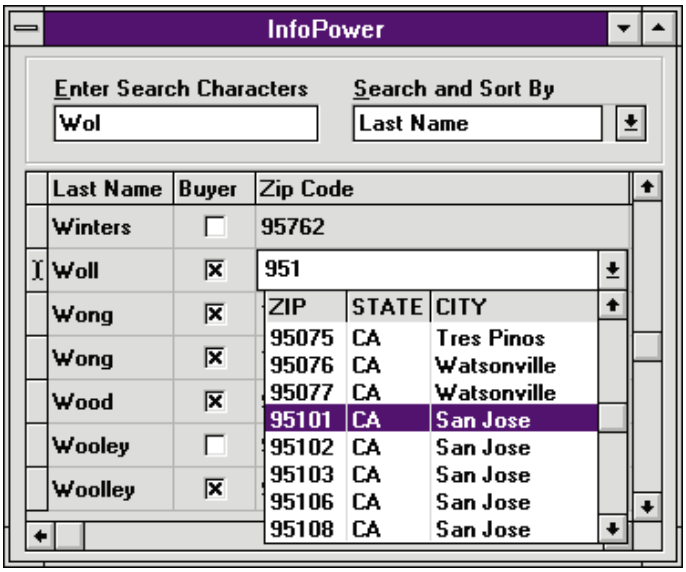
In addition, Info*Power* has a Sort component for selecting any primary or secondary index, an Auto-Expanding Memo component, and an Incremental Search component.

**Price:** US$149 until August 31, 1995 (price returns to US$199), includes a 30-day money-back guarantee.

**Contact:** Woll2Woll, 1032 Summerplace Drive, San Jose, CA 95122

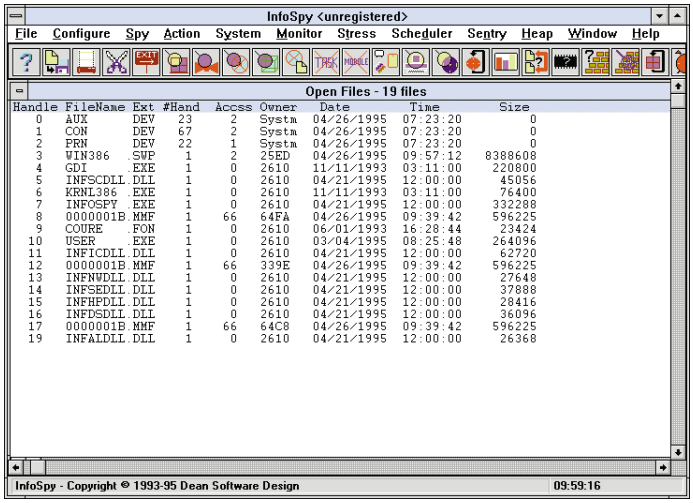
**Phone:** 1-800-WOL2WOL or (408) 293-9369

**Fax:** (408) 287-9374




## InfoSpy 2.3 for Delphi Developers

**Dean Software Design**, of Mill Creek, WA has released ***InfoSpy 2.3*** for Windows 3.1, Windows 95, and OS/2. InfoSpy is a Windows utility that provides spy, trace, monitoring, alarms, and scheduling functions in a single configurable MDI application. It can be hidden or set to require passwords to prevent changes to its configuration.




InfoSpy reports on heap, memory, DOS memory, Windows, tasks modules, classes, global atoms, multimedia, device drivers, DOS environment, open files and file handles, VxDs, Vms, CMOS, Version, timer information, and more.

Its screen capture capabilities will capture desktop, window, or cut a portion to the Clipboard or a .BMP file. Applications can be terminated and modules can be unloaded from the desktop. You can restart Windows — warm or cold boot — with a rapid shutdown option. Other functions include Compact Global Memory and the ability to prioritize tasks.

InfoSpy 2.3 is available on the *Delphi Informant* Companion Disk and for download from the Informant Bulletin Board. Library name: DSWINLIB. File name: ISPY230.ZIP

**Price:** US$19.99. Shareware version available on CompuServe's Delphi Forum, or on America On-Line's Windows Top Picks.

**Contact:** Dean Software Design, P.O. Box 13032, Mill Creek, WA 98082-1032

**Phone:** (206) 316-8645

**E-mail:** SteveDean@aol.com

Delphi TOOLS

New Products and Solutions

## New Delphi Books

**Delphi Programming Unleashed**
*By Charles Calvert*
Sams Publishing
ISBN: 0-672-30499-6
*Delphi Programming Unleashed* is written by a member of the Delphi Development Team. It contains tips and techniques for creating Delphi applications.
**Price:** US$36 (1000 pages)
**Phone:** (800) 428-5331

**teach yourself...Delphi**
*By Devra Hall*
MIS Press
ISBN: 1-55828-390-0
*teach yourself...Delphi* progresses from basic Delphi examples into a fully functional program. The book contains a diskette of custom controls and DLL functions by Sheridan Software Systems.
**Price:** US$27.95 (309 pages)
**Phone:** (800) 488-5233

# HyperAct Ships Pasterp 2.5

**HyperAct, Inc.** of Coralville, IA has released ***Pasterp 2.5***, an application script language for Delphi, C++, and Borland Pascal. It includes a function library, short examples, and demonstration projects. Pasterp supports object-oriented frameworks, dynamic dispatch functions, and arrays.

Pasterp can be linked directly to Delphi applications, with DLLs, VBXes, etc. It also has a small footprint of 150KB.

Used in Windows from Delphi, C++, or Borland Pascal applications, and in DOS from Borland Pascal real and protected mode applications, Pasterp can tie new functions, variables, and constants to your applications. It can also be used as a DLL and is compatible with event-driven applications.

In addition, Pasterp can be implemented as an object hierarchy that can be derived to provide application-specific functionality. It can create user configurable and extendible applications, automated tasks, scriptable macros, and expression evaluation.

Pasterp has no royalty fees. A demonstration copy is available from CompuServe's Delphi Forum, the *Delphi Informant* Companion Disk, and for download from the Informant Bulletin Board. Library name: DIWINLIB. File name:PTRPDEMO.ZIP.

**Price:** US$200; Pasterp with Object Pascal source code, US$800. (Prices do not include shipping and handling fees.)

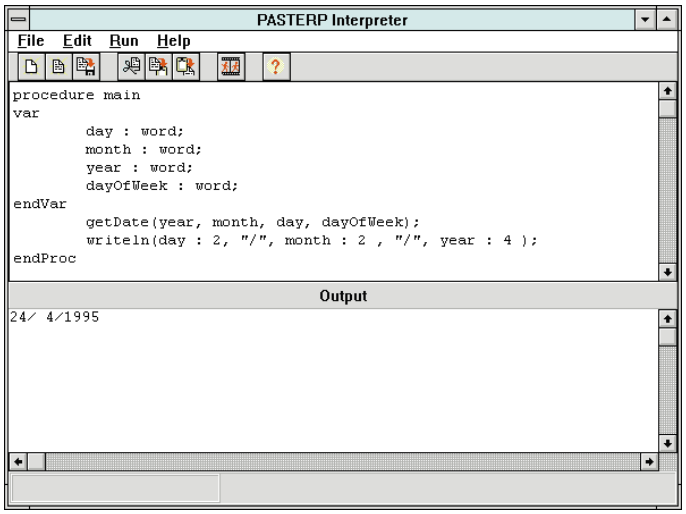
**Contact:** HyperAct, Inc. P.O. Box 5517 Coralville, IA 52241

**Phone & Fax:** (319) 351-8413

**CompuServe:** 76350,333

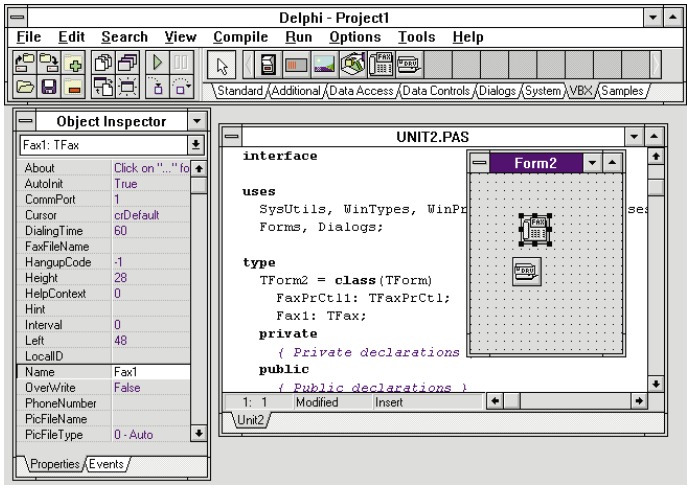
**Internet:** rhalevi@hyperact.com

**Internet Home Page:** http://www.hyperact.com/hyperact.html.




# Add Faxing to Delphi Applications

**MicroHelp Inc.** of Marietta, GA has released ***Fax Plus***, a control for adding fax sending and receiving capabilities to an application. It can use almost any Class 1, Class 2, or Class 2.0 fax modem (including many CAS-compatible modems).




Fax Plus allows users to send faxes from ASCII text, .BMP, .PCX, and .DCX file formats. Using Fax Plus' custom printer driver, users can also capture the output of any Windows application that supports printing, and then send that output as a fax.

Fax Plus is customizable. For example you can control the status display used when sending and receiving faxes by using those provided with the demonstration program, or by designing your own. Fax Plus provides low-level control over your fax or modem. Users can design phone-book databases, cover sheets, fax file archives, etc. Fax Plus includes a 100-page fully-indexed manual, a Windows help file, and free, full-time technical support.

There are no run-time royalties when distributing single user applications. Contact MicroHelp for more information about network "fax server" pricing.

**Price:** US$249, plus shipping and handling.

**Contact:** MicroHelp Inc., 4359 Shallowford Industrial Parkway, Marietta, GA 30068

**Phone:** (800) 777-3322

**Fax:** (404) 516-1099

**BBS:** (404) 516-1497

table of CONTENTS

# News

L I N E

June 1995

Starfish Software has released ***Sidekick Deluxe***, a multimedia CD-ROM version of Sidekick 2.0 for Windows. It includes over 40 content files called Sidekick Companions plus Dashboard 3.0, a personal utility for Windows. In addition, Sidekick Deluxe contains a video on getting organized with the Starfish 5-point program, featuring Starfish Software Chairman Philippe Kahn. According to the company, Sidekick Deluxe will be on store shelves nationwide by Memorial Day.

Object Management Group, co-sponsor of Object World, and Computerworld are seeking submissions for the ***Computerworld Third Annual Object Application Awards***. Winners will be featured in *Computerworld* and announced at Object World/San Francisco on Wednesday, August 16, 1995. The awards showcase innovative custom applications using object technology. Applications must be currently in use, and meet one of the following qualifications: built from scratch; a modification of an off-the-shelf application; or an object-oriented front-end for a host application. All entries must be postmarked by midnight, EST, May 16, 1995. All entrants are required to complete an official entry kit. For information call (508) 820-4300, or fax (508) 820-4303.

## Delphi World Tour

*Addison, IL* — Softbite International, Borland International, and Informant Communications Group have announced the 1995 Delphi World Tour. The two-day seminar will take place in Columbus, Boston, Los Angeles, Philadelphia, Chicago, Seattle, Dallas, New York, Atlanta, San Francisco, Washington DC, Denver, Minneapolis, Orlando, Phoenix, Detroit, Houston, Raleigh/Charlotte. International stops are also planned, but details weren't available at press time.

DELPHI WORLD TOUR

dates

| | | | |
|---|---|---|---|
| Columbus | July 6-7 | Orlando | September 14-15 |
| Boston | July 10-11 | Phoenix | September 18-19 |
| Los Angeles | July 19-20 | Detroit | September 21-22 |
| Philadelphia | July 25-26 | Houston | September 25-26 |
| Chicago | July 31-Aug 1 | Raleigh/ Charlotte | September 28-29 |
| Seattle | August 2-3 | | |
| Dallas | August 15-16 | Calgary | TBA |
| New York | August 17-18 | Toronto | TBA |
| Atlanta | August 21-22 | Sydney | TBA |
| San Francisco | August 24-25 | London | TBA |
| Washington DC | August 28-29 | Amsterdam | TBA |
| Denver | September 7-8 | Denmark | TBA |
| Minneapolis | September 11-12 | | |

The Delphi World Tour is a 2-day event. Attendees can attend both days or only one day. Day One, "The Main Event," will show how Delphi can be used to build stand-alone, LAN, and Client/Server applications. Components, form design, event handling, properties, database access, programming, and other aspects of Delphi development will be covered. Client/Server development in Delphi will also be introduced.

Day Two, "Delphi Extended and Delphi Client/Server," will cover Delphi development in more detail, including programming common tasks, accessing data via the Borland Database Engine, reporting with ReportSmith, writing and using VBX and stand-alone DLLs, and a detailed look at Client/Server development using Delphi.

Delphi World Tour attendees will receive a free copy of Delphi, a free starter subscription (three issues) to *Delphi Informant*, documentation and diskette containing Delphi code, a free copy of *Delphi Informant* at the event, and a copy of the *Delphi Power Tools Catalog* (Summer 1995).

In addition, attendees can meet with local third-party consultants, trainers, and book vendors during the "Delphi Networking Lunch."

Pricing for the Delphi World Tour is US$545. Those attending only "The Main Event" pay US$295. Discounts are available for three or more attending from the same company.

To receive a complete brochure via fax, dial (708) 833-9122 from a fax machine, and request document number 5. The brochure can also be requested by calling Softbite International at (708) 833-0006, or via Internet at register@softbite.-mhs.compuserve.com.

## Borland Rolls Out Delphi Client/Server Bundle

*Scotts Valley, CA* —Borland International, Inc. has announced it will offer a special Delphi Client/Server Bundle. This package includes five Delphi Client/Server units, maintenance contracts for one year, and a choice of customized technical support options.

According to Borland, the bundle offers one-year maintenance contracts that include a 32-bit release for Windows 95. This release will be fully compatible with the current version so that applications can easily be recompiled for Windows 95; (the release is scheduled to ship shortly after the commercial release of Windows 95).

The Delphi Client/Server Bundle provides two unique support options: an unlimited number of technical support calls for two designated corporate contacts for one year, or a 45-hour block of technical support time to be used by an unlimited number of corporate contacts.

The Delphi Client/Server Bundle's retail value is US$17,499.75, but through June 30, the package price is US$15,000 through Borland major account resellers: Egghead Software, Software Spectrum, Softmart, Corporate Software, and ASAP Software. The bundle is also available directly from Borland, by calling (408) 431-1064.

# News LINE

June 1995

**Lotus Development Corp.** has announced they will ask the US Supreme Court to review the First US Circuit Court of Appeals' decision to clear **Borland International** of copyright infringement. A Lotus spokesperson said the company would file its petition with the Supreme Court this month. The First US Circuit Court of Appeals recently reversed the District Court ruling that stated the Quattro and Quattro Pro spreadsheet products, formerly developed and marketed by Borland, infringed the copyright of Lotus 1-2-3. According to Lotus, this current ruling would promote plagiarism. Borland spokesman Steve Grady was confident that Lotus wouldn't convince the Supreme Court to rule in their favor. He said that even if the Supreme Court overturned the current decision, the case would still go to the appellate court for a hearing on Borland's defense. Lotus was expected to request US$100 million in damages — an amount that would have put Borland out of business.

The board of directors for the **Object Management Group (OMG)** has decided to address the need for industry specification in vertical application domains. They have chartered a committee to recommend a formalized process with supporting organizational structure to be instituted in 1995. OMG members work to develop and use integrated software systems, and believe that the object-oriented approach best supports these efforts. For more information contact OMG at (508) 820-4300.

## Informant Announces CompuServe Forum

*Elk Grove, CA* — Informant Communications Group, publisher of *Paradox Informant* and *Delphi Informant* has announced the company will go on-line with a new Forum on the CompuServe Information Service. The Forum will be available to the general public by early June 1995.

The Informant Forum will include message areas for both Paradox and Delphi users to exchange technical information, ask questions, and get technical help with each product. In addition, the Forum will contain code listings and support files appearing in each Informant magazine.

"We're extremely excited about being associated with CompuServe and having the Informant Forum" said Mitchell Koulouris, Publisher of *Paradox Informant* and *Delphi Informant*. "The Informant Forum allows us not only to offer superior service to our customers, but to get feedback from our readership so that we can make continuous enhancements to our magazines. We've been besieged by readers to provide a CompuServe Forum and we are proud to deliver this to our readers. We listen to our readers and the Informant Forum provides an excellent vehicle to get the important feedback we need to continue to improve our products on an ongoing basis."

Visitors to the Informant Forum will also be able to order back issues, place subscription orders, renew existing subscriptions, and inquire about advertising. In addition, the Forum will feature important figures in the Paradox and Delphi development communities in Conferences, allowing readers to ask questions on-line. There will also be areas for third-party vendors to provide support for their tools and services. The Library areas will include all code listings and support files appearing in each issue of the magazine, as well as third-party demonstration software and shareware.

Informant readers can obtain a free CompuServe starter kit that includes the latest version of WinCim and a US$15 usage credit by contacting Informant Communications Group at (916) 686-6610. To access the Informant Forum, type "GO ICG" from the CompuServe command prompt.

## Brainstorm Technologies Discloses Plans for Partnership with Borland

*Boston, MA* — Brainstorm Technologies Inc. has revealed the terms of a world-wide joint agreement with Borland. Brainstorm said it's exploring ways to expand to include products that integrate Lotus Notes across Borland's PC databases, programming languages, and development tools.

Initially, this agreement gives Borland exclusive rights to market Brainstorm's Delphi/Link for Lotus Notes software through Borland's channels under the Brainstorm brandname.

Current generation tools have supported Notes on an "ad-hoc/patch" basis. Brainstorm is planning to release several solutions in the next few months, providing Notes end-users with access to more tools, databases, and applications.

According to Mitchell Liu, Chief Technology Officer of Brainstorm Technologies, this agreement allows existing Borland client/server and database developers to architect enterprise-wide industrial strength Lotus Notes-enabled applications.

## New RAD Pack for Delphi

*San Francisco, CA* — Borland International Inc. has announced the release of the RAD Pack for Delphi, a powerful new companion tool set that combines many of Borland's products into one package.

The RAD Pack includes the Visual Component Library source code and the Resource Workshop for extracting and modifying standard Windows resources, such as icons, cursors, bitmaps, and dialog boxes. It also features a Resource Expert that converts standard resource scripts into Delphi forms, a Delphi Language Reference Guide, and the Turbo Debugger.

In addition the RAD Pack contains a Visual Solutions Pack with a collection of VBX custom controls including spreadsheet control, WYSIWYG word processors, asynchronous communications, image editors, and gadgets.

The RAD Pack will be available this month through major resellers or directly from Borland at a suggested retail price of US$249.95. For more information, call Borland at (800) 453-3375, extension 1309.

## Kahn Sends Open Letter to Lotus and Microsoft

*Scotts Valley, CA* — Philippe Kahn, CEO of Starfish Software and Chairman of Borland International Inc. recently posted an open letter to Lotus Development's Jim Manzi and Microsoft's Bill Gates. This letter, which was published in the Wall Street Journal in March, encouraged both Manzi and Gates to invest less in their legal departments and more in their research and development teams. According to Kahn, "a bit of creativity can save us from turning the software industry into a legal playground."

In the letter Kahn criticized Manzi for accusing Gates of anti-competitive practices. He wrote, "Let me refresh your memory. Over four years ago, you unfairly attacked Borland by alleging copyright infringement. You knew, like all of us, that systems and functionality cannot be copyrighted. Nevertheless, you relentlessly tried to put our company out of business. We had the best technology, but you decided to beat us in court, and not compete in the marketplace .... Unfairly you destabilized our company: you put doubt in the minds of our customers, you questioned our viability, you manipulated public opinion right up to the day when the Court of Appeals issued its opinion."

Kahn then turned the spotlight on Gates, asking him not to "stick it" to the industry. He asked Gates to use his position of leadership to "foster industry practices that will help the software industry grow" and assure customers that the software industry will remain fair and competitive.

Even in portions of the letter directed to Gates, Kahn kept Manzi's role in focus. He said Manzi's unfair competitive practices certainly helped Microsoft — and all Borland's competitors flourished as Manzi relentlessly attacked Borland's reputation.

In closing, Kahn reiterated Gates' leadership role with the release of Windows 95. He asked Gates to "take the high road" by making Windows 95 a level playing field where all can fairly compete. Kahn wrote, "We'll all get behind you and we'll all put more creativity to work as we continue developing the best software in the world. Out of courts of law and government intervention!"

## Microsoft's SQL Server 6.0's Planned Release

*Redmond, WA* — Microsoft Corp. recently announced that Microsoft SQL Server 6.0 is scheduled to ship this month. Microsoft SQL Server 6.0, code-named SQL Server 95, includes an all-new database technology designed for customers who are deploying distributed client/server systems.

The new version of Microsoft SQL Server features built-in data replication, parallel architecture enhancements, a centralized console, distributed management objects, and improved programmability and adherence to standards like ANSI SQL.

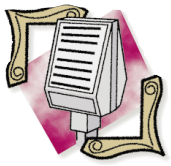
In addition, customers who license Microsoft SQL Server version 4.21a for Windows NT after March 15, 1995 will receive a free upgrade to Microsoft SQL Server 6.0 when it's available. Registered users of Microsoft SQL Server for OS/2 can also upgrade to SQL Server version 4.21a for Windows NT and receive a free upgrade to SQL Server 6.0 when it's released. After April 1, 1995, Microsoft SQL Server for OS/2 will no longer be available as a retail packaged product. It will be available through the Microsoft Select volume license program through June 30, 1995. However, Microsoft will continue to offer technical support to existing SQL Server for OS/2 customers.

In April, a beta version of Microsoft SQL Server 6.0 was made available to more than 2,500 customer sites worldwide. These sites included customers, developers, VARs, and Solution Providers. It was the fourth beta release of the Microsoft SQL Server 6.0 software, which has been undergoing testing at 250 sites since October 1994.

The scope of the release makes it one of the largest beta programs ever conducted in the database industry. The beta was feature-complete and was provided to customers to help them evaluate new functionality and plan migration strategies.

# News LINE

June 1995



IDG World Expo, Computerworld, and the Object Management Group announced the winners of ***Computerworlds's Second Annual Best New Object Technology Product Awards*** held at Object World Boston March 19-23, 1995. The Industry Judge Awards went to:

Best New CORBA-Based Product
NetLinks for the *ORBitize V1.1*

Best New OT Development Product
IBM for *VisualAge of C++ Version 3*
Rational Software Corporation for
*Rational Rose Family Version 2.7*

Best New Component or Library Product
Cadre Technologies for the
*ObjectTeam Application Factory*

The Attendees Choice for Best New Overall OT Product was given to IBM for the *VisualAge C++ Version 3*.

Boston University's Center for Information Technology is again sponsoring the ***International Developers Conference for Windows*** June 12 - 16, 1995. The 5th annual event features General Win32 Programming, Advanced Topics in Win32 Programming, Special Interest Windows Programming Topics, OLE Programming, Programming Windows Using C++ (MFC/OWL), and the Designer's Workshops. For more information, call (508) 649-4200 or fax (508) 649-2162.

table of CONTENTS




*By Mark Ostroff*

# OOP for the Uninitiated

## Fundamentals of an Object-Oriented Environment

A large part of Delphi's power comes from its robust support of object-oriented programming, or OOP. This article will introduce the basic concepts underlying OOP to an audience who is unfamiliar with object orientation. If you've heard all the talk about OOP and are wondering what it's about, this is the discussion you've been looking for.

### A Hardware Analogy

You already use object orientation without realizing it — in your PC hardware. Let's start with a familiar hardware example — that of upgrading the video board in your PC (see Figure 1.) The video board is an object. You really don't care how it works. You're just interested in the results. Because a standard interface has been defined for all video board "objects," you can upgrade easily. You don't need to replace the entire PC or become an electronics engineer and design your own board. You simply replace the old video board with a better one.

Another advantage of this type of *component architecture* is your ability to share components with other users. These components are reusable. For example, once you've upgraded your PC video board, that same standard interface allows you to pass your old video board to a co-worker. Thus, a single upgrade can have a "ripple effect" throughout various parts of an organization.

A third advantage of components involves the amount of expertise required. Using a collection of standard components, you can build functional items without having to know how to build the individual parts. The choice to "build or buy" is now available to you.

Finally, components provide easy support for maintenance. If a problem is discovered with your video board, you need only replace the video board, and not your entire PC.

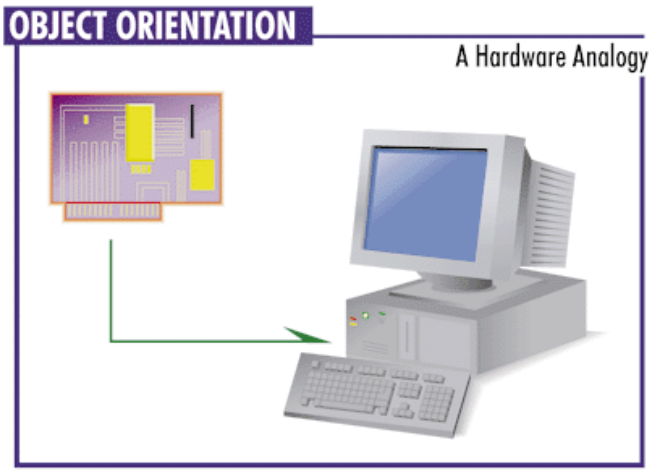



**Figure 1:** Hardware components (objects) in use.

This "plug and play" concept of creating reusable components is not new. It's the basis for the entire Industrial Revolution. What is new is the ability to apply these concepts to the world of software programming. The creation of reusable software components or "objects" is what OOP is all about.

## Foundation Concepts of OOP

OOP is based on four fundamental concepts. The first three concepts are well known and the technical terms for these concepts are *encapsulation*, *inheritance*, and *polymorphism*. The fourth concept, a division of programmer talent, involves leveraging the talents of people across your entire organization (more on this later).

## Encapsulation

The first OOP concept is known as *encapsulation*. The idea is that data structures and code routines are bound together, or encapsulated, into a single entity called an *object*. The data structures of an object are known as *properties* and the object's code routines are called *methods*. All access to the object's properties and methods must go through the object's interface (see Figure 2).

Relating encapsulation to our hardware analogy, a video board encapsulates all the I/O subroutines, internal data caches, and minute details of how to make a collection of dots appear on the screen. This one object embodies all those details so that the board's user doesn't have to be concerned with internal details. We just use the *API* (Application Programming Interface) of the board. For example, we simply want to say "draw a circle" and have it happen. We don't really care about how the computer accomplishes this task.

In the world of software, what are the main advantages of object encapsulation? Most programming problems arise from the issues of code re-entrancy and data access concurrency. Encapsulation eliminates these issues as sources of programming glitches. Since similar objects can be created that use the same code routines encapsulated within themselves, all program code is re-entrant. Object encapsulation protects data access in such a way that concurrency never becomes an issue.

## Encapsulation vs. Structured Programming

Before OOP, standard coding methodology involved a concept known as *structured programming*. Structured programming provided a clearly defined way for packaging code modules into subroutines, and for defining the way each code routine interfaced with each piece of code.

Structured programming is limited because it provides no way to structure the data used by an application. Therefore, although the code was structured, it sat atop a vast sea of unstructured data (see Figure 3). The programmer was responsible for insuring that each routine treated this data sea in a well-behaved manner.

This lack of data access structure led to a common problem — the ill-behaved routine. Since only the code was structured, an errant pointer or a memory glitch could send a routine off modifying data that other routines were using. This often led to mysterious and disastrous side effects (see Figure 4). For example, have you ever fixed one routine only to find that your "fix" broke something else that was totally unrelated?

A more common example of this in the software world is the improper set-up of *Expanded Memory Specification* (EMS) mem-

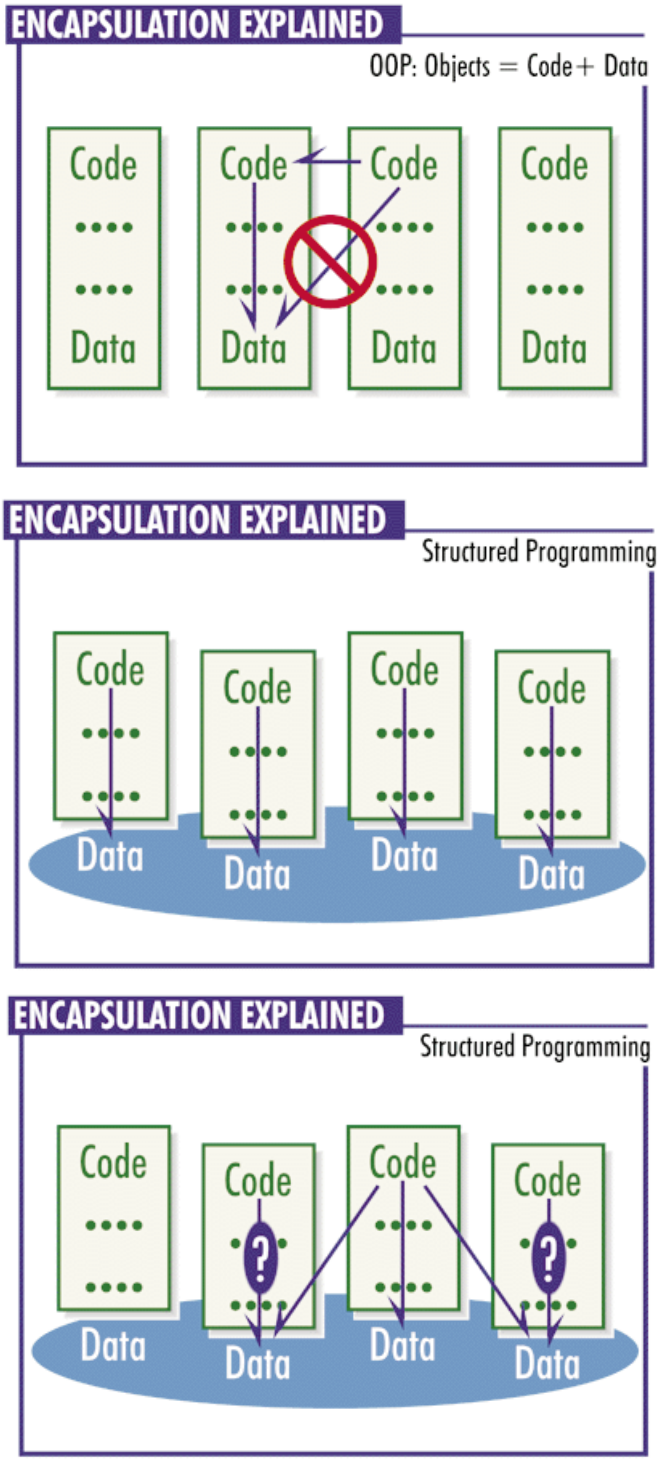



**Figure 2 (Top):** Encapsulation allows the creation of software components. **Figure 3 (Middle):** Structured code sitting on a sea of unstructured data. **Figure 4 (Bottom):** This schematic of encapsulation explains why structured programming is not enough to ensure that a routine will perform to expectations. Encapsulation allows the creation of software components.

ory managers. If your system has a video board with 1MB of RAM, the EMS driver must be told explicitly not to use that extra video RAM for EMS page framing. (In most cases, you must tell the EMS manager to exclude the memory locations C000-CFFF from use.) If this is not done, the system may run fine for awhile. When a software package that heavily uses video is invoked, both the graphic software and the EMS driver may try to use the same memory locations for different purposes.

First, let's say the graphics package draws a rich image on the screen. It stores video information in the C000-CFFF range. Then, the EMS manager looks at the same range of data. Unfortunately, the EMS manager thinks this information is memory page frame data and tries to jump to a non-existent "memory location" (since the data is really screen information, not address data). This often results in "strange" system crashes. Thus, protection from unwanted side effects is a major benefit of encapsulation.

You can actually see this concept of encapsulation in Delphi's user-interface as you build an application. When you look at the Object Inspector, you see a two-tab paged display (see Figure 5).

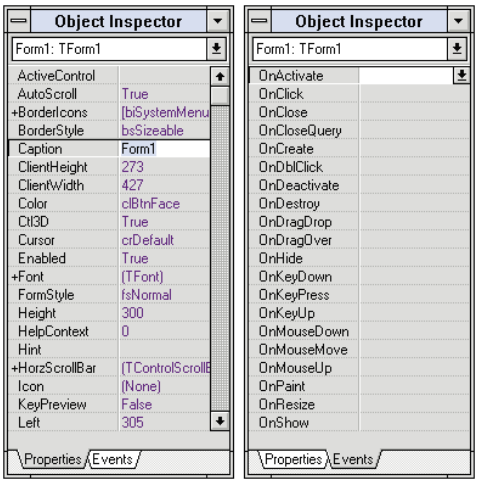



**Figure 5:** Delphi's visual display of encapsulation.

The visible API of each object includes a set of data values called *Properties* and a set of possible code routines labeled *Events* (or event handlers). As you select different types of objects, different properties and events appear in the Object Inspector.

## Beyond Strong Data Typing and Scoping

Encapsulation provides the structure needed to protect the data each routine uses from unexpected use by other routines. This structure is the mechanism whereby only methods that belong to an object can modify the data within that object. Methods from any other object must make calls to the object's methods to view and modify the properties in that object. (See Figure 2 for an illustration.)

Often, programmers say they can accomplish the same results that OOP provides simply by using strong data typing, variable scoping, and careful coding. However, using strong data typing and careful variable scoping does not afford the same level of data protection. Strong typing only identifies what type of information can be stored in a variable. It can only tell you what *doesn't* belong somewhere. Furthermore, it offers no information regarding which code modules should be allowed access to the data.

Variable scoping provides some amount of access control, but leaves many loopholes in providing data protection. For example, the documentation for many add-on code libraries includes statements such as, "Before using this library, make sure you define the following variables" or "This library reserves the following variable names for its own use." What prevents someone from using these reserved names? Faith!

Often, these variables need to be defined as globals, and that is one area where problems arise. An example from the real world is withdrawing cash from the bank. We all hate waiting in lines, so why not let people simply go behind the counter and process their transactions? Because that simply wouldn't be safe. What would prevent someone from taking cash from your account, or from withdrawing more money than they had? Therefore an intelligent agent is needed behind the counter.

In a bank, we need access to certain routines to be restricted. The same idea applies to application development. Yet, structured programming offers no way to hide program *code* modules from each other. OOP provides a way of creating scoping rules for your application's code as well as for your variables.

Let's return to the bank teller analogy. The idea of variable scoping can be seen in controlling account information. For example, I can only look at the information in my accounts and you can only see your accounts' data. Code scoping can be seen in controlling how that data is accessed and modified. I have no idea what internal procedures the bank teller performs to post a deposit to my account. That routine is "scoped" as being private to the "bank teller object."

In the same way, Delphi provides both data and method scoping capabilities. How does this work to provide added protection to your application? Let's say you need to create an object that has a property that you want to make sure is never overwritten. You want other objects to be able to see this property's value, but not have the ability to change it. The code sample in Figure 6 shows how to define such a "global read-only" property using Delphi's encapsulation and scoping constructs.

To see the effects of this VCL class, create a form with a *TRODemo* object and *TButton* object. Add the following code to the *TButton's OnClick* event, run the form, and click on **Button1**.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  RODemo1.Caption := RODemo1.GetROProp;
end;
```

Simply put, the code declares the property to be private to the object class (and thus inaccessible from direct manipulation by other objects). The code also defines a public method for reading that "invisible" property. Other objects can read the property, but only by using the public method defined.

That's what encapsulation is all about. Your application uses multiple co-equal intelligent agents to get the job done. Thus, encapsulation is a safer way to structure both your code and your data.

## Defining Some OOP Terms

Much of the confusion experienced by someone new to OOP is due to the huge number of new technical terms. Unfortunately, many of these terms seem to be defined in almost the same way. The subtleties of the differences can elude even the most technically-aware OOP neophyte. Problems in coding can arise when the distinctions between the following terms are not kept clearly in mind.

**Class:** A *class* or *class definition* is the blueprint of the OOP world. It determines what the defined methods and properties will be for all objects created from that class. Just as the blueprint for a Chevy Nova is not an actual car itself, so the class is not an object itself. It merely defines how to build an object, and what its capabilities will be once it's created.

**Object:** An *object* is an actual entity that is created at run-time and lives in active program memory. An object is created from a class definition that determines what methods and properties it has, and can also define what the initial values for those properties are.

Once an object is created, it no longer refers back to the class from which it was created. Thus, any run-time changes made to an object have *no* effect on any other objects created from the same class. (In the same way adding a ski rack to our Nova has no effect on any other cars made by GM.)

**Instantiation:** *Instantiation* is the name given to the process of creating an in-memory object from a class definition. This process creates an *instance* of the class, hence the name.

**Constructor:** Each class definition will have a section of code that tells the system how to instantiate an object of that class. This instantiation code is called the *constructor*. The constructor is called to allocate memory for an object instance, initialize its properties, and perform any other "startup code" the object may require. Delphi's components use the *Create* method as their constructor.

**Destructor:** Just as a class definition needs a constructor method, it also needs a section of code that controls how to remove an object instance from memory. This "clean up" code is referred to as the class' *destructor*. It should be defined to perform any "shut down" code needed, and then release all resources associated with a particular object instance. The use of destructors is vitally important to the issue of resource management. Delphi components use the *Destroy* method as their destructor.

**Ancestor Class:** An ancestor class is a class from which property and method definitions are inherited. This ancestor class may not be the immediate parent from which a child class was defined. It may be higher up on the inheritance hierarchy. For example, in Figure 9, *TControl* is an ancestor of *TRODemo*, but is not its immediate parent.

**Parent Class:** A parent class is the immediate class from which property and method definitions are inherited to create a new child class definition. Thus, "Automobile" would be the parent class for our Chevy Nova, "Shape" is the parent class for our *Rectangle* class, and *TLabel* is the parent class for *TRODemo*.

**Descendent Class:** Any class that has been defined based on inheritance from another class is known as a *descendent* class of that ancestor. The terms *child* class and descendent class are usually used interchangeably. A more strict definition would be that a descendant class is any class that inherits from an ancestor, and may or may not be the *immediate* child of that ancestor. In other words, parent and child classes describe immediate inheritance relationships, while ancestor and descendant are their more general corollaries.

**Sub-Classing:** The process of creating a descendent class definition through inheritance from a parent class is called *sub-classing*. Sub-classing only has meaning in the context of the methods and properties inherited from the parent class. The child class thus created will of course have additional properties and methods defined, or will redefine one or more of the parent's methods (i.e. use polymorphism), or both. After all, the whole idea behind sub-classing is to create a new class from the building blocks of an existing one.

**Single Inheritance / Multiple Inheritance:** Some OOP tools allow you to create a class definition that inherits from more than one parent class. This ability to have more than one parent class is called *multiple inheritance*. C++ tools typically support this type of inheritance.

However, most OOP tools (including Delphi) only allow inheritance from a single parent. These tools are said to support *single inheritance*. Since there are enough complexities in creating custom classes already, programmers just starting to define custom classes should limit themselves to the single inheritance model.

```
unit RODemo;

interface

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

type
  TRODemo = class(TLabel)

private
  { Private declarations - visible only to this class }
  ROProp : String;
  procedure SetROProp(ROValue: String);

protected
  { Protected declarations - visible to this class and
    its decendants }

public
  { Public declarations - visible to all classes }
  constructor Create(AOwner: TComponent); override;
  function GetROProp : String;

published
  { Published declarations - visible to all classes,
    and displayed in the Delphi Object Inspector }

end;

procedure Register;

implementation

constructor TRODemo.Create(AOwner: TComponent);
begin
  { Call the parent class' constructor }
  inherited Create(AOwner);
  { Set the default value of the R/O property }
  SetROProp('This value is scoped to be hidden');
end;

procedure TRODemo.SetROProp(ROValue: String);
begin
  ROProp := ROValue;
end;

function TRODemo.GetROProp : String;
begin
  Result := ROProp;
end;

procedure Register;
begin
  RegisterComponents('Samples', [TRODemo]);
end;

end.
```

**Figure 6:** RODEMO.PAS — using encapsulation to create a global *ReadOnly* property.

## Inheritance

The second OOP building block is called inheritance. Just like biological inheritance, OOP inheritance allows a programmer to create child object definitions (called *object classes*) that inherit the methods and data structures of the parent object class.

With inheritance, a programmer can simply use the inherited methods and properties from the parent class without having to recreate them from scratch. The programmer then simply codes only the differences between the parent and child objects. (This process is known as *programming by exception*.) The developer's time is then free to concentrate on creating methods and properties that are unique to the child object. Thus, inheritance allows developers to write less code. An example is shown in Figure 7.

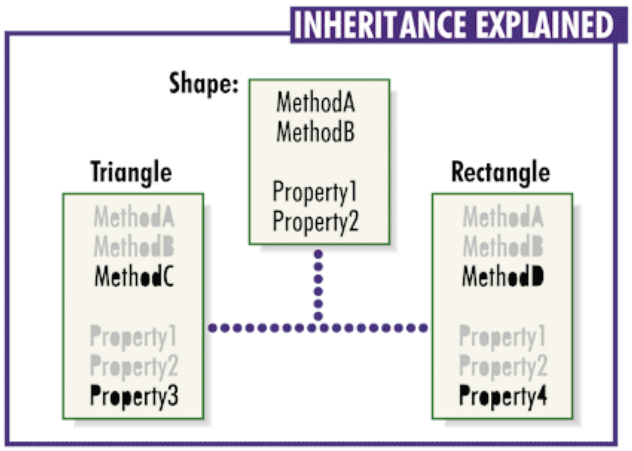



**Figure 7:** Inheritance allows programming by exception.

A second benefit to inheritance is it affords easier maintainability. Have you ever created an application that needed to implement a change that rippled throughout dozens of code modules? Or perhaps you've created a database application where the end-users wanted to add "just one more field" to multiple data entry screens? Inheritance provides a mechanism to make that change once, in the parent class. All object classes that inherit from that parent will automatically acquire the change.

Along with encapsulation, inheritance can make debugging your applications more secure as well. Encapsulation provides the "firewalls" to prevent unwanted side effects. Inheritance then safely makes the job easier. If a bug is found in a particular routine that is used in a variety of places, fixing it in the parent class will automatically "ripple" that fix to all child classes.

Delphi fully supports the use of inheritance in every area of application development. You can create custom Visual Component Library (VCL) classes from scratch, or you can inherit from existing VCL and Visual Basic Extension (VBX) components. Thus, if there is a component that is already close to what you want, inherit from that component and then program the differences.

Delphi makes this process of *sub-classing* a component especially easy with the Component Expert, shown in Figure 8. Select **File | New Component** from Delphi's menu to launch the Component Expert. Give your new component a class name, select an existing component from the **Ancestor type** drop-down list from which to inherit, and enter the name of the **Palette Page** where you want your new component to appear. Delphi will then build the skeleton of your new component, ready for you to code the new properties, methods, and behaviors that are unique to that component.

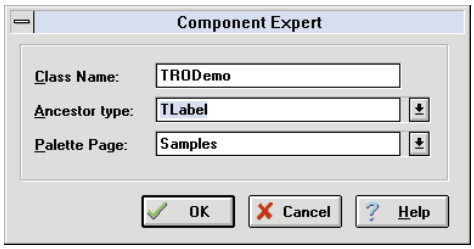



**Figure 8 :** Using Delphi's Component Expert to inherit from an existing control.

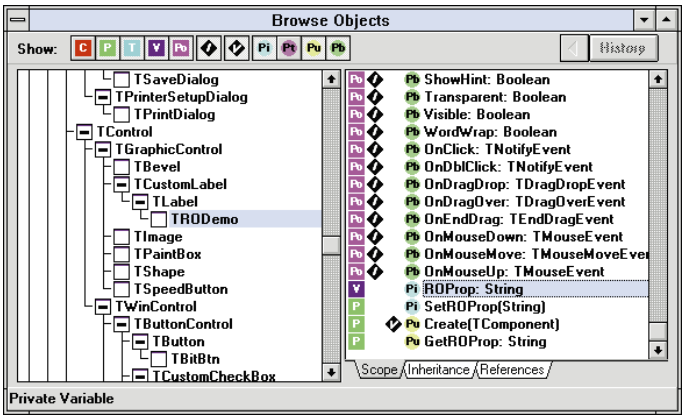



**Figure 9:** Delphi's Object Browser displays VCL inheritance.

Once you build your application (using either the **Compile | Build All** or **Compile | Run** menu choices), Delphi can also show a visual display of the inheritance hierarchy used in your application. Select **View | Browser** from the menu, and you will see Delphi's Object Browser appear (see Figure 9).

The Browser display is based on the actual code used in the currently open project. Figure 9 illustrates the Object Browser for an application that uses the *TRODemo* object shown in the code in Figure 6. Notice that the display of the *TRODemo* class shows that its parent class is *TLabel*, and that it contains only one property (a read-only variable named *ROProp*) that has not been inherited from *TLabel*.

You will find the Object Browser is particularly useful in determining *all* the available properties and methods for an object class. The Object Inspector and Delphi's documentation will only tell you about what has surfaced in the specific class you are viewing. There are many other properties and methods that may exist by virtue of inheritance from an ancestor class. Remember that the Object Browser is based on your actual code. Thus, the Browser is *always* correct and complete.

## Polymorphism

Polymorphism is the third building block of OOP. All programming languages give us the ability to create a set of publicly available functions (known as *public methods* in the OOP world) for each object. These public methods comprise the object's API. Polymorphism enables the programmer to use the same method name to perform vastly different operations, based on the type of object referenced by the method.

We make use of polymorphism all the time. For example, we use the same word to mean different things based on the context. We can apply a single word such as "run" to a person, nose, and computer to obtain vastly different results without any need for additional language commands. If we then add a new type of object, say a pair of stockings, we can still use the same "run" command. Now we have added yet another different result for this one word. The "code" remains the same regardless of the type of object on which it's used. Because of polymorphism, we can simply tell an object to perform a particular method without having to be concerned with any internal differences from one type of object to the next. Like our video hardware upgrader, we simply make use of the standard API of the object — its public methods. We can leave the details for the object to work out itself.

Returning to coding metaphors, let's take a look at the plus sign operator (+). The table in Figure 10 illustrates how polymorphism is used in the form of *operator overloading*. The plus sign performs a different function based on the objects upon which it's acting.

| Sample Code | Result | Operation Performed |
|---|---|---|
| 5 + 7 | 12 | Mathematical Addition |
| "This is " + "a string." | "This is a string." | String Concatenation |
| 1/25/95 + 14 | 2/8/95 | Date Math |

**Figure 10:** Polymorphism illustrated in the use of the plus sign (+).

OOP applies this concept to more than simple operators. Polymorphism can be applied to entire code routines. Let's say we are creating a screen shape-drawing package. We've defined a generic shape object class. From that parent, we've defined two child classes, a triangle and a rectangle, each with its *draw* method defined.

The internal details of how to draw a triangle are very different from the details of drawing a rectangle. In the past you would have had to create a *DrawTriangle* function, a *DrawRectangle* function, and so on. If you then wanted to add a new feature such as printing, you would have to create an entirely new set of custom functions. You would then have a giant **case** statement to process the selection of which function to call based on the type of object.

When it came time to develop version 2 with a new type of object, you'd have to go back through all that old code to make sure everything still worked. You also would have to make sure that you added this new object type to *every* **case** statement in *every* code module where the object type needs to be referenced. This massive code maintenance task becomes even more complex when the application is being written by a team of programmers. To be totally successful, nearly every programmer on the task would need to know all the details about the project's entire code set.

Polymorphism lets you ignore these details and simply say, "Object, draw thyself." In the real world, you can tell both your friend and your dog, "Let's go for a walk." Your dog may scratch at the door, but your friend will simply open it.

A code-based example from our shape drawing program is shown in Figure 11. The source code simply tells each object to

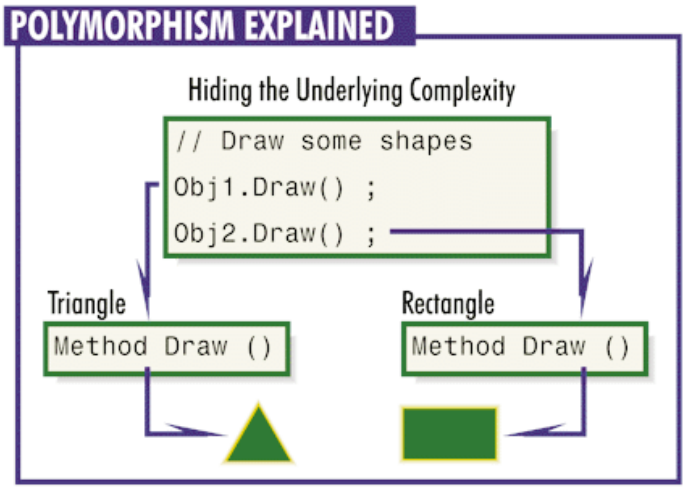



**Figure 11:** Polymorphism allows the developer to write simpler code.

*draw* itself. New types of objects can be added at any time and this same code will continue to work. As long as each new object type has some kind of *Draw* method defined, the developer doesn't have to be concerned about exactly what the *Draw* method does. Thus, polymorphism enables developers to create simpler code by insulating them from internal complexities.

Team development and code maintenance becomes a lot easier too. Most of the programmers on a project can ignore vast amounts of detail information. They simply need to know the API of the objects with which they are working. If a problem is found in a particular object's code, the fix can be propagated throughout the development team simply by fixing the parent from which all the other affected classes are derived.

Delphi provides a simple way for supporting polymorphism with the **override** keyword. In the code shown in Figure 12, we want to create a version of the DBNavigator control that defaults to being simply a record viewer. (It's rather tedious to be constantly turning off a whole lot of properties every time we want to create a navigate-only navigator.) We also want this navigator to default to showing the help hints for the buttons we use.

The way we do this is to override the *DBNavigator.Create* method (the object's constructor). In our *Create* method, we merely call the parent's constructor and then manipulate the properties to set them the way we want inside NavLite's *Create* method. Polymorphism lets us instantiate either a DBNavigator or a DBNavLite object with the same code — *ObjName.Create*.

To see the effect of this VCL class, create a form and drop a DBNavigator component and this new DBNavLite component onto it. You'll notice an immediate difference in the display of the two controls, although both are calling their respective *ObjName.Create* methods (see Figure 13).

## Developer Types: Producers and Users

Most people who are familiar with OOP will readily recognize the first three fundamental OOP concepts. There is however, a

```
unit Navlite;

interface

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

type
  TDBNavLite = class(TDBNavigator)
  { Declare which class to inherit from }
private
  { Private declarations - visible only to this class }
protected
  { Protected declarations - visible to this class and
    its decendants }
public
  { Public declarations - visible to all classes }
  constructor Create(AOwner: TComponent); override;
  { Polymorphism to produce a new effect }
published
  { Published declarations - visible to all classes,
    and displayed in the Delphi Object Inspector }
end;

procedure Register;

implementation

constructor TDBNavLite.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  { Call the parent class' constructor, then set new
    default values for properties }
  VisibleButtons :=
    [nbFirst,nbPrior,nbNext,nbLast,nbRefresh];
  ShowHint := True;
end;

procedure Register;
begin
  RegisterComponents('Data Controls', [TDBNavLite]);
end;

end.
```

**Figure 12:** NAVLITE.PAS — using inheritance and polymorphism to create a new VCL component.

very important fourth concept that is often ignored. With OOP, you need to divide development between two types of programming tasks: Producing new types of objects, and using objects to create applications.

Each task area requires a special, unique set of talents and programmer temperaments. Usually developers who are good at creating robust object classes will not be the same programmers you'll want to have using those objects to create end-user applications. You will typically have a team of *object producers* and a separate, larger group of *object users*. The developer is rare indeed who is equally adept at both of these areas of OOP.

This division of labor is one of the reasons why OOP techniques can offer significant productivity benefits to a wide-range of programmers. In our hardware analogy, only a few people have the expertise to design a video board. However, since board users don't need the same level of expertise in this

# Debunking Some OOP Myths

Object orientation is no magic bullet. The benefits come from a proper application of the four basic concepts of OOP. However, OOP is indeed a significant advancement from past technologies. As with any new idea, some people resist the change. Others find the change confusing. Still others find the idea stimulating and challenging, but have little guidance as to how to evaluate all the claims and counter-claims.

The truth about OOP has often been obscured by a variety of half-truths, deceptions, and misleading statements. We'll examine five major myths about OOP and set the record straight.

**OOP Myth #1: OOP is Just a Fancy Name for Structured Programming.** In the February 1993 issue of *DataBased Advisor* Magazine, Bill Gates and Dave Fulton were interviewed for their comments about OOP. Their take was that OOP was nothing more than a fancy name for structured programming.

**The Reality: OOP Is Much More.** Anyone who glosses over the basics of OOP with this kind of statement simply doesn't understand what issues OOP is designed to address. As we've seen, true object orientation involves a great deal more than just structured programming. True OOP is structured programming, plus structured data, plus structured access to that data. OOP also adds code scoping constructs to the existing variable scoping constructs seen in structured programming.

**OOP Myth #2: Faster Development?** Another common misconception about OOP is that it allows for faster development. The truth is that it takes time to create a proper foundation of object class definitions from which to build. Programmers also need to take the time needed to fully understand the ramifications that using OOP should have on the design process. For the first OOP project, overall development time may actually increase slightly (usually by about 20%) if you have to create your classes entirely from scratch.

**The Reality: Benefits for Version 1.0.** Although the initial version may take slightly longer to produce, there are significant benefits even for version 1.0 projects. First, there is significantly less code to debug. For example, one project started out with 400,000 lines of procedural code, and reduced the code count to around 140,000 lines after converting to OOP. The resulting code is cleaner and provides a much smoother debugging cycle. Thus, the initial release will be far more stable than most version 1.0 projects.

Delphi provides some additional benefits specific to its unique approach to *Rapid Application Development* (RAD). Even developers new to OOP can become productive almost immediately. Borland's Delphi research and development team has taken upon itself the initial overhead of creating a set of robust class definitions for you. The VCL includes 75 pre-built components that provide immediate productivity. Delphi's completely open architecture also lets you leverage the expertise of a wide array of third-party developers who have produced additional VCL components.

**The Reality: Benefits for Application Revision.** The big benefit of OOP is reuse. Thus, the reality is that OOP allows for faster revision for subsequent releases. One of the best examples of OOP's rapid revision capabilities is Quattro Pro for Windows (QPW). Version 1.0 of this product shipped on October 30, 1992. The next version, QPW 5.0, shipped a mere 11 months later with huge increases in functionality.

Due mainly to encapsulation and inheritance, OOP also allows for safer revisions. Again, QPW is a good example. Although QPW 5.0 had hundreds of new features, the product was solid as a rock from day one.

OOP adds up to easier code maintenance. Most organizations are finding the majority of their development time is spent in code maintenance. Thus, OOP provides major benefits in exactly the area where most people need it.

**OOP Myth #3: It Has to Be SmallTalk.** There are purists who claim that proper OOP can only be done in a language specifically designed for OOP from the beginning. In many cases, these purists will suggest a language such as SmallTalk.

**The Reality: What Counts is Encapsulation, Inheritance, and Polymorphism.** You have a project to get done. In today's fiscal climate, the ideal is to complete each project under budget and ahead of schedule. Practicality demands you select tools for which existing talent is widely available.

The main benefits of OOP are code reuse and code safety. These benefits are gained through proper use of encapsulation, inheritance, and polymorphism. These concepts are not dependent on the specific OOP language used.

**OOP Myth #4: GUI's Make OOP Difficult.** Many claim that the paradigm shift to graphical user interfaces (GUI) makes OOP development difficult for long time developers. Karl Steinle, president and CEO of Concepts Dynamic Inc. is quoted in the January 1995 issue of *Software* Magazine, "With the old programming methods, the programmer directed the user with the programmer's procedures. In the OOP arena, it's reversed because the system is event-driven. For the development staff, this is a challenge. The programmers have to change the way they think."

**The Reality: OOP and GUI's Are Different Topics.** While it's true that GUIs graphical user interfaces pose a severe challenge to procedurally-oriented programmers, this issue is not necessarily tied directly to OOP development. The confusion arises because most modern GUI development tools are *also* object-oriented in nature.

As the name implies, a graphical user interface has to do with the *interface* portion of an application, that is, the part the user sees. While the interface is an important part of any application, it's not the only part. OOP concepts have to do primarily with the underlying structure and foundation of an application's code, not its user interface.

The reality is that GUI concepts make creating a GUI application difficult, regardless of the underlying coding techniques. Condemning OOP because creating a good GUI is difficult is like discounting the skills of a world-class chef because she happens to cook a dish made from your most-hated vegetable.

OOP tools such as Delphi actually can make creating a GUI application much easier. GUI environments are designed around the concept of end-user manipulation of an application. OOP development treats code and data as if they were real, tangible objects with physical properties and abilities that can be manipulated. Therefore, the OOP style of development fits in very well with creating GUI applications. The best GUI development tools support OOP precisely for this reason.

Delphi's integrated development environment (IDE) is specially constructed to assist programmers who are new to GUI application design and to OOP. The Object Inspector visually surfaces the biggest area of confusion for new GUI developers — the idea of event-driven programming. The Object Browser is particularly useful for people new to the OOP concepts of inheritance and polymorphism.

**OOP Myth #5: Anything with "Objects" is OOP.** A lot of hype exists about OOP. This new buzzword is the development rage of the 90's. Many vendors try to jump on the OOP bandwagon by simply declaring that their products are object-oriented. "We have on-screen objects," they say. But do those objects encapsulate behaviors as well as display properties, or does the code simply sit at the application level? "We support inheritance down to the second level." Huh? True inheritance has no regard for the number of levels of inheritance. Sounds more like support for marketing concepts than for OOP concepts.

**The Reality: What Counts is Encapsulation, Inheritance, and Polymorphism (Again).** Always keep in mind that reuse is the ultimate goal of OOP. Any system that does *not* fully support all three major concepts is not truly object-oriented. Some development tools support one or two of the concepts, but not all three. These systems should be classified as object-*based*, not object-oriented. Some tools actually have *no* support for OOP at all. They only have so-called "on-screen objects" that are merely visual representations — graphical widgets rather than true objects in the OOP sense.

Good examples of object-based development tools for the PC are Paradox for Windows (no inheritance), Access (only partial encapsulation and no inheritance), and PowerBuilder (inheritance cannot be used on Data Windows). True OOP systems include Borland C++, Delphi, dBASE for Windows, Visual C++, NextStep, and the Taligent Application Environment.

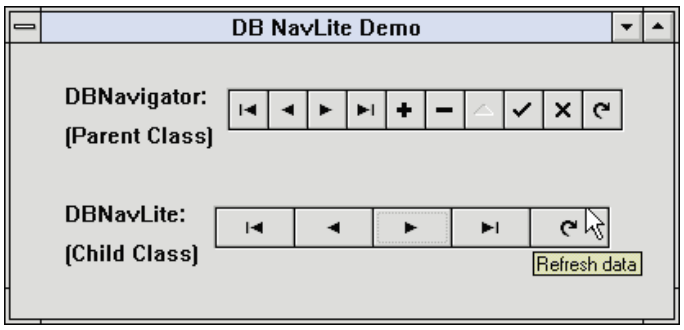



**Figure 13:** An example of the DBNavLite class (visible hints and fewer buttons).

area, far more people benefit from design advances than just a few engineers.

The same holds true for creating and using software components through OOP. Because "ordinary mortals" can inherit the work done by the more technical people in your organization, you can leverage that technical talent throughout your entire enterprise.

## Classes vs. Objects

You need to maintain a clear distinction between what a class is and what an object is. Although they are intimately related, code that is placed in a class will have radically different effects than application code that operates on run-time objects. The table in Figure 14 should help to keep these differences straight.

| | Usage | Creation Process | Coding Methodology |
|---|---|---|---|
| **Class** | Design-Time Blueprint | Sub-Classing | Programming by Exception |
| **Object** | Run-Time Operation | Instantiation | Programming by Properties |

**Figure 14:** Classes versus objects.

Programmers new to OOP have often complained that changes they made "magically" disappeared or reset themselves when they exited their applications and restarted them. They also became confused when changes to an object don't do what they thought they would do. This confusion is especially apparent when developers want to make a copy of an object.

You should constantly ask if you want to change the object's *blueprint* at design time, or change the *object* itself at run-time. For example, to make a copy of an *object*, you should instantiate a new object of the same class and then copy the values of the salient properties from the old object to the new instance. If instead you want to make a similar *blueprint* from which to instantiate objects, you should create a new class by sub-classing the original.

How you code each object is also subtly different. With a class definition, you want to get in the habit of *programming by exception*, that is programming only the differences. You accomplish this through the intelligent use of inheritance.

Run-time coding of object behavior should be accomplished via a technique known as *programming by properties*. That is, your classes should be designed in such a way that run-time code can easily affect the behavior of an object instance simply by changing the values of its properties.

## Conclusion

In summary, remember these key points about OOP:

- The ultimate design goal of any OOP development is to create reusable software components.
- Regardless of the language tool used, the hallmark of true object orientation is full support for encapsulation, inheritance, and polymorphism.
- Encapsulation provides code and data variable safety.
- The combination of inheritance and polymorphism provides increased programmer productivity. Δ

*The demonstration forms referenced in this article are available on the 1995 Delphi Informant Works CD located in INFORM\95\JUN\MO9506.*

Mark Ostroff has over 16 years experience in the computer industry. He began by programming minicomputer medical research data acquisition systems, device interfaces, and process control database systems in a variety of 3GL computer languages. He then moved to PC's using dBASE and Clipper to create systems for the US Navy and for IBM's COS Division. He volunteered to helped create the original Paradox-based "InTouch System" for the Friends of the Vietnam Veteran's Memorial. Mark has worked for Borland for the past 5 years as a Systems Engineer, specializing in database applications.

table of CONTENTS


## On The Cover

Delphi / Object Pascal

*By Zack Urlocker*

# The Triumph of Objects

## Delphi's RAD Approach to Object-Oriented Programming

**During the development of Delphi, one of our goals at Borland was to create a Rapid Application Development (RAD) environment that wouldn't hold you back. We wanted to ensure there weren't limitations that stopped developers from doing something complex or out of the ordinary. After all, if you're in the business of creating custom applications, the fundamental assumption is that unique tasks require custom code to complete.**

Several developers said the first generation of RAD tools often ran out of gas — leaving them with only about 80% of the application completed. In many cases, applications had to be recoded either completely or partially, in C or C++, in order to be completed and run with acceptable performance.

With Delphi, we wanted to provide the best of both worlds — give developers a program that featured the rapid development of a 4GL, with the performance and flexibility of an optimizing 3GL compiler.

Luckily, we had a secret weapon: *Objects*.

Surprised? If you thought object-oriented programming (OOP) was something for rocket scientists, you're in for a pleasant surprise. In the early days, OOP was definitely a lot less visual and required greater discipline — but the payoff was still there. Now it's more visual and the payoff is even sweeter.

### Do You Remember When?

My first object-oriented program for Windows was written on a 640KB 80286 with a 10MB hard disk and CGA graphics. I was running Actor 1.0 on Windows 1.0 — a beta of a beta, if ever there was one. At that time, the only other way to develop Windows applications was to use the outrageously difficult and expensive combination of Microsoft C and Windows SDK. I'd done a bit of graphical user interface (GUI) programming and wasn't eager to write hundreds of lines of code to deal with every Windows message (display context, handle, and all the low level details required). Actor was a very attractive alternative because it enabled me to build Windows applications from pre-existing components. You still had to write code to work in Actor, but the library of pre-built objects gave it a significant head start. Also, the ability to create new objects made it possible to easily extend the environment.

Of course, the industry has progressed a lot since then. Windows no longer runs on floppy disks, display adapters support graphics modes that require a magnify-

ing glass, and Microsoft has eradicated all UAEs in Windows in favor of general protection (GP) faults. We also now accept that hand-coding Windows applications using C and Windows API is generally to be avoided. Heck, if God wanted everyone to program in C, he would have given us pointers instead of fingers. But I digress.

## Your First Object

For a lot of programmers, Delphi's object-oriented environment is a new experience. We purposefully created Delphi so objects can be used easily with the user-interface builder without really having to know the details. First-timers can write a lot of code and complete many tasks without worrying about how the objects work. (You don't have to know pointers, inheritance, or constructors and destructors to get started.) Then, when you're ready to take your programming skills to the next level, the full power of OOP is already at your fingertips.

If we built Delphi right, hopefully many developers will be inspired to learn the concepts of OOP — namely inheritance, encapsulation, and polymorphism — and begin creating custom objects. After all, since Delphi is written in Delphi, there is no distinction between the components supplied and those built. So if you want to add some new control component that offers high performance WinG graphics, cool MIDI support, a rich text editor, or a better database grid, go ahead, you can do it.

But don't forget, you're not limited to creating visual objects, either. If you want to create new abstract objects that model customers, accounts, widgets, or meteors — just do it. The bottom line is simple: You can model whatever is needed in your application development and then reuse it.

## The Key Is Reusability

When you begin creating new objects in Delphi, you'll want to keep a few *Object Lessons* in mind. Most importantly, if you want to create reusable objects plan ahead and design with *reuse* in mind. Don't build a complex application and then try to "pull the objects out". Instead, consider the objects needed and analyze the current problem. If you can identify "clusters" of related data and functionality, these are good candidates for objects.

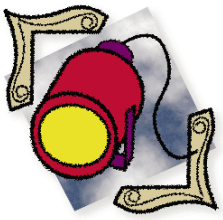
In the early stages of your design, select the data and functionality required in an object and don't worry too much about inheritance. First determine what the object requires, then decide where to get that functionality. In many ways, inheritance is a balancing act between getting the right functionality and interface, while attempting to limit the amount of superfluous baggage. Don't make the mistake of inheriting 20 unneeded capabilities just to get one piece of useful code.

As you create new objects, try to make them consistent with existing *protocols* or ways of interacting. For example, if you want to create a new high-capacity *TRichText* control in Delphi, it would be wise to implement the same key properties and methods found in *TMemo* (such as *Text, Alignment*, *SelectAll*, *GetSelTextBuf*, *CopyToClipboard,* and so on). You should also make sure the methods take the same parameters as an existing *TMemo* control. That way the new control is "plug compatible" with the existing one. This doesn't mean you should base your implementation of a Rich Text editor from the existing *TMemo* class. Actually, you'd probably want to inherit from much higher in the hierarchy.

It often takes two or three iterations before an object is fully reusable. After building a new object with all the properties, methods, and events you think it needs — you may discover it needs just a bit more code. So before you finalize your object, try using it in a complex application. Better yet, get someone else to test and reuse it. If you find code that is frequently implemented by users of your object, try to determine a way to generalize the code and apply it as a method that's built in. As we built Delphi and the Visual Component Library (VCL), there were many cases where new methods or properties were added to objects only after beta testers put them into real world circumstances. Δ

*Now go build some objects!*

Zack Urlocker is Group Product Manager for Delphi at Borland International. He is a frequent speaker at industry conferences on object-oriented programming. The views expressed here are his own.





*By Brian Johnson*

# VB to Delphi

## 10 Things You'll Want to Know to Get Started Fast

**You bought Delphi because it's a RAD (rapid application development) tool. You want to create stand-alone executables and you're not satisfied with the performance of interpreted p-code. You've got C/C++, but you can't stand using it because you spend more time trying to learn new things than building applications. If these are some of the reasons you wanted to begin developing applications in Delphi, then this article is for you.**

What follows are 10 topics you'll encounter when moving from Visual Basic (VB) to Delphi. There isn't a detailed explanation of each topic, instead we've constructed a down-and-dirty overview that will get you coding and building applications right away.

### Lay of the Land

Each Delphi Project is divided into at least three of the following files:

- the .DFM extension indicates a binary file containing form information
- .PAS indicates an Object Pascal unit file
- .DPR indicates the Delphi project file
- .RES indicates a Windows resource file
- .OPT indicates the options file (edited via **Options | Project**, this file saves compiler, linker, and other project options)

The .DFM file is binary and can be manipulated through the Delphi IDE. It can also be opened and viewed as a text file by selecting **File | Open** from the main menu. The .PAS files are text files that can be edited within the Code Editor in the IDE, or using a text editor. The .DPR file is also a text file and may be similarly edited.

Each .PAS file in a project is known as a *unit*. The unit itself is broken into blocks containing statements. The operational code is contained within the **unit...end** block. Your procedures are declared below the **type** keyword and used in the **implementation** section. Placing an object on your form and using the Object Inspector to access an object's events does most of this for you.

To declare variables within a procedure, you must add a **var** keyword before the **begin** keyword within the procedure. Variables declared outside the **implementation** section in the unit's **var** section become global to the unit (see Figure 1).

```
unit Unit1;

interface

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

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

var
  Form1: TForm1;
  y: LongInt;  { Global variable }

implementation

{ $R *.DFM }

procedure TForm1.Button1Click(Sender: TObject);
var
  x: integer;  { Local variable }
begin
end;

end.
```

**Figure 1:** A sample Object Pascal unit file. Note that some variables are declared in the unit's **var** section — above the **implementation** section — making them global to the unit.

## Variables

In VB, you may be accustomed to coding your application using *automatic* variables. That is, you might write:

```
x = "This is a text variable"
y = 500
z = Y / 2.5
```

In VB your variables are "automatic" because they can literally represent nearly any value you wish to use. Because it's a *strongly typed* language, things aren't quite that simple in Object Pascal. Data must be stored in precisely defined (or sized) locations you create within a program. You do this by declaring your variables by type. That is, they are *explicitly* declared (Option Explicit in VB). When working with compatible data types, this usually presents no problems. For example, consider this Object Pascal code fragment:

```
var
  x, y, z: integer;
begin
  x := y * z;
end;
```

However, if we change our equation statement a bit:

```
x := y / z;
```

we run into problems. This statement generates an error because a quotient produced using the / operator is always of type *real*. (Note: An integer divided by an integer using the **div** operator is of type integer.) To fix this, our variables must be declared a little differently:

```
var
  y, z : integer;
  x : real;
begin
  x := y /z;
end;
```

No error here. But if we want to plug x into another equation that returns an integer, we must make the data types compatible. In this case we'll use the *Trunc* function:

```
var
  y, z: integer;
  x: real;
begin
  x := y /z;
  y := Trunc(x) * z;
end;
```

(You could also use *Rnd* or another truncating function.) Now, what if we want to output some of this to the screen? Because the Label component's *Caption* property requires a string, we need to change any integer data to strings. This can be done with the *IntToStr* function:

```
Label1.Caption := IntToStr(y);
```

Of course if we want to get the label value and turn it into a number, we can do that too:

```
y := StrToInt(Label1.Caption);
```

Finally, to convert our real value (x) into a string, we simply nest the functions:

```
Label1.Caption := IntToStr(Trunc(x));
```

Remember there are many factors to consider when working with explicit data types. Learning the various data types, and how they can be converted (cast) into other types, will help you to avoid many mistakes.

## Conditional Terminology

Conditional terminology in Delphi is similar to that of VB. For example, you can use **if** statements to test the truth of a condition:

```
if CheckBox1.Checked = True then
  Object Pascal statement
else
  Object Pascal statement;
```

It's important to note that since there is only one statement for the **then** and **else** portions of this **if** statement, a semicolon is required only after the terminating **else** clause. If you need multiple statements within the **then** portion of an **if** statement (for example), you must use a **begin...end** block:

```
if CheckBox1.Checked = True then
  begin
    Object Pascal statement 1;
    Object Pascal statement 2;
```

```
  end
else
  Object Pascal statement;
```

In other words, there is no "EndIf" keyword in Object Pascal as there is in VB. You can also use the **case** statement to check conditions:

```
var
  NumMonth : integer;
  Month : string;
begin
  { Function returns month based on ordinal number. }
  case NumMonth of
    1 : Month := January';
    2 : Month := February';
    { and so on. }
  end;
end;
```

The **case** statement is very handy if you have a number of possible outcomes of the same variable type.

## File Functions

File I/O with Delphi components is fairly straightforward. For example, to load a file into a *TMemo* component, simply use the *LoadFromFile* method, passing the name of the file as the argument. Here's an example where `FileName` is a variable of string type:

```
Memo1.Lines.LoadFromFile(FileName);
```

Conversely, to write changes to a file, use the *SaveToFile* method:

```
Memo1.Lines.SaveToFile(FileName);
```

The same technique works for the contents of OLE containers and graphics components. As a VB programmer, you'll find a surprising number of operations can be called directly from within Delphi. As a rule, and to save some work, try finding a built-in method for performing an operation before venturing into the Windows API for a solution.

## Accessing Other Forms

This looks pretty straightforward. To open a second form from within your application you just add `Form2.Show`, right? Well, yes and no. Any external code — even if it's part of the current project — needs to be referenced in the **uses** section of the main unit. Therefore, you'll need to add Unit2 to Unit1's **uses** statement.

Further, to access Unit1 from Unit2, you must declare it in Unit2. Simply insert a **uses** clause in Unit2's **implementation** section and reference it from there.

## Making API Calls

Note there are references to `WinTypes` and `WinProcs` in a form unit's **uses** section. These files contain the information necessary to call Windows API functions in your application. In VB you might be used to declaring your functions before calling them. These two sections do this for you, and make calls to the Windows API straightforward.

Look at the WINPROCS.PAS file in your \DELPHI\SOURCE\RTL\WIN directory to see how these routines are declared. For example, the declaration for the Windows API function, **GetFreeSystemResources** is:

```
function GetFreeSystemResources(SysResource: Word): Word;
```

Since it's been declared for you, calling the function is simple:

```
begin
  Label1.Caption :=
    IntToStr(GetFreeSystemResources(
                GFSR_SYSTEMRESOURCES));
end;
```

Of course, you still need to be knowledgeable about the Windows API to use it effectively (and stay out of trouble). However, Delphi provides excellent on-line help to the entire Windows API. Just select **Help | Windows API**.

## Accessing Interrupts

In VB you may have used a DLL to access DOS interrupts. This is unneccessary in Delphi, and you don't need to know any assembler to access system data. All you need is a good reference to the DOS internals. First, declare the variables you're going to use to hold the returned data:

```
var
  SectorsPerCluster,
  AvailClusters,
  BytesPerSector,
  TotalClusters: Word;
```

Then you must call the interrupt function using embedded assembler statements. In Object Pascal, the **asm** keyword is used:

```
asm
  mov ah,36h { Get disk space }
  mov dl,3  { Select drive C }
  int 21h

  mov word ptr SectorsPerCluster,ax
  mov word ptr AvailClusters,    bx
  mov word ptr BytesPerSector,   cx
  mov word ptr TotalClusters,    dx
end;
```

Then we can assign the variables to Delphi component properties:

```
Edit1.Text := IntToStr(SectorsPerCluster);
Edit2.Text := IntToStr(AvailClusters);
Edit3.Text := IntToStr(BytesPerSector);
Edit4.Text := IntToStr(TotalClusters);
Edit5.Text := IntToStr(LongInt(SectorsPerCluster) *
                       BytesPerSector * TotalClusters);
Edit6.Text := IntToStr(LongInt(SectorsPerCluster) *
                       BytesPerSector * AvailClusters);
```

Notice that in the `Edit5.Text` and `Edit6.Text` assignments, that we must typecast one of our variables so we don't overflow the buffer holding our data.

If you've never accessed DOS internals, you'll find that they can hold a wealth of information. They can also be quite dangerous

if you're not careful. DOS instructions deal with everything from disk I/O, to low-level disk operations such as formatting, so double-check your numbers when accessing data to be sure you're calling the correct interrupt.

And don't create too much work for yourself. Delphi already encapsulates a lot of this functionality (for example, the Object Pascal *DiskSize* and *DiskFree* functions).

## Accessing a DLL

Accessing a DLL in Delphi requires a decision. You must choose whether to load the DLL at the same time the program runs (a *static import*), or when the function is required (a *dynamic import*). Your decision may depend on how the DLL is used in the application. Using a static import is the easiest way to access a DLL — and the one you probably used most often in VB.

Static importing of a DLL function requires declaring the function as external (i.e. using the **external** directive) and making a simple call to the routine. Let's say we have a DLL named DISKINFO.DLL with a routine named `diFreeSpace` that returns disk information based on an integer value (the drive number) passed as an argument:

```
{ In the implementation section }
function diFreeSpace(Drive: integer): LongInt;
  external 'DISKINFO';
```

Notice that ".DLL" is not appended to the `'DISKINFO'` specification; it's implicit in the declaration. To return our information we simply make the call and pass the drive number to the function:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := IntToStr(diFreeSpace(3));
end;
```

Remember you need to specify the far call model when importing a procedure or function. To do this, open the Project Options dialog box by selecting **Options | Project**. At the Compiler page, select **Force far calls** in the **Code generation** area. Or alternatively, you can add the **far** directive to a function declaration. For example:

```
function diFreeSpace(Drive: integer): LongInt;
  far; external 'DISKINFO';
```

If you have favorite libraries that you access often, consider creating a declaration file. (Again, see the WINPROCS.PAS file for an example.) Afterwards, you'll only need to add the name of the declaration file to your **uses** clause to call its routines.

## Adding Controls

If you want to add a VBX control to the Component Palette, select **Options | Install Components** to display the Install Components dialog box. Select **VBX** to insert the VBX you wish to add. [For a complete description of adding a component to Delphi's Component Palette, see Gary Entsminger's article "Approaching Components" in the Premiere issue of *Delphi Informant*.]

Like C/C++, Delphi can only use Level 1 VBX components. This should not be a problem for most VBXes — they are supposed to be written to be backward compatible with prior VB versions. However, you may have a problem with data access controls because the database features of VB weren't added until version 3.0.

## Exception Handling

Exception handling allows your program to recover from errors that might otherwise cause problems or instability on the user's system. An *exception* is raised when an error occurs. Exception checking is not required for every block of code that you write. However, it's highly recommended in situations where you allocate system resources, or where the outcome may cause a problem within the program. In Delphi, exception handling is implemented using the **try...finally** or **try...except** block. For example:

```
var
  x, y : real;
begin
  y := 0;
  try
    x := 1 / y
  except on EZeroDivide do
    MessageDlg(Divide by Zero error',
                mtInformation,[mbOk],0);
  end;
end;
```

[For an in-depth discussion of Delphi exception handling, see Gary Entsminger's article "Exceptional Handling" on page 23.]

## GO Delphi

One of the best sources for information and components for Delphi is the Delphi Forum on CompuServe. Also, a Delphi news group on the Internet is planned. If you struggle with a specific problem, you can bet another user has had the same problem — and the answer may be on the forum.

## Conclusion

In many ways, Delphi is an entirely new world for Visual Basic programmers. However, the environments and language syntax are similar enough to make the transition to Delphi fairly easy. You can find information dealing with this transition in the VB2DELPH.WRI file provided with Delphi. (It's in the \DELPHI\DOC directory.)

If you need to convert a large amount of VB code, you may want to purchase the Conversion Assistant. [EarthTrek's Conversion Assistant product is reviewed on page 42.]

Delphi is the "visual" C++ that VB programmers have been waiting for. With a little work, a Visual Basic programmer can be up to speed in Delphi quickly. Δ

Brian Johnson is a freelance writer and programmer in Orlando, Florida. He can be reached on CompuServe at 72322,3611.

table of CONTENTS



*By Gary Entsminger*

# Exceptional Handling

## Trapping Run-Time Errors with an Exception-Handling Component

*They looked as if they'd just come off the stage*
*or had arrived here straight from a costume ball.*
*Their headbands, patterned peploses, hairdos, bracelets,*
*and necklaces were all in imitation of the robes*
*of women of Delphi: they were slender and carefree.* — Pawel Huelle

**Unlike its ancestor, Borland Pascal 7.0, Object Pascal, the underlying language of Delphi, has an excellent mechanism for handling error conditions that can arise in code. Delphi uses exception handlers to respond to these errors.**

An *exception* in Delphi is both an error condition (something that halts an application's flow of execution), and an object (a *type* that contains information about the error condition). By *handling exceptions* you can prevent your applications from crashing after an exception occurs.

In this month's article, we'll discuss creating a general-purpose exception handler component that you can "plug into" your Delphi applications. This exception handler handles RTL (run-time library) integers and floating point math exceptions. You can also easily modify it to handle other exceptions.


### Exception Handling

An *exception handler* is code that responds to a specific exception or error that occurs within a protected block of code. If you don't protect code, and an exception or error occurs, Delphi will generate an error message and abandon execution of the code block that generated the error. Although your application might not crash at this point, this is definitely not how you want it to behave.

In general, you can protect code in two ways: within a **try...finally** block or a **try...except** block. A **try...finally** block establishes a resource-protection block. Use a **try...finally** block to recover from an exception and free any of the resources that were allocated by the application before the exception occurred. Without a **try...finally** block, memory allocated to resources may be lost.

For example, consider the following code, a button click event procedure:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Point1: Pointer;
  X, Y: Real;
begin
  X := 0;
  Y := 0;
  GetMem(Point1,1024);       { Allocate 1KB of memory. }
  X := 10 / Y;               { This generates an error. }
  ShowMessage('Continuing'); { Execution never gets here, }
  FreeMem(Point1,1024);      { or more importantly, here. }
end;
```

This code uses the *GetMem* procedure to allocate a 1KB block of memory. The address of the block is referenced by the pointer, `Point1`. In this procedure, you might want to protect the *GetMem* procedure. If you try to allocate a memory block, and there isn't enough free space in the heap (where dynamic variables are stored) to allocate the new pointer, *GetMem* generates a run-time error.

The **try...finally** block can solve this kind of problem. Consider the following revised button click event procedure:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Point1: Pointer;
  X, Y: Real;
begin
  X := 0;
  Y := 0;
  GetMem(Point1,1024);  { Allocate 1K of memory }
  try
    X := 10 / Y;  { This generates an RTL floating point
                    error }
  finally
    ShowMessage('Continuing'); { Execution DOES get here }
    FreeMem(Point1,1024);      { Despite the error }
    { Optionally, do something about exception here, }
  end;
    { or here }
end;
```

Placed in the **try** part of a **try...finally** block, the assignment to `X` generates an RTL floating point exception. If the application terminates at this point, the memory allocated by *GetMem* is lost because code execution never reaches the *FreeMem* procedure. In this case however, execution *does* continue after the exception, and the memory allocated for the pointer can be recovered by the *FreeMem* procedure.

The statements in a **try...finally** block execute normally unless an exception occurs. If an exception occurs, the **finally** part of the **try...finally** block executes. In fact, the statements in the **finally** part of a **try...finally** block always execute, even if no exception occurs.

Note that the **try...finally** block doesn't do anything about the exception. If an exception occurs in a **try...finally** block, execution proceeds to the **finally** part of the block, then exits the block. At this point you can optionally handle the exception.

If you're new to Delphi exception handling, you might become confused when you try to test your exception handlers. When you're working within the Delphi development environment, Delphi gets first crack at exceptions and reports the error. You must then use either the Run, Step Over, or Trace Into commands to actually see your exception handler work.

The best way to test your exception handlers is to exit Delphi entirely and run your application as a stand-alone .EXE. This way, you won't be bothered by Delphi's good-intentioned interruptions.

## The try...except Block

If you want to handle exceptions and don't need to worry about resources or dynamic variables, you can use a **try...except** block to protect your code.

Within the **try** part of a **try...except** block, statements execute normally unless an exception occurs. If an exception occurs, execution jumps to the **except** part of the **try...except** block. However, if no exception occurs, the block ends without executing the code in the **except** or **else** parts of the **protected** block. Thus, a **try...except** block won't necessarily protect resources.

Within the **except** part of the block, exception handling statements must be in the form:

```
on <type of exception> do <statement>
```

In other words, you evaluate individual exceptions.

If none of the **on...do** statements apply to the current exception, the default exception handler executes. Eventually, after a specific handler or the default handler handles the exception, the **try...except** block ends.

In short, in a **try...finally** block, execution resumes within it after the exception. In a **try...except** block, execution does not continue within the block after the exception.

The **try...except** block doesn't automatically give you information about an exception. However, you can obtain it from the exception instance if you need it. You access the exception instance through an alternative form of **on...do**:

```
on <temporary var...specific exception instance> do
```

This alternative form requires that you declare a temporary variable to hold the exception instance. The exception handler you develop next shows how to use the exception instance.

## Setting Up the Exception Handler

Setting up a general purpose **try...except** exception handler is relatively easy. You start by embedding the code you want to protect within an exception-handling block that begins with the reserved word, **try**. Then you specify your code's response to the exception after the reserved word, **except**. Here's how a **try...except** block might look:

```
try
  { Statements to protect. }
except
  { Exception-handling statements. }
end;
```

If an error condition (an exception) occurs while executing a statement in the **try** part, your application automatically creates an exception object. Then it begins executing statements in the **except** part of the block.

If the application locates an exception handler that handles the exception, it executes the statement, then automatically destroys the exception object. Finally, execution continues at the end of the **try...except** block. As noted earlier, execution does not continue within the **protected** block.

Any procedures or functions your code calls in the **try** part are also protected by this **try...except** block. For example, if code in the **try** part calls a procedure that doesn't define a **try...except** block, and an exception occurs, execution returns to this **try...except** block and tries to handle the exception in this **except** part.

## An Exception Handling Component

Although you can handle exceptions by adding new exception handling code each time you protect code, you can alternatively create a general-purpose exception handler. By making this general-purpose exception handler an object, you can modify and expand it. By making the exception handler a component, you can add it to the Component Palette.

The key to this exception handler is a general-purpose method (a procedure) that evaluates the exception. When an exception occurs in any **try...except** block, the application calls this general-purpose method to evaluate and handle the exception. This exception handler component publishes a property, called *Active*, that indicates whether the exception handler is active (available).

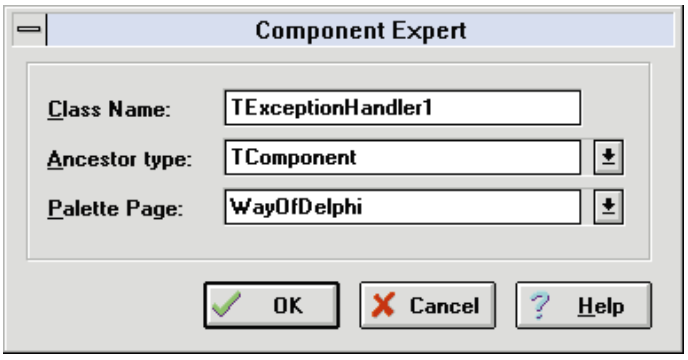
Begin a new project by selecting **File | New Project** from the menu. Delphi will automatically create a form and unit. We don't need them to create the exception handler, so close them without saving them. Then save the project as TExcept1.DPR.

## Deriving a New Component

Use the Component Expert to derive a new component from *TComponent*. As shown in Figure 1, enter the following parameters in Component Expert dialog box:

- **Class Name**: TExceptionHandler1
- **Ancestor type**: TComponent
- **Palette Page**: WayOfDelphi

Click on the **OK** button to accept the entries. The Component Expert automatically creates the Object Pascal code shown in Figure 2 in a unit called Unit1. Save this file as TExcept.PAS.




```
unit Unit1;

interface

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

type
  TExceptionHandler1 = class(TComponent)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('WayOfDelphi',[TExceptionHandler1]);
end;

end.
```

**Figure 1 (Top):** To access the Component Palette, select **File | New Component** from the menu. Now you can assign a class name, derive from an ancestor object, and designate a Component Palette page. **Figure 2 (Bottom):** Code generated by the Component Expert when you create the *TExceptionHandler1* component.

## Creating a Property

Next, modify this generic component by adding a property. Begin by declaring an object field (variable) in the **private** declaration of the class to store the data for the property:

```
private
  FActive: Boolean;
```

The following Object Pascal statement creates a property (*Active*) with a value that can be read from and written to the FActive field:

```
published
  property Active: Boolean read FActive write FActive;
```

Next, declare a method (a procedure), *OnDo*, in the **public** declaration. This method will accept an exception object and determine its type of exception. Make it a **virtual** method to

allow future descendants to override it:

```
public
  { Public declarations }
  procedure OnDo(E: Exception); virtual;
```

Here's the complete **type** description for the *TExceptionHandler* component:

```
type
  TExceptionHandler = class(TComponent)
  private
    { Private declarations }
    FActive : Boolean;
  protected
    { Protected declarations }
  public
    { Public declarations }
    procedure OnDo(E: Exception); virtual;
  published
    { Published declarations }
    property Active : Boolean read FActive write FActive;
  end;
```

## OnDo and the Is Operator

The key to handling each exception lies in the *OnDo* method. *OnDo* evaluates the exception, handles it (if it can) and then returns control to the **try...except** block that invokes it. Since an exception is an object type, *OnDo* uses the **is** operator to evaluate it.

The Object Pascal **is** operator is a dynamic type checker. It determines whether the run-time type of an object reference belongs to a specific class. For example, the following line of code tests whether the exception, `E`, belongs to the class, `EDivByZero`:

```
if E is EDivByZero then
  { Do Something }
  ShowMessage('EDivByZero')
```

The *OnDo* method of our exception handler component (see Figure 3) consists of a series of tests for the RTL integer and floating point exceptions.

Note that although a **case** statement is often more convenient to use than a nested series of **if** statements, we can't use it here because it doesn't test class types. A **case** statement can only test integers, chars, enumerated types, and subrange types.

Note also that after each test, *OnDo* simply displays an error message. However, this can be sufficient for application recovery (and is sufficient in the test application presented later in this article). However, you could easily create a more complex *OnDo* procedure to handle errors in more specific ways.

## Adding the New Component to the Palette

Now select **Options | Install** to access the Install Component dialog box. Click the **Add** button to access the Add Module dialog box. Select the **Browse** button to locate the TExcept.PAS file from the appropriate directory. Select TExcept.PAS to add and it will appear at the bottom of the **Installed Units** list box in the

```
procedure TExceptionHandler.OnDo(E: Exception);
begin
  { Integer math exceptions }
  if E is EDivByZero then
    ShowMessage('EDivByZero')
  else if E is ERangeError then
    ShowMessage('ERangeError')
  else if E is EIntOverFlow then
    ShowMessage('EIntOverFlow')
  { Floating point math exceptions }
  else if E is EInvalidOp then
    ShowMessage('EInvalidOp')
  else if E is EZeroDivide then
    ShowMessage('EZeroDivide')
  else if E is EOverflow then
    ShowMessage('EOverflow')
  else if E is EUnderflow then
    ShowMessage('EUnderflow')
  else
    ShowMessage('Unhandled Exception.');
end;
```

**Figure 3:** Code that tests for run-time library (RTL) exceptions.

Install Components dialog box. Its path will be added to the **Search path** edit box.

Finally, click **OK** to accept the changes. The library recompiles and the Component Palette is reconfigured, showing the new WayOfDelphi page and the new component (see Figure 4).

## Invoking Component Methods

Finally, let's create a form that tests the exception handler. Select **File | New Form** to create a new blank form. From the Standard page, add two Button components, and from the WayOfDelphi page add the *TExceptionHandler* component to the form. Then, change the captions of the buttons to `Integer exception` and `Floating point exception`. For this example, change the *Caption* property of the form to `Exception Handler Test`. Figure 5 shows this form in design mode.

Select the *TExceptionHandler* component and note the published properties, *Name*, *Tag*, and *Active* (see the Object Inspector in Figure 6).

Next, implement the *OnClick* events for the two buttons. The first button click event generates an RTL integer exception and checks the exception handler's *Active* property. If *Active* is set to *True*, it sends a message to the exception handler's *OnDo* method:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Num, X : Integer;
begin
  X := 0;
  try
    Num := 9 div X;    { Force an Integer exception. }
  except on E : Exception do
    if ExceptionHandler1.Active = True then
      ExceptionHandler1.OnDo(E)
    else
      ShowMessage('Custom Exception Handler not active.');
  end;
end;
```

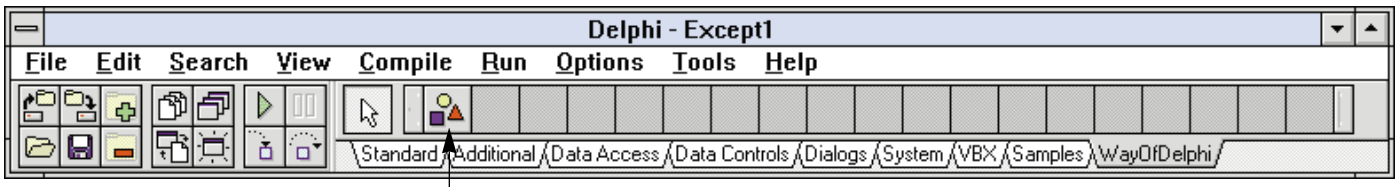



**Figure 4:** The Component Palette now has the WayOfDelphi page with the new *TExceptionHandler* component.

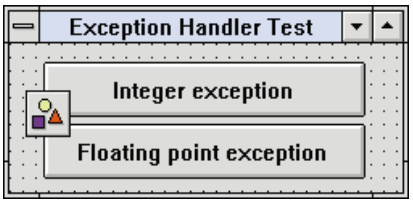



**Figure 5 (Left):** The new form in design mode.

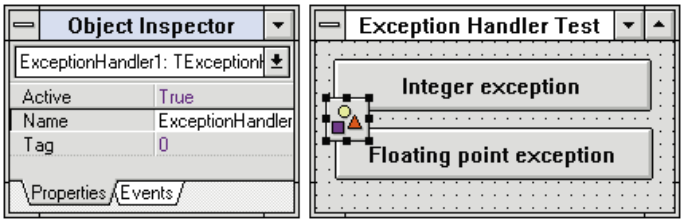



**Figure 6:** The Object Inspector showing the properties for the *TExceptionHandler* component.

The second button click event generates an RTL floating point exception and checks the exception handler's *Active* property. If *Active* is set to *True*, it sends a message to the exception handler's *OnDo* method:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Num, X : Double;
begin
  X := 0;
  try
    Num := 9 / X; { Force a floating point exception. }
  except on E : Exception do
    if ExceptionHandler1.Active = True then
      ExceptionHandler1.OnDo(E)
    else
      ShowMessage('Custom Exception Handler not active.');
  end;
end;
```

The Object Pascal code for the *TExceptionHandler* component and the demonstration application is shown in Listings One and Two (beginning on page 28), respectively.

Figures 7 and 8 show this application at run-time. As mentioned, remember to close Delphi prior to running the application.

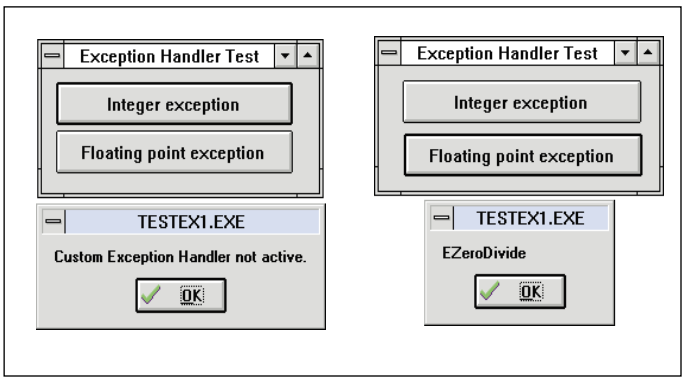



**Figure 7 (Left):** By clicking on the **Integer exception button**, the dialog box displays this message. **Figure 8 (Right):** Likewise, by selecting **Floating point exception**, the result is **EZeroDivide**.

## Conclusion

Delphi's exception handling capability is — well — exceptional. And it offers variations depending on your needs. If you need to protect resources, protect blocks of code with **try...finally**. If you need to handle an RTL exception, use a **try...except** block. If you need to access information about an exception instance, use the alternative form of **on...do**.

Use this discussion and the *TExceptionHandler* component as a basis for your exception handlers. For example, you might want to allow users to access the *Active* property of the *TExceptionHandler* component at run-time. This could enable them to turn the exception handler on and off. Use your imagination. Δ

*The component and demonstration application referenced in this article are available on the 1995 Delphi Informant Works CD located in INFORM\95\JUN\GE9506.*

Gary Entsminger is the author of *The Tao of Objects, an Introduction to Object-Oriented Programming, 2nd ed.* (M&T 1995) and *Secrets of the Visual Basic Masters, 2nd ed.* (Sams, 1994). He is currently working on *The Way of Delphi,* an advanced Delphi book for Prentice Hall, and is Technical Editor of *Delphi Informant.*

### Begin Listing One — TExcept.PAS

```
unit TExcept;

{ Generic exception handler component }

interface

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

type
  TExceptionHandler = class(TComponent)
  private
    { Private declarations }
    FActive : Boolean;
  protected
    { Protected declarations }
  public
    { Public declarations }
    procedure OnDo(E: Exception); virtual;
  published
    { Published declarations }
    property Active : Boolean read FActive write FActive;
  end;

procedure Register;

implementation

procedure TExceptionHandler.OnDo(E: Exception);
begin
  { Integer math exceptions }
  if E is EDivByZero then
    ShowMessage('EDivByZero')
  else if E is ERangeError then
    ShowMessage('ERangeError')
  else if E is EIntOverFlow then
    ShowMessage('EIntOverFlow')
  { Floating point math exceptions }
  else if E is EInvalidOp then
    ShowMessage('EInvalidOp')
  else if E is EZeroDivide then
    ShowMessage('EZeroDivide')
  else if E is EOverflow then
    ShowMessage('EOverflow')
  else if E is EUnderflow then
    ShowMessage('EUnderflow')
  else
    ShowMessage('Unhandled Exception.');
end;

procedure Register;
begin
  RegisterComponents('WayOfDelphi',[TExceptionHandler]);
end;

end.
```

### End Listing One

### Begin Listing Two — Tstexcep.PAS

```
unit Tstexcep;

interface

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

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    ExceptionHandler1: TExceptionHandler;
    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
  Num,X : Integer;
begin
  X := 0;
  try
    Num := 9 div X;

  except
    on E : Exception do  { Exception instance }
      if ExceptionHandler1.Active = True then
        ExceptionHandler1.OnDo(E)
      else
        ShowMessage
          ('Custom Exception Handler not active.');
  end;

end;

procedure TForm1.Button2Click(Sender: TObject);
var
  Num,X : Double;
begin
  X := 0;

  try
    Num := 9 / X;
  except
    on E : Exception do  { Exception instance }
      if ExceptionHandler1.Active = True then
        ExceptionHandler1.OnDo(E)
      else
        ShowMessage
          ('Custom Exception Handler not active.');
   end;

end;

end.
```

### End Listing Two

table of CONTENTS



*By Cary Jensen, Ph.D.*

# The TField Class

## Directing Stored Data with a Non-Visual Component

**A *TField* is a class of component that permits you to access and control data stored in tables. Unlike many other components, *TFields* are not visual. For example, you can place a DBGrid component (from the Data Access page of the Component Palette) on a form, and it appears as an object that can be selected, positioned, resized, and deleted.**

*TFields*, by comparison, don't appear on any of the Component Palette pages, and likewise cannot be physically placed on a form. Furthermore, once they are created, they don't appear on the form. Only after you place a Table or Query component on a form, can you begin creating *TField* components.

This article introduces the *TField* component and discuss its features. It will show you how to create *TField* components and manipulate their properties.

### What Are TFields?

A *TField* component is a descendant of the *TComponent* class. The *TField* component class itself has descendants, one for each type of data that can be handled in a field. The relationships between *TField*, its parents, and descendants are depicted in the Object Browser (see Figure 1).

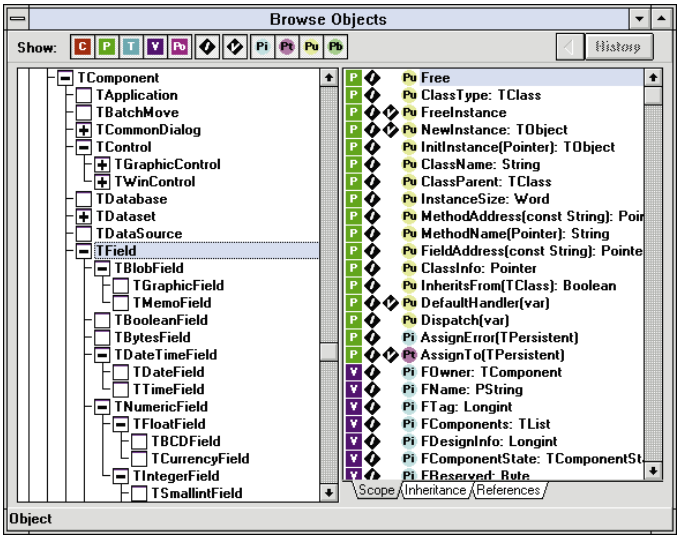



**Figure 1:** The Object Browser showing the *TField* objects descending from the *TComponent* object.

In fact, you don't actually work with *TField* components. Instead, you work with one of the descendant objects of the *TField* component class. For example, using a Table object you can create one *TStringField* object for each text field in the corresponding table. Although you don't use "pure" *TField* components, for the purpose of this discussion, all descendants of this class will be referred to generically as *TField* objects.

Although *TField* components are used to access information from a table, they are significantly different from components that appear on the Component Palette's Data Access page. For example, a DBEdit component may appear to be nearly identical to a *TStringField*, but it's actually quite different. Consider the Object Browser shown in Figure 2. Notice that the *TDBEdit* component is a descendant of the *TControl* class. On the other hand, both *TControl* and *TField* objects descend from the *TComponent* class.

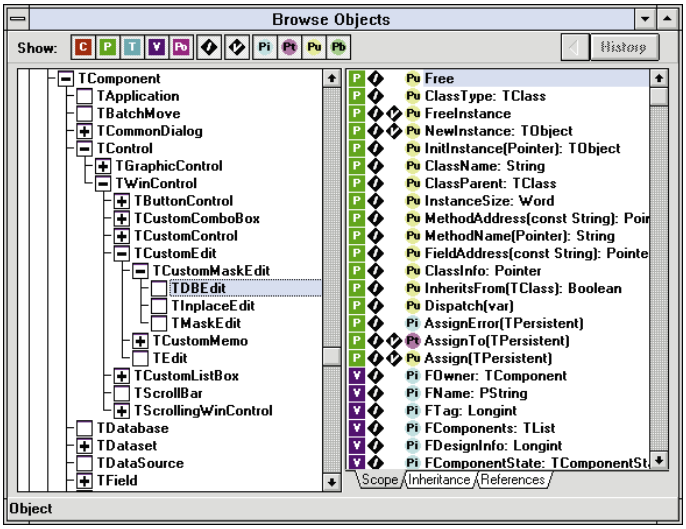



**Figure 2:** The *TDBEdit* object is a descendant of the *TControl* object.

Because *TField* and *TDBEdit* objects are descendants from different parent objects, they have inherited different properties and methods. For example, a *TStringField* object has an *EditMask* property that permits you to control how data is entered in the field. This property does not exist for DBEdit components. Conversely, a DBEdit component has an *OnDblClick* event, which the *TStringField* does not.

## Defining and Selecting TField Components

There are two ways to select a *TField* component: using the Fields editor, or the Object Inspector. However, you must first define a *TDataSet* object (*TTable* or *TQuery*). For this demonstration, it will be helpful to also create a *TDBGrid* component, which in turn requires a *TDataSource* object.

Use the following steps to create an active *TDBGrid* for the Delphi sample table named Employee:

1) Open a new project.
2) Place DataSource, Table, and DBGrid components on the form.
3) Position them so the DBGrid is in the center, and the DataSource and Table are at the top of the form.
4) Select DBGrid and set its *DataSource* property to `DataSource1`.
5) Choose DataSource and set its *DataSet* property to `Table1`.
6) Set the Table component's *DatabaseName* property to `DBDEMOS`. (This is a BDE alias that was created when you installed Delphi. If this alias isn't available, continue to the next step.)
7) Set the Table component's *TableName* property to `EMPLOYEE.DB`. (If the :DBDEMO: alias was not available, include the DOS path when entering the table name. If you used the default directory names when you installed Delphi, this path is C:\DELPHI\DEMOS\DATA.)

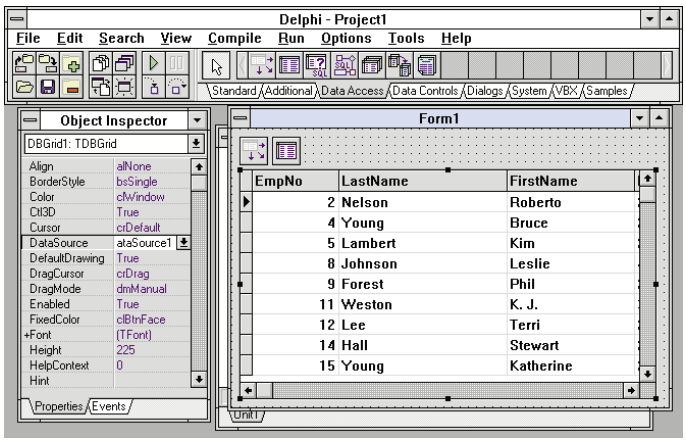



**Figure 3:** Designing a basic form to display table data.

8) Activate the database connection by toggling the Table component's *Active* property to *True*.

Your form should resemble Figure 3.

Note that the DBGrid displays data from the DataSource object, which points to the Table object that is connected to the Employee table. Still, the *TField* components haven't been instantiated (created). You can demonstrate this by selecting the Object Inspector's drop-down list (see Figure 4). Notice this list contains only the objects named `Form1`, `Table1`, `DataSource1`, and `DBGrid1` — no *TField* objects.


**Figure 4:** The current objects on the form.

Creating the instances of the *TField* objects is very important. Until you do, you can't manipulate individual fields in the table programatically. You instantiate the *TField* objects using the Fields editor. There are two ways to display the Fields editor. The easiest is to double-click the Table object on the form. Alternatively, right-click the Table object and select **Fields editor** from the displayed Speed Menu. The Fields editor, shown in Figure 5, is displayed.

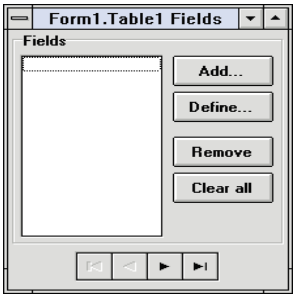
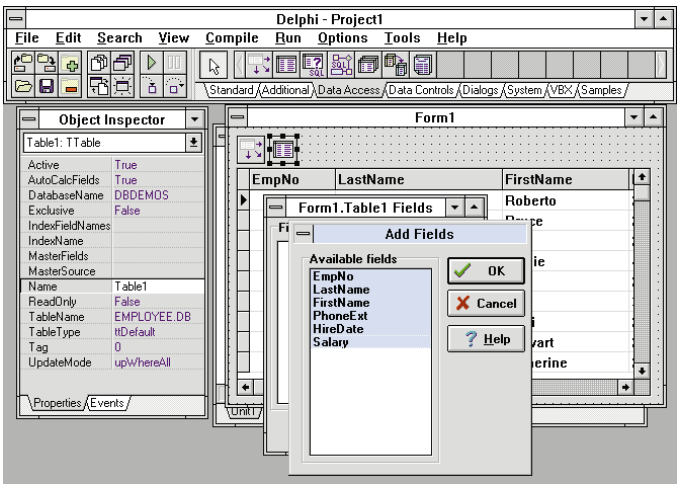



**Figure 5:** The Fields editor.

Initially the Fields editor is blank — no field names are listed in it. To add one or more fields, select the **Add** button. This causes the Add Fields dialog box to be displayed (see Figure 6). It contains a list of all fields not yet selected from the table pointed to by the Table object. All the displayed field names are selected by default. To create *TField* objects for each of these field names select **OK**. To create *TField* objects for a subset of these fields, highlight only those fields you want *TField* objects for, and click **OK**. (Hold down Ctrl when selecting and de-selecting individual fields.) For the current example, press **OK** to select all fields.



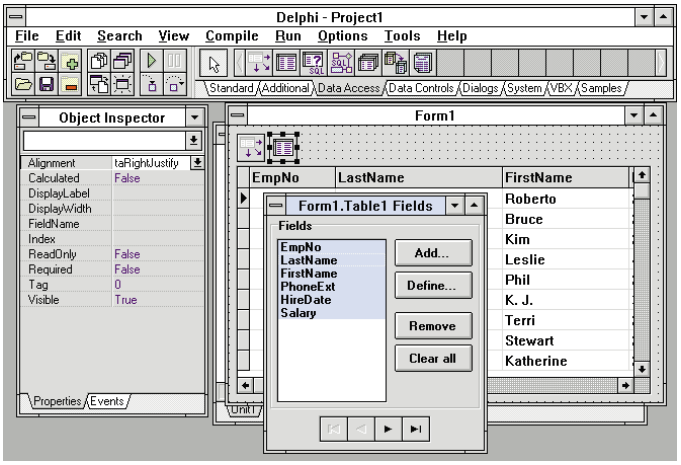



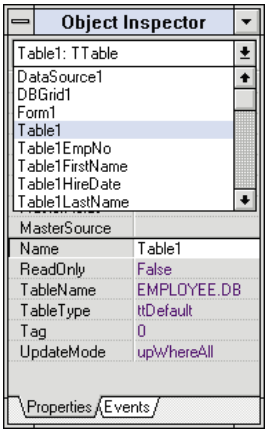

**Figure 6 (Top):** The Add Fields dialog box. **Figure 7 (Bottom):** *TField* objects will be instantiated for all fields selected in the Fields editor.

Once you have accepted the Add Fields dialog box, control returns to the Fields editor, and the selected fields appear in the **Fields** list, as shown in Figure 7. You can now close the Fields editor (press Alt F4 or double-click the Control menu) to instantiate *TField* objects for those selected fields.

Once you close the Fields editor, the DBGrid displays only those fields you selected. Consequently, the Fields editor is useful when you want to display only selected fields from a table. Also, all the selected fields now appear as objects in the Object Inspector's drop-down list box (see Figure 8).

Now that you have created *TField* objects for your table, you can access the properties and events for individual fields in the table. To do this, use the Object Inspector to select a field object. In this example, select the object named `Table1LastName`. When you do, the properties of this *TStringField* object are displayed in the Object Inspector, as shown in Figure 9.

The *TField* objects can also be selected in the Object Inspector using the Fields editor. To demonstrate this, double-click the Table object to display the Fields editor again. Begin by selecting one of the fields listed in this dialog box. Notice



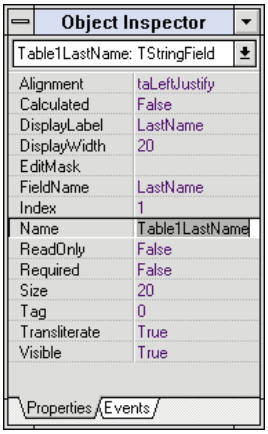




**Figure 8 (Left):** Once instantiated, the *TField* objects are selectable from the Object Inspector. **Figure 9 (Right):** A *TStringField's* properties in the Object Inspector.

that once you select a field, its properties and events are displayed in the Object Inspector. Now select another field in the Fields editor. Again, the Object Inspector updates itself and displays this newly selected field's properties and events.

## Some Selected TField Properties

While all these properties are important, let's consider a few properties that have special implications for the fields displayed in the DBGrid. For example, you can use the *Visible* property to control whether a field appears in the DBGrid. This property is especially important when you have selected all fields using the Fields editor (to instantiate a *TField* object for each field), but want to display less than all of these fields to the user (via the DBGrid). Any field not instantiated using the Fields editor must be manipulated using the *Fields* property of the Table component, instead of the *TField* object itself.

Another relevant property is *ReadOnly*. By default, all *TField* objects are created with their *ReadOnly* property set to *False*. Setting the *ReadOnly* property to *True* permits you to display the data of a field to the user, but prohibits the user from changing that value, regardless of the *State* property of the table. (The *State* property determines if, and to what extent, a user can edit a table.)

There is yet another property that is very important when your Object Pascal code needs to read or change the value in a field in a table — the *Value* property. And, because it's not a design time or published property, it's not listed in the Object Inspector.

To demonstrate the use of this property, add a Button to the form from the Standard page of the Component Palette. With the button selected, change its *Caption* property to `&Show Last Name`. Now double-click the button and enter the following code into its *Button1Click* procedure:

```
ShowMessage(Table1LastName.value);
```

and run the form. When the running form is displayed, click the button. Your form will display a dialog box, as shown in Figure 10. This dialog box displays the text that appears in the LastName field

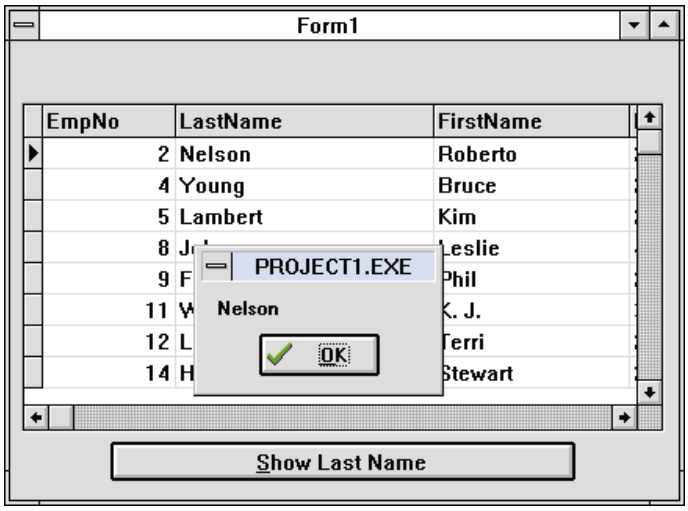



**Figure 10:** A dialog box displays the last name of the current Employee table record.

of the current record (which is the first record in this case). Accept the dialog box and change the current record in the DBGrid. Now click the button a second time. Again, the value displayed in the LastName field of the current record is displayed. Close the dialog box again.

While the code in the *Button1Click* procedure is simple, it cannot work for every *TField* descendant. For example, a compiler error would be generated if you enter the following code:

```
ShowMessage(Table1EmpNo.value);
```

This is caused by the EmpNo field (in the table Employee) not being an integer field, and the corresponding *TField* object being of the type *TIntegerField*. Furthermore, the *ShowMessage* procedure requires an actual parameter of the type *String*. Therefore, it's necessary to use the *AsString* property of the *TIntegerField* object to display the EmpNo value in this dialog box. For example, the following code will work:

```
ShowMessage(Table1EmpNo.AsString);
```

## TFields without DBGrids

At this point, it must be emphasized that instantiating the *TField* objects didn't require either the DBGrid or DataSource — only the Table object was required. At first, you may wonder why you would ever want to instantiate a *TField* object when no DBGrid exists. But, it's through the *TField* objects that you can easily edit and manipulate data in tables programatically.

This is demonstrated in the following steps. Begin by deleting the DBGrid and DataSource objects from your form. Now select the Standard page of the Component Palette and add two Button objects to the form. Change the *Caption* property of the first button to `&Next` and then double-click it to display its *OnClick* event handler. Enter the following code into the displayed procedure:

```
Table1.Next;
```

Now select the second button and change its *Caption* property to `&Previous`. Double-click it and add the following code to the displayed procedure:

```
Table1.Prior;
```

Now press F9 to run the form. Once the form is running, press the **Show LastName** button to display the value from the LastName field associated with the first record in Employee table. Accept this dialog box by pressing **OK**. Now click on the **Next** button. If you then again click the **Show LastName** button, the contents of the LastName field associated with the second record in the Employee table will be displayed.

## Conclusion

*TField* objects permit you to access and control the data stored in fields in a table. You use the Fields editor to instantiate *TField* objects at design time for a given Table object. You can modify the design time properties of these *TField* objects using the Object Inspector. Δ

Cary Jensen is President of Jensen Data Systems, Inc., a Houston-based database development company. He is a developer, trainer, and author of numerous books on database software. You can reach Jensen Data Systems at (713) 359-3311, or through CompuServe at 76307,1533.



*By Jim Allen & Steve Teixeira*

# A 3-D Label Component

## Designing and Creating a Component Step-by-Step

**So you want to build a Delphi Component? Notice the word *build* — it's a very appropriate verb to use in component design. Creating a component entails a brick-by-brick building approach, where the bricks are objects, variables, and type declarations. Once you've built your component with these bricks, you have something — hopefully something that's *useful*.**

In this article, we'll build a 3-D Label component called *TLabel3D*, using the criteria and techniques discussed in the first installment of "From the Palette." [See Jim Allen and Steve Teixeira's article "Component Basics" in the Premiere issue of *Delphi Informant*.] This article will show how easy building components can be, and delve into useful techniques of component design and programming.

### The Idea

One way to brighten a form's appearance is to use 3-D text. The standard technique of reproducing this effect is to layer three Label components that have the same caption. The bottom two labels have their colors set to white and black to emulate light and shadow, and the top label has any color (except white or black) as shown in Figure 1.

First, set the labels' *Transparent* properties to *True*. Then you offset the position of the white label from the top label by one position up and to the left, and the black label one position down and to the right. This will give the group of labels a 3-D effect. To give the labels a lowered appearance, simply switch the positions of the white and black labels. You can also easily change the text by selecting all three labels and editing the *Caption* property. This is a pretty cool idea, but a little tedious to get right.


After using this technique a few times it becomes readily apparent that a new component could be created to replace the whole process. It would be a 3-D Label that could be dropped on a form and used as a single label component. This would save time during development and you could use this 3-D Label more often since it's simple to use.

### Checking the Criteria

In the last article we discussed the three criteria for a good component idea: simplification, reusability, and uniqueness. These help you decide when to create a component and if it will be useful. We'll apply these criteria to our 3-D Label idea:

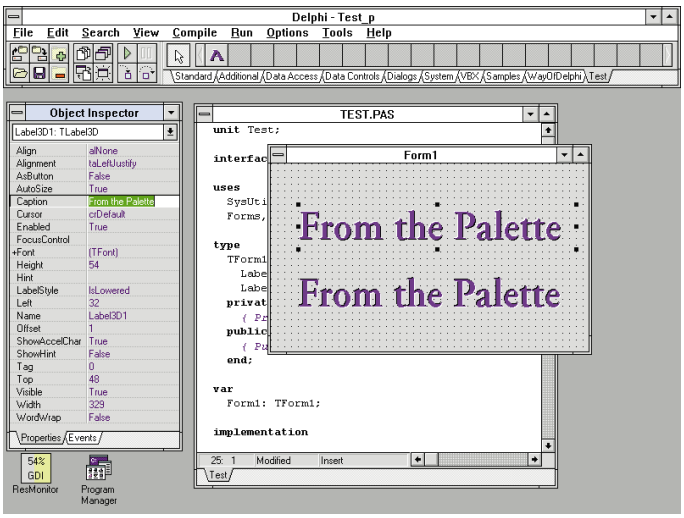



**Figure 1:** The custom 3-D Label component at design time. This form contains two 3-D Labels: one "raised" and one "lowered." Creating these effects by hand is tedious.

- **Simplification**. Would the 3-D Label component simplify a difficult or complex task? Considering the earlier tedious example of stacking three Label components, we know it's a "pain" so the answer is yes.
- **Reuse**. Would the 3-D Label component be easy to reuse? Again, yes. It can be dropped on the form like any other label component.
- **Uniqueness**. Would the 3-D Label Component be unique? Yes, because there is no built-in component with these attributes.

Now that the idea has successfully passed all the criteria, let's discuss how to implement it.

## Getting Started

What do we want the 3-D Label to do? It should, of course, display text in a 3-D manner in both the raised and lowered style (sounds like a great property to have, *LabelStyle*). There are a number of ways to implement this 3-D effect:

- Have a component that declares three *TLabel* components and sets the properties and positions of each to the appropriate value. However, although this technique is object-oriented, it violates the essence of object-oriented programming (OOP).
- Create a resource file with a 3-D bitmap for each character in the ASCII set and *BitBlt* the matching image to the screen. (Somebody actually suggested this technique.)
- Change the *Paint* method of a derived *TLabel* component to draw the text three times in the appropriate colors and positions. This sounds like the way to go because it's simple and doesn't require extensive coding.

Also, we may want to add a button-like action to the 3-D Label. That is, when you click on the label it recesses and then rises. While this may be just a novelty, it could come in handy. We can add another property called *AsButton*, that when set to *True*, makes the 3-D Label function as a button.

The implementation of the button functionality isn't as simple as the 3-D effect, but it shouldn't pose anything too difficult for us. We'll have to override the *MouseDown*, *MouseUp*, and *MouseMove* methods of the parent class (if it has them) to get the functionality we want.

## Choosing a Parent Class

Choosing the right ancestor class can take a few seconds or a couple of hours — depending on how well you know the Visual Component Library (VCL). Sometimes the ancestor is obvious because it only requires you to change or add to the functionality of an existing component (e.g. a password *TMaskEdit* or a default font setting on a *TLabel*).

In other instances, there are several classes that could be used as the ancestor of the new component. Figure 2 shows a decision tree to help you decide.

## Understanding the Decision Tree

Examining the Decision Tree can help you answer the following questions.

**1. Similar VCL object?** Is there an object in the VCL that's similar to the component you're creating? Consider this question carefully because most of the time the answer is yes, although some developers see their particular component as unique and therefore dissimilar

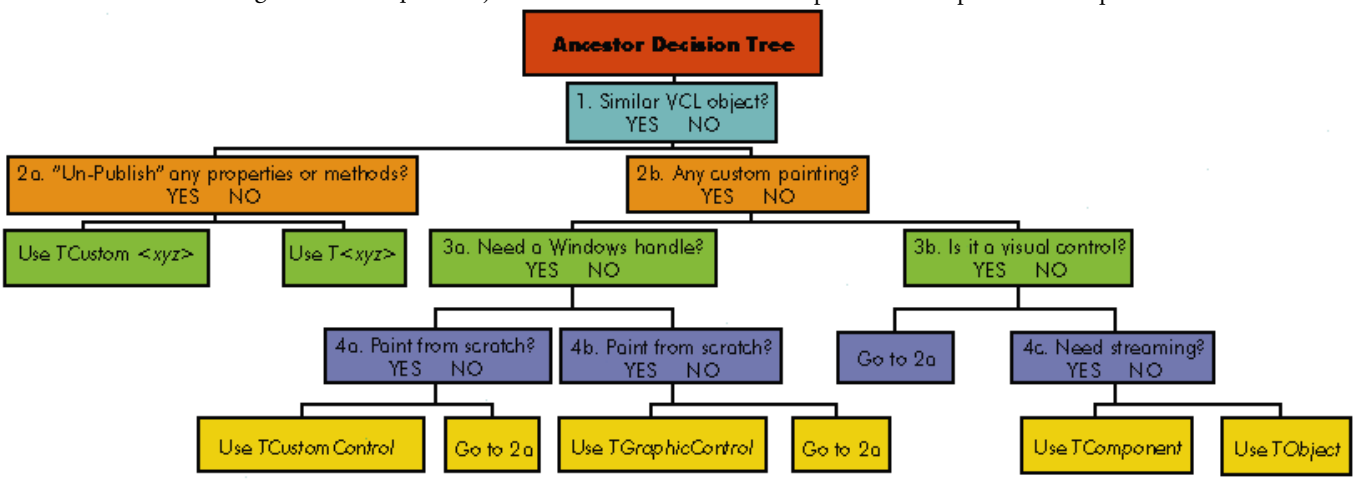



**Figure 2:** This decision tree can help you decide from which class to descend your new component.

| |
|---|
| *TCustomCheckBox* |
| *TCustomComboBox* |
| *TCustomControl* |
| *TCustomEdit* |
| *TCustomGrid* |
| *TCustomGroupbox* |
| *TCustomLabel* |
| *TCustomListbox* |
| *TCustomMaskEdit* |
| *TCustomMemo* |
| *TCustomOutline* |
| *TCustomPanel* |
| *TCustomRadioGroup* |

**Figure 3:** The custom classes.

to anything in the VCL. For example, if your component is a button type of control, then it will descend from one of the button classes (e.g. *TButtonControl*, *TButton*, or *TSpeedButton*). Our 3-D Label is almost identical to a *TLabel*, so the answer is yes.

**2a. "Un-Publish" any properties or methods?** If there's a similar component to derive the new component from, then are there any properties or methods that you *don't* want to include? If so, descend from the custom class of that component that doesn't publish any properties or methods, and publish only those items that you wish to surface. As a reminder, the custom classes are listed in Figure 3. If the properties and methods of the named class are adequate, then use the class itself to derive the new component.

There are a couple of properties that we don't want (since they're outside the scope of this month's article). We'll use *TCustomLabel* as our ancestor class.

**2b. Any custom painting?** In other words, will you use part of the existing *Paint* method, or re-write most or all of the visual portion of the component? This question asks you to consider the component's appearance. Again, if the component is a button and you want it to look like a regular Windows button, then don't change the *Paint* method. On the other hand, to get a unique looking component you'll probably have to write a new *Paint* method.

This doesn't really apply to our 3-D Label. Although we'll re-write some of the *Paint* method of a *TLabel*, we're not creating anything completely new. The 3-D Label has the same kind of functionality as the *TLabel*. It just changes the *Paint* method slightly.

**3a. Need a Windows handle?** Does Windows need to know about this component? Is it a Windows control that can receive input focus and needs a handle that it can pass to API functions? These questions are usually tough for new component builders for two reasons.

First, they aren't sure if the control is going to have direct interaction with Windows. Second, they don't know what a Windows handle can do for them. To decide if the control is a "gadget", ask yourself if this control is a major portion of the application or something that adds flavor. To understand what a Windows handle can accomplish, you're better off reading a Windows API reference. (Delphi ships with on-line Windows API help for quick reference.)

This doesn't apply to our 3-D Label component. Our component is more of an addition — it makes our applications more visually appealing. As for the button-like functionality mentioned earlier, it doesn't require a Windows handle. Speed buttons are descended from *TGraphicControl* and don't have a handle or receive input. Also, since *TGraphicControl* descendants are drawn on their parent's surface and don't have a Windows handle, they're easier on Windows resources.

**3b. Is it a visual control?** Is this something that will be visible at run-time? If it's a visual control without custom painting, then you may have made a mistake and there's most likely a VCL component that is similar. Go back and take a look.

Our control is definitely visual. This question doesn't apply to the 3-D Label component because we already know it descends from *TCustomLabel*.

**4a./4b. Paint from scratch?** If you're painting everything from scratch — meaning you're creating a graphic or "gadget" component and need a Windows handle, then use *TCustomControl*. However, if you don't need a Windows handle, use *TGraphicControl*. Lastly, if you're not painting from scratch — regardless of whether it needs a Windows handle — you should again look at one of the existing classes for an ancestor because you may have overlooked something. The 3-D Label doesn't require any of these modifications.

**4c. Need streaming?** Will you place the new component on the Component Palette, and drag-and-drop it onto the form at design time? Or will it be used in code like *TIniFile* or *TPrinter*? *Streaming* gives a component the ability to store its state to a form or binary file. This is the lowest common denominator for using the component as a visual design tool. If you want to drag-and-drop the new component, then you need to use *TComponent* as the ancestor class and do some streaming. If not, then *TObject*, *TList*, or *TString* might be the correct ancestor class. Our new label will have built-in streaming.

After filtering our 3-D Label through this tree, we have determined that *TCustomLabel* should be its ancestor class. It has all the functionality we want to either publish or override, and it doesn't publish anything of its own. We also have a name, *TLabel3D*. We can't use *T3DLabel* because when we create an instance of a variable, it cannot start with a number for the name. An instance of *T3DLabel* would become *3DLabel1*.

Before creating the class declaration for *TLabel3D*, we need to know exactly what the class definition will include. Object-oriented programming demands planning before coding.

## Property Considerations

While discussing ways of implementing the new component, we uncovered the *LabelStyle* and *AsButton* properties. Before completing their class definitions, we'll need to account for any special considerations. Luckily, *AsButton* is pretty straightforward. It should hold a *True* or *False* value so it's a variable of type *Boolean*.

However, *LabelStyle* is a bit more involved. We could call the property *Raised* (or *Lowered* for that matter), and give it a *True* or *False* value to determine its state based on the property name. For example, if the property is called *Raised* and the value is *True*, then Label3D would be raised. If the value is *False* it would be lowered.

Unfortunately, that's not descriptive enough for us. Remember, we are trying to simplify the whole 3-D process — not make it more cryptic. If we use an *Enumerated* type we can have almost any property values we want. In this case we want the values to be *Raised* and *Lowered*, or more specifically for our property of *LabelStyle*, *lsRaised* and *lsLowered*. The lowercase *ls* (LS) represents a property value of *LabelStyle*).

This requires us to have an additional **type** declaration in the unit that contains the code for *TLabel3D* (or at least in a unit used by *TLabel3D*). For example:

```
type
  TLabelStyle = (lsRaised, lsLowered);
```

Along with most of the traditional *TLabel* properties in the **published** section of our class definition, these are the only properties we want to include right now. There are a few properties we won't be using — for simplification purposes and because some properties would require additional coding. (Again, this is beyond the scope of this article.)

## Event Handler Considerations

There really aren't any considerations here for *TLabel3D* because we aren't going to create any event handlers. However, we are going to surface a few of the inherited event handlers from the ancestor *classes*. (Notice the word class*es* here.)

We can surface any **protected** or **public** property or event handler (which is essentially treated as a property) from any of the ancestor classes — provided it hasn't been overridden by a closer ancestor class. This is a very useful way of hiding implementation details throughout your class hierarchy.

## The Code

Figure 4 shows the code for the *TLabel3D* class. Where did the "F" variables in the **private** section of the class came from? And why are the *SetOffset*, *SetLabelStyle*, and *SetAsButton* procedures in there?

The F variables are the *fields* of the property. They are the private storage place for the information stored in the property. They are usually denoted by an "F" preceding the variable name. The **property** declarations are the interface to the values. This technique allows you to hide the implementation of the properties and allows you to control access, through those *Set...* procedures, to the actual data via the **read** and **write** definitions (the accessor methods).

We also added a property to the **public** section of the class — the *Offset* property — and its corresponding field variable and accessor methods. *Offset* will be used as a run-time only property that governs the position offset for the two background colors

```
TLabel3D = class(TCustomLabel)
private
  FOffset: integer;
  FLabelStyle: TLabelStyle;
  FAsButton:   Boolean;
  procedure SetOffset(Value: Integer);
  procedure SetLabelStyle(Value: TLabelStyle);
  procedure SetAsButton(Value: Boolean);
protected
  procedure Paint; override;
  procedure MouseDown(Button: TMouseButton;
                      Shift: TShiftState;
                      X, Y: Integer); override;
  procedure MouseMove(Shift: TShiftState;
                      X, Y: Integer); override;
  procedure MouseUp(Button: TMouseButton;
                    Shift: TShiftState;
                    X, Y: Integer); override;
public
  property Offset: Integer read FOffset
                           write SetOffset default 1;
  constructor Create(AOwner: TComponent); override;
published
  property Align;
  property Alignment;
  property AsButton: Boolean read FAsButton
                             write SetAsButton;
  property AutoSize;
  property Caption;
  property Enabled;
  property FocusControl;
  property Font;
  property LabelStyle: TLabelStyle read FLabelStyle
                                   write SetLabelStyle;
  property ShowAccelChar;
  property ShowHint;
  property Visible;
  property WordWrap;
  property OnClick;
  property OnDblClick;
  property OnMouseDown;
  property OnMouseMove;
  property OnMouseUp;
end;
```

**Figure 4:** The *TLabel3D* component's class declaration.

(white and black). This can be changed to define the amount of "depth" of the 3-D effect.

The **property** declarations not followed by accessor methods are the properties we're surfacing from the ancestor class — that's all we have to do to surface those. Now let's write the code for the methods.

## TLabel3D.Create

You usually override the *Create* method of your component, and the first line almost always calls the inherited *Create* method (see Figure 5). We also set all properties that are declared with the **default** keyword in the **property** declaration.

The **default** keyword only tells the Object Inspector what to display as the default selection. You must still set the property in the constructor. Properties of ordinal values, such as *AsButton* (Boolean: 0 = *False* or 1 = *True*) and *LabelStyle* (*TLabelStyle*: 0 = *lsRaised* or 1 = *lsLowered*), are internally defaulted to the first (0) value.

```
constructor TLabel3D.Create(AOwner: TComponent);
begin
  inherited Create(AOwner); { Call TCustomLabel's Create }
  FOffset     := 1;         { FOffset matches default }
  Transparent := True;      { Transparent must be True }
  with Font do              { Set up default font }
    begin
      Name  := 'Arial';
      Size  := 12;
      Color := clBlue;
      Style := [fsBold];
    end;
end;
```

**Figure 5:** Overriding the *TComponent*'s *Create* method to create the constructor for our custom component.

## TLabel3D.Paint

The *TLabel3D.Paint* method is the most time consuming to produce (see Listing Three). First set fonts, colors, and brush styles. Then, define a rectangle that is offset by the *Offset* property for the first (white) background text. Call the API procedure *DrawText* to place the text in the client area of the component. Repeat these steps for the second (black) background text.

After painting the background text, reset the *Font.Color* property to the regular color and call the inherited paint method of *TCustomLabel*. (Fortunately we don't have to offset the position.)

The raised and lowered effect is governed by the value of the *Offset* property. The *Offset* property uses a negative number to achieve a lowered style and a positive number for the raised style. We can modify the *Offset* value by changing the *LabelStyle* property.

## Accessor Methods

The *Set...* accessor methods do more than just set field values. This is one reason for using them instead of accessing the field directly from the property. For example:

```
property LabelStyle: TLabelStyle read FLabelStyle
                                 write FLabelStyle;
```

The *SetLabelStyle* method also toggles the *Offset* property between positive and negative, as discussed earlier, and changes *AsButton*. Also the *SetAsButton* method changes the *LabelStyle* to *lsRaised* when *AsButton* is set to *True*, as shown in Listing Three.

The *SetOffset* method:

```
procedure TLabel3D.SetOffset(Value: Integer);
begin
  if FOffset <> Value then
    begin
      FOffset := Value;
      Invalidate;
    end;
end;
```

invalidates the component, forcing Windows to repaint the object. This is the only place that the *Invalidate* method is called directly because any other change that might cause the component to need repainting calls (or calls something that calls) the *SetOffset* method.

## The Button Stuff

The methods that give *TLabel3D* its "button" functionality are as straightforward as the others. Each verifies that the component is set as a button (i.e. that *AsButton* is set to *True*) and takes the appropriate action based on the method. If the component isn't a button, each calls the inherited method. The *MouseDown* method sets the *LabelStyle* property to *lsLowered*, as shown in Listing Three.

*MouseMove* changes the *LabelStyle* property if the mouse cursor is moved off or onto the client area of the component while the mouse button is still being pressed. This is the same functionality that most buttons have. The *MouseUp* method assigns *lsRaised* to the *LabelStyle*. (The code for *MouseMove* and *MouseUp* is also shown in Listing Three.)

## The Icon

All that's left for us to do is create the icon that will represent our custom component on the Component Palette. Open the Image Editor by selecting **Tools | Image Editor**. Then choose **File | New**. At the New Project dialog box select **Component Resource (DCR)**. You'll want to make the image 20x20.

Considering the diversity of users, try to use only 16 colors. More importantly, the image needs to have the same name as the class definition. Also, the name for the .DCR file must be same as the unit. For example, the image should be Label3D.DCR and the file should be Label3D.PAS. Store the file in the same directory as the component unit. When you compile the *TLabel3D* component it will now have an icon on the Component Palette (see Figure 1).

## Enjoy

The *TLabel3D* component, while not spectacular, involves most of the concepts needed to become proficient at component design. In our next article, we'll take another look at the *TLabel3D* component and continue our trek into the world of Delphi component design. Δ

*The custom 3-D component referenced in this article is available on the 1995 Delphi Informant Works CD located in INFORM\95\JUN\JA9506.*

Jim Allen is an engineer in Borland's Technical Support Department. He supports object-oriented programming, and applications development for Borland's Delphi and Pascal. Jim also monitors the Delphi forum sections for Component Design and VBXes. If you have any comments, suggestions, or questions he can be reached on CompuServe at 70007,4655 or Internet at jallen@wpo.borland.com.

Steve Teixeira is a Senior Engineer at Borland International Technical Support. He writes for several industry periodicals, and he is the co-author of *Delphi Developer's Guide* from SAMS publishing. You can reach him on CompuServe at 74431,263 or on Internet at steixeira@wpo.borland.com.

### Begin Listing Three — Label3D.PAS

```
unit Label3d;

interface

uses
  SysUtils, WinTypes, WinProcs, Controls,
  StdCtrls, Classes, Graphics, Menus;

type
  TLabelStyle = (lsRaised, lsLowered);

TLabel3D = class(TCustomLabel)
private
  FOffset:     integer;
  FLabelStyle: TLabelStyle;
  FAsButton:   Boolean;
  procedure SetOffset(Value: Integer);
  procedure SetLabelStyle(Value: TLabelStyle);
  procedure SetAsButton(Value: Boolean);
protected
  procedure Paint; override;
  procedure MouseDown(Button: TMouseButton;
                      Shift: TShiftState;
                      X, Y: Integer); override;
  procedure MouseMove(Shift: TShiftState;
                      X, Y: Integer); override;
  procedure MouseUp(Button: TMouseButton;
                    Shift: TShiftState;
                    X, Y: Integer); override;
public
  constructor Create(AOwner: TComponent); override;

published
  property Offset: Integer read FOffset
                           write SetOffset default 1;
  property Align;
  property Alignment;
  property AsButton: Boolean read FAsButton
                             write SetAsButton;
  property AutoSize;
  property Caption;
  property Enabled;
  property FocusControl;
  property Font;
  property LabelStyle: TLabelStyle read FLabelStyle
                                   write SetLabelStyle;
  property ShowAccelChar;
  property ShowHint;
  property Visible;
  property WordWrap;
  property OnClick;
  property OnDblClick;
  property OnMouseDown;
  property OnMouseMove;
  property OnMouseUp;
end;

procedure Register;

implementation

constructor TLabel3D.Create(AOwner: TComponent);
begin
  inherited Create(AOwner); { Call TCustomLabel's Create }
  FOffset := 1;       { FOffset matches property default }
  Transparent := True;        { Transparent must be True }
  with Font do begin               { Set up default font }
    Name := 'Arial';
    Size := 12;
    Color := clBlue;
    Style := [fsBold];
  end;
end;

procedure TLabel3D.Paint;
const
  { Array that encapsulates some of
    WinProcs.DrawText's flags }
  Alignments: array[TAlignment] of Word =
   (DT_LEFT, DT_RIGHT, DT_CENTER);
var
  TempRect: TRect;
  Text: array[0..255] of Char;
  OldColor: TColor;
begin
  Canvas.Brush.Style := bsClear; { Set clear background }
  Canvas.Font := Self.Font; { Insure Canvas Font is set }
  OldColor := Font.Color;            { Save font's color }
  Canvas.Font.Color := clWhite;       { Make font white }
  { Create a rect that is offset for the top bevel }
  TempRect := Rect(ClientRect.Left - Offset,
                   ClientRect.Top - Offset,
                   ClientRect.Right - Offset,
                   ClientRect.Bottom - Offset);
  GetTextBuf(Text, SizeOf(Text));  { Get Label's Caption }
  { Draw offset text }
  DrawText(Canvas.Handle, Text, StrLen(Text),
           TempRect, DT_EXPANDTABS or
           DT_WORDBREAK or Alignments[Alignment]);
  Canvas.Font.Color := clBlack;     { Make font white }
  { Create rect that is offset for the bottom bevel }
  TempRect := Rect(ClientRect.Left + Offset,
                   ClientRect.Top + Offset,
                   ClientRect.Right + Offset,
                   ClientRect.Bottom + Offset);
  { Draw the offset text }
  DrawText(Canvas.Handle, Text, StrLen(Text),
           TempRect, DT_EXPANDTABS or
           DT_WORDBREAK or Alignments[Alignment]);
  Canvas.Font.Color := OldColor;{ Restore the old color }
  inherited Paint;               { Call inherited Paint }
end;

procedure TLabel3D.SetOffset(Value: Integer);
begin
  if FOffset <> Value then
    begin
      FOffset := Value;
      Invalidate;
    end;
end;

procedure TLabel3D.SetLabelStyle(Value: TLabelStyle);
begin
  if FLabelStyle <> Value then
    begin
      { Set AsButton property to False if it is True,
        and we're in design mode }
      if FAsButton and
         (csDesigning in ComponentState) then
        SetAsButton(False);
        SetOffset(FOffset * -1);    { Set FOffset field }
        FLabelStyle := Value;
    end;
end;

procedure TLabel3D.SetAsButton(Value: Boolean);
begin
  if FAsButton <> Value then
    begin
      if Value then
        { If label isn't already raised... }
        if LabelStyle <> lsRaised then
          begin
            SetLabelStyle(lsRaised);  { raise it }
          end;

      FAsButton := Value;
```

```
    end;
end;

procedure TLabel3D.MouseDown(Button: TMouseButton;
                             Shift: TShiftState;
                             X, Y: Integer);
begin
  if AsButton then
    begin
      { if it's a left mouse button and
        the label isn't lowered... }
      if (Button = mbLeft)         and
         (LabelStyle <> lsLowered) and
         Enabled                   then
        SetLabelStyle(lsLowered);{ Set Style to Lowered }
    end
  else
    inherited MouseDown(Button, Shift, X, Y);
end;

procedure TLabel3D.MouseMove(Shift: TShiftState;
                             X, Y: Integer);
var
  NewState: TLabelStyle;
begin
  if AsButton then
    begin
      { If it's a left mouse button... }
      if Shift = [ssLeft] then
        begin
          { Cursor is within Label's client area... }
          if (X >= 0)            and
             (X < ClientWidth)   and
             (Y >= 0)            and
             (Y <= ClientHeight) then
            NewState := lsLowered   { make it Lowered }
          else
            NewState := lsRaised;  { otherwise, Raised }
          if NewState <> FLabelStyle then
            SetLabelStyle(NewState);
        end;
    end
  else
    inherited MouseMove(Shift, X, Y);
end;

procedure TLabel3D.MouseUp(Button: TMouseButton;
                           Shift: TShiftState;
                           X, Y: Integer);
begin
  if FAsButton then
    begin
      { The cursor is within Label's client area... }
      if (X >= 0)            and
         (X < ClientWidth)   and
         (Y >= 0)            and
         (Y <= ClientHeight) then
        SetLabelStyle(lsRaised);  { make it Raised }
    end
  else
    inherited MouseUp(Button, Shift, X, Y);
end;

procedure Register;
begin
  RegisterComponents('Test', [TLabel3D]);
end;

end.
```

**End Listing Three**

# At Your Fingertips

BY DAVID RIPPY

DELPHI / OBJECT PASCAL

*If your rate of learning is slower than the rate of change, you are falling behind . . .*

— Jason B. Jones, Rocket Scientist

## How can I create an incremental search field for a DBGrid component?

This tip is a perfect example of what little code it takes to perform a significant programming task in Delphi. The form in Figure 1 shows a DBGrid component and an Edit component labeled **Search**. The Edit component allows the user to advance to a specific customer name quickly — as opposed to scrolling through the records in the customer table.


```
SEARCH.PAS
var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Edit1Change(Sender: TObject);
begin
  Table1.FindNearest([Edit1.Text]);
end;

end.
```

**Figure 1 (Top):** The record indicator automatically advances to the closest match on the DBGrid. **Figure 2 (Bottom):** Attach this code to the Edit component's *OnChange* event.

For example, to advance to the name "Smith", the user begins typing `Smith`. The application will advance to the record whose name most closely matches what the user is entering. When the user enters "S" the DBGrid will advance to the first record with a name beginning with "S", then "Sm", "Smi", and so on.

Using the amazing *FindNearest* method, only one line of code is required to create an incremental search field! Just add the Object Pascal code shown in Figure 2 to the *OnChange* event of the Edit component. — *D.R.*

## Why do the colors of my graphic images look wrong when I place them on a form?

Have you ever placed a graphic on a form, only to find it horribly discolored like the one shown in Figure 3? The problem is in the palette. Every Windows bitmap has a color palette associated with it that defines the set of colors for the picture. This holds true for the small bitmaps that are used as the glyphs on BitBtn components.

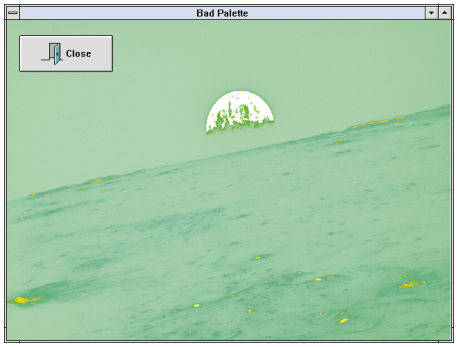



**Figure 3:** Conflicting color palettes can be pretty ugly!

The problem occurs whenever you place two or more bitmaps (including BitBtn components) with different color palettes on the form at the same time. Windows is forced to choose one of the color palettes at the expense of the other. This results in one of the pictures looking perfect, while the other looks like bad 60's art.

To prevent this, you should use the standard Windows color palette whenever possible. While it may not be the perfect palette for every picture, it will provide you with an acceptable standard palette to use within your application. By using the standard Windows color palette, you can place as many pictures and icons on a form as you need without the graphic images becoming distorted.

Figure 3 shows a form containing a graphic of the moon, and one BitBtn component containing a small graphic of a door. The glyph of the door is the culprit. It was originally clipped from another application using a non-standard color palette. Windows looked at the two graphics and decided to use the BitBtn component's color palette at the expense of the Earth and its moon.

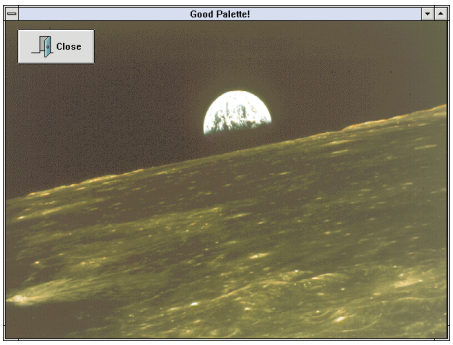



**Figure 4:** That's better! Here both pictures use the standard Windows color palette.

To remedy this problem, edit the door graphic in your favorite paint program that allows you to change color palettes (I use PaintShop Pro). Select the option to change the picture's palette to the standard Windows palette, and save the new version of the image. Now, when you reload the modified image into the BitBtn component's glyph, the two graphics on the form should co-exist peacefully, as shown in Figure 4.
— *Tony Goodman, Ensemble Corporation*

## How can I prevent certain columns from displaying in my DBGrid?

There are a few steps to take before you can remove a column from a DBGrid component. Our example form in Figure 5 shows a DBGrid with two columns, **Item** and **Price**. If we want to remove the **Price** field, we must first access the Fields editor by double-clicking on the Table object (see Figure 5). Next, press the **Add** button to define which fields from the Magic table we want to have access to in our application. Since we want to modify the *Visible* property of the **Price** field, select it from the list.

Now that we have selected the fields from the Magic table, they appear as objects that can be accessed from the Object Inspector (see Figure 6). The form is now "aware" of the fields as objects that can be manipulated individually, and gives us many additional properties that we can change at design-time and run-time.

Select the object named *Table1Price* from the Object Inspector. You will notice many properties that can be changed including the font, size, color, etc. We're interested in the *Visible* property near the bottom of the list. Change its value to *False*, and the **Price** column will be removed from the DBGrid, as shown in Figure 6. — *D.R.*

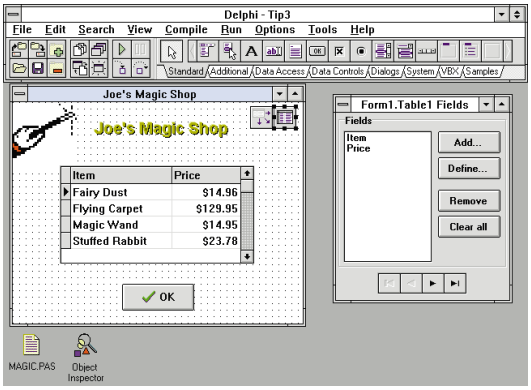



**Figure 5:** Double-clicking on a Table component invokes the Fields editor.

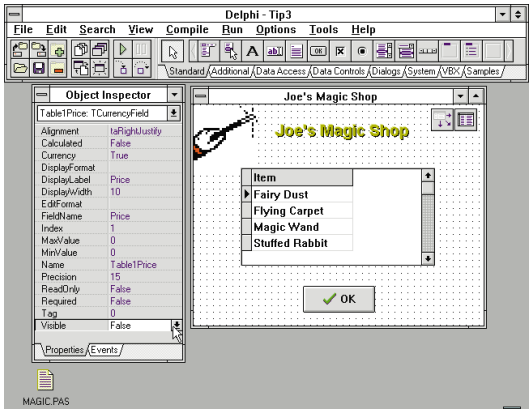
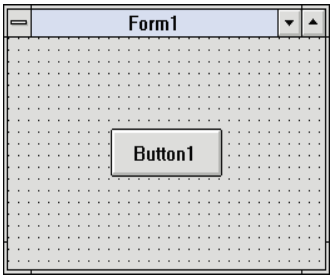



**Figure 6:** Set the *Visible* property to *False* and the **Price** column will magically disappear!



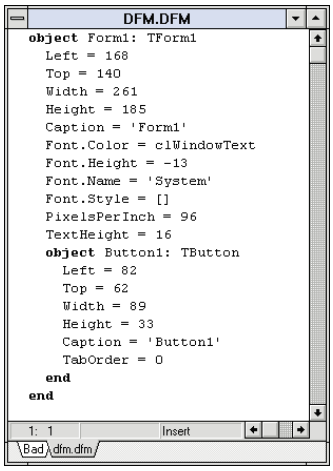




**Figure 7:** A simple form and its .DFM file.

## How can I document the objects and their attributes that I've placed on a form?

Soon there will be many third-party tools to help document forms. Until then, you can explore the .DFM file that is created when you compile your form. Every form has a .DFM file associated with it to define the objects that are placed on the form, as well as many of the object's key attributes such as location, size, color, and caption. Delphi uses the .DFM file to create the form at run-time. The source code attached to these objects is, of course, stored in the .PAS files that you edit when creating event handlers for the form.

To view a .DFM file, select **File | Open File** from the main menu. Next, choose **Form file (*.DFM)** from the **List Files of Type** drop-down list. Figure 7 shows a simplistic form and its corresponding .DFM file. Notice the object definition for both the form and button in the .DFM listing.

Hopefully you'll never need to recreate a lost or deleted form from scratch, but if you do, you can refer to the form's .DFM file to get a head start on the objects you need. Δ
— *Mike Leftwich, Ensemble Corporation*

*The demonstration forms referenced in this article are available on the 1995 Delphi Informant Works CD located in INFORM\95\JUN\DR9506.*

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, and is a contributing writer to *Paradox Informant*. David can be reached on CompuServe at 74444,415.

table of CONTENTS

BY GARY ENTSMINGER

# Conversion Assistant

## EarthTrek's Visual Basic-to-Delphi Conversion Tool

If you've developed a lot of great code in one language and then switch to another, nothing makes you lighter than an appropriate language translator. Since much of my code library is in Visual Basic (VB), and much of my current coding is in Object Pascal (the underlying language of Delphi), I was delighted to hear about EarthTrek's Visual Basic-to-Delphi conversion tool — Conversion Assistant.

The Conversion Assistant translates all the Visual Basic form (.FRM) and code (.BAS) files in a VB project to Delphi forms (.DFM) and code (.PAS) files. It then generates a corresponding Delphi project (.DPR) file from the VB project (.MAK) file.

While the Conversion Assistant isn't perfect, it does give you a good start toward translating your VB code. In this review, we'll walk through a project that includes some of the basic elements and components in a typical project. We'll look at the original VB code, the Conversion Assistant's translated code, and the code needed to make the application compile and work correctly in Delphi. We'll also discuss the specific changes made to duplicate the behavior of the original VB application.

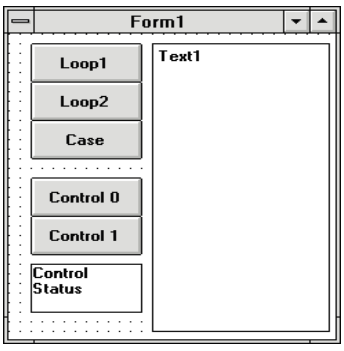



**Figure 1:** The sample VB form, LoopTst, in VB design mode.

### The Interface

Figure 1 shows the VB test project CALOOP.MAK. It consists of a form that has a text box, label, and five command buttons. Note that two of the command buttons are part of a control array.

A *control array* is a group of controls of the same type (for example, all command buttons or text boxes) that share a control name and set of event procedures. Each control in the array has a unique index that allows manipulation of individual controls. I use control arrays frequently in VB, and they're another good test of the Conversion Assistant's interface translation capability.

### A Prerequisite

If you're using VB 3.0, you must save VB files "As Text" not "Binary" before you try to convert them. Otherwise, the Conversion Assistant will flash an error message.

A typical conversion produces a Delphi .PRJ file, several Delphi form .DFM files, and several Delphi .PAS code files. Some conversions create a Delphi GLOBS code file, a consolidation of all the VB global .BAS code files.

### Testing 1, 2, 3

The first procedure, Loop1, creates a two-dimensional array. It then uses a nested For loop to assign values to each cell in the array. Finally, it uses a message box to report the results. Here is the code for the nested For loop:

```
Sub Loop1 ()
  Dim I As Integer
  Dim J As Integer
  Static TestArray(10, 10)
  For I = 1 To 10
    For J = 1 To 10
      TestArray(I, J) = I
    Next J
  Next I
  MsgBox "Test Array 10x10 filled."
End Sub
```

After you save your VB code as text, run the Conversion Assistant and open the VB .MAK file for the project to convert. Figure 2 shows the Conversion Assistant's user interface.


**Figure 2:** The Conversion Assistant's user interface.

The Conversion Assistant is easy to use. It's menu-driven with a few options. From within the Conversion Assistant, you can view each file after it's converted or wait until the end of the process.

You can also view a .LOG file that reports messages and the status of converted files. If the Conversion Assistant doesn't recognize a file in the project (for example, if it tries to convert a VB file that's still in binary form), it displays an error message.

After you open a VB project, you convert it. Figure 3 shows the Conversion Assistant during the conversion of Loop 2. This segment of the resulting code:

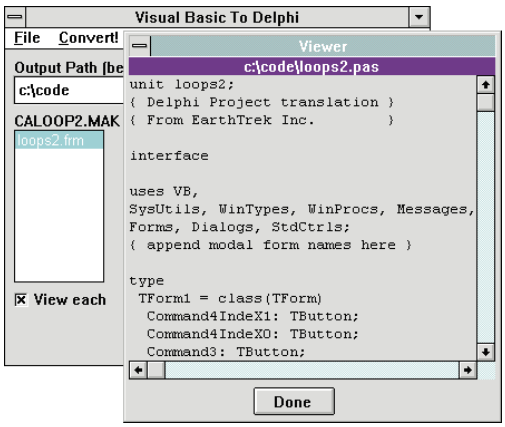



**Figure 3:** The Conversion Assistant Viewer during a conversion.

```
procedure TForm1.Loop1;
var
 I: integer;
 J: integer;
begin
  Static TestArray(10, 10);
  for I := 1 to 10 do
    begin
      for J := 1 To 10 do
        begin
          TestArray(I, J) := I;
        end;
    end;
  MsgBox 'Test Array 10x10 filled.';
end;
```

shows that the Conversion Assistant had some problems with the translation (indicated here in red). The smaller problem is that it didn't translate MsgBox. Fortunately, this is an easy fix. Each time you encounter MsgBox substitute with *ShowMessage*.

A bigger problem was caused by Conversion Assistant's inability to handle the array properly. Almost everything after the VB keyword, Static, is mush. However, this problem is fixable as well. Here's the corrected Object Pascal code:

```
procedure TForm1.Loop1;
var
  I : integer;
  J : integer;
  TestArray : array[1..10, 1..10] of integer;
begin
  for I := 1 to 10 do
    for J := 1 to 10 do
      TestArray[I, J] := I;
      ShowMessage('Test Array 10x10 filled.');
end;
```

## Loop2

The second procedure — Loop2 — uses a While loop, an If statement, the built-in VB Randomize procedure, and a Rnd function. Loop2 writes to a control's (Text1) text property. Here's the VB code:

```
Sub Loop2 ()
  Dim Dice1 As Integer
  Dim Flag As Integer
  Randomize
  Text1.Text = "Tossing"
  Flag = 77
  While Flag = 77
    Dice1 = Int(6 * Rnd + 1)
    If Dice1 < 6 Then
      Text1.Text = Text1.Text & Str(Dice1)
    Else
      Text1.Text = Text1.Text & Str(Dice1) & " !Bingo!"
      Flag = 1
    End If
  Wend
End Sub
```

The Conversion Assistant's translation is shown in Figure 4 with the problem areas highlighted in red.

The Conversion Assistant had two small problems with this code: it mishandled the Rnd function, and it didn't know what to do with the keyword, Wend. Again these are easy to fix, as shown in Figure 5.

```
procedure TForm1.Loop2;
var
  Dice1: integer;
  Flag: integer;
begin
  Randomize;
  Text1.Text := 'Tossing';
  Flag := 77;
  while Flag = 77;
    Dice1 := Int(6 * Rnd + 1);
    if Dice1 < 6 then
      begin
        Text1.Text := Text1.Text + VBstr(Dice1);
      end
    else
      begin
        Text1.Text :=
          Text1.Text+VBstr(Dice1)+' i.iBingo.';
        Flag := 1;
      end;
    Wend;
end;
```

**Figure 4:** The Conversion Assistant's translation of the Loop2 procedure.

```
procedure TForm1.Loop2;
var
  Dice1: integer;
  Flag: integer;
  Num: integer;
  DiceStr : string;
begin
  Randomize;
  Memo1.Text := 'Tossing';
  Flag := 77;
  while Flag = 77 do
    begin
      Dice1 := Random(6 + 1);
      if Dice1 < 6 then
        begin
          Str(Dice1,DiceStr);
          Memo1.Text := Memo1.Text + DiceStr;
        end;
      else
        begin
          Str(Dice1,DiceStr);
          Memo1.Text := Memo1.Text+DiceStr+' !Bingo!';
          Flag := 1;
        end;
    end;
end;
```

**Figure 5:** The corrected Object Pascal version of Loop2.

## CaseTest

The CaseTest procedure uses a Do loop, a Select Case statement, two comparison operators, an InputBox, and a MsgBox. Here's the VB code for CaseTest:

```
Sub CaseTest ()
  Dim NumToTest As Integer
  Dim Msg As String
  Dim Title As String

  Msg = "Enter a value between 1 and 10."
  Title = "CaseTest"
  Do
    NumToTest = InputBox(Msg, Title, "1")
    Loop Until NumToTest >= 1 And NumToTest <= 10
    MsgBox "You entered " & NumToTest ' Display message.
    Select Case NumToTest
  Case 1, 3, 5, 7, 9: Msg = "odd number"
  Case 2, 4, 6, 8, 10: Msg = "even number"
  End Select
  MsgBox Msg
End Sub
```

The Object Pascal code produced by Conversion Assistant is shown in Figure 6.

This procedure gave the Conversion Assistant more trouble. It wasn't sure how to handle the Do loop or the Select Case statement. It also has InputBox return an integer, although Delphi requires a string. This translation also has the same problem with MsgBox that we saw earlier.

Figure 7 shows the corrected Delphi code with the *Val* procedure added to translate the returned value of InputBox.

```
procedure TForm1.CaseTest;
var
  NumToTest: integer;
  Msg: string;
  Title: string;
begin
  (  )
  Msg := 'Enter a value between 1 and 10.';
  Title := 'CaseTest';
  Do;
NumToTest := InputBox(Msg, Title, '1');
  Loop Until NumToTest >= 1 And NumToTest <= 10;
  MsgBox 'You entered ' + NumToTest { Display message.};
  case NumToTest of
1: begin;
end;
2: begin;
end;
end; { c a s e }
  MsgBox Msg;
end;
```

```
procedure TForm1.CaseTest;
var
  NumToTest: integer;
  StrNum : string;
  Msg: string;
  Title: string;
  Code: integer;
begin
  Msg := 'Enter a value between 1 and 10.';
  Title := 'CaseTest';

  repeat
    StrNum := InputBox(Msg, Title, '1');
    Val(StrNum,NumToTest,Code);
  until (NumToTest >= 1) and (NumTotest <= 10);
    ShowMessage('You entered ' + StrNum);
    case NumToTest of
      1, 3, 5, 7,  9 : Msg := 'odd number';
      2, 4, 6, 8, 10 : Msg := 'even number';
  end; {c a s e}

  ShowMessage(Msg);

end;
```

**Figure 6 (Top):** Conversion Assistant's translation of the CaseTest procedure. **Figure 7 (Bottom):** Corrected Object Pascal version of the CaseTest procedure.

## Control Array Test

The VB Command4_Click event procedure handles the control array test. It uses an If test to test the Index value of each control array item. It then modifies the Caption property of the label depending on the result. The VB control array test procedure looks like this:

```
Sub Command4_Click (Index As Integer)
  If Index = 0 Then
    Label1.Caption = "Control array 0 pressed."
  Else
    Label1.Caption = "Control array 1 pressed."
  End If
End Sub
```

Here's the Object Pascal translation generated by Conversion Assistant:

```
procedure TForm1.Command4_Click(Sender: TObject);
begin
  if VBIndex(Sender) = 0 then
    begin
      Label1.Caption := 'Control array 0 pressed.';
    end;
  else
    begin
      Label1.Caption := 'Control array 1 pressed.';
    end;
end;
```

As you can see, the Conversion Assistant had no trouble handling this event procedure.

And that's it! Figure 8 shows the translated and corrected Delphi application at run-time.

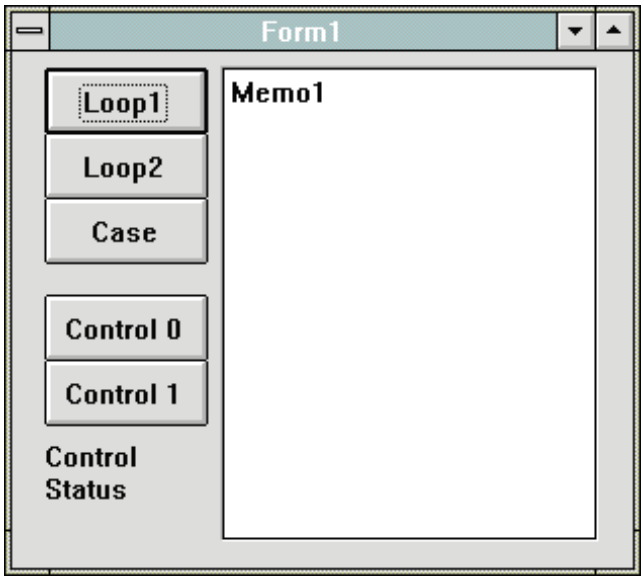



**Figure 8**: The translated and corrected Delphi application at run-time.

## The Bottom Line

Besides these tests, I translated several other VB applications to Delphi using the Conversion Assistant. In general, most application interfaces translated correctly or needed only small modifications.

The Conversion Assistant translated menus well and most code that contained Windows API functions. It also handles control arrays well.

However, Conversion Assistant did inappropriately translate a few interface object property values. For example, it translated a VB multi-line text box into a Delphi Edit component, not a Memo component (the Delphi equivalent of a multi-line text box). A snappier Conversion Assistant might check the VB textbox multi-line property, and seeing it set to True, create a Delphi Memo component.

Variable conversion is an issue as well. In VB, you can use a variable without declaring it, the variable then being variant by default.

You can also dimension variables by using suffixes (AnInt% for example). And you can declare several variables in the same line using commas (e.g. AnInt, AnotherInt, OneMoreInt As Integer). The Conversion Assistant had problems with all these scenarios.

In general, keywords in VB that do not have exact matches (for example, ReDim in VB) are ignored by the Conversion Assistant. They simply appear as VB code in the translation. Ignoring unrecognized keywords and identifiers is a reasonable approach, since this makes it easier to locate code that needs correcting.

VB dynamic arrays and error handling routines must be hand-coded after the translation. In these situations, the Conversation Assistant also retains the VB code.

## Conclusion

Conversion Assistant's on-line help system is incomplete. Also, when it encounters an error, it doesn't give you much information about the error. Otherwise, the interface is straightforward and easy-to-use.

**INFORMANT FACT FILE**

**The Conversion Assistant**
The Conversion Assistant converts Visual Basic project, code, and form files into equivalent files that can be read, modified, and executed in Borland's Delphi. It maintains references to VBX controls and has a compiler's syntax checking capability. Version 1.0 represents a good start on a great idea.

**EarthTrek**
79 Montvale Ave. #5
Woburn, MA 01801
**Phone:** (617) 273-0308
**Fax:** (617) 270-4437
**Price:** US$79

Conversion Assistant is based on a great idea and makes a good start toward becoming a valuable Visual Basic-to-Delphi translation tool. The more I used it, the more I appreciated what it accomplished. And with practice I learned how to more accurately "zero in" on code that needed modification.

Even with its flaws, the Conversion Assistant could save you programming time. Δ

*The complete Visual Basic code listing and Delphi code listings (Conversion Assistant-generated and corrected version) for this example conversion project are available on the 1995 Delphi Informant Works CD located in INFORM\95\JUN\CA9506.*

Gary Entsminger is the author of *The Tao of Objects, an Introduction to Object-oriented Programming, 2nd ed.* (M&T 1995) and *Secrets of the Visual Basic Masters, 2nd ed.* (Sams, 1994). He is currently working on *The Way of Delphi,* an advanced Delphi book for Prentice Hall, and is the technical editor for *Delphi Informant.*

# TEXT FILE

## Don't Be a Dummy, Get This Book

Or is that "Be a dummy! Get this book?" No, that doesn't sound right either. In any case, despite its title, Neil J. Rubenking's *Delphi Programming for Dummies* provides an excellent introduction to Delphi for dummies or anyone else. It's also very well organized and well written — in fact it's a page-turner.

I've read four of IDG's *Dummies* titles so far (the Paradox for Windows, Borland C++, and dBASE for Windows books were the others). They're all good, although the Delphi title is the best of the bunch. It contains remarkably few errors (typos, beta screen captures, etc.) and manages to pack an amazing amount of information between wisecracks into its 376 pages. This is especially praiseworthy since *Delphi Programming for Dummies* made it to the bookstores very quickly, and must have been substantially based on a pre-release version of Delphi. Several other early-to-market Delphi books show signs of the "rush to press".

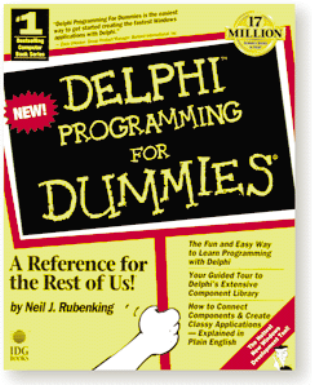
Just to demonstrate that *Dummies* isn't for the simple-minded, it begins with a description of Delphi's compiler options, before moving on to the other pages of the Project Options dialog box. It then quickly explains Delphi's Environment Options, Main Menu, SpeedBar, and other aspects of the IDE. These introductory sections also describe events, properties, and component interaction.

The next eight chapters are a whirlwind tour of the Component Palette. Each component is demonstrated with a step-by-step programming example that you build as you read. You will fall in love with Delphi (if you haven't already) long before you've completed these chapters. In fact, although I was reading *Dummies* to review it, I often found myself forgetting about the book and just enjoying Delphi. You just can't come away from this book without marveling at what a fabulous tool Borland has produced. From form design, to drag-and-drop, to file I/O, Delphi makes it simple. And Rubenking does an outstanding job of showing off Delphi's capabilities in an enjoyable way.

The next section, entitled "Real Programming", gets down to the nitty gritty of Object Pascal programming (e.g. units, project files, the interface and implementation sections, etc.) beginning with an examination of the code that Delphi generates by default for an "empty" form. Witty and succinct — like the rest of *Dummies* — it's the best introduction to the elements of Delphi programming I've read.

The next two chapters describe "Ten Common Mistakes" including caveats about semicolons and PChars; and "Ten Windows API Functions" including an application that displays Windows' system metrics, and a handy applet for exiting or restarting Windows, or simply rebooting the computer. In fact, several of the *Dummies* tutorials present useful programs. Another that comes to mind is an INI file viewer that displays INI files in outline form. The final two chapters introduce the Object Pascal run-time library, and work with "Five Hidden Delphi Objects" including the Printer object. An Appendix explores using VBXes in Delphi.

*Dummies* provides a comprehensive look at Delphi: the elements of a Delphi project, compiler directives, event handlers, using the **is** and **as** operators, variable declarations, variable scoping, string manipulation, calling DLL and Windows API functions, debugging, and much much more. In short, you'll feel




comfortable with the new tool and be well prepared to begin your own Delphi development project.

In brief, Neil J. Rubenking's *Delphi Programming for Dummies* is a non-stop romp through the new development environment's features, components, and language. It's also the best introduction to Delphi in the bookstores right now. Oh yeah — it's funny too.

— *Jerry Coffey*

***Delphi Programming for Dummies*** by Neil J. Rubenking, IDG Books Worldwide, 155 Bovet Road, Suite 310, San Mateo, CA 94402; (800) 762-2974 or (415) 312-0650.

**ISBN:** 1-56884-200-7
**Price:** US$19.99
376 pages

# The Gang of Four Speaks of Patterns

Software jargon changes constantly. Sure, we give new terms a chance, but we don't retain them all. Our brains can't hold them. And too many turn out to be empty hyperbole. If terms are useful they will endure. For example, some surviving terms from object-oriented jargon — *encapsulation*, *inheritance*, and *polymorphism* — are useful. They help us discuss a new approach to programming; they are the vocabulary of the object-oriented arena.

Now scouts are reporting from the OO frontier, but they're spending a long time in debriefing. Seems to be some problem understanding what the heck they're talking about! The frontier is Object-Oriented System Design and the scouts finally have a way to tell us what's "out there". They're called *patterns*.

In the mid-80s, Kent Beck read a book by the architect Christopher Alexander. Alexander's books explain why modern architecture produces many buildings that don't "work". The "pattern language" he invented makes it easier for designers to understand and create usable buildings. Patterns clarify when and why ideas work and how they reinforce each other. They also help users participate in design.

Beck saw many parallels to software issues, and he and Ward Cunningham applied the ideas to a current project. The results were encouraging, so Beck and Cunningham presented software-related patterns to OOPSLA 87 (the annual object-oriented programming convention). Since then, most participants at the OO conferences have accepted this Alexandrian Pattern form as a standard.

The authors of *Design Patterns* (published by Addison-Wesley) are members in good standing of the OO community, and with this book they have become almost a cult. They even have a nickname: the Gang of Four (or GOF).

*Design Patterns* shows why the patterns concept is so powerful when applied to OO design. But why do we need the new term? After all, it sounds a lot like a data structure specification, with perhaps a few ideas and rules for its application. It's mainly a difference of scale. Patterns can be generic solutions to large-scale system architecture problems.

Composing an architecture using patterns can save a good deal of strategic time, and provide a sounder architecture. The cost of getting things wrong in a system's architecture is high, so a pattern includes information to ensure it's applied in the correct cases, and that all alternatives and consequences are considered.

*Design Patterns* recommends that a pattern should contain four fully elaborated elements: a carefully selected name, the preconditions and indications for considering it, alternative implementations, and the pattern's consequences including the potential trade-offs. The pattern community allows a lot of latitude in how you fulfill these four requirements. In fact, the GOF authors present their patterns in far greater detail.

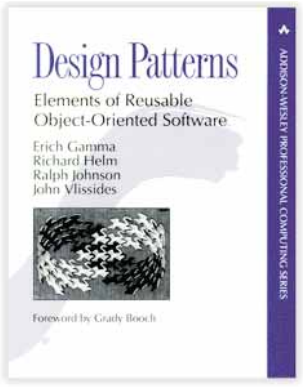
What kind of thing can be a pattern? Here are names and intents of two GOF patterns. 1) "Observer" defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. 2) "Mediator" defines an object that encapsulates the interaction of a set of objects. It promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interactions independently.

These two behavioral patterns make up about half of the 23 patterns in the book. Other sections deal with creational and structural patterns.

And there's more. An excellent and wise section discusses the realities of OO design and the common forces with which your design must contend. The "Designing for Change" section is full of useful insights, advice regarding the use of inheritance versus composition, and how to select and use a pattern. There is also a long case study that uses patterns to explain the design of a text editor program.

Each pattern section is preceded by an overview and followed by a "compare and contrast" summary. Examples focus on C++ and Smalltalk, but are just as applicable to Delphi OO programming.

If you want to know more, the Internet offers many




excellent papers. Different styles of patterns are described, along with general discussions and bibliographies. The best starting points are via WEB: http://st-www.cs.uiuc.edu/users/patterns/patterns.html, and FTP: st.cs.uiuc.edu/pub/patterns.

*Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, et al., is an important reference for all who design object-oriented software.

And by demonstrating the power of good object design, it will do much to explain and justify OO technology to those who remain doubtful or confused.

— *Richard Curzon*

***Design Patterns: Elements of Reusable Object-Oriented Software*** by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley Publishing Company, One Jacob Way, Reading, MA 01867; (800) 822-6339.

**ISBN:** 0-201-63361-2

**Price:** US$37.75
416 pages

table of CONTENTS