May 1996, Volume 2, Number 5

Cover Art By: *Doug Smith*

OLE Internet Access

Using Netscape Navigator as an Automation Server

ON THE COVER

9 **OLE Internet Access** — Joseph C. Fung

Netscape's Navigator dominates the Internet browser arena, but how can we leverage its power and popularity with Delphi 2? Mr Fung answers this question by presenting an example-laden guide to using Navigator as an automation server.

16 **An HTML Generator** — Keith Wood

Planning on building a Web site? Is your Object Pascal better than your HTML? Or is writing line after line of HTML simply too boring? If so, you'll be happy to read Mr Wood's HTML primer, and download his full-featured HTML-generating component.

FEATURES

26 **Informant Spotlight** — Ray Lischner

Virtual or **dynamic**? **Dynamic** or **virtual**? Which is faster? Which is "better?" Mr Lischner analyzes Object Pascal's two types of virtual methods, benchmarks their performance, and demystifies the arcana of DMTs and VMTs.

30 **DBNavigator** — Cary Jensen, Ph.D.

In this month's "DBNavigator," Dr Jensen provides us with an interactive tour of Delphi 2's enhanced Fields Editor and expands on his discussion of the new data module. It is now simpler than ever to create and place DBEdit and DBLookupCombo components.

34 **Dynamic Delphi** — Andrew Wozniewicz

In the third installment of his series on building DLLs in Delphi, Mr Wozniewicz shows us how to interface a DLL, export and import functions by ordinal numbers, explicitly load libraries, and more. He also provides us with step-by-step instructions to build the example application.

40 **In Development** — Craig Jones

Mr Jones finishes his three-part introduction to Quality Assurance. The focus this month is on the tools and processes you need to put in place to develop and deliver reliable software. The emphasis is on practicality, since these processes are worthless if they're never put to use.

45 **At Your Fingertips** — David Rippy

Our favorite tipster returns with tips and tricks for Delphi 1 and 2. This month, Mr Rippy reveals how to: tile bitmap images on forms and quickly make the same change to several objects programmatically. He also has some comments about Object Pascal comments.

REVIEW

47 ***Teach Yourself Delphi in 21 Days***
Book review by James Callan

47 ***Teach Yourself Database Programming with Delphi in 21 Days***
Book review by James Callan

DEPARTMENTS

2 **Editorial**
3 **Delphi Tools**
6 **Newsline**
49 **File | New**

When the snow falls the flakes / spin upon the long axis / that concerns them most intimately / two and two to make a dance ... — William Carlos Williams

Delphi and Java: Language Revolutions to Crown the 20th Century

Delphi is the crowning achievement of a programming environment revolution that began in 1983 when Turbo Pascal blasted us out of the CP/M command line and into the world of gorgeous IDEs (Integrated Development Environments). What a relief not to invoke a compiler and debugger from a command-line prompt. What a joy to have the cursor placed in the source code file at the position of the error with a useful explanation in a status bar. Even the assembly language demigods I rubbed desks with in those days were intrigued, while suggesting for weeks that Frank Borland and his cronies in some garage in Santa Cruz were promoting the hoax of the century.

But the rest, as they say, is history. And if Frank was promoting a hoax, it's still fool proof. Customizable speedy IDEs and programming languages have been sharing disk space for a decade, and we've been enjoying it. We enjoy it so much that we expect every new programming environment to have more than a few of our favorite things: code and string editors, object inspectors, project managers, built-in debuggers, component palettes, hooks to everything we can imagine, and so on. (Delphi… sigh…you gotta love it.)

But then, every now and then, something happens that shakes the foundations. This time the foundations are shaking from a revolution in the Online arena. Suddenly we're all drooling over the Internet, the World Wide Web, and Java. The hype is almost incredible, the enthusiasm contagious, the prospects staggering, and the products new and mostly as cyberspaced as cyberspace itself.

No…I haven't forgotten that there was a lot of hype about Windows 95. But that hype was mainly about an improvement in something we already knew about (Windows 3.x), not about something new and perhaps even a little crazy. The Internet and particularly the World Wide Web are attracting a kind of excitement and energy that was rampant in mid-80s PC development when hackers experimented with their computers and gave their code freely to the public domain. When it seemed like everyone had a new idea about modifying computers, designing peripherals, or generating code.

The new Net/Web hackers, like those early PC hackers, see opportunity and challenges everywhere they turn. Web hackers show their creations to the world. Anyone with a connection to the Net and a few tools can download a truckload of new nifty Net-Web applications for free. And everyone and their sisters want to be involved somehow, some way.

But perhaps the most telling aspect of the Web-o-lution, the clue that lets us know something really interesting is happening, is the enthusiasm the computing community has bestowed on Java.

A taste of Java. Java is both a general purpose programming language and a Web applet development language. Applets are Java programs (or applications) that you download from the Web with a browser such as Netscape 2.0 and run on your computer.

As a general purpose programming language, Java may be the answer to the prayers of frustrated C++ programmers everywhere. Java is based on C++ syntax, but is simpler and easier to use than C++, and it's portable across many platforms. Java developers can write an application once for any operating system and never need to port it — the Java applications you create on one platform will run without modification on other operating systems and computers. (Talk about a breath of fresh air.)

Java, like C++ and Delphi, is object-oriented and provides a run-time environment. Although Java is interpreted for faster prototyping and quicker development cycles, it's also robust and potentially quite high-performance. Many C++ programmers abandoned C++ because of the painful C++ compile-link-load-test-crash-debug-crash cycle.

Java applications are robust. The Java run-time system manages memory and garbage collection automatically. Multiple threading (which allows more than one flow of control within a single application) is built into the Java environment to allow CPU and resource intensive applications such as number crunchers and interactive graphics to process in the background while users perform other tasks.

Java applications are flexible, dynamic; they adapt to changing environments by dynamically downloading code modules throughout a network. Java can be used for stand-alone, Internet, and intranet application development. To run a Java applet for example, all one needs is a Java-compatible browser such as Hot Java or Netscape 2.0, or the Applet Viewer that comes with Sun Microsystems' JDK (Java Development Kit).

The Java run-time system has built-in protection against viruses and break-ins. In theory at least, end-users can feel secure downloading Java code from anywhere on the Internet.

Java on the rocks. But there is a rub, one I'm finding as a Java explorer, more than a little taxing. While Java might be the language of the future, Java development environments are a blast from the past. Java developers compile Java from the command line, and there's nothing to drag and drop. Although Java represents one future vision of programming, with respect to developer convenience, it's a vision of the past.

Although heavy hitters such as Borland have announced Java development tools, I've seen nothing yet that's better than Java extensions placed over a C++ environment. And speaking from the satisfied point of view of a Delphi programmer, I gotta say "no thanks." What Delphi programmer in his right mind wants to deal with a C++ environment? That's too many steps backward to get ahead.

But what to do? After all, I agree with the pack: the Web and Java are too exciting to ignore, even without a nice IDE. One solution, primitive yes, but better than Java straight at least, is Java on the Rocks, a little Java IDE I created (you guessed it) in Delphi to unprimitivise my Java development environment.

Java on the Rocks is an IDE composed of Delphi components (menu, edit boxes, option dialogs, etc.) that gives me a little room to develop and connects me a little less painfully to the DOS command line. Although Java on the Rocks is also primitive, it at least lets me explore Java without giving up all my hard-earned Delphi conveniences. Now I can enjoy the latest language revolution a bit less painfully.

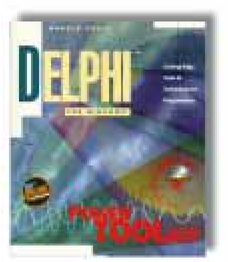
I'll describe the code and give it to you in an upcoming "The Way of Delphi." Meanwhile, I'm translating Java on the Rocks to Java and trying to decide whether Java or Delphi has more to offer explorers at the end of the 20th century.

Gary Entsminger is a Contributing Editor to *Delphi Informant*.

New Products and Solutions

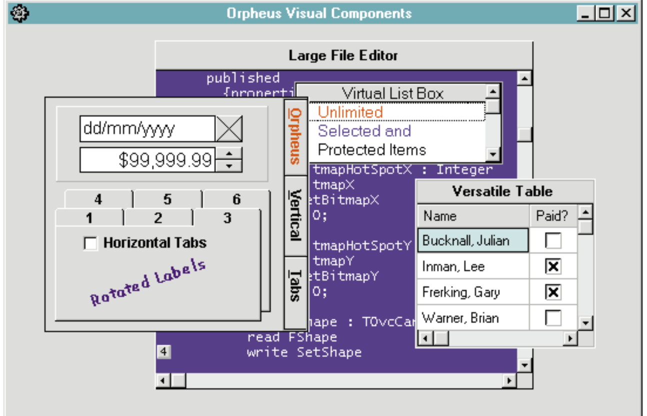
New Delphi Book

Delphi Power Toolkit for Windows
Harold Davis
Ventana



ISBN: 1-56604-292-5
Price: US$49.95
(750 pages, CD-ROM)
Phone: (800) 743-5369

TurboPower Releases Orpheus 2.0 For Delphi and Delphi 2



TurboPower Software Co. of Colorado Springs, CO has announced ***Orpheus 2.0*** for use in Delphi and Delphi 2. Orpheus is a collection of true VCL data entry components for Delphi. This new version offers 32-bit support, a data-aware table, a data-aware array editor, a formatted data-aware picture label, and alarm components.

Orpheus also includes validated data-aware fields for string, numeric, currency, and date/time variables. It has a text editor with 16MB capacity, undo/redo, and word wrap. Orpheus includes a list box with unlimited capacity, multiple selection, and colors, along with a notebook page with top and side tab options. It has a flexible table that holds edit fields, combo boxes, bitmaps, and check boxes. It also includes two-, four-, and five-way spinners.

A free trial version is available via the Internet and TurboPower's BBS, and includes the full functionality of Orpheus. However, it runs only when the Delphi development environment is operating. A Delphi help file is included.

Price: US$199, includes source, documentation, free technical support by phone, e-mail, and fax, and a 60-day money-back guarantee.

Contact: TurboPower Software Co., PO Box 49009, Colorado Springs, CO 80949-9009
Phone: (800) 333-4160 or (719) 260-9136
Fax: (719) 260-7151
BBS: (719) 260-9726
CIS Forum: GO PCVENB
Web Site: http://www.tpower.com

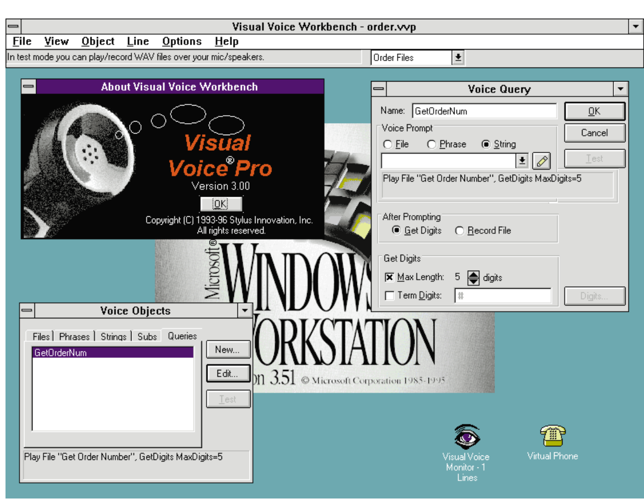
Stylus Releases Toolkit for Windows 95 and Windows NT

Stylus, of Cambridge, MA, announced the release of ***Visual Voice Pro Version 3.0*** for Windows 95 and Windows NT. Visual Voice 3.0 turns any OLE control-compatible environment, such as Delphi 2.0, C++ 4.0, PowerBuilder 5.0, Visual Basic 4.0, or Paradox 7 for Windows 95 and Windows NT into a full-featured telephony application development toolkit. Visual Voice 3.0 is a 32-bit upgrade to Stylus' telephony software toolkit. It uses the multi-tasking features of Windows NT to support 72 simultaneous phone line connections per PC.

Using Visual Voice 3.0, developers can create applications with touch-tone data access, fax-on-demand, voice mail, and Internet telephony applications. Systems developed with Visual Voice 3.0 can interact with any available data source, network, and groupware platform because of third-party built-in support available for Windows-based environments. Typical business systems built with Visual Voice 3.0 include 24-hour customer order services, benefits enrollment hotlines, fax-on-demand, and unified messaging systems.

In addition to the OLE control interface, Visual Voice 3.0 provides a 32-bit DLL and class library interface for Visual C++.

Visual Voice 3.0 also includes documentation, online help, a tutorial, and sample applications for distribution as-is, or for use in development. The sample applications include voice mail, fax-on-demand, IVR order status, and an out-dating application.

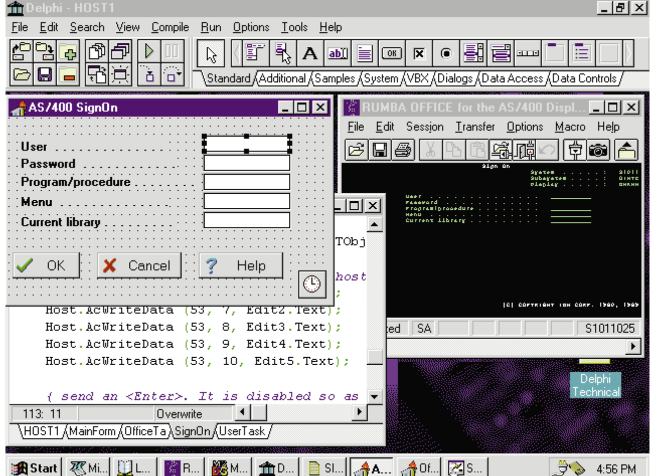


Price: Visual Voice Pro 3.0 is available for multi-line Dialogic voice boards. Pricing starts at US$495 for Windows 95 and US$795 for Windows NT.

Contact: Stylus, 201 Broadway, Cambridge, MA 02139
Phone: (617) 621-9545
Fax: (617) 621-7862
E-Mail: Internet: sales@stylus.com or info@ stylus.com
Web Site: http://www.stylus.com.

New Products and Solutions

Software Development Tools Releases AppBridge AutoCode for Delphi



Software Development Tools, Inc., of Boston MA, announced a Delphi version of its ***AppBridge AutoCode***, a tool that generates Delphi host navigation code and graphical user interface (GUI) versions of IBM mainframe or AS/400 screens.

With AppBridge AutoCode users can create host navigation code, thus creating GUI versions of partial or complete host screens, and integrate generated code into a range of development environments. This is accomplished by turning on AutoCode and navigating the host system.

AutoCode provides additional value by enabling companies to modify how users interact with host-based applications. With AutoCode, multiple host screens can be combined into a single screen. Using a variety of development tools, companies can re-engineer their applications from the client end by changing the work flow, without altering the host side.

AppBridge AutoCode is part of the SDTI WorkBench, a family of client/server migration tools that enable customers to move from host-based systems to client/server at their own pace, while leveraging their tool and application investments along the way.

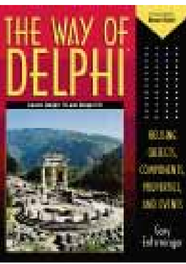
AppBridge AutoCode is available as a stand-alone product, as part of AppBridge, or as part of the suite of SDTI WorkBench migration tools. AppBridge AutoCode runs on Windows 95, Windows NT, IBM mainframe, and AS/400 environments.

Price: AppBridge AutoCode for Delphi, US$1,295 per developer license.

Contact: Software Development Tools, Inc., 60 State St., Suite 700, Boston, MA 02109
Phone: (617) 854-7454
Fax: (617) 854-7453
E-Mail: CIS: 75553,3027

New Delphi Book

The Way of Delphi
Gary Entsminger
Prentice Hall PTR



ISBN: 0-13-455271-7
Price: US$39.95 (400 pages)
Phone: (800) 947-7700

Sherlock Releases Formations 2.1 for Delphi

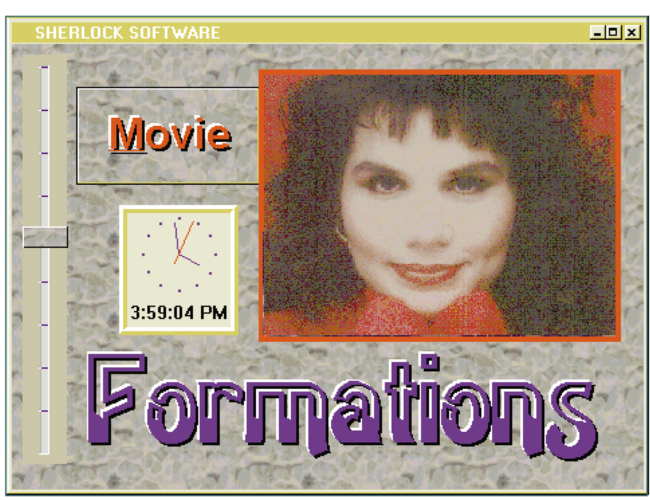
Sherlock Software of Baton Rouge, LA is shipping ***Formations 2.1***, a set of 20 graphics VCLs for Delphi.

Formations is a toolkit that offers 256-color form background alterations for Delphi. The Granite, Marble, Brick, and Stone components transform backgrounds into 30 styles of rock sculpture in 16- and 32-bit environments.

In addition to backgrounds, Formations offers 16 components that enhance form design. These include a ClearButton component that displays a 3D button with the texture of the background beneath it. Formations' four SlideBar components show handles that match the 30 background styles, producing the same three-dimensional appearance.

Formations' other components include two running clocks (analog or digital display), and a SlideShow component that allows programmers to "slide" one photo on top of another from 12 different angles. In addition, you can create full-color animation using the Movie component.

Other VCL components include: Scrolling text in a marquee fashion, a FontCombo, alternate radio buttons and check boxes, password dialog, and a multi-style 3D Label component.



Price: US$199. No royalties required; includes online and written documentation, free support via telephone or fax, two additional components free with registration, and a 30-day money-back guarantee.

Contact: Sherlock Software, 8386-D Airline Highway, Baton Rouge, LA 70815
Phone: (504) 924-2511 or (504) 924-2566
Fax: (504) 924-2572

New Products and Solutions

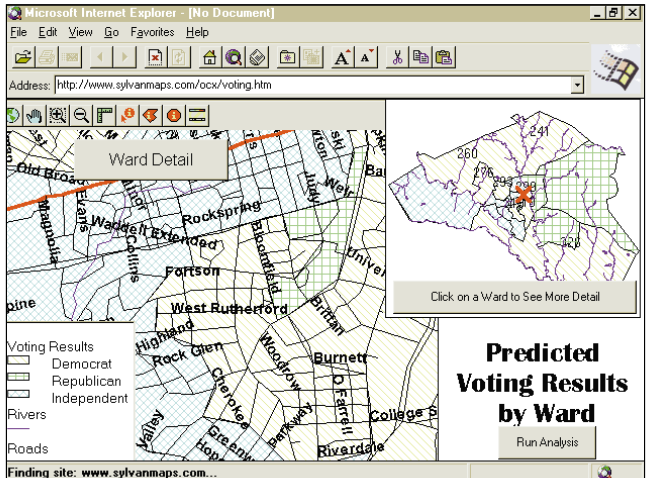
Sylvan Ascent Announces Geographic Mapping Capabilities for Web Sites

Sylvan Ascent, Inc., of Santa Fe, NM, recently released a new version of ***SylvanMaps/OCX*** for embedding mapping capabilities into interactive Web pages. This version will allow programmers to embed mapping functionality into programs written in Delphi and other programming languages that support OCXes.

Like the original Windows version, the Web version of SylvanMaps/OCX enables mapping from any database that follows the Open Database Connectivity (ODBC) standard.

A range of mapping functions are built into SylvanMaps/OCX, including display, SQL and geo-querying, distance measurement, visual spatial analysis, address matching, and a map-key control.

The installation program adds the map control and map-key control to the programming software's tools palette. Programmers can then select the control, drag and drop, and set the required parameters. The result is live mapping from any size database built into a Web site.



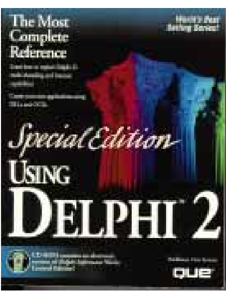
SylvanMaps/OCX is bundled on a CD-ROM and includes mapping data. In addition, translators will be provided for ArcView shape files, US Census Bureau TIGER files, USGS DLG, AutoCAD, DWG, and Sylvan Ascent's own CD\MAPs.

Price: US$495, includes free technical support via e-mail and telephone. A demonstration version is available at Sylvan's Web site.

Contact: Sylvan Ascent, Inc., PO Box 4792, Santa Fe, NM 87502
Phone: (800) 362-8971 or (505) 986-8739
Fax: (505) 986-0906
E-Mail: Internet: sylvan@sylvanmaps.com
CIS: GO SYLVANASCENT
Web Site: http://www.sylvanmaps.com/ocx

New Delphi Book

Special Edition Using Delphi 2
Jonathan Matcho, et al.
Que



ISBN: 0-7897-0591-5
Price: US$49.99
(891 pages, CD-ROM)
Phone: (800) 428-5331

Open Horizon's Connection Application Broker Supports Delphi 2

Open Horizon, Inc. of Belmont, CA has released ***Connection Application Broker***, an add-on to Open Horizon's Connection family of database access and enterprise services connectivity software. Application Broker enables Delphi developers to link 2-tier applications to shared business logic created with standard programming languages (e.g. C, C++, and COBOL), transaction processing monitors (including CICS, TUXEDO, Encina, and TOP END), and enterprise development environments (such as Forte and Dynasty). The Application Broker uses the built-in capabilities of the Delphi toolkit to provide transparent integration.

The Application Broker is a server-resident module that snaps on to the base Connection connectivity product. Developers of server-based application logic can register their business rules inside the Application Broker, making those services available to clients throughout the enterprise. Front-end GUI tools such as 16- and 32-bit Delphi Client/Server 2.0 are able to invoke the application services through built-in capabilities that come bundled with the product. Through the Application Broker's implementation of dynamic binding, Delphi applications can automatically discover new business rules on the server as they become available.

By combining Connection Database Broker module(s) with the Application Broker, Delphi developers can support 2- and 3-tier systems simultaneously in one application.

The Connection Client supports Windows 3.1, Windows NT, Solaris, AIX, and HP/UX. Future platforms will support Macintosh, OS/2, and Windows 95.

Price: The Connection Application Broker is US$149 per client workstation; there's no charge for the server.

Contact: Open Horizon, Inc., 1301 Shoreway Rd., Ste. 126, Belmont, CA 94002
Phone: (415) 598-1200
Fax: (415) 593-1669
E-Mail: Internet: info@openhorizon.com
Web Site: http://www.openhorizon.com.

L I N E

May 1996

Arthur Andersen Selects Delphi: Joins Partner Program

Dallas, TX — Arthur Andersen announced it has adopted Delphi as its development environment corporate-wide. In addition, Arthur Andersen will now train all new employees on Delphi.

Arthur Andersen has also joined Borland's Premier Value Added Partner Program. This alliance integrates Arthur Andersen's Business Consulting services with Borland's client/server technology.

Arthur Andersen's business consulting practice assists mid- and large-sized companies in improving their business processes and technologies by specializing in operational and organizational improvement, performance measurement, and middle market technology implementation services. The company provides creative solutions for its clients through audit, tax, business advisory, and specialty consulting services.

Borland International Inc. recently launched the Premier Value Added Partner Program for client/server system integrators and consultants who provide corporations and government clients with applications, consulting, and training services for Borland's client/server software products (Delphi Client/Server, InterBase, ReportSmith, and Paradox Client/Server).

Borland's Premier Value Added Partner Program's annual fee includes software, rebates, technical support, and marketing tools and programs. Each partner must also complete Borland sales and product training. For more information, contact Borland at (408) 431-5117.

Borland Announces Support for ActiveX Controls

San Francisco, CA — Borland International, along with other vendors, has announced support for Microsoft ActiveX Controls (formerly called OLE Controls). ActiveX Controls provide functionality for Web pages, including sound, video, animation, security, credit-card approval, and more.

ActiveX Controls are an open architecture supported by software vendors, corporations, and tools vendors. A developer can use any tool or language to create controls that support specific platform functionality such as multimedia, graphics, data access, and more. For details, visit Microsoft's Web site at http://www.microsoft.com.

Blue Sky Software Corp.	http://www.blue-sky.com/
Borland International Inc.	http://www.borland.com/
Brainstorm Technologies	http://www.braintech.com/
Crystal, A Seagate Software Company	http://www.seagate.com/software/crystal/
DART	http://www.dart.com/
Digital Equipment Corp.	http://www.digital.com/
FarPoint Software	http://www.fpoint.com/fpoint/
LEAD Technologies	http://www.leadtools.com/
Macromedia	http://www.macromedia.com/
MicroHelp	http://www.microhelp.com/
Nesbitt Software	http://www.nesbitt.com/
NuMega Technologies Inc.	http://www.numega.com/
Oracle	http://www.oracle.com/
Pinnacle Publishing Inc.	http://www.pinpub.com/
Powersoft Corp.	http://www.powersoft.com/
ProtoView Development Corp.	http://www.protoview.com/
Quarterdeck Corp. Inc.	http://www.quarterdeck.com/
Sax Software	http://www.saxsoft.com/
Sheridan Software Systems Inc.	http://www.shersoft.com/
Stylus Innovations	http://www.stylus.com/
SuccessWare (NetSync)	http://www.gosware.com/
Sylvan Ascent Inc.	http://www.sylvanmaps.com/ocx/
Visual Components	http://www.visualcomp.com/
Voysys	http://www.voysys.com/

Microsoft Announces ActiveX Technologies

San Francisco, CA — Microsoft Corp. has announced ActiveX Technologies, small, full-featured components for the Internet, intranets, and PCs. Using ActiveX Technologies, developers can add active content, including animation, 3D virtual reality, video, and other multimedia content to static Web pages.

ActiveX Controls form a framework for creating interactive content using software components, scripts, and existing applications. Specifically, ActiveX Technologies enable developers to build Web content using ActiveX Controls (formerly OLE Controls), active scripts, and active documents.

ActiveX Controls work with several programming languages, including Delphi, Microsoft Visual C++, Visual Basic, and others. ActiveX Controls enable developers to embed a variety of software components, such as graphics viewers, animation sequences, credit-card transaction objects, or spreadsheet applets, into hypertext markup language (HTML) pages. In addition, Java applets can co-exist with ActiveX Controls on an HTML page.

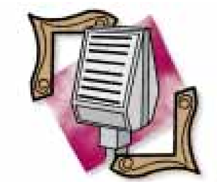
A key benefit of using ActiveX Technologies is its ability to integrate applica-

"*Microsoft Announces ActiveX Technologies*" continued on page 7

Developers Competition Heads to Chicago, San Jose, and Boston

Droege Computing Services and Digital Consulting Inc. (DCI) will be co-producing a series of Developers Competitions to be held in Chicago, San Jose, and Boston. Each competition will require contestants to develop a typical database application based on specifications detailing a real-life problem. Chicago will feature the RAD Developers Competition held during Database and Client/Server World, December 10-12, 1996. In San Jose, developers can compete in the Internet Developers Competition, scheduled during the Internet Expo, February 18-20, 1997. The Client/Server Developers Competition will take place during the Database and Client/Server World, May 20-22, 1997 in Boston. Competition winners will advance to the 1997 Developers Competition, scheduled for October 1997 in Raleigh/Durham, NC. For more details visit http://www2.interpath.net/devcomp/.

May 1996



Seventh Annual Borland Developers Conference Nears

Scotts Valley, CA — Borland International's 7th Annual Borland Developers Conference (BDC) is coming to Anaheim, CA on July 27 - 31, 1996. This year's conference will feature more than 200 sessions hosted by development experts, a free CD-ROM including proceedings, samples, and source code from all conference tracks, and your choice of a free copy of Borland's Delphi Developer, Borland C++ Developer and Suite, Paradox, Visual dBASE Professional, or InterBase.

BDC will offer core tracks focusing on: Delphi, Borland C++, Paradox, Visual dBASE, InterBase, and Java. In addition, elective tracks covering industry trends, business solutions, operating systems issues, and more are planned. These topic threads include solutions, programming, tools and techniques, methodologies, client/server, and the Internet.

Attendees can also access a computer lab containing the latest Borland products in a network environment, and discuss issues with Borland technical experts.

Early registration costs US$945 (if received by June 7, 1996), a savings of US$250 off the regular conference price of US$1,195. To register, call (800) 350-4244. If outside the US and Canada, call (805) 495-7800, ext. 239.

Third-Party Vendor Support for Delphi 2 on the Rise

Scotts Valley, CA — Borland International's Delphi 2 is gaining support from third-party vendors.

Over 1,300 components, such as OCX and VCL controls, support Delphi 2, as well as automated testing, CASE/data modeling, team development, data warehousing, online analytical processing (OLAP), and distributed computing and transactional middleware tools.

Book publishers are also entering the Delphi 2 market. Those producing Delphi 2-related books include SAMS, Addison-Wesley, Que, Osborne/McGraw-Hill, IDG Books, Sybex, M&T Books, Ventana, and MIS Press.

Over 130 training centers are supporting Borland products, and most are adding Delphi 2 courseware and instruction to their training programs.

For more information, call Borland's developer partner program, Borland Connections, at (800) 353-2211.

For technical information, call Borland's Tech Fax service at (800) 822-4269.

Category	Vendors
Testing	SQA, Segue Software, and Mercury Interactive
OOP Design	Computer Systems Associates, Logic Works, Rational Software, db Logic, and MicroGold
Version Control	Intersolv (PVCS version control system is included in Delphi Client/Server Suite 2), and MKS
Database	AppSource, Arbor Software, Attachmate, Platinum Technologies, Popkin Software, LBMS, Asymetrix, Brainstorm, Sequelink, and SDP Technologies
Interface Components	Woll2Woll Software, Shoreline Software, Apiary, TurboPower, and more
Communications	Dart, Silverware, TurboPower, and MicroHelp
Installation	InstallShield Corp. (InstallShield Express is included in Delphi Client/Server Suite 2), Eschalon Development, Great Lakes Business Solutions, 20/20 Software, and Sax Software
Conversion	Eagle Research and more
Graphics/Imaging	DFL Software, Mobius, Lead Technologies, and GigaSoft
Emerging Technologies (vector maps, virtual reality environments, telephony, etc.)	TIMC, Voysys, and GIS Systems
Internet/Web	Sax Software, HREF Tools, and Nesbitt Software
Expert/Code Generator	Apiary
Network	Apiary
Help Systems	MicroHelp and Blue Sky Software Corp.

Microsoft Announces ActiveX Technologies (cont.)

tions into Web browsers so data managed by those applications becomes accessible as Web pages. This technology, called ActiveX Documents, lets a user navigate a corporate intranet to view a department's Web page, examine spreadsheets, and query databases — all from within the Web browser and without converting it into HTML.

ActiveX Technologies includes the ActiveX Server Framework, based on the Microsoft Internet Information Server (IIS) integrated with the Windows NT Server.

The ActiveX Server Framework is composed of ActiveX Server Scripting and ActiveX Server Controls. ActiveX Server Controls can be used to build server-driven active content, allowing customers to tie into legacy systems or build applications from reusable object components.

ActiveX Server Scripts can be written using most scripting languages.

In addition, Microsoft has co-developed an ActiveX plug-in for Netscape Navigator with nCompass Labs Inc., enabling Netscape Navigator browsers to view active content.

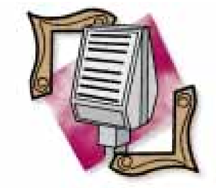
ActiveX Technologies are available in the Microsoft ActiveX Development Kit. The kit has over 600MB of Internet information.

It includes Microsoft Internet Explorer 3.0 (developer pre-release), Microsoft Internet Information Server, a sample application, Help files, and Windows NT 3.51 updates to support IIS. A CD-ROM version of the ActiveX Development Kit will be available for US$99.

For details visit http://www.microsoft.com/intdev/. Online product information is available at http://www.windows.microsoft.com/.

LINE

May 1996



ICG to Host Internet Developers' Forum for Database Professionals

Elk Grove, CA — Informant Communications Group, Inc. (ICG) will host the Internet Developers' Forum 96 for database professionals, April 28 through May 1, 1996, at the Santa Clara Convention Center in Santa Clara, CA.

The three-day conference will feature a keynote presentation by Marc Benioff, Senior Vice President, Web/Workgroup Systems Division, Oracle Corp. Benioff will discuss Oracle's Internet products, and future directions for implementing client/server applications with Oracle tools and the Internet.

Conference attendees will also benefit from over 60 educational product-training sessions, covering a wide array of Internet and database application products, add-ins, development tools, and World Wide Web database techniques.

"Our *Delphi Informant*, *Paradox Informant*, and *Oracle Informant* readership has been demanding more information about Internet Database development, and they view the Web as the next step for client/server applications," said Mitchell Koulouris, president and CEO of ICG.

The Internet Developers' Forum 96 will feature five concurrent tracks, including: CGI Tools and Languages, covering Java, JavaScript, Delphi, and Perl; Database connectivity, featuring a panel of database vendors discussing new product features for the Web; and sessions on general database development for the Internet. Other sessions will focus on server security optimization and maintenance, including firewall and security issues, and server optimization; Web site design and maintenance, covering HTML programming and case studies of hot Web sites; and business on the Web.

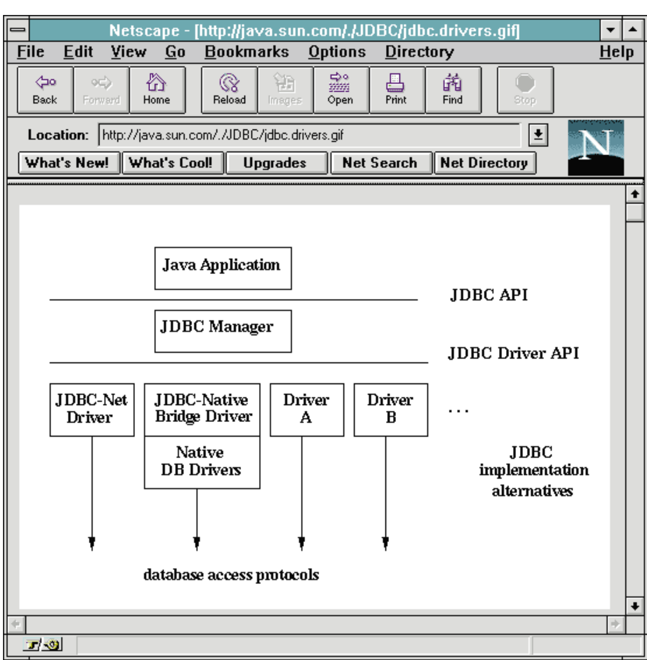
A variety of special events will also be featured, including a Sunday evening book-signing party at an opening reception. Renowned technical authors will be onsite to mingle with attendees and sign their books; each attendee will have a choice of one signed book. In addition, a vendor reception will feature table-top displays of their database services and products.

Approximately 1,500 conference-goers are expected to attend Internet Developers' Forum 96, ranging from database developers and consultants to corporate MIS representatives to VARs and network/datacomm managers. Conference attendees will represent users of Oracle, Access/NT/BackOffice, Delphi, Paradox, and other RDBMSs and tools.

For details, call Outstanding Marketing Events at (408) 462-4777 or visit the Forum's Web site at http://www.devforum.com.

JavaSoft Announces Java Database Connectivity

Palo Alto, CA — JavaSoft, an operating company of Sun Microsystems, Inc., announced the release of Java Database Connectivity (JDBC), a database access application programming interface (API) that enables developers to write Java applications that access databases. The API specification is available on the Internet at http://java.sun.com.

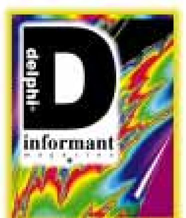


JDBC gives developers a framework to create portable solutions that access a variety of databases. JDBC supports interchangeable DBMS drivers through a driver manager that automatically loads the correct JDBC-compatible driver for the specific database.

JavaSoft will bundle the JDBC Driver Manager into future releases of Java products. In addition, JavaSoft will release a software bridge from JDBC to Microsoft's database driver interface (ODBC).

Several companies are using the JDBC API, including: Gupta Corp., Informix Software, Inc., IBM's Database 2 (DB2), Object Design, Inc., Oracle Corp., Sybase, Inc., as well as database connectivity and tools vendors Borland International Inc., Intersolv, Open Horizon, OpenLink Software, Persistence Software, RogueWave Software Inc., SAS Institute Inc., Visigenic Software, Inc., and WebLogic, Inc.

JavaSoft will support compliance test suites that developers will use to test JDBC-based products, ensuring that each application has the highest level of Java power and compatibility. Initial test suites are expected to be available in June.



Delphi 2.0 / OLE / Object Pascal / Netscape

By *Joseph C. Fung*

Using OLE Automation to Access the Internet

Netscape Navigator as an Automation Server

Your Web browser makes it simple for you to surf the World Wide Web and gain access to a wealth of information. The current crop of Web browsers — or "Internet clients" as they are becoming known — can retrieve hypertext Web documents, download files, handle e-mail, and let you read newsgroups.

To use your Web browser to visit a particular Web address or Internet site, simply enter the site's Web address — the URL (Uniform Resource Locator) — and the browser connects and retrieves information for you. A forms-capable Web browser, such as Netscape Navigator or Microsoft Internet Explorer, even lets you submit information to a Web server by filling in fields on a form. A CGI (Common Gateway Interface) program on the server can process this information, perform a desired action, then return a response to you. The Web browser takes care of details such as network access, file retrieval, and security, so you can concentrate on exploring the Internet.

Delphi lets you build Internet and intranet programs that range from client/server applications to CGI programs, but you may need to obtain third-party Internet components or tools. [For more information on forthcoming third-party tools, see the sidebar "Delphi and the Internet" on page 14.]

Fortunately, you can also use Delphi 2's OLE automation capabilities to leverage Netscape Navigator's built-in feature set and experiment with creating your own simple Internet-enabled program.

What Is OLE Automation?

OLE automation defines a standard way in which an application can programmatically control or be controlled by another program. This ability lets you tap the functionality of many applications to create a single, integrated solution. Before OLE automation, the few ways to do this included using macro or key stroke recorders, or using the proprietary macro language of another application.

OLE automation changes this by letting an application known as an *automation server* expose one or more objects that surface functionality you can access. The program that works with these objects is called an *automation controller*. A Delphi program can have automation server and automation controller capabilities. Netscape Navigator is an automation server so you can use a Delphi program to control it.

Using OLE Automation to Control another Application

To access an automation object, you use the Object Pascal *CreateOLEObject* function to instantiate the object. *CreateOLEObject* accepts the *ProgID* of an automation object and returns a reference to it as a *Variant* (a special data type that can store different types of data and is intended primarily for OLE programming). *ProgID* is the programmatic string identifier, or name, of the automation object. Each automation server registers the *ProgID* of all its automation objects into the system registry when you

install the server. The *ProgID* lets you identify and instantiate an automation object without knowing the file name or location of the automation server that creates it.

Once you've instantiated an automation object in your program, you use standard Object Pascal syntax to work with it. The automation object can be treated as a normal object with the following exceptions. First, all the properties and method arguments must be passed as Variants. This means that any value that you use must be a Variant or can be implicitly or explicitly converted into a Variant. Second, calls to the automation object's methods are *late-bound*, so checking the method's validity, type, or number of arguments, is not done until run time. Therefore, if you call a method that is not part of the automation object, an exception is raised in your program.

The code fragment in Figure 1 illustrates how to use *CreateOLEObject* to instantiate an OLE object for Excel 7 and use it to add a new workbook.

```
var
  // Declare variable to reference object.
  ExcelObject : Variant;
begin
  // Create instance with Excel.Application ProgID.
  ExcelObject := CreateOLEObject('Excel.Application');
  // Add a new workbook.
  ExcelObject.Workbooks.Add;
end;
```

Figure 1: Sample code to control an Excel automation object.

Using Netscape Navigator as an Automation Server

One of the automation objects that Netscape Navigator 2.0 exposes has a *ProgID* of *Netscape.Network.1*. This automation object lets you use Netscape Navigator's Internet access and file retrieval capabilities in your application.

The automation object defines many methods and properties. Here are some of the methods:

- ***Open***: Connects to and begins retrieving from a specified URL. If the method fails (e.g. Navigator is busy), the method returns *False*. *Open* takes five arguments: *pURL*, *iMethod*, *pPostData*, *lPostDataSize*, and *pPostHeaders*. The first argument, *pURL*, is a string that specifies the URL of the Web site. The second argument, *iMethod*, indicates the type of operation to perform. When reading from a Web site, you pass a 0 to indicate that you want to read. The remaining arguments pertain to posting data to the Web site. Since you are only reading, you can also pass 0 as values for these arguments.
- ***Close***: Disconnects any current connection and resets the automation object. You should call *Close* when you are finished with the automation object.
- ***GetStatus***: Returns an integer indicating the status of the load. It returns a non-zero value in the case of an error.
- ***Read***: Retrieves data from the Web site and stores it into a buffer that you specify. *Read* returns -1 if there is no more data to read, 0 if there is currently no data to read, or a positive number indicating the number of bytes actually read. *Read* takes two arguments, *pBuffer* and *iAmount*.

  The first argument, *pBuffer*, is an OLE *BSTR* type and refers to the buffer that receives the data. The second argument, *iAmount*, specifies the size of the buffer. To keep things simple, you can use a string as the type for the buffer instead of a *BSTR*; Delphi automatically converts the string into a *BSTR* for you. When Netscape Navigator stores any returned data into your buffer, it does not update the internal byte count of the string or *BSTR*, for that matter. Because of this, you must correct the byte count manually after calling *Read*.

This brief description only describes the methods used in the examples in this article. If you are interested in learning more about the objects surfaced by Netscape Navigator, you can find the documentation for the OLE automation interface at http://home.mcom.com/newsref/std/oleapi.html.

Using Netscape Navigator to Connect to a Web Site

The following two examples show how you can use Netscape Navigator to connect to the Internet and download useful information. Each sample application uses the Netscape Navigator automation object to connect to a particular Web site. Specifically, the application passes a URL to the automation object and requests some data to retrieve. During this interaction, Netscape Navigator is not visible so all you'll see is your main form.

To run these examples, you'll need to establish an Internet connection and have a copy of Netscape Navigator 2.0. The Internet connection should be active before you try any of the samples. If you're using a dial-up connection, you may want to use Windows 95 Dial-Up Networking or a third-party dialer to connect to your Internet Service Provider (ISP). If you don't have a copy of Netscape Navigator, you can obtain one by purchasing the Netscape Internet Starter Kit at a store, or by downloading an evaluation copy from the Netscape home page at their Web address, http://www.netscape.com.

Quote.Com Background

The first example uses the Quote.Com Web site (www.quote.com) to download stock quotes. Quote.Com is a company that provides financial market data, such as stock quotes, financial commentary, and business news to Internet users. Normally, you must have a paid subscription to the service to use it. New users who don't want to pay can still try the service on a limited basis, but they must register online to obtain a user account and password. Users who do not register may still try the service, but are limited to using the stock ticker "MSFT" (Microsoft) in the demonstration area (see Figure 2). This example assumes that you are not registered, so it uses MSFT as the default ticker. If you register with Quote.Com, you can use other stock tickers with the sample application.

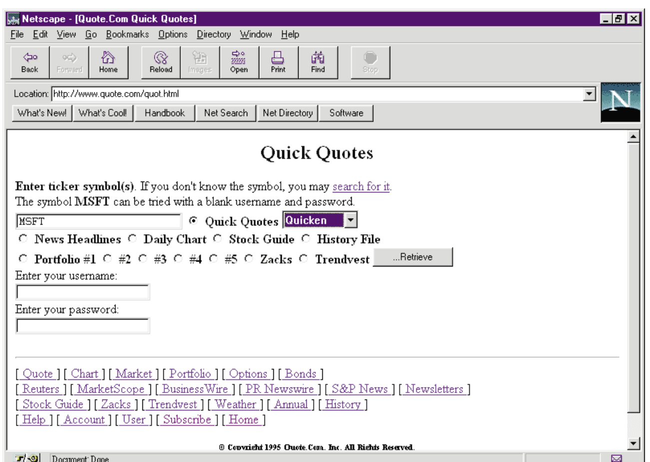


Figure 2: A Web page for quick quotes.

This screen allows you to enter your user name and password, a stock ticker, and the type of information you want. When you press the **Retrieve** button, a URL containing these parameters is sent to the Web server. A CGI program at the Web server parses this URL and returns the requested information as a hypertext document.

The Stock Quote Example

The Stock Quote example program shows you how you can use a Delphi program to connect to the Web server and download a stock quote for a specific ticker. The program has a single form that lets you enter a stock ticker and press a button to obtain a current quote for the stock. The screen in Figure 3 shows the appearance of the Delphi program after you request a stock quote for MSFT.

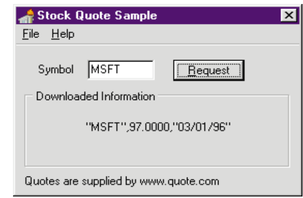


Figure 3: The form for our stock quote sample.

The main unit, STOCKFRM.PAS, contains a definition for the form class, *TStockForm* (see Figure 4), and also declares a Variant, *NetscapeObject*, to reference the Netscape automation object after you've created it.

TStockForm has two private variables, *LoginName* and *Password*. By default, the form's *OnCreate* handler assigns a blank string to both of these variables. Therefore, you must use MSFT as the stock ticker. *LoginName* and *Password* are sent to the server when you request a quote. If you register with Quote.Com, be sure to change the code in *OnFormCreate* so that your user name and password are used (see Figure 5).

On the form is a *TEdit* component named *StockSymbol* that lets you enter a stock ticker. After entering a valid stock ticker, you can press the **Request** button. This executes the *StockButtonClick* method that does the work of retrieving the stock quote and displaying it on the *QuoteLabel* label component.

```
type
  TStockForm = class(TForm)
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Help1: TMenuItem;
    About1: TMenuItem;
    Exit1: TMenuItem;
    Bevel1: TBevel;
    StockButton: TButton;
    Label1: TLabel;
    StockSymbol: TEdit;
    GroupBox1: TGroupBox;
    QuoteLabel: TLabel;
    Label2: TLabel;
    procedure FormClose(Sender: TObject;
                        var Action: TCloseAction);
    procedure Exit1Click(Sender: TObject);
    procedure About1Click(Sender: TObject);
    procedure StockButtonClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    LoginName: string;   // Registered user login name.
    Password:  string;   // Registered user password.
  public
    { Public declarations }
  end;

var
  StockForm      : TStockForm;
  NetscapeObject : Variant;
```

```
procedure TStockForm.FormCreate(Sender: TObject);
begin
  // This sample application only works if you use MSFT as
  // the stock symbol. To obtain stock quotes for other
  // companies, you must register with Quote.Com and obtain
  // a login name & password. When you do, enter them here.
  LoginName := '';
  Password  := '';
end;
```

Figure 4 (Top): The *TStockForm* class and variable declarations.
Figure 5 (Bottom): The *FormCreate* event handler.

The *StockButtonClick* method (see Figure 6) starts the Netscape Navigator automation server by calling *CreateOLEObject* with the *ProgID* of the automation object, then assigns it to the Variant, *NetscapeObject*. Next, the automation object's *Open* method is called with the correct address for the quote server. The text for the URL is composed so that all the necessary values are passed to the CGI program at the Web site to retrieve the stock quote. This includes the site address, stock ticker, quote type, and the user name/password combination. Since only reading is being performed, zeros can be passed as the remaining arguments to the *Open* call.

If the site is successfully contacted, the server's *Read* method is used to download the stock information into a buffer that is then displayed on the form. Before the information is displayed, the buffer is checked to see if the server returned an error message instead of the stock information. Normally, the stock quote is returned as a string containing comma-delimited values. In an error situation, this particular server returns a hypertext document indicating the error. To deter-

```
procedure TStockForm.StockButtonClick(Sender: TObject);
var
  Buffer     : string;
  BytesRead  : Integer;
begin
  Screen.Cursor := crHourGlass;
  // Launch Navigator if we haven't done so already.
  if VarIsEmpty(NetscapeObject) then
    begin
      QuoteLabel.Caption :=
        'Please wait. Loading Netscape for the first time';
      QuoteLabel.Refresh;
      try
        NetscapeObject :=
          CreateOLEObject('Netscape.Network.1');
      except
        ShowMessage('Could not start Netscape 2.0.
                     Please exit.');
        Screen.Cursor := crDefault;
        Exit;
      end;
    end;

  // Begin the process of connecting and downloading.
  try
    QuoteLabel.Caption := 'Downloading data';
    // Attempt to open the URL.
    if NetscapeObject.Open('http://www.quote.com/cgi-bin' +
                           '/quote-form?symbols = ' +
                           StockSymbol.Text + '&login=' +
                           LoginName + '&passwd=' +
                           Password + '&quotetype=Quicken',
                           0,0,0,0) then
      begin
        // Check status first.
        if NetscapeObject.GetStatus = 0 then
          begin
            // Set size of Buffer.
            SetLength(Buffer,2048);
            // Loop until something is read.
            while True do
              begin
                // Read just a buffer full.
                BytesRead:=NetscapeObject.Read(Buffer,2048);
                // Nothing read. Server busy?
                if BytesRead = 0 then
                  continue
                else
                  break;
              end;
            // Correct the Buffer size.
            SetLength(Buffer,BytesRead);
            if (Copy(Buffer,1,1) = '<') or
               (BytesRead < 1)          then
              // Error at site or time-out.
              QuoteLabel.Caption :=
                'No information downloaded.'
            else
              QuoteLabel.Caption := Buffer;
            end;
          end
        else
          begin
            // Any one of several problems may have
            // occurred here. The DNS server may be
            // having trouble locating the URL or
            // the connection is bad or the URL has
            // moved or is no longer there, etc.
            QuoteLabel.Caption := 'Can not connect';
            Exit;
          end;
   finally
    Screen.Cursor := crDefault;
  end;
end;
```

Figure 6: The *StockButtonClick* handler.

```
procedure TStockForm.FormClose(Sender: TObject;
                               var Action: TCloseAction);
begin
  // Shut down Netscape if we started it.
  if not VarIsEmpty(NetscapeObject) then
    begin
      // Call Netscape's Close method.
      NetscapeObject.Close;
      // Free the variant.
      NetscapeObject := Unassigned;
    end;
end;
```

Figure 7: The *FormClose* handler.

Figure 8: The Web page for NBC's Intellicast weather site.

mine if the latter occurred, the first character of the buffer is compared to `'<'` to see if the buffer contains the start of an HTML tag.

After you are finished with the program, you can close it. When the form closes, its *FormClose* handler (see Figure 7) uses the *VarIsEmpty* function to determine if the automation server needs to be cleared. If so, the server is closed, then the *NetscapeObject* variable is assigned a value of *UnAssigned*.

Intellicast Background

The second example uses the NBC News Intellicast Web site (www.intellicast.com) to display a GIF graphics image that depicts the 4-day weather forecast for a city (see Figure 8). The NBC News Intellicast Web site provides weather and skiing information for major locations around the world. This service is free to online users and no registration is required.

When you request a forecast for a particular city, the Web site returns a hypertext document containing summary information and the image of the weather forecast. The URL of the weather image for each city takes the form:

```
www.intellicast.com/weather/xxx/4-day.gif
```

where *xxx* is a three-letter city abbreviation.

This URL is static, but the image is continually updated by NBC News so that it displays the current weather forecast.

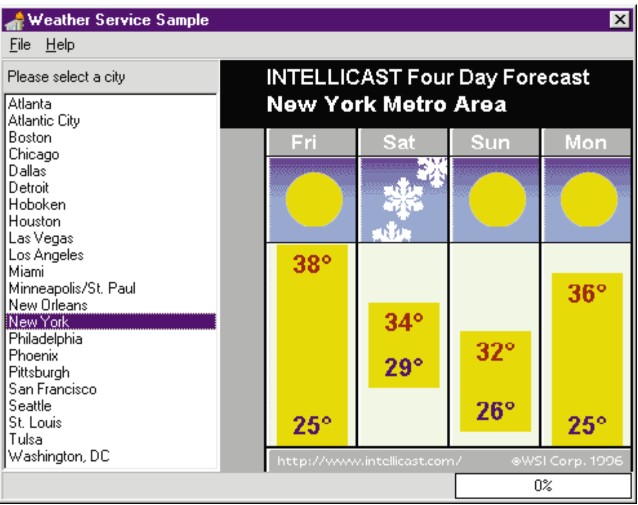


Figure 9: The form for the weather service sample.

The Weather Forecast Example

The Weather Forecast example program has a main form, *WeatherForm*, with a *TListbox* component of major cities and a *TImage* component. When you double-click on a city, the application connects to the Intellicast site and downloads its weather image. This image is then displayed in the *TImage*. Figure 9 shows a weather forecast for the New York metropolitan area.

The main unit, WEBFORM.PAS, defines the form class *TWeatherForm* and the Variant *NetscapeObject*, to reference the automation server (see Figure 10).

The form contains the list box, *CityListBox*, which stores the name of each city and an associated *TMapURL* object for that city. Each *TMapURL* object stores the three-letter abbreviation for a city and has a method, *DisplayWeather*, that is executed when you double-click on a city in the list box. The form's *OnCreate* handler, *FormCreate*, populates the list box with the list of cities and *TMapURL* objects (see Figure 11).

The MAPURL.PAS unit defines the *TMapURL* class. *TMapURL* has a *Create* constructor that accepts a three-letter city abbreviation as an argument. This abbreviation is stored in a private string, *FCityAbbrev*, which is used by *DisplayWeather* to build the correct URL for the city's weather forecast. *DisplayWeather* does all the work of connecting to the Web site, downloading the weather image, then displaying it on the form:

```
type TMapURL = class(TObject)
  private
    FCityAbbrev : string;
  public
    constructor Create(CityAbbrev: string);
    procedure DisplayWeather;
end;
```

DisplayWeather defines a *TMemoryStream* (*GIFStream*), a string (*Buffer*), and an integer (*BytesRead*). These variables are used to download the GIF image and convert it into a bitmap. The weather image is a GIF image that cannot be used in a *TImage*, so it is translated in memory into a bitmap (.BMP) before displaying it.

```
type
  TWeatherForm = class(TForm)
    StatusPanel: TPanel;
    Image1: TImage;
    Gauge1: TGauge;
    Label1: TLabel;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Help1: TMenuItem;
    About1: TMenuItem;
    Exit1: TMenuItem;
    Bevel1: TBevel;
    Panel1: TPanel;
    CityListBox: TListBox;
    Panel2: TPanel;
    procedure FormCreate(Sender: TObject);
    procedure CityListBoxDblClick(Sender: TObject);
    procedure FormClose(Sender: TObject;
                        var Action: TCloseAction);
    procedure Exit1Click(Sender: TObject);
    procedure About1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  WeatherForm    : TWeatherForm;
  NetscapeObject : Variant;
```

Figure 10: The **type** declaration for the sample form.

```
// Initialize city list.
procedure TWeatherForm.FormCreate(Sender: TObject);
begin
  with CityListBox.Items do begin
    AddObject('Atlanta', TMapURL.Create('atl'));
    AddObject('Atlantic City', TMapURL.Create('acy'));
    AddObject('Boston', TMapURL.Create('bos'));
    AddObject('Chicago', TMapURL.Create('ord'));
    AddObject('Dallas', TMapURL.Create('dfw'));
    AddObject('Detroit', TMapURL.Create('dtw'));
    AddObject('Hoboken', TMapURL.Create('ewr'));
    AddObject('Houston', TMapURL.Create('iah'));
    AddObject('Las Vegas', TMapURL.Create('las'));
    AddObject('Los Angeles', TMapURL.Create('lax'));
    AddObject('Miami', TMapURL.Create('mia'));
    AddObject('Minneapolis/St. Paul',
              TMapURL.Create('msp'));
    AddObject('New Orleans', TMapURL.Create('msy'));
    AddObject('New York', TMapURL.Create('lga'));
    AddObject('Philadelphia', TMapURL.Create('phl'));
    AddObject('Phoenix', TMapURL.Create('phx'));
    AddObject('Pittsburgh', TMapURL.Create('pit'));
    AddObject('San Francisco', TMapURL.Create('sfo'));
    AddObject('Seattle', TMapURL.Create('sea'));
    AddObject('St. Louis', TMapURL.Create('stl'));
    AddObject('Tulsa', TMapURL.Create('tul'));
    AddObject('Washington, DC',TMapURL.Create('dca'));
  end;
end;
```

Figure 11: The form's *OnCreate* handler.

When *DisplayWeather* is executed, it creates the Netscape Navigator automation object by passing a *ProgID* of *Netscape.Network.1* to *CreateOLEObject*.

Delphi and the Internet

Delphi is a very powerful development tool that can be used today and tomorrow to create useful Internet applications. Examples of the types of Internet programs you can create include: client/server database programs, Netscape plug-ins, Web browsers and servers, and CGI programs that run on Web servers. Delphi also lets you take advantage of emerging technologies.

Borland is currently working on a Delphi-type development environment for Java, Sun Microsystems' hot Internet programming language. This evolving language has many strengths, and creates applets that are executed by a cross-platform interpreter. If you want to create a Web application that can be accessed publicly by any Windows, Macintosh, or UNIX user, Java is compelling. Future versions of Delphi will be closely tied to Borland's overall Internet strategy. Microsoft is also working on a set of Internet development technologies based on Win32 and OLE for the Windows platform.

The Sweeper API enables developers to Internet-enable Win32 programs using tools such as Delphi, Visual Basic, and C++. The Internet Server API (ISAPI) and CryptoAPI let you extend Microsoft's Internet Information Server and add cryptography to your program. OLE Document Objects (*DocObjects*) will let you host OLE controls in Web pages, much as you do with Java applets. Because OLE controls are automation objects, they have automation capabilities.

Delphi and Delphi users are in a unique and positive situation. Delphi can be used *immediately* to build many types of Internet applications. Not only can you use today's technology, you can take advantage of future technologies as Microsoft evolves and extends the Internet APIs for the Windows platform.

— *Joseph Fung*

Next, the server's *Open* method is called with the URL of the city's weather image. The URL for the image can be easily constructed because its format is known. Zeroes are passed as the remaining arguments to *Open*.

Once the connection is established, the image retrieval process begins. The image is downloaded as 2K chunks into *Buffer* by executing the server's *Read* method. After each chunk is received, it is transferred into *GIFStream* using the *TMemoryStream* method *WriteBuffer*. These steps are repeated until the image is fully retrieved and *Read* returns a value of -1.

After the image is downloaded, it's converted into a bitmap format using a freeware GIF-to-BMP conversion routine, *GIFConvert*. Finally, the image is displayed on a *TImage* on the form.

.GIF to .BMP Conversion Routine

Listing One on page 15 adapts the GIF-to-BMP graphics conversion routine (GIF2BMP.PAS) developed originally by Sean Wenzel and updated for Delphi by Richard Dominelli. The comments in the GIF2BMP.PAS source code indicate that the code may be freely used provided the original authors are credited. The routines contained in GIF2BMP have been minimally changed so that they can be used for this article. The source code is not listed here, but is available (see end of article for details). The original source code can be found on CompuServe and several Delphi-related Web sites.

Conclusion

Building custom applications for the Internet and intranets requires sophisticated tools and experience, but this doesn't need to be a road block. OLE automation lets you use an existing Web browser and experiment with adding browser-type Internet capabilities to your Delphi application in a low-cost way. To explore this avenue further, you should visit the Netscape home page to learn more about the additional objects that Navigator exposes. ∆

Portions of this article were adapted from material for *Delphi In Depth* [McGraw-Hill, 1996] by Cary Jensen, Loy Anderson, Joseph C. Fung, Ann Lynnworth, Mark Ostroff, Martin Rudy, and Robert Vivrette.

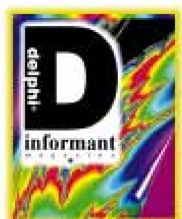
The demonstration projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\MAY\DI9605JF.

Joseph C. Fung is Director of Technology and Tools at PCSI, a leading client/server and Internet/Intranet consulting and development firm. He is the co-author of the forthcoming *Delphi In Depth* [McGraw-Hill, 1996], and *Paradox for Windows Essential Power Programming* [Prima, 1995], and is the architect of AppExpert and ScriptView. Recently, Mr. Fung chaired an Advisory Board for the Borland Developers Conference. He can be reached at (201) 816-8002.

Begin Listing One — An Adaptation of GIF2BMP.PAS

```
constructor TMapURL.Create(CityAbbrev: string);
begin
  inherited Create;
  // Store the city abbreviation.
  FCityAbbrev := CityAbbrev;
end;

procedure TMapURL.DisplayWeather;
const
  BaseURL : string = 'http://www.intellicast.com/weather/';
var
  GIFStream  : TMemoryStream;
  Buffer     : string;
  BytesRead  : Integer;
begin
  Screen.Cursor := crHourGlass;
  // Launch Netscape Navigator.
  if VarIsEmpty(NetscapeObject) then
    begin
      WeatherForm.StatusPanel.Caption :=
        'Please wait. Loading Netscape for the first time';
      WeatherForm.StatusPanel.Refresh;
      try
        NetscapeObject :=
          CreateOLEObject('Netscape.Network.1');
      except
        ShowMessage(
          'Could not start Netscape 2.0. Please exit.');
        Screen.Cursor := crDefault;
        Exit;
      end;
      WeatherForm.StatusPanel.Caption := '';
    end;

    try
      WeatherForm.StatusPanel.Caption :=
        'Downloading weather information';
      WeatherForm.StatusPanel.Refresh;
      // Attempt to open the URL for the weather
      // forecast GIF.
      if NetscapeObject.Open(
        BaseURL + FCityAbbrev + '/4day.gif',0,0,0,0) then
        begin
          // Create temporary memory stream to store GIF.
          GIFStream := TMemoryStream.Create;
          try
            while NetscapeObject.GetStatus = 0 do begin
              // Set size of Buffer.
              SetLength(Buffer,2048);
              // Read a chunk.
              BytesRead := NetscapeObject.Read(Buffer,2048);
              if BytesRead = -1 then
                // Nothing left to read.
                Break
              else
                begin
                  // Correct the Buffer size. Read does
                  // not update it correctly.
                  SetLength(Buffer,BytesRead);
                  // Store GIF segment into stream.
                  GIFStream.WriteBuffer(
                    Pointer(Buffer)^,Length(Buffer));
                  // Loop up to read more chunks ...
                  Continue;
                end;
            end;

            // Convert the GIFStream to a BMP so that we
            // can display it in a TImage.
            with TGIF.Create do
              try
                SetIndicators(WeatherForm.Gauge1,
                              WeatherForm.StatusPanel);
                Stream := GIFStream;
                // Reset position of GIF stream to origin.
                Stream.Seek(0,0);
                GIFConvert;
              finally
                Free;
              end;
          finally
            GIFStream.Free;
          end;
        end
      else
        begin
          // Any one of several problems may have occurred
          // here. The DNS server may be having trouble
          // locating the URL or the connection is bad or
          // The URL has moved or is no longer there, etc.
          WeatherForm.StatusPanel.Caption := '';
          ShowMessage('Sorry! URL for the Intellicast ' +
                      'weather site cannot be located.');
          Exit;
        end;
    finally
      Screen.Cursor := crDefault;
      WeatherForm.StatusPanel.Caption := '';
      WeatherForm.Gauge1.Progress := 0;
    end;
  end;

end.
```

End Listing One



ON THE COVER

Delphi 1 / Object Pascal / HTML

By Keith Wood

An HTML Generator

Use a Delphi Component to Build Your Web Site

Building your own World Wide Web (WWW) site can be a daunting task. You need to understand HTTP, MIME, HTML, and possibly even CGI [see the sidebar "Internet Definitions" on page 23]. However, by using the encapsulation available in Delphi's components, you can hide much of this complexity from the novice user.

This article describes a component that allows us to generate HTML with a Delphi program, without an in-depth understanding of HTML. The component, *THTMLWriter*, allows values from other sources — typically a database — to be included in the documents produced, creating truly up-to-date pages for display.

HyperText Markup Language

HTML is a language that describes how a page will be presented. It's interpreted by various browsers that display the page to the user. HTML is not WYSIWYG since the final display is determined by the browser. The advantage of this is that browsers can be implemented on many different platforms with various capabilities.

```
<html>
<head>
<title>Keith Wood's Home Page</title>
</head>
<body>
<h1>Keith Wood's Home Page</h1>
<hr>
<p>Welcome to your own home page.</p>
<p>Text can be easily <strong>highlighted</strong> or
<i>italicised</i>. Special characters can be inserted :
&#169;,&AElig;, &eacute;. Lists can be added :</p>
<ul>
<li>First list item.
<li>Second list item.
<li>Third list item.
</ul>
<p>Include an image if you want :<img src="athena.jpg"
border=0 alt="Demonstration image" align=middle></p>
<p>Links can also be added. This one
<a href="sourceb.htm">shows the code that produced
this</a>.</p>
</body>
</html>
```

Figure 1: A sample HTML document. The use of tags makes the code easily understandable. The result of this code is shown in Figure 2.

Let's discuss the HTML elements that are normally implemented on a home page.

Tags. An HTML document consists of a straight text file with markup instructions encoded within *tags*. These tags are delimited by angle brackets (< >) allowing them to be easily identified. Each tag has a name indicating its purpose, and may have one or more attributes to control its function. Many tags contain text or other tags, and some remain empty (i.e. they stand alone).

To indicate the end of a particular tag, its name is enclosed in angle brackets as shown above, but with a slash preceding the name. An example of a container tag is the paragraph marker <p>, which matches its closing version, </p>.

An example of an empty tag is the image directive:

```
<img ...>
```

Figure 1 is a sample HTML document and Figure 2 displays its appearance in a browser.

Head and Body. An HTML document is divided into two parts: the *head* and *body*. Among others, the head contains tags that provide information about the

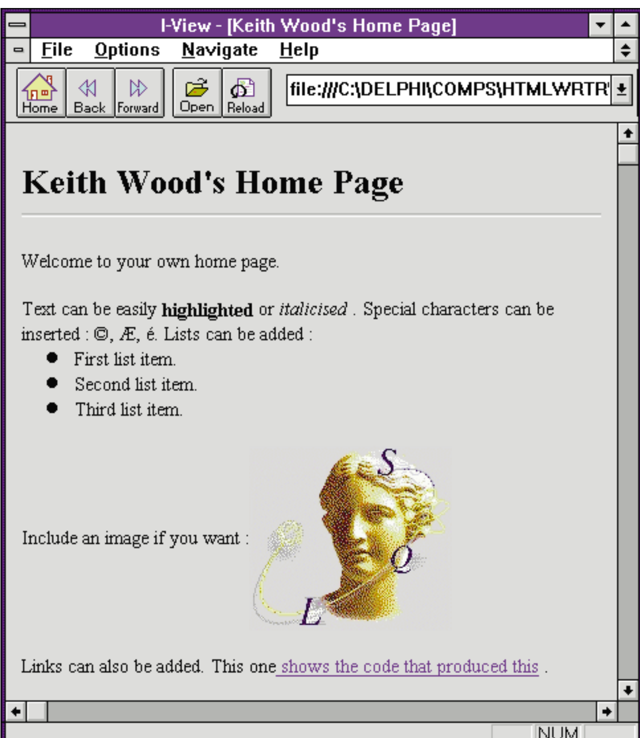


Figure 2: The visual representation of the HTML code shown in Figure 1. The I-View browser is being used to view the page.

document as a whole, including the document's title and the base URL for this and related documents. The title is the only mandatory tag in the header block. An example of a document header is:

```
<head>
<title>Keith Wood's Home Page</title>
</head>
```

The body of the document is the part that is displayed to the user. It can contain text, images, tables, lists, and links to other documents. HTML allows up to six levels of headings, `<h1>` to `<h6>`, with each heading displayed in a slightly different way. HTML also allows text to be grouped into paragraphs. Sections of text can be formatted in various ways with physical tags such as `<b>` (for bold) and `<i>` (for italics), or with logical tags, such as `<strong>` and `<code>`. How the latter are displayed is determined by the browser. For example:

```
<h1>Keith Wood's Home Page</h1>
<p>Welcome to your own home page.</p>
<p>Text can be easily <strong>highlighted</strong> or
<i>italicised</i>.</p>
```

Images. Images can be included in the document, with the standard formats being X-bitmap, .GIF, and JPEG files. For example:

```
<img src="athena.jpg" border=0 alt="Demonstration image"
align=middle>
```

These can be positioned relative to surrounding text and resized if required. A text alternative to images is also available for those browsers that cannot handle images, or have been disabled by the user to save load time.

Lists. Several styles of lists can be displayed in HTML documents, with the main ones being unordered (usually displayed with bullets) `<ul>`, ordered `<ol>`, and glossary `<dl>`, consisting of a series of terms and their definitions. Some lists can be embedded to create multi-layered structures. Here's an example of an unordered list:

```
<ul>
<li>First list item.
<li>Second list item.
<li>Third list item.
</ul>
```

Links. The most important part of an HTML document is its *links*. Using the tag `<a>`, links provide references to other parts of the same document or related documents, which can be anywhere in the world.

A section of text or an image is marked as the *anchor* (or hotspot) for the link, and is presented to the user in some highlighted format. Clicking on an anchor causes the browser to request the specified document (identified by its URL) from its server and display the results:

```
<a href="sourceb.htm">shows the code that
produced this</a>
```

The Table Construct. Some columnar formatting can be provided by the table construct, `<table>`. Tables are comprised of rows, `<tr>`, which are themselves comprised of headings, `<th>`, or cells, `<td>`. These headings and cells contain text and other HTML tags. The borders can be sized or made invisible, and even colored in some browsers. Items within headings and cells can be aligned both horizontally and vertically. For example:

```
<table>
<tr><th>Column 1</th><th>Column 2</th></tr>
<tr><td>Cell 1,1</td><td>Cell 2,1</td></tr>
</table>
```

Forms. Forms provide a way for users to interact with the document, supplying entered information to the server. Normally a CGI program receives the parameters and processes them accordingly. This can be used to search Web indexes, respond to surveys, or just about anything else you can imagine. The processing of CGI programs is beyond the scope of this article, but the following code demonstrates how simple form input capability can be implemented:

```
<form method=get action="search">
<p>Enter word to search for: <input type=text name="value">
<input type=submit name="action" value="Search"></p>
</form>
```

Spacing, etc. Additional spacing within an HTML document is ignored, unless it occurs within a preformatted

section. This means that consecutive spaces, tabs, and carriage returns are all treated as if they were a single space. The document's formatting is done by the browser, based upon the embedded tags. Tags that are not understood by a browser are ignored. This allows new features to be added incrementally without adversely affecting existing browsers and documents.

The basic definition of HTML is referred to as version 2.0. This includes all the basic tags, but excludes tables. Further extensions to this standard have been proposed and are being finalized into HTML 3.0. These include tables, client-side image maps, and common attributes for many existing tags.

In the meantime, various browser providers are including non-standard extensions to HTML that work within their products. The leading organization in this category is Netscape. Their extensions include specifying colors for the page background, text and links, centering text and images, customizing lists and horizontal rules, etc. Netscape's other principal extensions are tables and background images, which have now been included in the proposed HTML 3.0 standard. Microsoft's new Internet Explorer has added its own extensions, primarily in the area of multimedia. (For additional resources see Figure 12.)

An HTML Writer Component

The Delphi component that helps us generate HTML is *THTMLWriter*. Since it's non-visual, it's derived directly from *TComponent*. Therefore, *THTMLWriter* can be placed on the Component Palette and easily incorporated into our projects.

THTMLWriter consists of only four properties and many methods to produce the various tags available in HTML. The properties are:

- *Filename*: The name of the file to which the generated HTML is directed. *Filename* defaults to `HTMLWRTR.HTM` (in Windows, HTML files have the .HTM extension). When changed, the previous file is closed and the new one is opened, or created, ready for output.
- *Errors*: We'll discuss the *Errors* property in "Errors and Warning."
- *IncludeMIMEType*: Set to *True* if the MIME header is automatically included in the output. This would normally be done if the program is functioning in a CGI environment.
- *Version*: A read-only property that shows *THTMLWriter*'s current version.

A Two-Tiered Approach

To maximize the usefulness of *THTMLWriter*, we implement a two-tiered approach for generating HTML. First, there is a set of functions that return string values. These correspond to the different HTML tags and allow the HTML formatted text to be returned for further manipulation in the program. Then a set of procedures that correspond to the functions writes the formatted text to the output file.

At each point, appropriate error processing is performed. In the functions, this relates to mandatory attributes and warnings about various HTML extensions. For the procedures, the error processing involves relationships between tags. Note that *THTMLWriter* is not intended to support every possible combination of tags and attributes, just the most common. Of course, you can always extend it.

Since a container tag can surround an arbitrary number of characters, we cannot guarantee that a string value (limited to 255 characters) will always satisfy the requirement. Thus all such tags come in opening and closing versions, with convenient shortcuts for smaller text values. A good example of this is the paragraph tag. A paragraph in HTML begins with `<p>` and ends with `</p>`. This is implemented in the *FormatParagraphStart* and *FormatParagraphEnd* functions, with the corresponding *ParagraphStart* and *ParagraphEnd* procedures (see Figure 3).

The HTML tag is built by the *FormatParagraphStart* function, including the alignment attribute if it's not the default. If the alignment attribute is used and we want to be informed about HTML 3.0 extensions, then a warning exception is raised. In the corresponding *ParagraphStart* procedure, the value returned above is written to the output file. The additional code ensures that the tag is not being placed in an invalid position in the document, and then registers its own presence.

The *sTags* variable is a *TStringList* containing all the opening tags that haven't been closed. This allows for warnings to be generated later when tags are closed out of sequence. The **try..except** block ensures that the tag is written to the output even when the warning in the function is raised. It's then re-raised to allow the generating program to see and handle this exception.

The *ParagraphEnd* procedure first checks that the paragraph tag is the next one that must be closed — an error is raised if it isn't. *ParagraphEnd* must then remove the tag from the list of those open and write the result to the output. Note that the ending tag is written followed by an end-of-line, using *Writeln*. Recall that spacing within an HTML document is generally ignored and that this is done merely to improve the readability of the generated document. Thus, these methods allow us to write the following Object Pascal code:

```
ParagraphStart(ahDefault);
Text('This text makes up the entire paragraph.');
ParagraphEnd;
```

which in turn generates this HTML:

```
<p>This text makes up the entire paragraph.</p>
```

```
{ Return start of paragraph as a string. }
function THTMLWriter.FormatParagraphStart(
  ahAlign: THTMLAlignHoriz): string;
begin
  Result := '<p';
  if ahAlign <> ahDefault then
    Result := Result + ' align=' + sAlignHoriz[ahAlign];
  Result := Result + '>';

  if (ahAlign <> ahDefault) and (erHTML3 in Errors) then
    raise EHTMLWarning.Create(tcParagraph,
           'ALIGN is HTML 3.0extension',Result);
end;

{ Write start of paragraph. }
procedure THTMLWriter.ParagraphStart(
  ZhAlign: THTMLAlignHoriz);
begin
  CheckNesting(tcParagraph,False,True,False,True,True);
  sTags.Add('p');
  try
    Write(fOutput,FormatParagraphStart(ahAlign));
  except on e: EHTMLWarning do
    begin
      Write(fOutput,e.Result);
      raise;
    end;
  end;
end;

{ Return end of paragraph as string. }
function THTMLWriter.FormatParagraphEnd: string;
begin
  Result := '</p>';
end;

{ Write end of paragraph. }
procedure THTMLWriter.ParagraphEnd;
begin
  CheckClosing(tcParagraph,(sTags.IndexOf('p') < 0),
              'p','paragraph');
  sTags.Delete(sTags.Count - 1);
  Writeln(fOutput,FormatParagraphEnd);
end;
```

Figure 3: Procedures and functions for starting and ending paragraphs.

To make it easier to create small paragraphs, two extra methods are defined: the *FormatParagraph* function and the *Paragraph* procedure. They combine the paragraph start and end methods above with the specified text in between. This allows us to generate an entire paragraph in one statement:

```
Paragraph('This text makes up the entire paragraph.',
         ahDefault);
```

Some tags have many attributes, most of which are not normally required. To make these tags easier to use, they can be invoked in two formats: with all the available parameters specified, or with just those parameters that are most commonly used (including none).

An example of this is the `img` tag that displays an image in the document. In its full format, `img` takes 10 parameters, of which three are common. Thus, one method, *FormatImageParams*, is defined that takes all the parameters and processes those that are not default values. The shorter version, *FormatImage*, simply calls the full one, passing across those parameters that have been supplied and sending default values for the remainder (see Figure 4).

Doing it this way means that the actual processing is only done in the one place, making maintenance easier, with changes to the full version being immediately reflected in the shorter version. Other tags that have similar versions include the body, the horizontal rule, lists, and tables.

Another thing to notice about the image processing code is the call to the function *CheckIfPercentage*. This is required because an image's height and width can be specified as an absolute value (the actual number of pixels), or as a relative amount (the percentage of the current browser size). To enable the methods to handle both cases without additional parameters, we make use of the fact that a size must be a positive value. We can then flag the fact that a percentage is required by using a negative value (with zero indicating that the default value be used).

To avoid having to remember that we must negate the value (and to reduce the confusion in the next person that has to maintain our code) a function is provided with the *THTMLWriter* component that converts the value for us. The *Percent* function takes a positive integer value and returns its negative counterpart. The negative value is then decoded in HTML generation and produces the required percentage value.

Thus, we can write the following code to have the image fill the entire browser:

```
ImageParams('athena.jpg','Athena',EmptyStr,aiDefault,
          Percent(100),Percent(100),0,0,0,False);
```

These relative/absolute values are also used in horizontal rules, marquees, and tables.

Colors in HTML are represented differently than those in Delphi. To overcome this, the processing converts from Delphi *TColor* values to a format suitable for HTML. This format is *#RRGGBB* where *RR* is the hexadecimal notation for the red component, *GG* for the green, and *BB* for the blue. Thus *clYellow* is converted to the string representation `'#FFFF00'`. All this is hidden behind the scenes and need not concern us.

Maps

Another interesting Object Pascal construct arises in the generating of areas in an in-line (or client-side) map. An area is defined by its shape (rectangle, circle, or polygon) and the appropriate coordinates. For example:

- A rectangle is defined by the x and y coordinates of its top-left and bottom-right corners.
- A circle is defined by the x and y coordinates of the center and its radius.

```
{ Return image, with all parameters, as string. }
function THTMLWriter.FormatImageParams(
  sImage, sAlt, sMap: string; aiAlign: THTMLAlignImage;
  Height, iWidth: Integer; iHSpace, iVSpace, iBorder: Byte;
  bIsMap: Boolean): string;
begin
  if (sImage = EmptyStr) and (erErrors in Errors) then
    raise EHTMLError.Create(tcImage,'Missing image source');

    Result := '<img src="' + sImage + '" border=' +
              IntToStr(iBorder);

    if sAlt <> EmptyStr then
      Result := Result + ' alt="' + sAlt + '"';
    if aiAlign <> aiDefault then
      Result := Result + ' align=' + sAlignImage[aiAlign];
    if iHeight <> 0 then
      Result := Result+CheckIfPercentage('height',iHeight);
    if iWidth <> 0 then
      Result := Result + CheckIfPercentage('width',iWidth);
    if iHSpace <> 0 then
      Result := Result + ' hspace=' + IntToStr(iHSpace);
    if iVSpace <> 0 then
      Result := Result + ' vspace=' + IntToStr(iVSpace);
    if sMap <> EmptyStr then
      Result := Result + ' usemap="' + sMap + '"';
    if bIsMap then
      Result := Result + ' ismap';

    Result := Result + '>';

    if ((iBorder<>0) or (iHSpace<>0) or (iVSpace<>0)) and
       (erNetscape in Errors) then
      raise EHTMLWarning.Create(tcImage,
        'BORDER,HSPACE,VSPACE are Netscape extensions',
        Result);
    if ((iWidth <> 0) or (iHeight <> 0) or
       (aiAlign <> aiDefault) or (sMap <> EmptyStr)) and
       (erHTML3 in Errors) then
      raise EHTMLWarning.Create(tcImage,
       'WIDTH,HEIGHT,ALIGN,USEMAP are HTML 3.0 extensions',
       Result);
end;

{ Write image, with all parameters. }
procedure THTMLWriter.ImageParams(sImage,sAlt,sMap: string;
  aiAlign: THTMLAlignImage; iHeight,iWidth: Integer;
  iHSpace,iVSpace,iBorder: Byte; bIsMap: Boolean);
begin
  CheckNesting(tcImage,False,True,False,True,True);

  try
    Write(fOutput,FormatImageParams(sImage,sAlt,sMap,
                    aiAlign,iHeight,iWidth,iHSpace,
                    iVSpace,iBorder,bIsMap));
  except on e: EHTMLWarning do
    begin
      Write(fOutput,e.Result);
      raise;
    end;
  end;
end;

{ Return image as string. }
function THTMLWriter.FormatImage(sImage,sAlt: string;
  aiAlign: THTMLAlignImage): string;
begin
  Result := FormatImageParams(sImage,sAlt,EmptyStr,aiAlign,
                              0,0,0,0,0,False);
end;

{ Write image. }
procedure THTMLWriter.Image(sImage,sAlt: string;
                             aiAlign: THTMLAlignImage);
begin
  ImageParams(sImage,sAlt,EmptyStr,aiAlign,
              0,0,0,0,0,False);
end;
```

Figure 4: Image processing methods.

- With a polygon, a list of x and y coordinate pairs define the shape, with the last pair combining with the first.

To handle this in *THTMLWriter* we need to pass across a variable number of parameters. Fortunately, this can be done in Object Pascal by declaring the parameter as an array of values, integers in this case. Normally an array's size must be declared, but by omitting the array size in the declaration, we can pass arrays of different sizes to the one routine (provided that the elements are of the correct type):

```
iCoords: array of Integer;
```

This is known as an *open array*. We can then process the array elements as necessary. To determine the number of elements in the array we use the *High* function. It returns the number of items less one, since the elements are numbered from zero. Figure 5 shows the implementation of this construct. Note that this can be extended to cater to a variable number of variable type parameters.

Straight Text

Straight text can be added to the generated document with the *Text* method, which simply writes the supplied value directly to the file. If the text contains any of the reserved characters, then these must be "escaped", i.e. converted to non-active values. These characters are the angle brackets (< >), ampersand (&), and quotation mark ("). Two methods are provided to perform this: *FormatEscapeText*, which returns the string in escaped form, and *EscapeText*, which writes the escaped text to the file.

Similarly, lists of strings can be written directly into the document or can be escaped first with the *TextList* and *EscapeTextList* methods. Note that these methods can be used to include any HTML constructs that have not been covered in the rest of *THTMLWriter*. Simply build the required syntax in a string (or list of strings) and then write it directly into the generated document.

The *InsertFile* method allows large sections of HTML or text to be copied into the generated document. No escaping of characters is performed by *InsertFile* as it is assumed that the document has already been processed in this way if necessary. This allows common sections of documents to be maintained in one place and used by many documents. Note that the file is inserted as static text. (We'll discuss allowing for dynamic replacement of values in the section, "HTML Templates.")

The *Initialise* method resets all internal variables that are used in checking relationships between tags. This should be called before any HTML is generated to ensure that

```
{ Return area for in-line map as string. }
function THTMLWriter.FormatMapArea(shShape: THTMLShapes;
  iCoords: array of Integer; sUrl,sAlt: string): string;
var
  i     : Integer;
  sSep  : string;
begin
  if erErrors in Errors then
    begin
      i := High(iCoords) + 1; { Num of entries in array. }
      if ((shShape = shRect)   and (i<>4))  or
         ((shShape = shCircle)  and (i<>3)) or
         ((shShape = shPolygon) and ((i<6) or Odd(i))) then
    raise EHTMLError.Create(tcMap,
                       'Invalid number of coordinates for ' +
                       sShape[shShape]);
  end;

  Result := '<area shape=' + sShape[shShape];
  if shShape <> shDefault then
    begin
      sSep := ' coords="';
      for i := 0 to High(iCoords) do
        begin
          Result := Result + sSep + IntToStr(iCoords[i]);
          sSep := ',';
        end;
      Result := Result + '"';
    end;
  if sUrl = EmptyStr then
    Result := Result + ' nohref'
  else
    Result := Result + ' href="' + sUrl + '"';
  if sAlt <> EmptyStr then
    Result := Result + ' alt="' + sAlt + '"';
  Result := Result + '>';
end;
```

Figure 5: The *FormatMapArea* function with an open array parameter.

errors are not introduced from any earlier processing. The *Finalise* method checks that all the open tags have been correctly closed, before writing the terminating HTML tags (unless a file insertion or merge was performed) and then closing the output file. *Finalise* should be the last thing called in generating the document.

Figure 6 presents a complete list of the methods available in *THTMLWriter*. Procedures are listed on the left and their corresponding functions (where applicable) are displayed on the right. Each set of procedures and functions are grouped by purpose.

HTML Templates

In addition to directly generating HTML for our page, *THTMLWriter* can also generate a *template* document. A template is basically just another HTML document, but has special tags indicating where variable information should be inserted. This replacement is done under programmatic control. There are advantages to generating a page with a template:

- Most of the document can be written directly in HTML, making the document easier to maintain, and allowing for minor alterations without having to recompile the program.
- The template cleans up the code in the generating program. Instead of describing each line and construct individually, the document can be produced with a single method call.

The implementation of variable tags within a template is different than in standard HTML. Braces ({ }) are used to identify these tags, and hold the name of the variable whose contents will appear here. For example, let's say a variable name *User* has been assigned the value `'Keith Wood'`. Using a template that contains this code:

```
<h1>{user}'s Template Demonstration</h1>
```

generates this HTML:

```
<h1>Keith Wood's Template Demonstration</h1>
```

This has been extended to allow for entire files to be inserted into the template. Follow the opening brace with a caret (^) and the name of the file to be added, and the file will be inserted at that point. For example, if the file FOOTER.FTP contains:

```
<hr>
<p><font size=-1>Generated by the THTMLWriter
component.</font></p>
```

then, a template with the following code:

```
<p>The footer for the page comes from another file.</p>
<p>View the <a href="sourcem.htm">source code
and templates</a>.</p>
{^footer.htt}
</body>
```

generates this:

```
<p>The footer for the page comes from another file.</p>
<p>View the <a href="sourcem.htm">source code and
templates</a>.</p>
<hr>
<p><font size=-1>Generated by the THTMLWriter
component.</font></p>
</body>
```

This technique can be used for common sections of HTML, allowing them to be maintained in a single place while having their effect apply to the entire site. Examples of their use are the body tag, allowing a common color scheme to be applied in document footers and in toolbars.

Both of these replacement strategies are applied recursively. This means that a file can be inserted that also contains variable tags that are themselves replaced. Similarly, a variable can be replaced with another variable tag causing different files to be inserted depending on some condition.

As a convention, these template files should have an extension of .HTT, to differentiate them from straight HTML files (.HTM).

A New Class

To enable these variables and their values to be specified, a new class has been defined in the unit along with the *THTMLWriter* component. This is the *THTMLDictionary* class. It's based on *TStringList*, but adds two new methods. The first is *AddFieldAndValue*

Procedures	Functions
Content	*FormatContent*
Head	*FormatHead*
Title	*FormatTitle*
IsIndex	*FormatIsIndex*
Base	*FormatBase*
Meta	*FormatMeta*
Comment	*FormatComment*
BodyParams	*FormatBodyParams*
Body	*FormatBody*
Sound	*FormatSound*
HeadingStart	*FormatHeadingStart*
HeadingEnd	*FormatHeadingEnd*
Heading	*FormatHeading*
ParagraphStart	*FormatParagraphStart*
ParagraphEnd	*FormatParagraphEnd*
Paragraph	*FormatParagraph*
ImageParams	*FormatImageParams*
Image	*FormatImage*
MapStart	*FormatMapStart*
MapEnd	*FormatMapEnd*
MapArea	*FormatMapArea*
ListStartParams	*FormatListStartParams*
ListStart	*FormatListStart*
ListEnd	*FormatListEnd*
ListItemParams	*FormatListItemParams*
ListItem	*FormatListItem*
HorizRuleParams	*FormatHorizRuleParams*
HorizRule	*FormatHorizRule*
LineBreak	*FormatLineBreak*
WordBreak	*FormatWordBreak*
TextEffectStart	*FormatTextEffectStart*
TextEffectEnd	*FormatTextEffectEnd*
TextEffect	*FormatTextEffect*
FontStart	*FormatFontStart*
FontEnd	*FormatFontEnd*
Font	*FormatFont*
BaseFont	*FormatBaseFont*
SpecialChar	*FormatSpecialChar*
SpecialCharValue	*FormatSpecialCharValue*
MarqueeStart	*FormatMarqueeStart*
MarqueeEnd	*FormatMarqueeEnd*
Marquee	*FormatMarquee*

Procedures	Functions
LinkStart	*FormatLinkStart*
LinkEnd	*FormatLinkEnd*
Link	*FormatLink*
TableStartParams	*FormatTableStartParams*
TableStart	*FormatTableStart*
TableEnd	*FormatTableEnd*
TableRowStartParams	*FormatTableRowStartParams*
TableRowStart	*FormatTableRowStart*
TableRowEnd	*FormatTableRowEnd*
TableHeadingStartParams	*FormatTableHeadingStartParams*
TableHeadingStart	*FormatTableHeadingStart*
TableHeadingEnd	*FormatTableHeadingEnd*
TableHeadingParams	*FormatTableHeadingParams*
TableHeading	*FormatTableHeading*
TableCellStartParams	*FormatTableCellStartParams*
TableCellStart	*FormatTableCellStart*
TableCellEnd	*FormatTableCellEnd*
TableCellParams	*FormatTableCellParams*
TableCell	*FormatTableCell*
FormStart	*FormatFormStart*
FormEnd	*FormatFormEnd*
TextField	*FormatTextField*
PasswordField	*FormatPasswordField*
CheckboxField	*FormatCheckboxField*
RadioField	*FormatRadioField*
SubmitField	*FormatSubmitField*
ResetField	*FormatResetField*
ImageField	*FormatImageField*
HiddenField	*FormatHiddenField*
SelectStart	*FormatSelectStart*
SelectEnd	*FormatSelectEnd*
SelectOption	*FormatSelectOption*
TextAreaStart	*FormatTextAreaStart*
TextAreaEnd	*FormatTextAreaEnd*
TextArea	*FormatTextArea*
Text	
EscapeText	*FormatEscapeText*
TextList	
EscapeTextList	
InsertFile	
MergeFile	
Initialise	
Finalise	

Figure 6: The *THTMLWriter* component's methods.

Internet Definitions	
CGI	Common Gateway Interface. The standard for external gateway programs to interface with information servers such as HTTP servers. The program should generate an HTML or other document for return to the user.
GIF	Graphic Interchange Format, a file format standard developed by CompuServe as a device-independent method for storing images. GIF allows high-quality, high-resolution graphics to be displayed on a variety of graphics hardware and is intended as an exchange and display mechanism for graphic images.
HTML	HyperText Markup Language. The page description language used on the World Wide Web.
HTTP	HyperText Transfer Protocol. A generic, stateless, object-oriented protocol, suitable for hypermedia information systems. It's the main protocol for moving HTML and other documents around the Web.
JPEG	A standard image compression mechanism, JPEG (pronounced "jay peg") is the acronym for Joint Photographic Experts Group, the original name of the committee that wrote the standard. JPEG is designed for compressing either full-color or gray-scale images of natural, real-world scenes.
MIME	Multipurpose Internet Mail Extensions. A standard for describing the contents of documents transferred through e-mail and HTTP. The default type for HTML documents is text/html.
URL	Universal Resource Locator. The address of a document, it takes the form: *//host.domain*[*:port*]*/path/document*.
WinCGI	The Windows version of CGI.
WWW	World Wide Web. The hypertext/graphical part of the Internet.

by its name) and its associated value to the list of variables to be inserted. *AddFieldAndValue* is the method used in our programs. The other method, *GetValue* is called during *THTMLWriter*'s template processing. Given a field name, *GetValue* returns its corresponding value. If the variable does not exist in the dictionary, the variable tag in the template is simply deleted.

The dictionary makes use of the *TStringList*'s ability to store an associated object with each string. Unfortunately, all we want to store is another string, but *TStringList* is expecting a value derived from *TObject*. One of the ways to overcome this is by defining a new class, *THTMLFieldValue* (derived directly from *TObject*), that has a single property which is the string we require.

All this is handled behind the scenes and should not be manipulated directly. This allows us to write code similar to this example:

```
dicDictionary := THTMLDictionary.Create;
try
  dicDictionary.AddFieldAndValue('user',Username2.Text);
  with HTMLWriter do begin
    Initialise;
    MergeFile('template.htt',dicDictionary);
    Finalise;
  end;
finally
  dicDictionary.Free;
end;
```

Errors and Warnings

Various errors can occur while generating HTML. These are all handled as exceptions within *THTMLWriter*, allowing us to trap, identify, and handle them as necessary. To ease this process, a special base exception, *EHTMLException*, is defined in the unit with *THTMLWriter*. The error and warning exceptions, *EHTMLError* and *EHTMLWarning*, are then derived from *EHTMLException*. This means that these exceptions can be easily identified as having been raised by *THTMLWriter*.

Each of these exception classes has a *Tag* property that identifies the type of construct that caused the error. These are defined by the *THTMLTagCategory* type and can be translated using the *TagCategory* array, with *Tag* as the index. Additionally, the *EHTMLWarning* class has a *Result* property that contains the formatted HTML text for further manipulation within the generating program.

THTMLWriter can produce two types of exceptions: errors and warnings. Errors are situations that should not be ignored, such as missing mandatory values for tags, while warnings cover circumstances that may not be what we intended. Another major difference in their implementation is that when a warning is raised, the original action requested is guaranteed to have been completed. This allows the warning to be reviewed and processing to continue. (Note that not all error conditions are trapped in the *THTMLWriter* component, only those that are most common.)

Warnings come in four flavors. These are basic warnings, such as not having a **Submit** button on a form, and constructs that are HTML 3.0, Netscape, or Microsoft Internet Explorer extensions to HTML 2.0. This means that we can check on tags that may not be correctly interpreted in various browsers.

To allow us to control the kind of errors and warnings that are generated, *THTMLWriter* publishes an *Errors* property, which is a set of the different error and warning types described above (see Figure 7). Simply include in the set those errors and warnings that you want to understand. The default value is to only be notified of errors and basic warnings. Note that this set can be empty, meaning that no errors or warnings are raised. I suggest that this is done only after debugging the generating program.

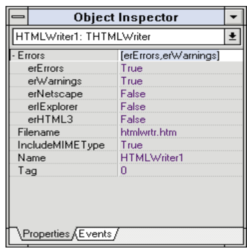


Figure 7: The Delphi Object Inspector showing the nested values for the *Errors* property.

Demonstration

The demonstration project included with this article shows some of *THTMLWriter*'s functionality (see Figure 8). It presents four options for generating HTML, each of which takes a parameter for inclusion in the output. In each case, the HTML is written into a predefined file whose name is then displayed. This file should then be loaded into your browser to view its appearance. Links from these generated pages show you the Object Pascal code that produced them using the *THTMLWriter* component. If you don't have a browser, you can download one (e.g. Netscape); or download I-View (a browser intended for use on local documents only); or use a Delphi viewer component such as *THtmlViewer*.

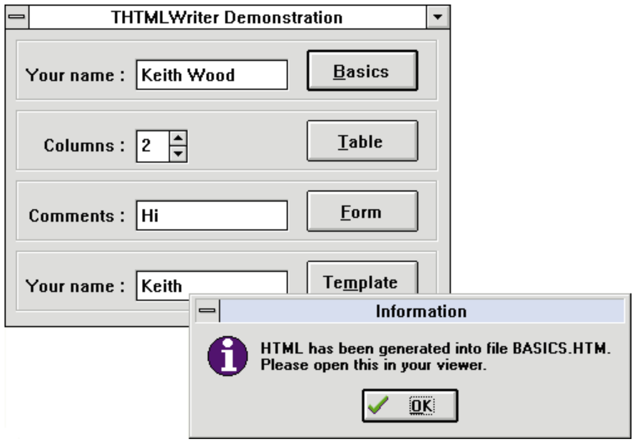


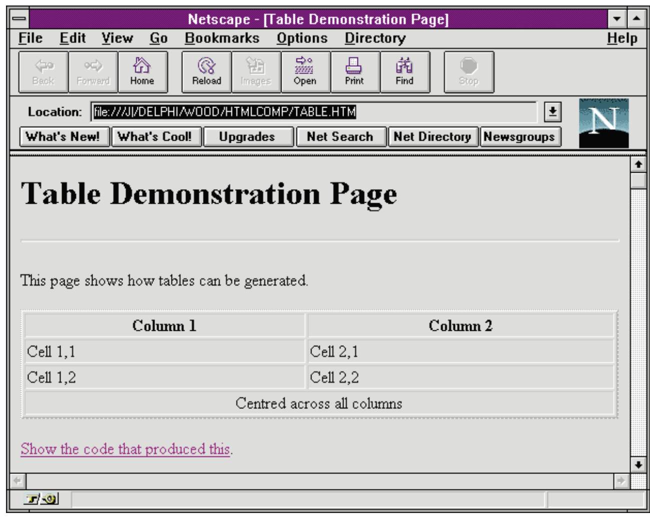
Figure 8: The sample application, THTMLWriter Demonstration. After entering a value in the **Your name** field, the user clicks the **Basics** button. An information dialog box appears and instructs you to view the updated .HTM file in your browser.

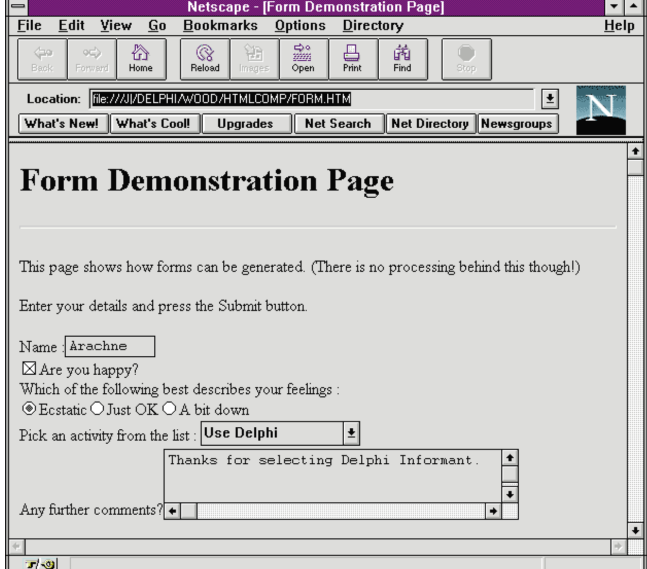
The basic demonstration shows the variety of constructs that can be produced through *THTMLWriter*. It includes examples of headings, text formatting, images, lists, and links. The user's name, as entered on the screen, is inserted into the title and the first heading in the document (as is shown in Figure 1). Follow the link to see how all this was done.

The table demonstration asks for the number of columns to be generated, between two and five, and then produces a table accordingly (see Figure 9). Note that most of the table procedure calls use the simplest version, while the last table cell requires all the possible parameters to be specified so it can be centered across all the columns.

In the form example (see Figure 10), the various interactive controls available in a form are shown. The text entered on the screen becomes the default value for the Comments field. After entering any values required, you press the **Submit** button to send the form. This one doesn't go anywhere of course, but simply calls itself. This should display the parameters entered in the `Location` box at the top of the browser.

Finally, Figure 11 shows the result of using a template for generating HTML documents. The name entered on the screen is inserted into the template, along with the





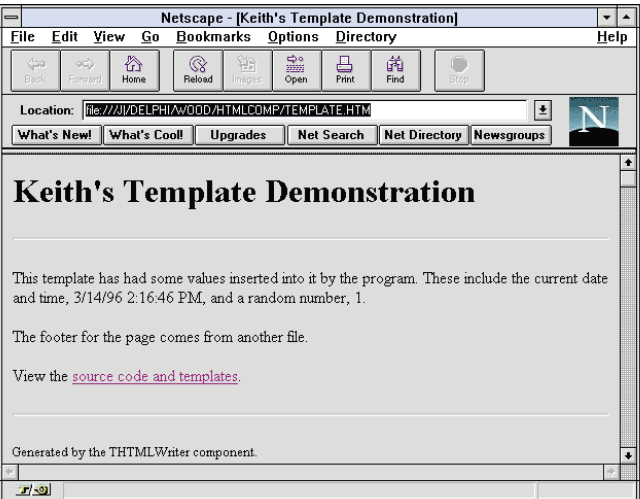


Figure 9 (Top): The table demonstration page showing how the cells of an HTML table are formatted. **Figure 10 (Middle):** The form demonstration page with entry fields, radio buttons, and check boxes. **Figure 11(Bottom):** The *THTMLWriter*'s template document feature was used to create this page.

HTML 2.0 Reference	http://www.w3.org/hypertext/WWW/MarkUp/html-spec/html-spec_toc.html
HTML 3.0 Reference	http://www.w3.org/hypertext/WWW/MarkUp/html3/Contents.html
Netscape Extensions	http://www.netscape.com/assist/net_sites/html_extensions.html
Microsoft Internet Explorer Extensions	http://www.microsoft.com/intdev/browser/iexplore.htm
HTML Tutorial	http://www.utirc.utoronto.ca/HTMLdocs/NewHTML/htmlindex.html
CGI	http://hoohoo.ncsa.uiuc.edu/cgi/
WinCGI	http://www.city.net/win-httpd/httpddoc/wincgi.htm
I-View Browser	http://www.talentcom.com/iview.htm
THtmlViewer Component	http://www.empire.net/~dbaldwin/

Figure 12: HTML-related references available on the Internet.

current date and time and a random number between 1 and 100. A second document is included in the first, showing how common sections can be maintained in one spot only and used in many documents as required.

(Note that the demonstration program runs best outside the Delphi IDE. Otherwise, exceptions that are trapped internally appear and disrupt the flow of the program.)

Conclusion

The *THTMLWriter* component allows us to generate HTML on-the-fly. It provides access to most of the features of the various HTML versions, along with many Netscape and some Microsoft Internet Explorer extensions. It detects many errors and warns us of possible problems while generating our pages.

Combine *THTMLWriter* with a CGI component and you have a Web site just waiting to happen. All from our familiar Delphi environment.∆

The THTMLWriter component and demonstration project referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\MAY\-DI9605KW.

Keith Wood is an analyst/programmer with CSC Australia, based in Canberra. He started using Borland's products with Turbo Pascal on a CP/M machine. Although not working with Delphi currently he has enjoyed exploring it since it first appeared. You an reach him via e-mail at kwood@nla.gov.au or by phone (Australia) 6 291 8070.

Informant Spotlight

Delphi / Object Pascal

By Ray Lischner

Virtual or Dynamic?

An Examination of Object Pascal's Virtual Methods

Delphi's dialect of Object Pascal is unusual among object-oriented programming languages in that it gives programmers a choice of using **virtual** or **dynamic** methods. Although their semantics are the same, the two kinds of methods have different performance characteristics, so you need to decide which to use when devising new classes. This article reveals the details of **virtual** and **dynamic** methods. Using this information, you can decide which best suits your needs.

Borland's *Object Pascal Language Guide* offers this guideline on page 92: "In general, virtual methods are the most efficient way to implement polymorphic behavior. Dynamic methods are useful only in situations where a base class declares a large number of virtual methods, and an application declares a large number of descendent classes with few overrides of the inherited virtual methods." Let's take a look inside Delphi to understand why this is true.

Due to an unfortunate collision of terminology, **virtual** and **dynamic** methods are both referred to as virtual methods. To keep the distinction clear, the **virtual** and **dynamic** keywords are shown in boldface when referring to the specific kinds of methods, and the word *virtual* is in a plain typeface when referring to a generic virtual method, that is, a method that can be **virtual** or **dynamic**.

```
{ Simple class declarations for virtual and dynamic
  methods. Measure the relative performance of calling
  the four methods, to compare the speed of virtual
  method calls with that of dynamic method calls. }
type
  TBaseClass = class
    procedure DynamicMethod; dynamic;
    procedure OverrideDynamic; dynamic;
    procedure VirtualMethod; virtual;
    procedure OverrideVirtual; virtual;
  end;
  TIntermediateClass = class(TBaseClass)
    procedure OverrideDynamic; override;
    procedure OverrideVirtual; override;
  end;
  TDerivedClass = class(TIntermediateClass)
    procedure OverrideDynamic; override;
    procedure OverrideVirtual; override;
  end;
```

```
{ Basic timing loop. Duplicate this loop
  for each method. }
Obj := TDerivedClass.Create;
try
  Ticks := GetTickCount;
  for I := 1 to Iterations.Value do
     Obj.DynamicMethod;
  Ticks := GetTickCount - Ticks;
finally
  Obj.Free;
end;
```

Figure 1 (Top): Simple class declarations for measuring performance. **Figure 2 (Bottom):** A timing loop to measure performance.

Performance of Virtual and Dynamic Methods

Virtual and **dynamic** methods have identical syntax. You can change a **virtual** method to a **dynamic** method, or vice versa, without changing the behavior of your program. All that changes is the running time and amount of memory used. Let's compare the running time for calling the two kinds of virtual methods.

Figure 1 shows some simple class declarations. The goal is to compare the speed of calling the virtual methods. *VirtualMethod* and *DynamicMethod* are declared in the base class and called for an instance of the derived class. *VirtualOverride* and *DynamicOverride* are overridden in the derived class. The performance tests measures the calling time for all four methods.

Figure 2 shows a timer that measures the relative performance of a method call with a loop that calls the method many times. The timer creates an instance of *TDerivedClass* and repeatedly calls one of the methods. To account for the time taken by the loop itself, the timer separately measures a loop that does nothing, and subtracts that time from the other measurements.

Figure 3 shows a graph of the elapsed time for varying numbers of iterations. The actual time depends on the system and environment where the tests are run. In any case, the relative performance is more important than the absolute numbers. Steeper lines indicate slower performance.

As you can see, **dynamic** methods are slower than **virtual** methods. Overriding a **virtual** method has no impact on performance. (This is hard to see because the two lines coincide, indicating identical performance.) Overriding a **dynamic** method improves performance.

Therefore, if speed were your sole concern, you should always use **virtual** methods. There are times, however, when you might want to consider **dynamic** methods, namely, when space is at a premium.

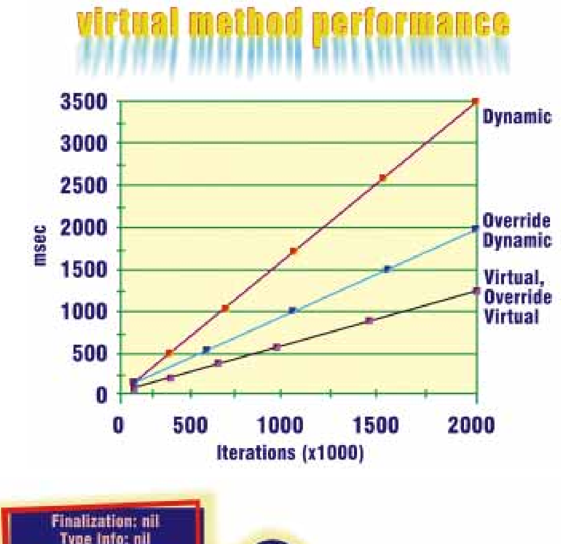
In this contrived example, **virtual** and **dynamic** methods take up about the same amount of memory, but as you add more methods and more classes, **dynamic** methods tend to use less space than **virtual** methods. To understand this, it is necessary to look at how **virtual** and **dynamic** methods are implemented in Delphi.

The Virtual Method Table

Every class has a Virtual Method Table (VMT) that contains a pointer for every **virtual** method defined by the class and its base classes. The VMT also points to a Dynamic Method Table (DMT) that contains entries for every **dynamic** method defined or overridden by the class. Unlike the VMT, the DMT contains no entries for methods defined in a base class.

Figure 4 illustrates the VMTs for the example classes. A VMT contains two parts. The first part contains the instance size and several pointers: to other tables, to the class name, to the VMT for the base class, and to the **virtual** methods defined by *TObject*. The second part of the VMT is a table of code pointers for all the **virtual** methods defined by the class and its base classes.

In Delphi 1, the "pointers" in the first part of the VMT are near pointers: offsets into the code segment that contains the methods for the class. In Delphi 2, the pointers are 32-bit pointers. Thus, the VMT is a different size in Delphi 1 and Delphi 2, and using the pointers is different in Delphi 1 and Delphi 2. That's one reason why Borland does not document these details. They are free to change the VMT implementation without dealing with backward compatibility issues.



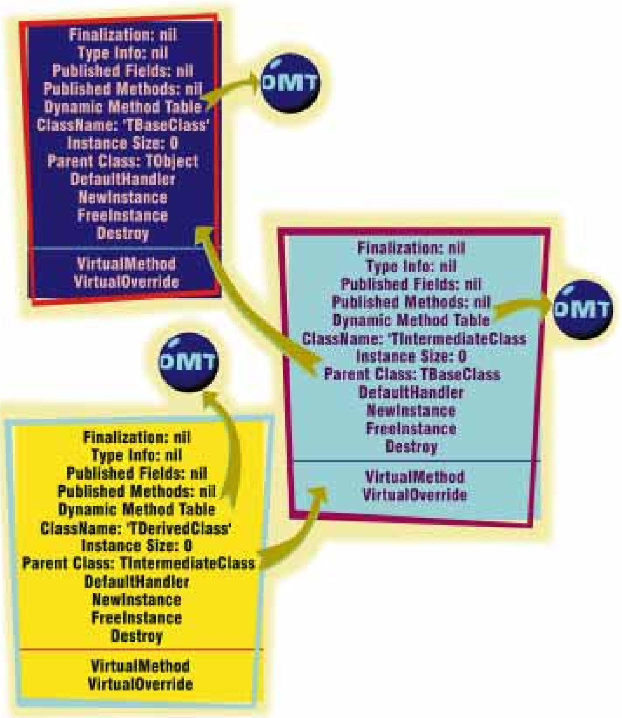


Figure 3 (Top): Relative performance of **virtual** and **dynamic** methods. **Figure 4 (Bottom):** Virtual Method Tables (VMTs).

Nonetheless, it is a simple matter to hide these differences in some simple functions, providing portable access to the VMT.

Every object has a pointer to its class' VMT. This pointer is stored as the first field of the object. The *ClassType* method returns a pointer to the second part of the VMT. To access the first part, you need to subtract the VMT size from the *ClassType* pointer.

To access the second part of the VMT, simply treat the pointer value that *ClassType* returns as a pointer to an array of code pointers. The compiler assigns a number to a **virtual** method; this number is an index into the VMT pointers. A call to a **virtual** method, therefore, is very simple and quick: just index into the VMT, and call that method. Figure 5 shows the equivalent Pascal code.

```
procedure CallVirtualMethod(Obj: TObject;
                            MethodIndex: Integer);
type
  PVmt = ^TVmt;
  TVmt = array[0..MaxVmt] of Pointer;
  TProcedure = procedure of object;
var
  Vmt: PVmt;
  Method: TProcedure;
begin
  { Get the pointer to the VMT. }
  Vmt := Obj.ClassInfo;
  { Lookup the method by its index. }
  TMethod(Method).Code := Vmt[MethodIndex];
  TMethod(Method).Data := Obj;
  { Call the virtual method. }
  Method;
end;
```

Figure 5: Pascal equivalent to calling a **virtual** method.

The Dynamic Method Table

Dynamic methods are also assigned numbers, but the numbers are not indexes into tables. Instead, a Dynamic Method Table (DMT) is a sparse table. Each entry consists of a method number and a method's code pointer. A call to a **dynamic** method is compiled into a search through the DMT for a matching method number. If the method number cannot be found, the search continues with the parent class' DMT.

The search for a **dynamic** method is performed by a special subroutine which is written in assembly language for maximum speed. Looking up a **dynamic** method requires understanding the format of the DMT. Borland does not document the format of the DMT, so let's take some extra time to understand what the DMT looks like. Figure 6 depicts the DMTs for the example classes.

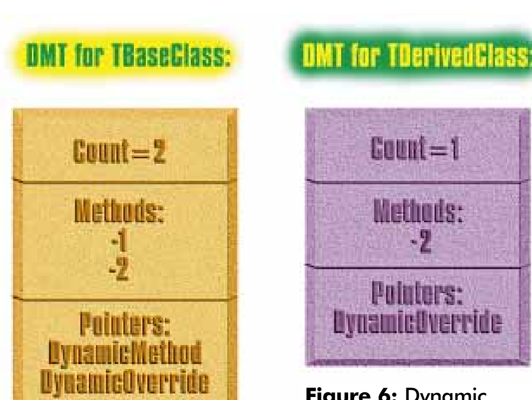


Figure 6: Dynamic Method Tables (DMTs).

A DMT is organized to help Delphi look up **dynamic** methods rapidly. The DMT starts with a count of the number of **dynamic** methods in the table. Then, the table contains a list of the **dynamic** method numbers for every **dynamic** method defined or overridden by the class. This list is followed by a list of code pointers for the **dynamic** methods. Each method pointer corresponds to a method number.

The DMT is organized this way to allow rapid lookup for a **dynamic** method number. The Intel x86 instruction set has a special instruction (SCAS) that scans a sequence of numbers, looking for a specific number.

```
procedure CallDynamicMethod(Obj: TObject;
                            MethodNumber: Integer);
type
  TProcedure = procedure of object;
var
  Vmt: PVmt;
  Method: TProcedure;
begin
  Vmt := GetVmt(Obj.ClassType);
  TMethod(Method).Data := Obj;
  TMethod(Method).Code :=
    GetDynamicMethod(Vmt,MethodNumber);
  Method;
end;
```

Figure 7: Pascal equivalent to calling a **dynamic** method.

Figure 7 shows the equivalent Pascal code for looking up and calling a **dynamic** method. The code depends on several routines that get information out of the VMT and DMT. These routines are included in the full code listings, in the VmtInfo unit.

Why Dynamic Methods?

Dynamic methods have an advantage over **virtual** methods because a DMT does not contain any entries for ancestor classes. This means DMTs can be much smaller than VMTs. Adding a **virtual** method to a base class has a multiplicative effect: the number of **virtual** methods times the number of base classes.

Conversely, **dynamic** methods avoid the multiplicative effect of **virtual** methods. Changing one method from **virtual** to **dynamic** might eliminate hundreds of VMT entries, while adding one DMT entry. The downside of a DMT is that searching for a **dynamic** method entry is slower than looking up a **virtual** method.

It is a classic trade-off: **dynamic** methods usually have smaller tables and slower lookup, and **virtual** methods usually have larger tables and faster lookup.

However, this is only half the story. Consider what happens when a **dynamic** method is overridden in every derived class. A DMT contains a method number and a method pointer, but a VMT contains only a method pointer. The result is that the DMTs take up more space than the equivalent VMTs, so **dynamic** methods can be larger and slower.

Therefore, it is proper to use **dynamic** methods only when you know that a method will rarely be overridden. The benefit is greatest when there are many derived classes, few of which override the **dynamic** method.

Dynamic methods slow down even more as the class inheritance tree grows. Calling a **dynamic** method requires searching in every DMT from the derived class

to the base class that defines the method. If a class overrides the method, then the search can end early. However, most **dynamic** methods are not overridden (for very good reasons, as described earlier), so a **dynamic** method call usually searches almost every ancestor class.

When you use **dynamic** methods, you can help the performance by leaving out DMTs whenever possible. If a class does not declare or override any **dynamic** methods, then it has no DMT, which means the DMT pointer in its VMT is **nil**. This helps to speed up the search for the **dynamic** method because it is faster to skip over a missing DMT than it is to search a small DMT.

Message Handlers Are Like Dynamic Methods

Another consideration is that message handlers are also stored in the DMT, mixed with the dynamic methods: **dynamic** methods are assigned negative numbers; message handlers use positive numbers. This is why you cannot declare a message handler for any message number greater than $7FFF. It would be considered a negative number, which interferes with **dynamic** methods.

Message handlers are called in a manner similar to **dynamic** methods. The message number is looked up in the DMT, and the corresponding method is called with the message record as the sole argument. Thus, when considering whether a class should have any **dynamic** methods, realize that a message handler takes up space in a DMT, just like a **dynamic** method.

If a class has at least one message handler or at least one **dynamic** method, then it has a DMT. If a class has no DMT, then looking up **dynamic** methods or message handlers is slightly faster. If a class has many message handlers, then looking up a **dynamic** method or message handler in the DMT is slightly slower because the DMT is larger.

Real Numbers

To help understand the impact on the space occupied by a DMT, let's consider a real class: *TControl*. Every visual control in Delphi (including forms) inherits from *TControl*, so it has 189 subclasses. (Note that these numbers are for the December Beta release of Delphi 2. The figures for the final release may vary slightly.) *TControl* and its base classes define 25 **virtual** methods. Every one of the 189 subclasses of *TControl* must contain an entry for every one of the 25 **virtual** methods, plus any other **virtual** methods that the derived classes define. Every control and form you create in a project also has its own copy of these 25 **virtual** method pointers. In Delphi 1 and Delphi 2, a code pointer is always four bytes long.

TControl and its base classes also define 26 **dynamic** methods. Most of these methods are rarely overridden in a derived class. If these **dynamic** methods were **virtual**, then all 189 derived classes would have their VMTs grow by 26 entries, or a total growth of 19K bytes.

There are times when saving 19K bytes in an application is significant. There are other situations where the use of **dynamic** methods does not result in nearly so large a space reduction.

Let's compare *TControl* with another Delphi class. *TGraphicsObject* has one **dynamic** method and seven **virtual** methods. It has three derived classes (*TFont*, *TPen*, *TBrush*), none of which override the **dynamic** method (*Changed*). This results in a space saving of four bytes, since *TGraphicsObject* would not have a DMT if the *Changed* method were **virtual**.

The *Changed* method is called when any aspect of a graphical object changes, such as font size, pen color, or brush style. In some applications, such as a word processor or bitmap editor, this might happen often. For a method that might be called often, **virtual** is usually better than **dynamic**. When the use of **dynamic** saves only four bytes, the choice is clear — in this case, clearly wrong.

Conclusion: When in Doubt, Go Virtual

This excursion into virtual methods supports the guidelines of the *Object Pascal Language Guide*, but the *Guide*'s advice is not strong enough. Although **dynamic** methods are useful in the root classes of the VCL, there is almost no other situation that calls for **dynamic** methods.

For the component writer and application programmer alike, it is best to use **virtual** methods:

- Certainly, you should use the **virtual** directive for any method called from a time-sensitive or compute-intensive loop.
- If a class doesn't have many derived classes, then use **virtual**.
- If most of the virtual methods are overridden in derived classes, use **virtual**.
- If you can afford a little extra space, use **virtual**.
- If the classes in the inheritance tree contain many message handlers — which slow down dynamic method calling — use **virtual** methods.

Only when space is at a premium should you use **dynamic** methods, and then only in the situations defined by the *Object Pascal Language Guide*: a class with many derived classes that do not override many methods.

Perhaps the best and simplest rule is: When in doubt, use **virtual**. ∆

The demonstration project referenced in this article is available on the Delphi Informant Works CD located in INFORM\96\MAY\DI9605RL.

Ray Lischner is the founder and president of Tempest Software, which specializes in object-oriented components and tools. Mr Lischner has been a software developer for over a decade, at large and small firms across the United States. His current work in progress is *Delphi Secrets*, a book that reveals undocumented aspects of Delphi, to be published by the Waite Group Press in 1996.

DBNavigator

Delphi 2 / Object Pascal

By Cary Jensen, Ph.D.

Elysian Fields

Leveraging the Delphi 2 Fields Editor

Delphi 2, the new 32-bit version of Delphi, has been available for a couple of months now. And although you might have expected this version to be little more than a 32-bit port of the existing product, it is much more. Delphi 2 contains so many new features that it represents a major leap forward for this already revolutionary product.

Fortunately for database developers, many of the enhancements are database related. This month's "DBNavigator" takes a closer look at one of these enhancements, the new Fields Editor. This powerful tool will be demonstrated by creating several simple data entry forms.

In Delphi 2 this will almost always start with creating a data module.

The Data Module

As you learned in last month's column, the data module is a form-like object on which you place data access components. Specifically, you place your DataSource and DataSet (Table, Query, and StoredProc) components on a data module. These components can then be made available to any form in your application by adding the data modules' units to the form unit's **uses** clause.

An example of a data module is shown in Figure 1. It contains a DataSource and Table component. Just as you would on a form, you link the DataSource to the Table by setting the DataSource's *DataSet* property to *Table1* (or whatever name you have assigned to the table). Likewise, the Table is defined by setting its *DatabaseName* and *TableName* properties. In the data module shown in Figure 1, the *DatabaseName* has been set to `DBDEMOS` (an alias installed by Delphi), and the *TableName* to `CUSTOMER.DB`.

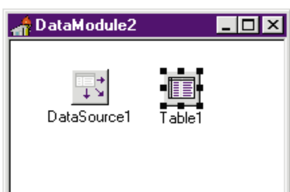


Figure 1: A data module including a DataSource and a Table.

As mentioned earlier, you now can add the data module unit's name to the **uses** clause for any unit that needs to access the DataSource or Table. This can be done manually by typing in the **uses** clause, or by selecting **File | Use Unit** from the form that needs this access. There is also a third way in which Delphi automatically adds this unit to the form's **uses** clause — a technique that employs the new Fields Editor.

The Delphi 2 Fields Editor

The Fields Editor is a tool you use to instantiate (create) *TField* components for the fields of a DataSet. In addition, the Fields Editor permits you to create calculated fields (fields whose read-only contents are based on a calculation), as well as lookup fields (fields that display associated data from another DataSet).

The Fields Editor in Delphi 2 is similar in several respects to that found in Delphi 1. However, it sports a slimmed-down interface, as well as a number of powerful new features.

To access the Fields Editor, right-click a DataSet component and select **Fields Editor** from the SpeedMenu displayed, or simply

double-click the DataSet component. Figure 2 shows the new Fields Editor (on the left) beside its Delphi 1 counterpart. As you can see, the new Fields Editor is smaller, and omits the button panel. However, it retains the "VCR" buttons used to navigate the records of an active DataSet during design time.

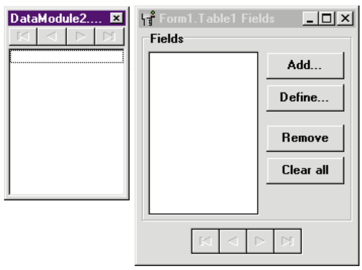


Figure 2: The Delphi 2 Fields Editor (on the left) and its Delphi 1 predecessor.

You use a SpeedMenu to control many of the features of the new Fields Editor (see Figure 3). For example, to instantiate one or more fields for a DataSet, you right-click the Fields Editor and select **Add fields**. To create a new calculated field or a lookup field, select **New field**. Each of these selections results in the display of an appropriate dialog box you use to select or define the fields you are creating.

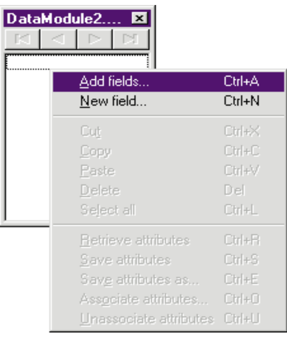


Figure 3: Many of the features of the Fields Editor are accessed using its SpeedMenu.

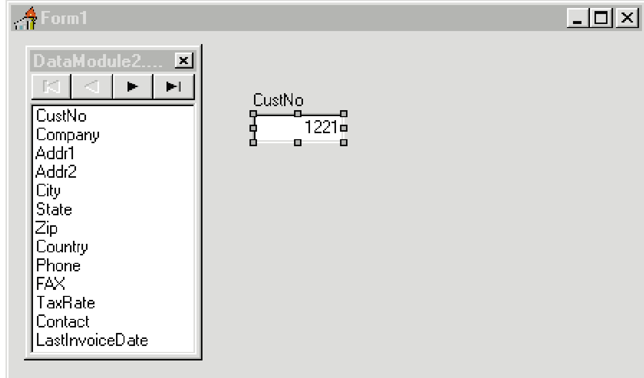
Once you accept the dialog box, control returns to the Fields Editor, and the fields created appear in the Fields Editor list box. Figure 4 shows how the Fields Editor appears when all fields from the CUSTOMER.DB table are instantiated.

Figure 4: All fields from the CUSTOMER.DB table have been instantiated, and are listed in the Fields Editor.

Placing Fields from the Fields Editor

One of the major new features of the Fields Editor is that it permits you to drag and drop fields onto a form. For example, if you want to place the CustNo field of the CUSTOMER.DB table onto a form, simply select the **CustNo** field in the Fields Editor, drag it onto the form, and release. Delphi will place the field where you specified, as shown in Figure 5.

Earlier in this article I mentioned that Delphi automatically adds the unit name of a data module to a form. This is how it happens: If you use drag-and-drop to add a field from the Fields Editor to a form, and the field you are dropping is associated with a data module, and that data module's unit has not yet been added to a **uses** clause on the form's unit, Delphi displays the dialog box shown in Figure 6. If you select **Yes** in this dialog box, Delphi adds the appropriate **uses** clause to the **implementation** section of the form's unit.



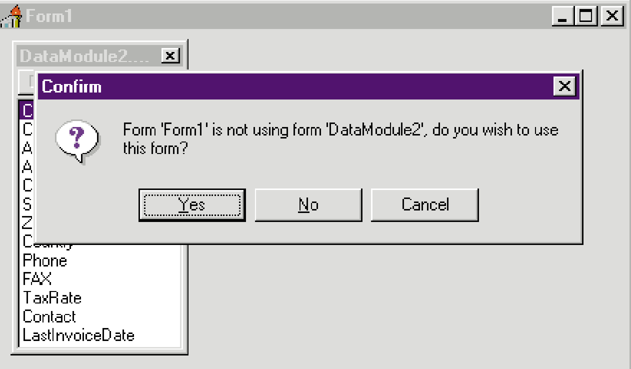


Figure 5 (Top): This DBEdit component and associated Label component were created on this form by dragging the CustNo field from the Fields Editor and dropping it onto the form. **Figure 6 (Bottom):** Delphi displays this dialog box the first time you drag-and-drop a field from a Fields Editor onto a form that is not yet using the field's associated data module.

This drag-and-drop technique can greatly increase your productivity when designing forms, because you no longer need to place each DBEdit and Label separately for each field, then assign the appropriate properties to make them active. Although you could use the Form Expert (the Database Form Expert in Delphi 1), the drag-and-drop technique provides greater flexibility. For example, instead of dragging and dropping directly onto a form, you could drop your new fields into a DBCtrlGrid, a multi-record object (MRO) component that ships with the Delphi 2 Developer and Client/Server editions.

You can also drag-and-drop multiple fields using standard Windows techniques. To select more than one field, hold down Shift to select sequential fields, or Ctrl to select any combination of fields, while you select fields in the Fields Editor. The selected fields will be highlighted (see Figure 7). You can then drag them as a group by grabbing any part of the highlighted selection. When you perform the drop, all of the selected fields are placed on the form (see Figure 8).

Creating Lookup Fields

In addition to instantiating the fields of a DataSet, the Fields Editor permits you to define two new types of fields: calculated and lookup. Delphi 1 also permits you to define a calculated field, and has already been described in this column (see *Delphi Informant*, July, 1995).

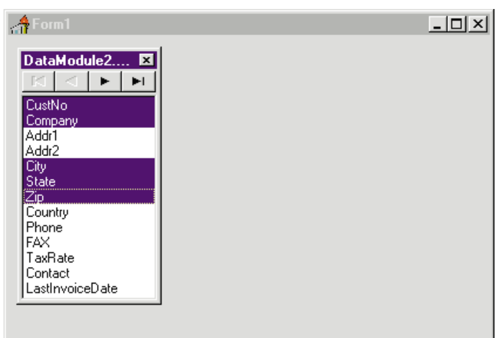


Figure 7: Select more than one field in the Fields Editor by Shift-clicking or Ctrl-clicking.

Figure 8: If you select more than one field in the Fields Editor, the selected fields can be dropped onto a form in a single drag-and-drop operation.

The lookup field, however, is new with Delphi 2. A lookup field is a field in one DataSet that displays data from another DataSet, based on an association between the two. For example, the ORDERS.DB table in the DBDEMOS directory has a field named CustNo that contains the customer number the order is associated with. The table does not hold the customer name — this information is stored in the CUSTOMER.DB table (specifically, in the Company field of the CUSTOMER.DB table). By creating a lookup field for the ORDERS table, you can define a new field that displays the customer's name based on the customer number. The lookup field performs the task of "looking up" the customer's name in the CUSTOMER.DB table and displaying it.

This technique is demonstrated by creating a customer name lookup field for the ORDERS.DB table. The first step is to create a data module that contains two DataSets, one for the table we will create the lookup field in, and one for the lookup table (see Figure 9). It contains one DataSource and two DataSets. The *DataSet* property of the DataSource is set to *Table1*. The *DatabaseName* properties of both *Table1* and *Table2* are set to `DBDEMOS`. The *TableName* property of *Table1* is set to `ORDERS.DB`, and the *TableName* property of *Table2* is set to `CUSTOMER.DB`. The *Active* property of both of these *Tables* has been set to *True*.

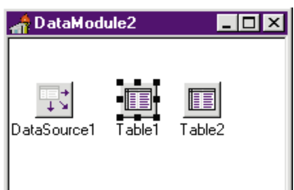


Figure 9: A data module for the example form.

The next step involves the Fields Editor. Here all fields are instantiated, as described earlier in this article. (Note that it is not necessary to instantiate all, or even any, of the fields in a DataSet to create a lookup field.) We can now create the lookup field. Right-click the list box in the Fields Editor and select **New**. Delphi responds by displaying the New Field dialog box (see Figure 10).

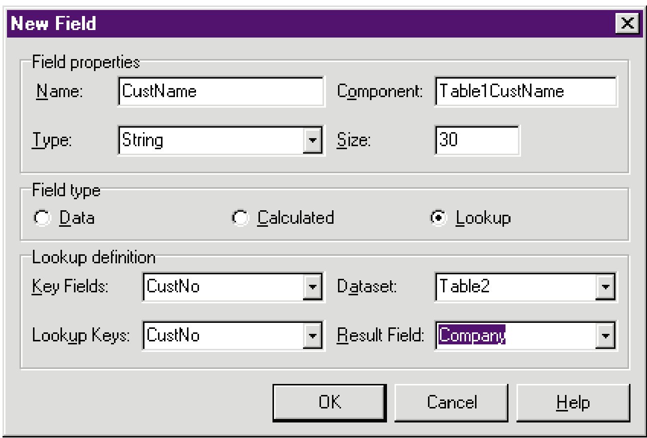


Figure 10: Use the New Field dialog box to create **Calculated** and **Lookup** fields. The dialog box in this image is being used to define a lookup field.

The New Field Dialog Box

At **Name**, enter the name of the field you are creating. For this example we will use the name `CustName`. As you type the field name, the name of the *TField* component that will be created is entered for you in the **Component** box. You can change the name of the *TField* component, but this is rarely necessary.

In the **Type** field enter or select the type of field. In this example, customer name is a string field, so we enter `String` here. You then use the **Size** field to define the size of the field. This is only necessary with *String*, *Bytes*, and *VarBytes* fields. In this case we will enter `30`, which is the size of the **Company** field in the CUSTOMER.DB table.

You use the radio buttons in the **Field type** group located in the center of this dialog box to define the type of field you are creating. Select **Lookup**. Doing so enables the fields in the **Lookup definition** group of the New Fields dialog box.

You use the fields in the **Lookup definition** group of this dialog box to identify the association between the current table and the lookup table. Begin by setting **Key Fields** to the field in the current table that has a counterpart in the lookup table. The value of this field is used to perform the lookup. In this example, select the ORDERS.DB field named CustNo, the identifier for the customer.

While the **Key Fields** drop-down menu only includes single fields, it is possible to define a lookup based on more than one field. To do so, you must manually enter the fields that associate the two DataSets, separating the field names with semicolons.

Next, set **DataSet** to the DataSet that points to the lookup table. Here we'll select *Table2*.

Once you have defined the DataSet, the **Lookup Keys** and **Result Field** edits become enabled. For **Lookup Keys** select the field in the lookup table that corresponds to the field in the **Key Fields** list. If you have entered two or more fields into **Key Fields**, you must enter the corresponding fields in **Lookup Keys**. In this example, select the CUSTOMER.DB table field named **CustNo**. In this example, both the **Key Fields** and **Lookup Keys** have the same name, but this is a coincidence and won't necessarily be true in all cases.

Finally, define which field in the lookup table will supply the data for the lookup field you are defining. Select the lookup table field from the **Result Field** drop-down menu. In this case, select the CUSTOMER.DB table field named **Company**.

This completes the lookup field definition. Save it by selecting the **OK** button on the New Fields dialog box.

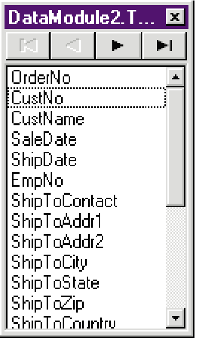


Figure 11: The new lookup field, **CustName**, has been dragged to a new position in the Fields Editor list box.

Once you have saved your lookup field definition, the new field name will appear in the Fields Editor. By default, new fields are placed at the bottom of the list of fields. You can move this field to a new position in the Fields Editor list box by dragging it. This is desirable because if you associate a DBGrid with a DataSet, and you have instantiated fields for that DataSet, the order of the fields in the DBGrid is based on the order of the fields in the Fields Editor list box. Figure 11 shows this new field, **CustName**, dragged to the third position in the Fields Editor.

Figure 12 shows a DBGrid that has been placed on a form, and whose *DataSet* property has been set to *DataSource1* on the data module. This required the data module's unit first be added to the form unit's **uses** clause. (This is easily accomplished by selecting **File | Use Unit** from Delphi's main menu while on the form.) Notice that the company name associated with the customer number appears in every record.

Lookup Field Example

OrderNo	CustNo	CustName	SaleDate	ShipDate
1003	1351	Sight Diver	4/12/88	5/3/88 12:0
1004	2156	Davy Jones' Locker	4/17/88	4/18/88
1005	1356	Tom Sawyer Diving Centre	4/20/88	1/21/88 12:
1006	1380	Blue Jack Aqua Center	11/6/94	11/7/88 12:
1007	1384	VIP Divers Club	5/1/88	5/2/88
1008	1510	Ocean Paradise	5/3/88	5/4/88
1009	1513	Fantastique Aquatica	5/11/88	5/12/88
1010	1551	Marmot Divers Club	5/11/88	5/12/88
1011	1560	The Depth Charge	5/18/88	5/19/88
1012	1563	Blue Sports	5/19/88	5/20/88
1013	1624	Makai SCUBA Club	5/25/88	5/26/88

Figure 12: A DBGrid containing a lookup field.

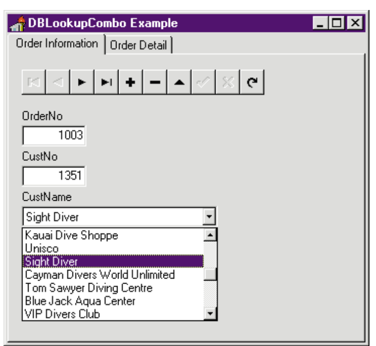


Figure 13: Dropping a lookup field onto a form produces a DBLookupComboBox. This form demonstrates using this combo box to select the customer for a new order. Once the customer name has been selected, the customer's number will automatically appear in the **CustNo** field. This requires that you set the *ReadOnly* property on the DBLookupComboBox to *True*.

Dragging Lookup Fields

There is one final interesting feature of the Fields Editor. When you drag a lookup field from the Fields Editor onto a form, Delphi does not just place a DBEdit and Label — it instead places a DBLookupComboBox. The form shown in Figure 13 was created by dragging the OrderNo, CustNo, and CustName fields from the Fields Editor onto the form.

A DBLookupComboBox permits the user to drop-down a menu of all possible lookup values in the lookup table. When added to a form by dropping a lookup field from the Fields Editor, the default DBLookupComboBox's *ReadOnly* property is set to *False*, permitting only a view of other customer names at run time. However, by setting this property to *True*, the user can easily change the customer associated with a particular order by selecting a new name from the combo box. Doing so automatically updates the CustNo field in the table.

Conclusion

Many new features await the database developer upgrading to Delphi 2. The new and improved Fields Editor is just one of them, but a welcome one indeed. ∆

The demonstration projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\MAY\DI9605CJ.

Cary Jensen is President of Jensen Data Systems, Inc., a Houston-based database development company. He is author of more than a dozen books, including the upcoming *Delphi In Depth* [Osborne/MacGraw-Hill, 1996]. He is also Contributing Editor of *Paradox Informant* and *Delphi Informant*, and this year's Chairperson of the Paradox Advisory Board for the upcoming Borland Developers Conference. You can reach Jensen Data Systems at (713) 359-3311, or through CompuServe at 76307,1533.

Delphi 1 / Object Pascal

By Andrew Wozniewicz

DLLs: Part III

Interfacing, Importing, and Exporting Functions

Over the last two months, we've introduced Delphi as an environment for developing dynamic link libraries. Now it's time for us to create a corresponding import unit for the custom DLLFirst library we've created. [To create the DLLFIRST.DLL, see Andrew Wozniewicz's articles "DLLs: Part I" and "DLLs: Part II" in the March and April issues of *Delphi Informant*.]

Creating the Import Unit

Let's develop a test application that will use the subroutines implemented in the DLLFirst unit. Follow these steps:

- Create a new, blank Delphi project and call it DLLTest (DLLTEST.DPR). Save the main form unit of the new project as FRMFIRST.PAS.
- Select **File | New Unit**. A new, minimal unit file is created by Delphi. This will be the **import** unit that enables your application to use the subroutines in DLLFIRST.DLL.
- Save the new unit file as FIRST.PAS.

Interfacing the DLL

The **interface** section of the library **import** unit appears identical to the **interface** section of any regular unit. It simply lists the subroutines available, which in this case, are the subroutines exported by the DLL.

The difference between an **import** unit and a regular unit lies in its implementation. Whereas the **implementation** section of a regular unit must provide the actual implementation code, the **implementation** section of a library **import** unit delegates the implementation's details to the external library. It merely binds the subroutine headings declared in the interface to the implementation provided elsewhere.

Remember that we already have implemented the string-handling subroutines inside the DLLFirst library. There is no point repeating that code in the library **import** unit. The only missing link is the ability to tell the using application which subroutine listed in the unit's **interface** corresponds to the appropriate entry points of a DLL.

To delegate the details of the implementation of a subroutine to an external module, and to allow the program to compile without actually "seeing" the subroutine's implementation directly, the **external** directive is used. This replaces the implementation of a subroutine, indicating that it's given outside the current project. The **external** directive is placed after the **implementation** heading of a subroutine. For example:

```
procedure Clear; external 'CUSTOM';

function StripStr; external 'DLLFIRST';
```

The **external** directive completely replaces the implementation of a subroutine. Instead of the familiar **begin..end** keywords and the implementation code inside the block they delimit, the **external** directive indicates that the implementation code is compiled and deployed separately, outside the current project.

Several possibilities can follow the **external** keyword. The simplest of these calls for the library's file name, – `DLLFIRST` – in this case, to be included after the directive. This enables the compiler to bind the subroutine declared inside the **import** unit to its implementation that resides in a DLL by the declared name. When giving the name of the external library in this way, a file name

extension of .DLL is assumed (the library on disk must actually have the .DLL extension). It's possible to have DLLs with other extensions, but Windows can automatically load libraries only with the default extension of .DLL.

When we run the program using this type of binding, after we make a call to the DLL, Windows attempts to resolve it by using the subroutine name declared in the **import** unit and searching for the same name inside the indicated DLL. For example, this statement:

```
function StripStr; external 'DLLFIRST';
```

indicates that the implementation for the *StripStr* function resides in the external library DLLFIRST.DLL. Windows automatically loads DLLFIRST.DLL and searches for a subroutine *StripStr* that matches the requested function by name to resolve the call.

We'll also discuss other ways of binding subroutines declared on the application (.EXE) side to their DLL-based implementations. Right now, let's continue with the DLLFirst example, using the simplest possible binding: binding by the declared name.

Interfacing DLLFIRST

Based on the comments provided in our previous articles, you're ready to provide the **interface** unit for the DLLFirst library. The good news is that the declarative part of the unit (the subroutine headings that you need to list there) are nearly identical to those we entered within the interface of the XString unit. Figure 1 implements a library **import** unit for the DLLFirst library we implemented previously.

The **interface** section provides declarations of the subroutines imported from the external library. Note that the function headings are (and in fact, must be) identical to those in XSTRING.PAS, with the exception of the **export** directive, which is missing in the **import** unit. The **import** unit does not implement these functions, but instead provides a way of importing their implementations from an external DLL. Therefore, an **export** directive is unnecessary here.

The **implementation** section — instead of providing the implementations of the functions listed in the unit's interface — provides their external bindings that enable Windows to find the appropriate implementation at run time. In addition, we can see the implementation headings for each of the imported subroutines, followed by the **external** directives indicating the library from where the functions will be imported.

Note that you have a great deal of flexibility in how you can specify the library's name. In Figure 1, the name of the library is provided as a string constant so that the constant name is repeated with the **external** directives, and the constant itself is defined. This approach makes it easy to change the name of the external library because the constant declaration is the only place where you would need to change it before recompiling the **import** unit.

```
unit First;
{ (Implicit) Library Import Unit for DLLFIRST.DLL.
  This unit is to be listed in the uses clause
  of applications wishing to use the services
  of DLLFIRST.DLL. }

interface
  function FillStr(C : Char; N : Byte): string;
  function UpCaseFirstStr(const s: string): string;
  function LTrimStr(const S: string): string;
  function RTrimStr(const S: string): string;
  function StripStr(const s: string): string;

implementation

const
  LibName = 'DLLFIRST';

  function FillStr;         external LibName;
  function UpCaseFirstStr;  external LibName;
  function LTrimStr;        external LibName;
  function RTrimStr;        external LibName;
  function StripStr;        external LibName;

end.
```

Figure 1: FIRST.PAS is an import unit for the DLLFirst library.

Remember that recompiling the **import** unit does not cause the DLL itself to be recompiled. The two entities are totally separate. In fact, it's possible to create discrepancies between the two entities. For example, you can declare the same subroutine with a different number of parameters on either side. This leads to potentially serious bugs that can be difficult to find. So be forewarned: Always double-check that the subroutine declarations inside the **import** unit correspond exactly to those declared in the DLL.

The **import** unit in Figure 1 is a typical example of a simple **import** unit, using the implicit binding. Implicit here means that we don't have to do anything special for the calls to any of the listed subroutines to be resolved automatically at run time — beyond declaring them with the **external** directives as shown. Windows handles the details of when and how to load the external library and how to obtain the actual run-time addresses of the subroutines you are calling.

We've already taken the first step toward creating the unit in Figure 1. Namely, we've created the FIRST.PAS **import** unit, whose function is to import implicitly the subroutines from the DLLFirst module. Complete the library **import** unit, FIRST.PAS, by entering the code as shown in Figure 1. Make sure that you save the unit file before proceeding.

Creating the Example Program

We're now ready to try the external subroutines we've created and imported via the FIRST.PAS import unit. To visualize the results returned by the various external functions, let's create a simple form-based Delphi application. Follow these steps:

1) Select the *FrmFirst* unit and bring the Form Designer window to the foreground.

2) Inside the Form Designer, place two copies of the Edit component. These are found on the Standard page of the Component Palette. Place *Edit1* and *Edit2* (as they are named by default) one below the other. Clear their *Text* property so that they appear blank on the form.
3) Change the *ReadOnly* property of the second edit box, *Edit2*, to *True*. Change its *Font.Color* property to *clRed*.
4) From the Standard page, select a Label component and place it on the form between the two Edit components. By default, the Label's name is *Label1*. Change the Label's *AutoSize* property to *False*, and extend the enclosing box so that it accommodates the names of the external subroutines. Clear the Label's *Caption* property so that it initially appears blank.
5) Place several Button components on the form, one for each of these four functions: *UpCaseFirstStr*, *LTrimStr*, *RTrimStr*, and *StripStr*. Rename the buttons (named by default *Button1*, *Button2*, etc.) to *ButtonUpCase*, *ButtonLTrim*, *ButtonRTrim*, and *ButtonStrip*, respectively. Change the *Caption* property of each button to reflect the functions they represent: `UpCaseFirst`, `LTrim`, and so on. Figure 2 shows the completed main form of the example program.

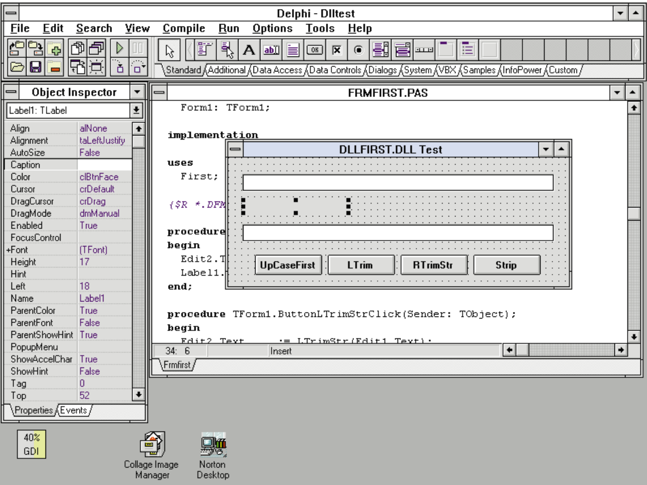


Figure 2: The DLLTest program's main form.

6) Modify the *ButtonUpCaseFirst* button's *OnClick* method thus:

```
procedure TForm1.ButtonUpCaseFirstClick(Sender:
Tobject);
begin
  Edit2.Text     := UpCaseFirstStr(Edit1.Text);
  Label1.Caption := (Sender as TButton).Caption;
end;
```

The first of the two statements exercises one of the subroutines we've implemented inside the external library: *UpCaseFirstStr*. After the user clicks the **UpCaseFirst** button, the contents of the *Edit1* edit box are converted via a call to the external *UpCaseFirstStr* function and the result is displayed inside the *Edit2* box.

The second of the two statements inside the event handler updates the contents of the *Label1* component so that its *Caption* property reflects the function most recently applied. The Label's caption is set to the sending button's caption. The button's label provides a visual clue to the user.

Note how the type of the argument to the *TForm1.ButtonUpCaseFirstClick* method is assumed to be a *TButton*. The **as** type conversion is used on the

```
unit FrmFirst;

interface

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

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    ButtonUpCaseFirst: TButton;
    ButtonLTrim: TButton;
    ButtonRTrim: TButton;
    ButtonStrip: TButton;
    procedure ButtonUpCaseFirstClick(Sender: TObject);
    procedure ButtonLTrimClick(Sender: TObject);
    procedure ButtonRTrimClick(Sender: TObject);
    procedure ButtonStripClick(Sender: TObject);
  private
  { Private declarations }
  public
  { Public declarations }
end;

var
  Form1: TForm1;

implementation


uses
  First;

{$R *.DFM}

procedure TForm1.ButtonUpCaseFirstClick(Sender: TObject);
begin
  Edit2.Text     := UpCaseFirstStr(Edit1.Text);
  Label1.Caption := (Sender as TButton).Caption;
end;

procedure TForm1.ButtonLTrimClick(Sender: TObject);
begin
  Edit2.Text     := LTrimStr(Edit1.Text);
  Label1.Caption := (Sender as TButton).Caption;
end;

procedure TForm1.ButtonRTrimClick(Sender: TObject);
begin
  Edit2.Text     := RTrimStr(Edit1.Text);
  Label1.Caption := (Sender as TButton).Caption;
end;

procedure TForm1.ButtonStripClick(Sender: TObject);
begin
  Edit2.Text     := StripStr(Edit1.Text);
  Label1.Caption := (Sender as TButton).Caption;
end;

end.
```

Figure 3: The completed form unit of the example DLLTest program.

generic *TObject*-typed *Sender* argument to obtain the desired caption.

Complete the form unit by providing event handlers for the *OnClick* event of each remaining button. The resulting unit is shown in Figure 3. Be careful to copy everything shown in Figure 3 inside your form unit, including the following **uses** clause in the **implementation** section:

```
uses
  First;
```

This is where we are making the imported, external subroutines visible to the code inside the `FrmFirst` unit. Including `First` in the **uses** clause makes it possible to call the external functions that are declared there. Without having the unit `First` listed in the **uses** clause, attempting to call any of the imported subroutines would result in a compile-time error: "Unknown identifier."

Compile and run the example program. The DLLTest example should appear as shown in Figure 4 and should enable us to enter a string value inside the top edit box (the *Edit1* component). After entering a text string inside the first edit box, press one of the function buttons. The result of applying the selected external function appears within the second edit box.

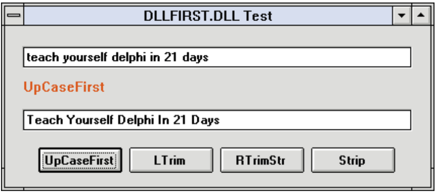


Figure 4: The DLLTest program.

The fact that we are calling external functions each time a button is pressed is transparent to the user. The functions just appear to work, regardless of where they were implemented. You call them in exactly the same way you would call functions implemented in a unit that is statically linked to the program.

Exporting by Ordinal

Consider a variation on the way in which we exported the string-handling functions from the DLLFirst library. Recall that the **exports** clause of the DLLFirst library module:

```
exports
  FillStr,
  UpCaseFirstStr,
  LTrimStr,
  RTrimStr,
  StripStr;
```

In this case, the functions were exported by simply declaring their names in Object Pascal code. However, there is another way of exporting subroutines from DLLs: by ordinal number. Here's an Object Pascal example with two custom functions:

```
exports
  LTrimStr index 13,
  RTrimStr index 14;
```

To export a subroutine by an ordinal number, we must list it inside the **exports** clause of the library module with the additional **index** directive specifying the ordinal number by which the subroutine will be known. The ordinal number we specify after the **index** keyword must be a true constant expression evaluating to a 16-bit (Word) value at compile time.

Remember that exporting a subroutine by ordinal number is always in addition to exporting it by name. The client application will have the capability to bind dynamically to the exported subroutine by ordinal number, in addition to being able to bind to it simply by the declared name, as before.

We now can change the **exports** clause of the DLLFirst library to the following:

```
exports
  FillStr        index 11,
  UpCaseFirstStr index 12,
  LTrimStr       index 13,
  RTrimStr       index 14,
  StripStr       index 15;
```

By doing so, we have introduced an explicit ordinal number by which each of the routines may be identified by Windows at run time in addition to being identified by their names.

Importing by Ordinal

The client application can import the subroutines by their declared names, even if they were explicitly exported by an ordinal number. If it's more convenient, however, the client can also import the subroutines by their ordinal numbers.

Importing subroutines by an ordinal results in programs running slightly faster and it's more efficient than importing by name. In the latter case, Windows must exert extra effort to search for the subroutine's name and to convert it into an actual callable address.

However, importing by ordinal runs a potentially greater risk of making an incorrect connection to the DLL. Importing by name is less error-prone.

To import a subroutine by a specific ordinal number rather than by a name, place the **index** directive — followed by a number, after the **external** part of the subroutine external binding. The number after the **index** directive may be a decimal or hexadecimal constant, or a constant expression using symbolic names. For example:

```
procedure Clear; external 'CUSTOM' index $11;
function StripStr; external 'DLLFIRST' index 47;
```

As an example of how to import subroutines by ordinal numbers rather than by their names, consider the modified FIRST.PAS library import unit shown in Figure 5. It uses the "import by ordinal" feature to rename the subroutines being imported.

```
unit First;

{ Modified Library Import Unit for DLLFIRST.DLL.
  This unit is to be listed in the uses clauses
  of applications wishing to use the services
  of DLLFIRST.DLL. }
interface
   function XFillString(C : Char; N : Byte): string;
   function XTitleString(const s: string): string;
   function XLeftTrimString(const S: string): string;
   function XRightTrimString(const S: string): string;
   function XStripString(const s: string): string;

implementation

const
  LibName = 'DLLFIRST';

   function XFillString;      external LibName index 11;
   function XTitleString;     external LibName index 12;
   function XLeftTrimString;  external LibName index 13;
   function XRightTrimString; external LibName index 14;
   function XStripString;     external LibName index 15;

end.
```

Figure 5: A modified library import unit for the example DLLFIRST.DLL using the "import by ordinal" feature.

This library **import** unit is functionally equivalent to the **import** unit shown in Figure 3. The difference here is that this modified **import** unit renames the imported functions so that they are known to the using application under different names instead of their declared names inside the DLL that implements them.

This trick is possible thanks to the "import by ordinal" feature of Windows. This enables us to bind a subroutine prototype inside an **import** unit to an arbitrary external subroutine. The external declarations assume that the subroutines were exported by the specified ordinal numbers from the DLL.

Note that the changes to the **import** unit do not affect the signatures of the imported functions: their parameter lists and return types remain identical to those in Figure 3.

This is understandable once you realize that we are importing the same subroutines as before, implemented inside the DLLFirst library. The signatures of these subroutines did not change just because we chose to import them in a different way.

ortin i erent e

The flexibility of choosing a different name for an imported subroutine does not depend on it being exported by an ordinal number. It's still possible to import a subroutine by a name different than the export name specified within the DLL.

Correspondingly, it is possible to export a subroutine from a DLL under a different name than the name declared in the source code. Both these feats are accomplished with the **name** directive.

Here's an example:

```
exports
  LTrimStr name 'TrimStringOnLeft',
  RTrimStr name 'TrimStringOnRight';
```

The **name** directive is used much like the **index** directive inside the **exports** clause to make the exported subroutines known externally under their assumed names. The **name** directive also can be used to import subroutines under different names than those under which they were exported. For example:

```
procedure Clear; external 'CUSTOM' name 'CustomDLLClear';
function StripStr; external 'DLLFIRST' name 'XStripStr';
```

The **name** directive therefore can be used on both the exporting and importing side to resolve naming conflicts, or simply to change the name of the subroutine being bound dynamically for convenience reasons.

So far, we've learned the techniques involving **import** units where the subroutines to be imported are identified with the **external** directive. **Import** units enable the subroutines in DLLs to be imported implicitly; the process of loading the library and resolving the address of the subroutine at run time is automatic and transparent to the programmer using the DLL. It is the responsibility of Windows.

The implicitly loaded DLLs usually are loaded by Windows at the time the application starts. If any of the DLLs imported implicitly are missing or simply cannot be found, the application fails to load. The application never gets a chance to correct the problem by prompting the user to specify the location of the required files and changing to the indicated directory, for example. Windows prevents the application from loading before it can respond to user events.

Another serious drawback to using an implicit import is that the name of the DLL has to be hard coded, or fully known at compile time. As you may recall from the earlier sections, the name of the implicitly imported DLL must be a string constant.

Fortunately, there is a way to gain an even greater control over the process of loading DLLs. Instead of importing the required subroutines implicitly via an external statement, you can take the responsibility for loading the DLL into memory and retrieving the addresses of the subroutines inside that DLL explicitly. This way, you can defer loading the library until you can determine at run time what the name of the library is and where it is located.

Next month, we'll conclude our series on creating DLLs with Delphi. ∆

This article was adapted from material for *Teach Yourself Delphi in 21 Days* [SAMS, 1995], by Andrew Wozniewicz and Namir Shammas.

The source files referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\MAY\DI9605AW.

Andrew J. Wozniewicz is president and founder of Optimax Development Corporation (http://www.webcom.com/~optimax), a Chicago-based consultancy specializing in Delphi and Windows custom application development, object-oriented analysis, and design. He has been a consultant since 1987, developing primarily in Pascal, C, and C++. A speaker at international conferences, and an early and vocal advocate of component-based development, he has contributed articles to major computer industry publications. Andrew can be contacted on CompuServe at 75020,3617 and on the Internet at optimax@optidevl.com.

In Development

Software Development Techniques

By Craig L. Jones

PQA: Part III

Practical Quality Assurance Techniques for Delphi

As every programmer knows, much can go wrong with even the simplest of programs. Worse yet, the number of possible problems grows geometrically with the size of an application. The good news is that just a little quality assurance (QA) savvy can go a long way towards heading off disaster. As with any discipline, the fastest route to success lies in acquiring the proper tools of the trade and learning how to use them correctly.

This is the third installment of a three-part series on assuring the quality of Delphi programming projects. The series is primarily directed towards Delphi programmers and assumes no prior knowledge on the subject of QA. It is hoped that programmers who previously gave quality assurance little thought will discover some enthusiasm for applying the techniques presented here.

The emphasis of this series has been on exploring practical techniques for quality assurance. In the previous installments, some general QA theory was presented along with how it applies to the different stages of program development. Eight types of testing were outlined: requirements verification, design validation, unit testing, integration testing, user acceptance testing, alpha testing, beta testing, and gamma testing. And a tool kit was established for performing unit testing, probably the most important of the eight types of testing.

Testing Tools

In this installment we'll go on a whirlwind tour of several other types of testing tools that are available, and some techniques for using them effectively. Some of these tools are commercially available, while others need to be developed specifically for the project at hand. In all cases, these are tools no serious software developer should be without.

(Please note that this article only describes these tools in terms of general concepts, and does not endeavor to name any particular products or vendors. An effort was made to seek out commercial products with Delphi-specific features worthy of mention. However, this author was not made aware of any such tools in release as of the time of this writing. Perhaps the upcoming Software Testing Analysis and Review trade show, "STAR '96," will feature Delphi-specific products. In any case, the May 15th event in Orlando, FL, will certainly provide ample opportunity to explore the offerings of all of the major players in the software quality assurance industry. Call (800) 423-8378 to inquire about the US$30 exhibits-only admission, or to check on the possibility of obtaining an exhibitors guide without attending.)

User Action Automation Tools

User action automation tools, the most common type of commercially available testing tools, work by watching the events that occur as a tester runs through the execution of a program being tested. The key stroke events, mouse events, and Windows messages are recorded in a script that can be played back at high speed to repeat the test at will. Hundreds

of separate scripts might be recorded to test different aspects of the program in different ways, and then played back sequentially (unattended).

Screen snapshots are taken along the way which can be compared to the snapshots taken during previous runs. At the end of a new run, any differences in the screen snapshots are presented to the tester who can take one of three actions: 1) flag the difference as representing a bug that was introduced, 2) accept the difference as a positive change, or 3) ignore the difference as inconsequential.

Usually, the scripts that are created can be edited, either to change the key stroke, mouse, or message actions that occur, or to change when the snapshots are taken. Such editing can be tricky, so it's often easier to just re-record the script in question. On the other hand, some changes are readily invoked. For example, if the program being tested is changed so that the "Find" function is moved from the "Edit" menu to its own "Search" menu, then it could be a simple matter to change any tests that use Alt E (being the hot key for the **Edit** menu) to use Alt S instead, where appropriate.

The scripting languages interpreted by these testing tools can be quite robust, providing flow control (looping and branching), for example. Various timing factors can be customized, making it easy to slow down playback and pinpoint trouble spots. Some of the scripting languages are modeled after complete, standard programming languages such as Basic, and are even user-extensible.

The variety of comparisons that can be achieved by these tools can be sophisticated as well. Some tools can only compare window images as bitmaps. Others are able to apply optical character recognition (OCR) scanning to the bitmaps and thus compare the text found within. Yet other tools are able to hook into the application development environment via specialized application programming interface (API) kits and deal directly with the underlying objects and properties.

One of the pitfalls with this type of testing tool is that it is almost too easy to use. It's tempting to jump right in, creating scripts left and right, without planning. The result is that some functions are tested 10, 20, or even 100 times, while other functions are neglected. Without proper direction, user-interface (UI) intensive functions tend to monopolize a tester's time, while more important functions that aren't as UI-intensive are left until the end, and then rushed (if they are tested at all).

Load Testing

Once a suite of tests are developed and the application is made to pass those tests one-by-one, the next challenge is to run the tests in a multi-user environment. This is where "load testing" comes in. Here the object is to identify not only glaring problems such as table and record locking conflicts, but also to measure performance under varying loads.

Load testing can be achieved to some degree by simply setting up the tests in your test suite to run on multiple workstations in a parallel, offset, and/or random manner. Some testing tools provide specific support for load testing by coordinating the execution of tests on the various workstations, allowing the test sequence timing to be programmed for specific possible conflicts.

Data Generators & Sanitizers

A comprehensive test plan will eventually require that the system being tested is populated with a substantial set of data. A certain small amount of data might be provided by virtue of testing the data entry functions, but at some point in the testing cycle — especially during load testing — large amounts of data will be required. Entering such data by hand will likely be cost- and time-prohibitive. Thus, developers often turn to one or more data generator utilities.

Commercial data generators do exist for the more common types of data (company names and addresses, inventory parts, invoices, etc.), but it is usually necessary to develop such utilities in-house. If the project under development is a system rewrite, then perhaps one or more databases already exist that can be tapped.

If the data is at all sensitive, then a "sanitizer" should be written to scramble the identities. This obviously pertains to medical and legal records, but it also applies to business information that might hint at ways for a competitor of the data's owner to gain an advantage.

If an available existing database proves to be too large and unwieldy, then yet another tool, a purge mechanism, might be crafted and applied. Such a mechanism might be based on a pivotal data structure, such as the customer list, and then radiate out to touch each of the related data structures (invoices, products, vendors, etc.). To prime the process, first reduce the customer table to a handful of customers and then let your purge mechanism delete all of the extraneous data having nothing to do with those customers, directly or indirectly. With a little planning, portions of the mechanism might be used in the application itself as a year-end cleanup feature.

The use of standardized data generators and sanitizers should be encouraged at the earliest possible juncture in the project development cycle. Left on their on, programmers can come up with some of the strangest and silliest sample data that is sometimes downright embarrassing. Furthermore, it is amazing how easily such inappropriate data can find its way into specification documents, user manuals, help screens, marketing literature, demo disks, and other materials.

Defect Tracking

Defect (bug) tracking software merely provides facilities for keeping track of the problems discovered during test-

Category	Severity Level	Description
Specification compliance	3	Vital problem; program is unusable
	2	Major problem; fails to meet major requirement
	1	Minor problem
	0	Cosmetic problem
Crash level	3	System crashes and data corruption is likely
	2	System crashes, but data corruption is unlikely
	1	System continues after error/warning message
	0	System continues normally
Availability of a work-around	3	No work-around
	2	Work-around is difficult/tedious/error-prone
	1	Work-around is simple
	0	Work-around is unnecessary

Figure 1: Three ways a problem report might be ranked by severity. By combining such factors in a weighted average, resources for solving the problems can be allocated efficiently.

ing. Defect tracking software can be as simple as a home-grown, single table database, or it can be in the form of a comprehensive commercial package. The primary purpose of a bug tracking system, of course, is to make sure that the nastiest bugs are eliminated and that no bugs, no matter the severity, are forgotten.

It is unrealistic to expect every bug in any given project to be squashed completely. As Stella, Lady Reading, is attributed with saying, "The whole point of getting things done is knowing what to leave undone." Therefore, bug trackers allow for the problem reports to be graded according to their severity, usually in multiple categories such as the likelihood of occurrence and how difficult it is for the user to work around the problem (see Figure 1).

Bug trackers can provide some interesting metrics that are highly useful in managing the project. Severity factors, turn-around times, reoccurrence frequencies and the like, can all be combined to provide valuable projections. The project manager can then examine possible alternate scenarios. For example, what if instead of having to allocate 25 hours of an analyst's time to the complete redesign of a module, a way could be found to work within the existing design and settle for only reducing the problem from a level 3 severity to a level 1 severity?

Version Control Systems

This is not a quality assurance topic per se, but the subject of version control systems (VCS) goes hand-in-hand with quality assurance under the heading of software configuration management (SCM). Briefly, a VCS maintains libraries of source code and related files, keeping track of every change that is made to every file throughout the development cycle. If ever a question arises about how a file, or set of files, appeared before they were changed, the VCS can be directed to disgorge the files in that previous form.

Using a good version control system to keep track of software revisions has numerous benefits, only one of which is making it easier to track down the cause of an elusive bug. Several days or even weeks may pass between the time that a defect is first noted and resources can be allocated to attend to it. In the meantime, the source code in question could have been modified a dozen times, making it impossible to recreate the defect, much less fix it. The solution is to have your version control system provide you with a copy of the software as it was at the time the defect was first discovered. Once the error is discerned within the old source code, it becomes an easy matter to assure that the newest version of the code is free of the error.

The Build Process

A second QA issue under the SCM heading is that of assuring consistency with your build process. "Build process" refers to whatever step-by-step procedures must be followed in order to produce a deployable product. At the root of the process are the particular steps necessary to compile the software into the deliverable EXE and DLL files, to prepare any associated data files, and then to make them ready for installation.

Such a build process will have to be followed every time a test candidate is needed. More than anything else having to do with QA, the watchword here is "consistency," for no matter how well the software is debugged, all is for naught if the delivery mechanism is unreliable. This applies not only to the build process itself, but also to numerous ancillary aspects.

Figure 2 contains a rough checklist to serve as a basis for the build process of any software project. It is divided into six sections. The first two sections are to be completed by the developers before the build takes place. The latter four sections are completed by personnel taking on the roles of SCM and SQA as appropriate.

The following is a summary of all six sections:

1) **Complete any pending changes.** These are some simple measures that developers can take to avoid introducing new problems.
2) **Check in all files that were checked out.** There is no point in performing a build if half-completed changes will interfere with successful operation.
3) **Plan the build.** Prepare to inform the users (beta testers) of the significant changes, and any special handling that will be required to install and use the new build.
4) **Make a build candidate.** This is the root of the build process.
5) **Verify the build.** Perform primary testing of the new build.
6) **Publish the build.** Distribute the new build for further testing or final release.

Coding Standards & Practices

Obviously, the best way to eliminate bugs from a project

A Software Build Process Checklist

Complete Any Pending Changes
- Search for place-holder comments that still need to be replaced with code.
- Update the file and function prologs and add other necessary comments.
- Verify that all new code meets the established coding standards.
- Check the spelling of all new user-interface text (prompts, error messages, etc.).
- Create or modify appropriate unit test drivers and run them.
- Remove extraneous debugging code.
- Turn off debugging compiler switches.

"Check in" Files
- "Check in" all modified files to the version control system (VCS).
- Compare files that might have been modified without having been checked out first and resolve conflicts.

Plan the Build
- Compile a list of significant changes (compare the new code base to the code base from the previous build and/or confer with VCS logs, checked off task lists, status reports, etc.).
- Determine if any changes will require special handling during this build (because of changes to data files, .INI files, etc.).
- Prepare a build memo informing users of the significant changes, standard or special installation notes, known problems with the new build, etc.

Make a Build Candidate
- Check out a fresh set of source code files.
- Update the build number (and/or other relevant numbers and/or dates: version, revision, release) as appropriate.
- Insure that the proper build environment is in place.
- Run the "Make" scripts, batch files, etc. to build all of the deliverables (executables, run-time libraries, etc.).

Verify the Build
- Run whatever build tests have been established (a non-stop sequence of unit test-drivers, an automated integration test, a manual testing checklist, etc.).
- Inspect the deliverables for missing files, wrong file names, unusually sized files, etc.

Publish the Build
- Transfer the new deliverables to the delivery medium (network volume, distribution diskettes, etc.) as appropriate.
- Finalize and distribute the build memo.
- Monitor at least one installation to check the build memo instructions.

Figure 2: A generic checklist for the process of building the program deliverable(s).

is to prevent them from being introduced in the first place. This is just one of the ways that establishing and enforcing a thorough set of coding standards can help. Standards help to avoid unreliable coding practices, to increase effective communications between team members, and to present a consistent user interface.

Some of the issues that should be covered by a comprehensive set of standards include:
- Programming conventions relevant to Pascal in general and Delphi in particular (e.g. when and how to use exception handling)
- Naming conventions for variables, constants, types, procedures, functions, objects, methods, files, directories and their corresponding aliases, database tables and their fields, compiler directive tokens, etc.
- Comment formatting and content (e.g. the layout of file and function prologs)
- User interface issues (e.g. using complete sentences in all error and warning messages)

The most difficult problem to overcome when developing a set of standards is the tendency to overdo it. If the standards document is unwieldy, it won't get used. So it may be necessary to compromise, to keep the standards simple enough that the programmers can remain mindful of them.

One way to enforce the chosen standards is via the use of a smart text editor with programmable macros. For example, just as the editor might automatically follow an **if** keyword with **then begin end;**, so could it be programmed to automatically follow a *GetMem* call with a corresponding *FreeMem* statement. In the case of the **if**-statement macro, the programmer is merely saved some typing time. In the latter case, however, the programmer is reminded of an all-important cleanup function that might otherwise be forgotten.

The Role of the Chief Engineer

When one thinks of the personnel assigned to any given software development effort, it is easy to visualize the traditional roles and titles. Top down, the positions go something like:
- Director
- Project Managers
- Team Leaders
- Analysts, Programmers, Testers, Writers, and (these days) Graphic Artists

No matter how many people are available to cover these roles, be it 30 or just one, they all have to be covered. There is one additional role, however, that ought to be taken on as well. This role is variously named; it is sometimes called Project Coordinator or Chief Engineer. By whatever title, this role is vital. Yet it is too often underplayed in large organizations and completely ignored in small ones.

The Chief Engineer is someone assigned to remain omniscient ("all seeing") in his or her approach to the development of the project, or projects, at hand. The Chief Engineer is charged with maintaining the "big picture," but unlike the managers and the team leaders, does so from the perspective of the analysts, programmers, and testers who are down in the trenches. In other words, this person needs to be someone who works with specific implementation issues on a daily basis.

Some of the tasks taken on by a Chief Engineer can include:

- overseeing the details of inter-project and inter-module communications/interfaces
- overseeing adherence to whatever global requirements have been put forth (e.g. user interface "look and feel" issues), which are too easily overlooked
- coordinating and/or performing code and design reviews
- continuously reexamining whatever coding standards are adopted by the organization to keep them simple and practical so developers will actually apply them
- troubleshooting any inter-team problems that may arise
- acquiring, building, or commissioning the QA and development tools needed for optimum productivity
- training the team members to properly apply the tools, techniques, standards, and practices
- otherwise attend to the productivity capability of the line workers

Conclusion

According to Dr Dennis Waitley, "If you fail to plan, then by default, you plan to fail."

Throughout this three-part series we've seen how using the proper tools, with proper planning, can make it relatively painless to successfully assure software quality. In Parts I and II we built a QA tool kit to accommodate the task of systematically developing unit test drivers that are machine-executable. This month we explored a number of other tools and techniques applicable to software quality assurance. ∆

Craig Jones is a contract software engineer in Southern California, with over 14 years of programming and consulting experience. He is the programming standards SIG leader for the Orange County Delphi Users Group. He is also a member of Team Borland, supporting Paradox and Delphi on the GEnie network. Mr Jones can be reached at craig.jones@genie.geis.com or on CompuServe at 71333,3515.

At Your Fingertips

Delphi 1 / Delphi 2 / Object Pascal

By *David Rippy*

If one advances confidently in the direction of his dreams, and endeavors to live the life which he has imagined, he will meet with a success unexpected in common hours.

— *Henry David Thoreau*

How can I tile a bitmap image on my form?

Here's a cool trick that you can implement on any Delphi form to give it a visual edge. With just a few lines of code, you can tile a bitmap of your choice onto the form's canvas as shown in Figure 1.

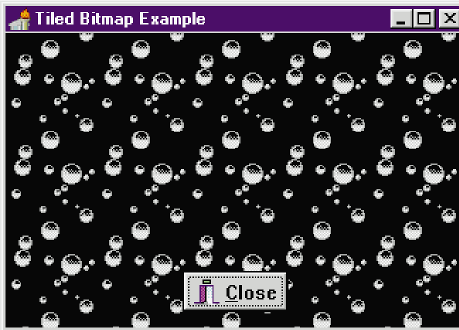


Figure 1: A sample form with a tiled bitmap in the background.

The code needed for this tip is located in two of the form's event handlers: *OnCreate* and *OnPaint* (see Figures 2 and 3). The code in the *OnCreate* event handler creates an instance of the bitmap (*myBitmap*) that will be tiled on the form. The *OnPaint* event handler tiles *myBitmap* onto the form starting from the top left corner and continuing to the lower right corner. The placement of the code is important. By placing the tiling code in the form's *OnPaint* event handler, the bitmaps are repainted appropriately if the form is moved or resized.

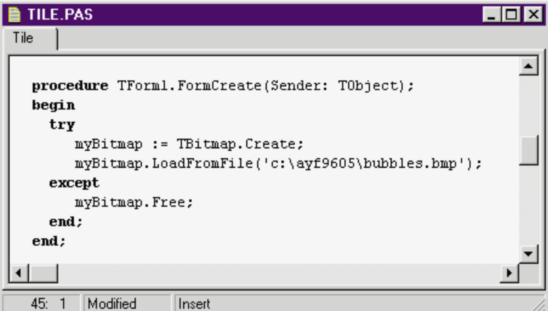


Figure 2: This code is found in the Form's *OnCreate* event handler.

Bitmaps of any size can be used, but you typically want to use something small (40x40 pixels or so). Also, it's important to select an image that tiles well. Appropriate selections will appear to be a single, large image when they are placed adjacent to one another. The nerd bitmap in Figure 4 is an example of an image that does not tile well — plus your clients may not appreciate the humor. — *D.R.*

How can I quickly make the same change to several objects programmatically?

Have you ever wanted to reference several objects on a form without hard-coding the name of every object? For example, the form shown in Figure 5 contains six CheckBox components and two buttons, labeled **Check All** and **Uncheck All**. The purpose of the **Check All** button is to place a check mark in every CheckBox on the form. To accomplish this, you might first be tempted to explicitly change the *Checked* property of each CheckBox individually:

```
CheckBox1.Checked := True;
CheckBox2.Checked := True;
...
```

This works, but if the number of *CheckBox* components you want to reference increased to 20 or 30, you would quickly find yourself writing a lot of code unnecessarily.

Here's a better way. Examine the *OnClick* method for the **Check All** button shown in Figure 6. Two properties are key: *Components* and *ComponentCount*. The *Components* property is an array that

TILE.PAS

```
procedure TForm1.FormPaint(Sender: TObject);
var
  xPos, yPos, bmpWidth, bmpHeight : LongInt;
begin
  with myBitmap do
  begin
     bmpWidth := Width;
     bmpHeight := Height;
  end;
  yPos := 0;
  while yPos < Height do
  begin
     xPos := 0;
     while xPos < Width do
        begin
           Canvas.Draw(xPos, yPos, myBitmap);
           xPos := xPos + bmpWidth;
        end;
    yPos := yPos + bmpHeight;
  end;
end;
```

40: 1 Modified Insert

Figure 3 (Top): This code is located in the Form's *OnPaint* event handler.

Figure 4 (Left): The nerd bitmap doesn't tile very well.

contains a list of all the components owned by an object. In our case, we want to know all the components owned by the form. The *ComponentCount* property tells us how many components are owned by the form. Once we have the number of components and their names, a **for** loop is used to iterate through the CheckBoxes, setting each *Checked* property to *True*.

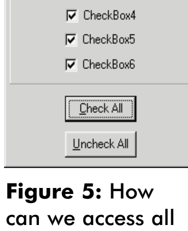


Figure 5: How can we access all six *CheckBoxes* without hard-coding their names?

You could use code such as this to perform just about any type of operation on a set of objects. For instance, you could change the style or color of each of the objects, perform math on their values, or alter their *Visible* property. There are lots of applications. Use your creativity! — *D.R.*

Comments on Comments

As a programmer you likely "comment out" large sections of code to help zero in on a bug. Since Object Pascal does not support nested comments, this can be difficult. There is a trick, however! Object Pascal supports two kinds of comments — "standard" Pascal comments and Turbo Pascal comments. For example, both of the following comments are valid:

```
(* This is a Pascal comment *)
{ This is a Turbo Pascal / Object Pascal comment }
```

control.pas

```
procedure TForm1.Button2Click(Sender: TObject);
var
  iCount : SmallInt;
begin
  for iCount := 0 to ComponentCount-1 do
     begin
        if Components[iCount] is TCheckBox then
           TCheckBox(Components[iCount]).Checked := False;
     end;
end;
```

51: 10 Insert

Figure 6: This code is found in the *OnClick* event handler of the **Check All** button.

Luckily, you're allowed to nest the different types of comments. This means the following code is a comment:

```
(* for x:= 0 to 100
     myArray[x] := 0; {initialize array to zero} *)
```

With Delphi 2, you also have the option of using C++ style comments. To comment out a single line of code, just preface the line with a double-slash (//). The compiler considers everything to the right of the two slashes to be a comment.

Warning: Don't use double-slashes if you're developing for 16-bit and 32-bit platforms. These comments are not supported by the Delphi 1 compiler. And they don't turn blue either. — *David Faulkner and Russ Acker*

Quick Tip: Delphi 2 Trim Functions

Delphi 2 includes three new functions (defined in the SysUtils unit) that allow you to trim white space with a minimum of effort: *Trim*, *LeftTrim*, and *RightTrim*. Each of these functions accepts a single string as a parameter, and returns the trimmed string.

Trim removes both leading and trailing "white space" and control characters from a string. *LeftTrim* and *RightTrim* remove white space and control characters from the beginning and end of a string, respectively.

Here's an example of the *Trim* function:

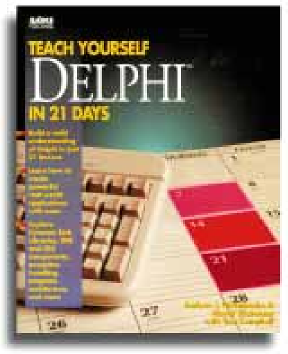
```
Edit3.Text := Trim(Edit1.Text) + ', ' + Trim(Edit2.Text)
```

— *Russ Acker, Ensemble Corporation* ∆

The demonstration projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\MAY\DI9605DR.

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

Kick the Habit and *Teach Yourself Delphi in 21 Days*



If you're a novice programmer, a bit rusty with Pascal, or are simply seeking to learn more about the exciting world of object-oriented Windows programming, then buy and read *Teach Yourself Delphi in 21 Days*. You can kick the habit of programming ignorance and join the ranks of knowledgeable Delphi enthusiasts.

Teach Yourself moves the reader — day by day — towards a fairly extensive understanding of the Delphi environment, the built-in components, and the Object Pascal programming language. Authors Andrew J. Wozniewicz and Namir Shammas focus on the fundamentals of Object Pascal, and through the insightful use of example programs, questions and answers, and end-of-chapter workshops, subtly teach object-oriented programming.

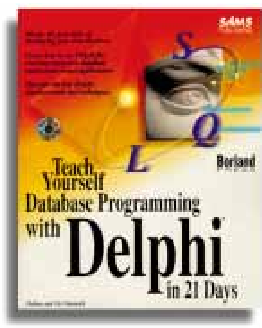
The first week introduces Windows programming basics, Delphi tools, and the IDE, laying a firm foundation for the Object Pascal language. After building the familiar "Hello World," readers quickly move on to a desktop calculator, and top the week off with a file browser. Along the way, the well chosen projects demonstrate and build familiarity for data types, operators, branching, and loops. The Q&A and Workshop sections at the end of each chapter assist the reader to begin generalizing the concepts of Delphi and encourage self-exploration. Novices should be warned that the first four chapters contain numerous syntax, keyboard, and diagramming errors. But don't give up; play around with different key strokes and try to look ahead — persistence rewards those who make it beyond day six.

In week two, you begin by mastering arrays, strings, structures, and user-defined data types, and by mid-week you've used subroutines and are learning object-oriented programming. Day 11 builds understanding of encapsulation and inheritance, and by day 12 you can polymorph with the best. The VIRTUALS program that you create on day 12 (and the authors' subsequent explanation of virtual and dynamic methods) is instructive even for many seasoned programmers who may not understand how function tables operate. By the end of the second week you can define properties, respond to Windows events, use the Windows API, use Run-Time Type Information, and handle exceptions.

Week three brings graphics, drawing on the canvas, and the visually satisfying rewards of programming. Building upon the foundation of Object Pascal, *Teach Yourself* brings it all together with printer output, menus, edit boxes, scroll bars, and the litany of standard Delphi components used to construct an editor. The DO and DON'T advice for

"Kick the Habit and Teach Yourself Delphi in 21 Days" continued on page 48

Twenty-One Days to Delphi/Paradox Knowledge



Many programmers turn to Delphi when considering desktop database or client/server applications. After all, Delphi was designed in-part to ease writing database applications. Finally, from SAMS Publishing, comes *Teach Yourself Database Programming with Delphi in 21 Days*, a book exclusively dedicated to database application development. If you are new to programming or are considering Delphi to maintain desktop databases, you should consider this book. However, if you have been using Delphi a while and have been eager for an advanced client/server guide, your wait continues.

Visual Basic and C++ programmers will recognize the authors, Nathan and Ori Gurewich, from their previous books on database programming and multimedia development. Fans of the authors will be happy to note that they have included a CD that contains plenty of .AVI, .MIDI, .WAV, and .DB files, as well as all the program examples described in the book. The authors don't assume readers have database programming experience, and the abundance of diagrams and figures generously interspersed ensures that no previous programming experience is required to complete the exercises.

The authors take you day-by-day, starting with "Hello World," and then plunge into the Database Desktop to create the tables used in subsequent lessons. You use the Database Expert to create sample applications and then dissect the generated applications to determine how they are built. After a thorough discussion of the

"Twenty-One Days to Delphi/Paradox Knowledge" continued on page 48

Kick the Habit and *Teach Yourself Delphi in 21 Days* (cont.)

many of these components can save you hours. Day 19, dedicated to database programming, provides concise and critical information to get you started writing database applications. The 38-page chapter on database programming teaches basic database concepts, master/detail forms, and even illustrates field linking and the use of calculated fields. You're off to creating custom components on day 20, and building DLLs on day 21. To top off your Delphi whirlwind, authors Wozniewicz and Shammas toss in an extra day so that the truly addicted can play with DDE and OLE.

If you are completely new to programming, fear not. The authors have gone to extremes to make the first week promote knowledge with a minimum of assumptions. You'll find the book clearly and carefully written. Experienced programmers will be impressed with, and refreshed by, the clarity and depth of explanation provided. For those who are wondering, as we all do when facing a "21 day" book, just how long each of your 21 days will become: expect to spend anywhere from 20 minutes to one hour and 40 minutes, with the average being about 40 minutes per day.

Although it does contain more than its fair share of typos and printing gaffs (which will be especially troublesome to the beginner audience that will find this book the most useful) *Teach Yourself* is the best structured path to Delphi knowledge. Delphi converts, beginners, and programmers wanting a better understanding of Object Pascal, will have a difficult time finding a better introduction than *Teach Yourself Delphi in 21 Days*.

— *James Callan*

Teach Yourself Delphi in 21 Days by Andrew J. Wozniewicz and Namir Shammas, SAMS Publishing, 201 West 103rd St., Indianapolis, IN 46290, (800) 428-5331

ISBN: 0-672-30470-8
Price: US$29.99
912 pages

Twenty-One Days to Delphi/Paradox Knowledge (cont.)

navigator, you use the Fields Editor, introduced on day four, to alter field properties, and by day five you're adding calculated fields. Your first week wraps up by linking master-detail tables, using *FindKey* to search for records, and familiarizing yourself with the important topic of data validation and exceptions.

In week two, while laying out data entry forms, you begin to explore table lookups and static validation lists. Although the book makes clear that its intent is not to explain data modeling and database design, some readers may be disappointed to find the important topic of referential integrity hidden away as an answer to the exercises in chapter nine. By then, the reader, having an acute sense of déjà vu, would probably prefer discussing field events like *SetText*, converting case on data entry, customizing complex validation routines, cascading deletes, and learning other meaty database topics, rather than trudging over another half page devoted to implementing an exit button.

Day 10 introduces SQL, and by day's end, you're implementing queries and ranges. Multi-form projects are lightly glossed on day 11 in order to provide a context for the interesting topic of bookmarks. The real fun begins on day 12 with an animated dance routine featuring an Amazon "cutie" in a tight fuchsia dress and tiny Texas Tom, both doing the two-step thanks to a DBImage. You add synchronized sound to programs with a VBX included on the CD, and by the end of the second week, you're listening to banjo picking and space-age reggae.

Week three has you watching .AVI flicks of Einstein and former presidents, while setting up dynamic SQL statements behind the scenes. A lengthy read on day 16 rewards you with lively animation and virtual reality, also assisted by a VBX included on the CD. The book ends with some helpful, albeit often ignored, insights into writing reports with ReportSmith. Here you'll find multi-table reports and groups well covered, and derived fields and the crosstab given equally complete treatment.

Written for new and casual users, *Teach Yourself Database Programming with Delphi in 21 Days* promotes pragmatics over purity. It offers practical step-by-step database fundamentals, spiced with multimedia and virtual reality. Although the book starts slowly and repeats topics, later chapters and the CD resources add adventure to weeks two and three. When using the CD, make sure that you create a /Dprog directory, preferably off your C drive, or you may experience difficulty running the examples. Except for obvious spelling oversights found on both the CD and in the book, you will find the book an accurate and easy read.

Experienced programmers, however, should pass on this book. It's also important to note that the "database" referred to in the title is Paradox. You'll find nothing about working with true RDBMSs (e.g. Oracle or InterBase), transaction processing, or even the Database component. So the wait continues. Although you'll find broader database coverage and application scaling techniques in *Delphi Unleashed* [SAMS, 1995] or *Delphi: A Developer's Guide* [M&T Books, 1995], a comprehensive guide for the Delphi database developer still eludes us.

— *James Callan*

Teach Yourself Database Programming with Delphi in 21 Days by Nathan and Ori Gurewich, SAMS Publishing, 201 West 103rd St., Indianapolis, IN 46290, (800) 428-5331

ISBN: 0-672-30851-7
Price: US$39.99
569 pages, CD-ROM

Down and Out

We talked last month about the decentralization of the computer world and its implications for Web development. This month we continue that theme by looking at a technology that deals with decentralization within the database realm.

Corporate data in the 1990s is moving down and out. The popularity of client/server architecture has certainly been a driving force in this decentralization of data. Not only are PCs replacing mainframe terminals, people are doing more with company data through powerful query and analysis tools on the front-end. Additionally, data is moving increasingly outward rather than being stored and maintained at a single site. Geographically dispersed companies that have already invested in client/server technology are seeking solutions that can deal with dispersed data sets. Perhaps as important, the popularity of notebook computing has created a demand by a mobile work force to work with data remotely and periodically synchronize with their office. These trends point out that there are more people doing more with data — data that is spreading out to more places.

For developers, the task of designing client/server applications that can keep up with these trends is proving difficult. For many, the solution is fast becoming replication. *Asynchronous replication* (sometimes referred to as "store and forward") is a technology that allows you to maintain a single set of data among two or more separate locations. To achieve synchronization, replication tracks updates made to a specific data set and then ensures these changes are shared with other replicated databases. Replication is emerging as the most important database technology in the mid-1990s. While server-to-server replication is nothing new, what is becoming evident is the fact that replication will eventually emerge as a technology for the "masses," not just the Fortune 500. Smaller firms and departments can employ replication to meet a variety of needs unthinkable just a few years ago when floppy disks and ZIP files were the principle means of refreshing data.

Dealing with Distributed Data. Replication is not the first nor only approach to dealing with distributed databases. Many companies have a centralized database operating over a Wide Area Network (WAN), but available bandwidth and high cost make this idea unattractive to many. Web technology offers a possible alternative to WANs for a centralized solution, but as we discussed in last month's "File | New," Web applications are not a client/server panacea. A third alternative, commonly referred to as the "two-phase commit," has largely been considered a failure. A common element of these three approaches is that they require a "live" link somewhere in the process in order for users to actually work with data. However, synchronous communication is not only impractical in certain contexts, it also raises cost, bandwidth, and reliability concerns. Replication is thus proving to be more cost effective, faster, and more reliable.

Synchronizing Data. A fundamental issue you face with replication is how dispersed data can be synchronized when data is exchanged from one site to another. There are two approaches. First, a *primary site* (or "publish & subscribe") scheme has a single site as the owner of the data; this site publishes the data to subscriber databases that have only read-only access to the original data. Second, *peer-to-peer* (or "update anywhere") replication allows data to be updated at more than one site at the same time. While the latter approach is much more flexible, it also leaves open the possibility of "colliding records" (records that are simultaneously updated in two or more sites). These collisions can be risky. Conflicts have to be resolved in an efficient manner, but not even a sophisticated conflict resolution scheme can deal with the non-database actions taken after the data is committed locally, but before it is rejected during synchronization. Warnings aside, I am discovering that many companies are demanding an "update anywhere" solution, simply because it is the "real world" way in which their business works.

Shrink-Wrapped vs. Custom Solution. Replication support is becoming as ubiquitous to SQL servers as spell checkers are to word processors. Not just a value added feature, it is now becoming essential to being competitive in the SQL database marketplace. However, as robust as the current offerings of database vendors are, they are difficult to administer (especially Oracle) and remain fairly strict and narrow in terms of how you can implement them. First, not all products support all forms of replication. Oracle7 is the only major database to support the "update anywhere" model; in contrast, Microsoft SQL Server and Sybase support a primary site model only. Second, while Oracle and Sybase have solutions for notebook-to-server replication, Microsoft SQL Server currently remains limited to a server-to-server model (although Microsoft should have a desktop solution before the end of the year). Third, when you buy into any of these solutions, you are limiting yourself to a vendor-specific solution; none of these products currently support replication across multiple vendor databases. The third party replication products on the market currently do not help much either; they fail to pack the power or speed you need for many real world situations. Therefore, as much as replication technology has matured, there are holes remaining in the marketplace. Given these limitations, developers continue to find the need to develop custom solutions. We'll look at some of the issues involved with designing your own replication scheme in a future "File | New."

Balancing Act. The early days of PCs brought users to computers in an effort to increase productivity. Today, we are doing just the opposite: moving the computers out to the users. The challenge we have as application developers is to support this flexibility while maintaining the data integrity of a centralized environment. Replication can be a solution to this dilemma, but be sure to use caution as this technology continues to evolve.

— Richard Wagner

Richard Wagner is the Chief Technical Officer of Acadia Software in Boston, MA. He welcomes your comments at rwagner@acadians.com.