May 2004 | Volume 10 Number 5

INSIDE

Reviews: Doc-To-Help and Delphi Developer's Guide to XML v2
Delphi Informant
www.DelphiZine.com
May 2004 vol.10, no. 5
The Complete Monthly Guide to Delphi Development
extending
DREAMWEAVER
Not Just for C Programmers Anymore
Troubleshooting Typed DataSets
.NET Streams & Serialization
Transitioning to ADO.NET
RSA Encryption

Cover Art by Arthur A. Dugoni Jr.

MySQLDirect .NET 2.00 Released

Core Lab announced the release of *MySQLDirect .NET 2.00*, an ADO.NET data provider for direct access to the MySQL database server for the Microsoft .NET Framework.

MySQLDirect .NET is based on ActiveX Data Objects for the .NET Framework (ADO.NET). ADO.NET provides a rich set of components for creating distributed, data-sharing applications. It is an integral part of the .NET Framework, providing access to relational data, XML, and application data.

MySQLDirect .NET can be used in the same way as the SQL Server .NET or the OLE DB .NET Data Provider.

MySQLDirect .NET provides functionality for connecting to the MySQL database, executing commands, and retrieving results. Those results can be processed directly or placed in an ADO.NET DataSet for further processing while in a disconnected state. While in the DataSet, data can be exposed to the user, combined with other data from multiple sources, or passed remotely between tiers. Any processing performed on the data while in the DataSet can then be reconciled to MySQL database.

MySQLDirect .NET is designed to be lightweight. It consists of a minimal layer between the MySQL database and your code.

Also new is a database explorer tool integrated in Visual Studio. NET. DBExplorer lets you view, create, and manipulate database connections, schema objects, and data using a grid-based editor.

The standard edition includes ADO.NET core classes and classes for native data types only. The professional edition includes additional classes, design-time extensions, and wizards.

MySQLDirect .NET supports Delphi 8 and Microsoft ASP.NET Web Matrix. MySQLDirect .NET Data Provider is licensed per developer and registered users can deploy run-time assemblies with executable applications royalty fee. Free support is provided for registered users. A trial version is available for free download from the Core Lab Web site.

Core Lab
Price: MySQLDirect .NET 2.00 Standard, US$99; MySQLDirect .NET 2.00 Professional, US$149. Upgrade from version 1, US$39.
Contact: support@crlab.com
Web Site: www.crlab.com

TurboDemo 5.0 Available

Bernard D&G announced the release of *TurboDemo 5.0*, a Windows-based application to create professional demonstrations and interactive tutorials with no programming knowledge required.

Even though the finished demonstrations or tutorials may include animation, background music, verbal instructions, interactivity with the learner, and special effects, the generated demos/tutorials have extremely small file sizes (about 100KB per minute of playback). This allows them to be easily distributed; they can be accessed and played online to sell products and services, e-mailed to a customer to provide technical support, circulated via CD for internal training, or incorporated into a Windows Help file to provide animated, interactive user support.

Even a novice can quickly create a dynamic demonstration in three simple steps: 1) Capture the screens and/or import pictures; 2) Enhance the slides with special effects such as images and balloons, add verbal cues or sound, and include hot spots to allow the viewer to interact with the demonstrations; and 3) Compile the slides into the demonstration or tutorial format of your choice (such as Flash, GIF, or Java).

Once created, the demonstrations or tutorials can be played on all operating systems and the viewer is not required to have the TurboDemo software, or even an Internet connection. With TurboDemo, your audience is not limited by computer preference or language.

TurboDemo 5.0 offers many new functions and features, including: a network version that allows the program to launch from a CD or the server without installation; faster recording capabilities with DirectX technology and an automatic capture function; a new Memory Manager that uses less RAM; more styles of balloons, notes, and interactive text objects to direct the attention of the viewer on each slide; new export formats, animated objects, and many other enhancements.

See the Bernard D&G Web site for complete details on the Standard and Professional versions, as well as all the upgrades to TurboDemo 5.0.

Bernard D&G
Price: See Web site.
Contact: sales@turbodemo.com
Web Site: www.turbodemo.com

RemObjects SDK 3.0 for Delphi Announced

RemObjects Software released RemObjects SDK 3.0 for Delphi and Kylix. RemObjects SDK allows you to remotely access objects residing on a server from clients inside the LAN or across the Internet. It supports object pooling, asynchronous invocation, compression, encryption, and a variety of protocols such as TCP/IP, HTTP, UDP, POP3/SMTP, NamedPipes, etc. It includes the RemObjects Service Builder and allows you to expose your services as SOAP Web services.

RemObjects SDK 3.0 features a completely rewritten Service Builder 3, the RAD service-modeling tool; new extensive plug-in architecture for Service Builder 3; a new Service Tester that allows benchmarking and load-testing of your services; a new Master Server database application to easily share common session data and event repository among multiple servers; integrated load balancing and failover support; new server to client events and callbacks; and much more.

In addition, you can now access your RemObjects Servers from any COM-compatible language, and from Active Scripting environments such as Office and ASP.

RemObjects Software
Price: Single-developer license (includes source), €299; single-developer license upgrade (includes source) €199.
Contact: info@remobjects.com
Web Site: www.remobjects.com

Packet Sniffer SDK for Windows

MicroOLAP Technologies released *Packet Sniffer SDK*, a development suite for network packet capture that works without any preinstalled network drivers. This Windows library set includes ActiveX, DLL, VCL, and static library editions for Microsoft VC and Borland C compilers.

Previously available only to Delphi/C++Builder developers, the suite has been rewritten in Microsoft Visual C++, and is now available to any development environment, including Borland Delphi/C++Builder, Microsoft Visual Basic, Microsoft Visual C++, Microsoft Visual C#, Sybase PowerBuilder, Intel C++ for Windows, and many others.

Packet Sniffer SDK doesn't require any external network drivers to be installed, but instead uses its own built-in network driver with a very high working efficiency rate. This driver works directly with the network adapter selected in the application. Upon initiation and termination of the application using Packet Sniffer SDK, a built-in network driver loads and unloads, not interfering with the previously installed networking software.

Packet Sniffer SDK features a built-in BSD Packet Filtering engine, which leads to an even more increased performance.

MicroOLAP Technologies LLC
Price: See Web site.
Contact: info@microolap.com
Web Site: www.microolap.com

Reg Organizer 2.1 System Utility Announced

ChemTable Software released *Reg Organizer 2.1*, a registry maintenance tool for Windows 95/98/ME/NT/2000/XP/2003.

Reg Organizer allows Windows users to take full control over the registry database using a set of easy-to-use tools for optimizing system performance. The application offers advanced registry management methods that are not included in Windows, making it possible to preview .reg files before adding data to the registry; automatically find, replace, and delete multiple instances of user-specified registry keys and values; and clean up obsolete registry entries and monitor certain registry key modifications.

Reg Organizer is intended both for home and corporate users who would like to take advantage of hidden functionality to keep their registry clean and the overall system operation at a maximum. Besides the built-in functionality, Reg Organizer is capable of learning new file formats and processing .ini file settings on-the-fly, which turns the application into a handy tool for managing any data associated with any program installed on the system. Reg Organizer boasts a comprehensive help system, an intuitive user interface, and tight Windows shell integration.

ChemTable Software
Price: Single-user license, US$29.95; small-business license, US$199.95; site license, US$500.
Contact: support@chemtable.com
Web Site: www.chemtable.com

Elcor Software Releases Registry Defragmentation 5.0

Registry Defragmentation 5.0 from **Elcor Software** physically defragments the Windows registry.

After a single run of Registry Defragmentation, the computer speed increases up to 100 percent and the computer itself stops freezing and crashing. As a result of regular registry defragmentation, computer users get a more stable operating system, shorter application/system response time, and the most optimal linear registry structure.

Registry Defragmentation targets all those who frequently install/uninstall new software applications. Uninstalling an application doesn't completely remove all program components. These components start hindering the computer performance, consuming system resources that should be allocated otherwise. As more and more applications get installed and uninstalled, the problem keeps growing, making computer performance unpredictable. This and many other registry-related problems can be fixed by Registry Defragmentation.

Registry Defragmentation 5.0 is distributed electronically over the Internet; a free demo version is available from the Elcor Web site.

Elcor Software
Price: Single copy, US$11.95; site license, US$35.95
Contact: support@elcor.net
Web Site: www.elcor.net

Powerful Set of Tools for JBuilder Developers

jProductivity has released *Productivity! 2.0*, a set of tools for JBuilder that simplify routine coding and navigation operations.

Productivity! makes you aware of any errors in your code and gives you immediate assistance to resolve them. The built-in Task List ensures that you'll always be on schedule, and the documentation support lets you easily write bulletproof documentation for your code.

With Productivity! it's easy to write well-composed and easily maintainable code, and you can reuse your favorite code fragments. Also, Productivity! lets you avoid annoying dialog boxes and wizards while you are coding. With Productivity! you can discover context and navigate through it, use hyperlinks to surf, and navigate freely through your classes, methods, and fields.

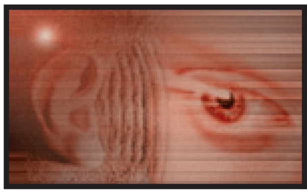
Using Productivity! you can obtain quick help on classes and methods exactly where and when you need it. You also can add super interfaces and change super classes in several simple steps, or override methods and constructors in a couple of clicks. Productivity! allows you to add access methods for your fields instantly, and use your own unique naming standards.

Productivity! Pro includes additional code generation tools, IDE improvements, editor enhancements and options, and navigation tools. A comprehensive description of features can be found on the company's Web site.

Productivity! is platform independent and requires Borland JBuilder X. Versions of Productivity! are also available for earlier versions of JBuilder. You can download trial versions of Productivity! from the company's Web site.

jProductivity, LLC
Price: Productivity! Standard, US$99; Productivity! Pro, US$189. Multi-user discounts are available.
Contact: (212) 918-1500
Web Site: www.jproductivity.com

table of CONTENTS



SOUND + VISION

MACROMEDIA DREAMWEAVER ■ HTML ■ JAVASCRIPT ■ DELPHI 5-7

By Stefan van As

Extending Dreamweaver

Not Just for C Programmers Anymore

Most developers have probably heard about Macromedia Dreamweaver. It's a visual DHTML editor for building Web sites and Web applications. It supports CSS (Cascading Style Sheets), cross-browser validation, and several middleware solutions, such as ColdFusion, ASP, and PHP. What most people don't realize, however, is that Dreamweaver is an extendible product. It comes with a huge JavaScript-based API that allows developers to create their own Dreamweaver extensions. Some of these third-party extensions are published at the Dreamweaver Exchange Web site: www.macromedia.com/cfusion/exchange/index.cfm?view=sn120.

In general, you'll develop Dreamweaver extensions using HTML and JavaScript. However, some extensions are developed in C. A C header file named mm_jsapi.h is available for download at the Macromedia Web site. This article describes the Pascal translation of this file (mm_jsapi.pas), and how to use Delphi to create Dreamweaver extensions. This article assumes you're working with Dreamweaver MX and Delphi 5, but the techniques discussed also apply to Dreamweaver MX 2004 and Delphi versions 6 and 7.

Dreamweaver extensions typically automate changes to the user's current document, such as inserting HTML or JavaScript, changing text or image properties, or inserting and managing blocks of server code in the current document. Most extension developers write an extension to handle a common, repetitive task.

There are three main components to Dreamweaver extensibility:
1) An HTML parser/renderer, which makes it possible to design user interfaces for extensions. Dreamweaver has a built-in HTML parser/renderer for this.
2) A JavaScript engine, which executes the JavaScript code in extension files. Dreamweaver MX uses the Mozilla JavaScript 1.5 engine (also known as SpiderMonkey) for this. This is the same JavaScript engine that's used in Web browsers such as Netscape Navigator.
3) A series of APIs that provide access to Dreamweaver functionality through JavaScript. These are generally referred to as the Dreamweaver extensibility APIs.

The Mozilla JavaScript engine that's used by Dreamweaver extensibility (and some Web browsers such as Netscape Navigator) comes with several built-in JavaScript objects, such as *Array*, *Date*, *RegExp*, etc. These objects are a good foundation on which to build extensibility into Dreamweaver. However, several tasks unique to Dreamweaver extensibility cannot be accomplished with the objects available in the Mozilla JavaScript engine.

To make creating useful Dreamweaver extensions possible, Dreamweaver introduces a number of additional JavaScript objects that are specific to Dreamweaver extensibility (in addition to the JavaScript objects that are provided by the Mozilla JavaScript engine). The most

```
var theDOM = dw.getDocumentDOM();
if (theDOM) {
  var theImg = theDOM.images[0];
  if (theImg) theImg.src = "MyImage.gif";
}
```

Figure 1: Getting a reference to the user's document DOM.

common (and most important) one that you'll probably use the most is the *dw* object (short for Dreamweaver object). This *dw* object exposes more than 400 properties and methods. For example: the *dw.getDocumentDOM* method returns a reference to the user's document DOM.

It's important to distinguish between the DOM of the user's document and the DOM of the extension UI; the way that you reference each DOM is different. You reference objects in your extension UI via *document*, e.g. `document.forms[0]`. To reference objects in the user's document, however, you must call what is probably the single most important Dreamweaver extensibility API function, *dw.getDocumentDOM*. For example, to refer to the first image in the user's document, you can store the user's document object in a variable, as shown in Figure 1.

"Macromedia Dreamweaver is a visual DHTML editor for building Web sites and Web applications. What most people don't realize, however, is that Dreamweaver is an extendible product."

The *getDocumentDOM* method of the *dw* object is a custom object implemented by Dreamweaver, and is widely used within extensions. For additional information on the properties and methods of the *dw* object, see the *Extending Dreamweaver MX* manual.

Most Dreamweaver extensions are simple and consist of only one or two files: an HTML document that "acts" as the UI to the extension, and a JavaScript document that does the work. Most of the time these two files have the same name, but there are also extensions that consist of several dozen, if not several hundred, files. On multiple user systems, such as Windows XP, these extension files are placed in the folder: C:\Documents and Settings\[UserName]\Application Data\Macromedia\Dreamweaver MX\Configuration. Otherwise, they're placed in C:\Program Files\Macromedia\Dreamweaver MX\Configuration.

Your extension files must be placed in either one of these folders for Dreamweaver to recognize them. You can do this by hand, or you could package your extension files using the Macromedia Extension Manager (EM), which comes with various Macromedia products. (In Dreamweaver, select Commands | Manage Extensions to launch the EM.) When the end user installs a packaged extension, your extension files are placed in the appropriate folder automatically. In other words, the EM facilitates the process of installing extensions in the correct folder(s).

The files you already have in your Configuration folder contain the extensions that come with the Dreamweaver product. Macromedia has its own extension development group, and a lot of features that are perceived as "native" to the product are actually extensions. You can use the files that come within the Configuration folder as examples, but these files are generally more complex than the average extension that's available on the Macromedia Exchange Web site. When you modify or delete these files, you potentially compromise the stability of the Dreamweaver product.

The folder structure of the Configuration folder corresponds to a specific extension type, e.g. Behaviors are in the /Configuration/Behaviors folder, Commands in the /Configuration/Commands folder, and Property Inspectors in the /Configuration/Inspectors folder. By familiarizing yourself with these folders, you can discover the interface for extensions and working examples of each extension type.

One folder within the Configuration folder does not correspond to a specific extension type. The /Configuration/Shared folder is the central repository for functions, classes, and images that are used by more than one extension. Look here for functions that perform specific tasks, such as creating a valid DOM reference to an object, escaping special characters in strings, and more. For more information on the contents of each subfolder within the Configuration folder, view the Configuration_ReadMe.htm file.

Dreamweaver automatically calls any extension that exists in an appropriate Configuration folder when specified conditions are met. In most cases, this means that the user initiates a task, and then Dreamweaver identifies a related extension in the Configuration folder, calls the various functions in the extensions, and expects a valid return value from each. Let's take a look at a simple Object extension to see how this works in practice.

Creating a Simple Object Extension

An Object extension is typically used to automate the insertion of HTML into a document. The UI to an Object gathers input from the user. (If you plan to submit your extension for Macromedia certification, your UI will need to follow certain guidelines.) Dreamweaver supports HTML form elements as the building blocks for extension UIs, and displays the UI using its built-in HTML parser/renderer. Object extensions are stored in the /Configuration/Objects folder.

The Object extension we're about to develop will insert the following HTML into the user's document:

```
<embed src="" width="" height=""></embed>
```

Our Object will consist of two files. The first is an HTML document that defines both the UI to the Object (the BODY contains an HTML form that accepts parameters for the extension) and what is inserted into the user's document (the HEAD contains JavaScript functions that process HTML

```
<html>
<head>
<script type="text/javascript" language="JavaScript">
function browseForFile() {
  var theFile = browseForFileURL();
  if (theFile) document.forms[0].filename.value = theFile;
}

function objectTag() {
  var theHTML = '<embed src="' +
              document.forms[0].filename.value;
  theHTML += '" width="' + document.forms[0].width.value;
  theHTML += '" height="' + document.forms[0].height.value;
  theHTML += '"></embed>';
  return theHTML;
}

</script>
<title>Insert MediaPlayer</title>
</head>
<body>
<form>
  <table>
    <tr>
      <td align="right" valign="baseline">Filename:</td>
      <td valign="baseline" nowrap>
        <input type="text" name="filename" size="30">
        <input type="button" onClick="browseForFile()"
               name="button" value="Browse...">
      </td>
    </tr>
    <tr>
      <td align="right" valign="baseline">Width:</td>
      <td valign="baseline">
        <input type="text" name="width"
               size="6" value="200">
      </td>
    </tr>
    <tr>
      <td align="right" valign="baseline">Height:</td>
      <td valign="baseline">
        <input type="text" name="height"
               size="6" value="200">
      </td>
    </tr>
  </table>
</form>
</body>
```

Figure 2: The HTML/JavaScript source for the Object extension.

form input from the BODY and control what is added to the user's document). The second is an 18x18 pixel image that will appear on the Insert panel.

When the user selects the Object by clicking the 18x18 pixel image in the Insert panel, the HTML document is scanned for a FORM tag. If a FORM tag exists and the Show Dialog When Inserting Objects option is selected in the General preferences, Dreamweaver displays the UI, the user enters parameters for the Object in the UI and clicks OK. If no FORM tag exists in the HTML document, Dreamweaver doesn't display a dialog box. Eventually, the *objectTag* function is called, and its return value is inserted into the user's document. The Objects API contains additional functions, but they're optional. At a minimum, you must define the *objectTag* function. The *objectTag* return value is inserted into the user's document. Returning an empty string, or null, is a signal to Dreamweaver to do nothing.

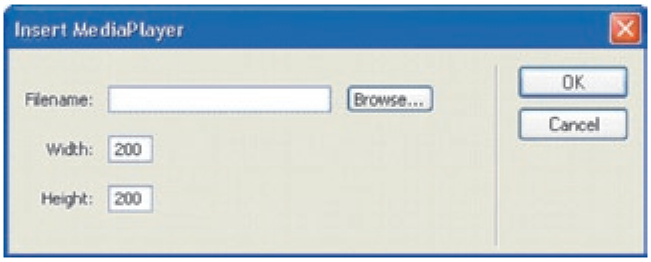


Figure 3: This dialog box is the UI to the Object extension.

Figure 2 shows the source for the Object extension. Because we're inserting an EMBED tag, we'll save this document (along with the accompanying 18x18 pixel image) in the /Configuration/Objects/Media folder (if this folder doesn't exist, create it). This will make the Object extension appear on the Media category of the Insert panel.

Now that we've saved the Object extension (and the accompanying 18x18 pixel image) in the /Configuration/Objects/Media folder, it's time to (re)launch Dreamweaver, or hold down Ctrl, right-click the icon in the upper-right corner of the Insert panel, and select Reload Extensions. The latter is an undocumented shortcut that allows the extension developer to reload extensions without having to close and restart Dreamweaver. The Object extension we've just created should appear in the Media category of the Insert bar. When the user clicks the 18x18 pixel icon, the dialog box in Figure 3 will appear.

So far, we've used the Dreamweaver extensibility API and created a simple Object extension. We could — and should — include additional parameters supported by the EMBED tag, such as *autoStart*, *showControls*, and *showStatusBar* to enrich the Object extension, but we'll stick with what we have for simplicity's sake.

So far, we've used only HTML and JavaScript. Now it's time to include the C-level extensibility layer into our extension.

The C-level Extensibility Layer

The C-level extensibility mechanism lets you develop extensions using a combination of JavaScript and your own Delphi code. You define functions using Delphi, bundle them in a DLL, save the DLL in the /Configuration/JSExtensions folder, and then call the Delphi functions from JavaScript. The functions you'll write in Delphi will eventually "act" as methods to a custom JavaScript object that's named identically to your DLL file.

Why would a developer use the C-level extensibility layer in the first place? Doesn't the JavaScript-based Dreamweaver extensibility API support everything we could possibly want to do? Yes and no. Sure, there's nothing you can instruct Dreamweaver to do in the C-level extensibility layer that you cannot do using the JavaScript-based APIs. But the JavaScript language itself is limited, mostly because of built-in security features. (Remember, the same JavaScript engine is running client-side in some Web browsers such as Netscape Navigator.)

For example, there's no file I/O in client-side JavaScript. If you want to create a Dreamweaver extension that inserts the contents of a user-specified file in the current user's

```
function browseForFile() {
  var theFile = browseForFileURL();
  if (theFile) {
    document.forms[0].filename.value = theFile;
    var docPath = dw.getDocumentPath("document");
    theFile =
      dw.relativeToAbsoluteURL(docPath, "", theFile);
    var theArray = MediaPlayer.getMediaDimensions(theFile);
    if (theArray && theArray.length > 1) {
      document.forms[0].width.value = theArray[0];
      document.forms[0].height.value = theArray[1];
    }
  }
}
```

Figure 4: Calling a Delphi function from JavaScript.

```
library MediaPlayer;

{$R *.RES}

uses
  Forms, SysUtils,
  mm_jsapi in 'mm_jsapi.pas';

begin
end.
```

Figure 5: Beginning to create a Dreamweaver extension in Delphi.

document, you will have to write the function using the C-level extensibility layer. This example is partly false, because Macromedia has already provided a library for file I/O named DWfile.dll in the /Configuration/JSExtensions folder, but they used their own C-level extensibility layer to do it. This is the same C-level extensibility mechanism we'll use to create our own DLLs.

Suppose we'd like to query the Media Control Interface (MCI) driver to automatically determine the width and height parameters for our Object extension, when the user presses the **Browse** button and selects a file. There isn't anything we can do in JavaScript to extract this information (at least not without embedding the MediaPlayer ActiveX into our extension UI, which would be extra overhead, visible to the user, and considered bad practice), but we can with Delphi. Figure 4 shows a modified version of the custom *browseForFile* function in our Object extension. Notice that the *browseForFile* function calls a Delphi function named *getMediaDimensions*, which is stored in a DLL named MediaPlayer.

The *getMediaDimensions* function accepts an argument, unpacks this argument, opens the specified filename in an instance of Delphi's *TMediaPlayer* class, and then returns the width plus the height of the specified filename as the return value.

Launch Delphi; select **File | New**, click **DLL**, and press **OK**. Save this project as MediaPlayer.dpr in the /Configuration /JSExtensions folder (if this folder doesn't exist yet, create it). Select **Project | Add to Project**, and select the mm_jsapi.pas unit (this unit is available for download, see the end of the article for details). mm_jsapi.pas

```
library MediaPlayer;

{$R *.RES}

uses
  Forms, SysUtils,
  mm_jsapi in 'mm_jsapi.pas';

function GetMediaDimensions(cx: JSContext; obj: JSObject;
  argc: UINT; argv: Ajsval; var rval: jsval): JSBool;
  cdecl;
begin
end;

procedure Init;
begin
  JS_DefineFunction(
    'getMediaDimensions', GetMediaDimensions, 1);
end;

begin
  MM_Init := Init;
end.
```

Figure 6: Registering a Delphi function with the Dreamweaver JavaScript engine.

includes definitions for the data types and functions in the C-level extensibility layer. After a bit of clean up, the project source looks like that shown in Figure 5. Choose **Build**. When the build operation finishes, a file named MediaPlayer.dll should appear in the /Configuration/ JSExtensions folder.

The Delphi code in your library must interact with the Dreamweaver JavaScript engine at three different times:

1) At startup, to register the library's functions. Dreamweaver will call the procedure you've assigned to the *MM_Init* variable at initialization time. This variable is defined in the mm_jsapi.pas unit. You'll register the functions you would like to make available to the Dreamweaver JavaScript engine inside this procedure.
2) When the function is called, to unpack the arguments passed from JavaScript to Delphi.
3) Before the function returns, to package the return value.

Everything you need to accomplish these tasks (including the *MM_Init* variable) is in mm_jsapi.pas, the unit that defines the data types and functions you'll need.

Registering a Delphi Function with the Dreamweaver JavaScript Engine

Let's start with registering the function that we want to make available to the Dreamweaver JavaScript engine. We'll create a procedure named *Init*, and upon initialization of our DLL, assign this procedure to the *MM_Init* variable (see Figure 6). This will instruct Dreamweaver to call the procedure named *Init* when Dreamweaver is launched. By the way, if you need to do any processing while Dreamweaver displays its splash screen, the *Init* procedure is the place to do it.

Dreamweaver calls the *Init* procedure to get the following three pieces of information:

- the JavaScript name of the function,
- a pointer to the Delphi function, and
- the number of arguments that the function expects.

To supply Dreamweaver with this information, we'll call the *JS_DefineFunction* function from inside the *Init* procedure. *JS_DefineFunction* is defined in mm_jsapi.pas, and registers a Delphi function with the Dreamweaver JavaScript engine. There's no need to call *JS_DefineFunction* from any place other than the procedure pointed to by the *MM_Init* variable, which Dreamweaver calls during startup. *JS_DefineFunction* takes three arguments:

- the name of the function as it is exposed to JavaScript (you can invoke the Delphi function from JavaScript by referring to it with this name)
- a pointer to a Delphi function (this function must accept certain arguments, which are discussed later)
- the number of arguments that the function expects

The function we've just defined must accept the following arguments:

- *cx* is a pointer to a *JSContext* structure, which holds the Dreamweaver JavaScript engine execution context. Some functions in mm_jsapi.pas accept this pointer as one of their arguments.
- *obj* is a pointer to a *JSObject* structure that contains the JavaScript object. It may be an array object or some other object type. While the JavaScript is running, the keyword *this* is equal to this object.
- *argc* contains the number of JavaScript arguments that are passed to the function.
- *argv* is a pointer to an array that contains the actual JavaScript arguments that are passed to the function. This array is *argc* elements in length. Each element in the array is of type jsval, an opaque data structure that can contain an integer, or a pointer to a float, string, or object. Some functions in mm_jsapi.pas can be used to read and convert these jsval values to Delphi-compatible data types.
- *rval* is a pointer to a single jsval. The function's return value should be written to *rval*. Again, mm_jsapi.pas includes several functions that you can use to convert Delphi data types to a jsval.

The function we've just defined must return a JSBool value, indicating success (JS_TRUE) or failure (JS_FALSE). JSBool is a simple data type that stores a Boolean value. If the function result value equals JS_FALSE, the current JavaScript stops executing and Dreamweaver will prompt a JavaScript error message. (There is, by the way, a function named *JS_ReportError* in mm_jsapi.pas that allows you to generate your own custom JavaScript errors).

Last but not least, because Dreamweaver expects a C function (not a Delphi function), the Delphi function should use the **cdecl** calling convention.

In JavaScript, our *getMediaDimensions* function will accept one argument from the user. We'll need to unpack this argument that contains a filename, convert the file:// URL to a DOS path, instantiate Delphi's *TMediaPlayer*, and package the width plus the height of the specified filename as the return value.

Unpacking JavaScript Arguments

Now that we've defined the Delphi function, let's start with unpacking the JavaScript argument(s) that are passed to it. The mm_jsapi.pas unit contains the following functions to do so:

- *JS_ValueToString*
- *JS_ValueToInteger*
- *JS_ValueToDouble*
- *JS_ValueToBoolean*
- *JS_ValueToObject*

All of these functions take a pointer to a *JSContext* structure as their first argument (this is the *cx* argument that's passed to our Delphi function), and a jsval from which the function argument is to be extracted as their second argument.

JS_ValueToObject is a special function; it converts a function argument to an object. Most of the time, this will probably be an array object. If this is the case, use *JS_GetArrayLength* and *JS_GetElement* to read its contents (more on these functions later in this article).

In our case, the *getMediaDimensions* function expects a single string argument, so we need to call the *JS_ValueToString* function, as shown in Listing One (beginning on page 9). Before doing so, however, we need to validate that we actually received at least one argument. The *argc* argument will provide us with that information.

"The Delphi code in your library must interact with the Dreamweaver JavaScript engine at three different times."

Because Dreamweaver will pass a file:// URL, and because Delphi's *TMediaPlayer* expects a DOS path, we'll need to convert the former to the latter, using a helper function named *JS_FileURLToDOS*. The function converts a file:// URL to a DOS path (for example: file:///C|/Tools/Example.avi > C:\Tools\Example.avi). Notice this function isn't part of mm_jsapi.pas, but a custom function we will have to write and include with the project.

Ideally, the *JS_FileURLToDOS* function should decode any HTTP escape characters in the string that is passed to it, but for simplicity's sake, we'll stick with what we have for now. If you're ambitious, you could use Dreamweaver's *dw.doURLDecoding* or Borland's *HTTPDecode* function. The latter is included with Delphi's HTTPApp.pas unit.

If the DOS filename exists, we'll create a *TMediaPlayer* object, assign the DOS filename to the appropriate property, and call the *Open* method. But before we do so, we'll need to create a parent form because *TMediaPlayer* needs a windowed parent control (again, see Listing One). Assuming *TMediaPlayer* didn't receive an MCI error, we'll read the *TMediaPlayer.DisplayRect* property and retrieve the width and the height of the specified media.

Packaging the Return Value

So far, we've defined a function and unpackaged the argument(s). Now it's time to package the return value. To do so, the mm_jsapi.pas unit contains the following functions:

- *JS_StringToValue*
- *JS_IntegerToValue*
- *JS_DoubleToValue*
- *JS_BooleanToValue*
- *JS_ObjectToValue*

Because our function will need to return more than one value, we'll return an array object. But before we can call *JS_ObjectToValue* and store the return value in *rval*, we'll need to create the array object using *JS_NewArrayObject* and *JS_SetElement* (again, refer to Listing One). *JS_NewArrayObject* creates a new object that contains an array of jsval values. *JS_SetElement* writes a single element to an array object.

"The mm_jsapi.pas unit provides a full translation of the mm_jsapi.h header file. This is everything you'll need in Delphi to create a Dreamweaver extension using Macromedia's C-level extensibility layer."

In addition to *JS_NewArrayObject* and *JS_SetElement*, mm_jsapi.pas includes the following functions that will work on JavaScript arrays:

- *JS_ObjectType* returns the class name of the object referenced by the specified *JSObject*. In the case of an array object, this function would return "Array".
- *JS_GetArrayLength* returns the number of elements in an array object.
- *JS_GetElement* writes a single element to an array object.

If everything goes well, *GetMediaDimensions* should eventually return a JS_TRUE value.

Before we can compile the project that now includes implementation of both the *GetMediaDimensions* and *JS_FileURLToDOS* functions, we'll need to add Delphi's MPlayer.pas unit to the project's **uses** clause.

Now select **Build**. If the library is already loaded by a Dreamweaver instance, the Delphi compiler won't be able to create the output DLL, and you'll need to close down Dreamweaver. If the project compiled okay, launch Dreamweaver, click the MediaPlayer icon on the Insert panel, select an existing .avi filename somewhere on your computer (Delphi 5 installs a nice one named speedis.avi in C:\Program Files\Borland\Delphi5\Demos\Coolstuf if you have installed Delphi's demo projects), and the width and height parameters should get "calculated" and filled in automatically.

Conclusion

The mm_jsapi.pas unit provides a full translation of the mm_jsapi.h header file. This is everything you'll need in Delphi to create a Dreamweaver extension using Macromedia's C-level extensibility layer.

There's much more to Dreamweaver extensibility in general, and the C-level extensibility layer in particular, than an article like this could possibly cover. For example, there's a function named *JS_ExecuteScript* that compiles and executes a JavaScript string from within Delphi. Imagine the possibilities.

If you run into trouble, your first line of defense should be the *Extending Dreamweaver* manual (once an addition to *Using Dreamweaver*, but now a separate 600+ page manual). The *Extending Dreamweaver* manual is available via **Help | Extending Dreamweaver**, or you can order a printed version at the Macromedia store.

To communicate with other developers who are involved in extension writing, you might want to join the Dreamweaver extensibility newsgroup. You can access the Web site for this newsgroup at www.macromedia.com/go/extending_newsgrp.

The examples in this article are available for download on the Delphi Informant Magazine Complete Works CD located in INFORM\2004\MAY\DI200405SV.

Stefan van As *is an independent contractor, specialized in Borland (Delphi) and Macromedia (Authorware, Dreamweaver) development. Based in The Netherlands, his portfolio consists of several companies inside and outside the US, including Macromedia, Inc. He is considered an expert on e-learning development, has co-authored an e-learning book, and is a frequent speaker at e-learning conferences. If you would like to consult Stefan, contact him at svanas@xs4all.nl.*

Begin Listing One — The MediaPlayer library; a Dreamweaver extension

```
library MediaPlayer;

{$R *.RES}

uses
  Forms, SysUtils, MPlayer,
  mm_jsapi in 'mm_jsapi.pas';

function JS_FileURLToDOS(const S: string): string;

  function ReplaceChar(const S: string;
    Old, New: Char): string;
  var
    i: Integer;
  begin
    Result := S;
    for i := 1 to Length(Result) do
      if Result[i] = Old then
        Result[i] := New;
  end;

begin
  Result := S;
  if AnsiSameText(Copy(Result, 1, 8), 'file:///') then
```

```
    Delete(Result, 1, 8);
  Result := ReplaceChar(Result, '|', ':');
  Result := ReplaceChar(Result, '/', '\');
end;

function GetMediaDimensions(cx: JSContext; obj: JSObject;
  argc: UINT; argv: Ajsval; var rval: jsval): JSBool;
  cdecl;
var
  i: UINT;
  S: string;
  A: JSObject;
  v, w, h: jsval;
  F: TCustomForm;
  M: TMediaPlayer;
begin
  Result := JS_FALSE;
  if argc > 0 then begin  // We need at least one argument.
    // Convert argument from jsval to string.
    S := JS_FileURLToDOS(JS_ValueToString(cx, argv[0], i));
    if FileExists(S) then begin
      F := TCustomForm.CreateNew(nil);
      try
        // Instantiate TMediaPlayer (requires parent form).
        M := TMediaPlayer.Create(F);
        M.Parent := F;
        M.FileName := S;
        try
          M.Open;
          if M.Error = 0 then
            try
              // Convert width/height integer values
              // to jsval values.
              with M.DisplayRect do begin
                W := JS_IntegerToValue(Right - Left);
                H := JS_IntegerToValue(Bottom - Top);
              end;
              // Create new object that contains an array.
              A := JS_NewArrayObject(cx, 0, v);
              JS_SetElement(cx,A,0,W); // Add width value.
              JS_SetElement(cx,A,1,H); // Add height value.
              // Return array object.
              rval := JS_ObjectToValue(A);
              Result := JS_TRUE;
            finally
              M.Close;
            end;
        except
        end;
      finally
        F.Free;
      end;
    end;
  end;
end;

procedure Init;
begin
  JS_DefineFunction('getMediaDimensions',
                    GetMediaDimensions, 1);
end;

begin
  MM_Init := Init;
end.
```

End Listing One

■ By Bill Todd

ADO.NET Data Access Components

Part IV: Troubleshooting Typed DataSets

Part III of this series showed how to call the *DataAdapter* class' Fill method to automatically create a *DataTable* in a *DataSet* and fill the *DataTable* with the records returned by the *DataAdapter*'s *SelectCommand* object. Figure 1 shows a diagram of the *DataSet* object and the objects it references through its properties. When you call the *DataAdapter.Fill* method, a *DataTable* and its associated *DataRow*, *DataColumn*, and *Constraint* objects are created.

This sounds easy, but it isn't — and there was a hint of the problem in Part III of this series.

The sample application in Part III used a DataGrid to display the data from the Employee table in the *DataSet*. However, because the Employee *DataTable* doesn't exist at design time, there's no way to connect the DataGrid to the *DataTable* at design time. The solution was to add the following line of code to the **Open** button's *Click* event handler after the call to the *Fill* method:

```
EmployeeGrid.DataMember := 'Employee';
```

This works because the Employee *DataTable* exists in the *DataSet* after the *Fill* method has been called. However, if you consider a more realistic data entry form that contains perhaps 20 *TextBox*, and other, controls that are bound to columns in *DataTables*, the task of typing one line of code to bind each control to its data source becomes onerous. There are two solutions. The first is to use a typed *DataSet*. The second is to create the *DataTables* and their associated *DataColumn*, *DataRow*, *Constraint*, and *DataRelation* objects at design time.

A typed *DataSet* is a custom *DataSet* descendant that has all the *DataColumns* included as properties. To create a usable typed *DataSet*, you need a design-time tool that lets you generate the *DataTables* automatically from their respective *DataAdapters*. This tool must also let you define the *DataRelation* objects and any calculated or aggregate data column objects that you need. After you've supplied the required information, the tool should then generate the XSD schema file, and the .pas source code file for the typed *DataSet*.

Typed *DataSets* have two advantages. First, the columns are accessed through properties, so you get compile-time type checking. For example, the compiler will catch the error if you try to assign the value from a string column to an integer variable. The second advantage is that the syntax for accessing the value of a column requires a bit less typing than the syntax used with untyped *DataSets*.

You may read that accessing the value of a column in a typed *DataSet* is faster than accessing a column's value in

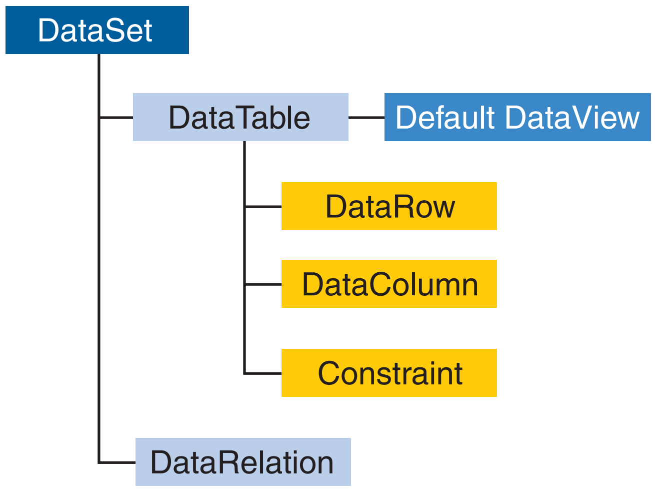


Figure 1: The ADO.NET *DataSet* and associated objects.

an untyped *DataSet*. This is not true. There are two ways to access values in an untyped *DataSet*. One technique is slower than a typed *DataSet*, but the other is as fast or faster. Both techniques will be covered later.

Unfortunately, the tool for generating typed *DataSets* in Delphi 8 doesn't allow you to define the relationships between tables, so typed *DataSets* won't have any *DataRelation* objects. This means you must write code to manage all master-detail relationships. In addition, the typed *DataSet* wizard doesn't provide any way to define calculated or aggregate columns. The result is that typed *DataSets* aren't very useful in Delphi 8.

Untyped *DataSets* would be easy to use if there were a wizard that would do at design time what the *DataAdapter*'s *FillSchema* method does at run time — namely, add the *DataTable*, *DataColumn*, and *Constraint* objects to the *DataSet* component. There is no such wizard, however, so you must create all the schema for the *DataSet* manually using the property editors.

Building the Sample Application Shell

Before building the schema for the *DataSet* you must create the shell of the sample application that accompanies this article (see end of article for download details). First, create a new WinForms project. Then drag the Department table to the form, from the Employee database in the Data Explorer. This will add a BdpConnection and a BdpDataAdapter to the tray. Name the BdpConnection `EmployeeConnection` and name the BdpDataAdapter `DeptAdapter`. Drag the Employee table to the form. Then add a second BdpDataAdapter and name it `EmpAdapter`. Finally, add a DataGrid to the form. The form should now look like Figure 2.

In a real application you wouldn't load all the rows from the tables into the *DataSet*. Right-click **DeptAdapter** and choose **Configure Data Adapter**. Add the following WHERE clause to the SELECT statement in the Data Adapter Configuration dialog box:

```
SELECT DEPT_NO, DEPARTMENT, HEAD_DEPT, MNGR_NO, BUDGET,
       LOCATION, PHONE_NO FROM DEPARTMENT
 WHERE HEAD_DEPT = '100'
```

You'd normally use a parameter (instead of the literal '100') to give the user a way to change the parameter value. However, I'm going to skip that and stay focused on building the *DataSet* schema. Next, change the SELECT SQL for the EmpAdapter as shown below (the JOIN and the WHERE clause will retrieve only the employee records for the departments selected by the DeptAdapter):

```
SELECT EMP_NO, FIRST_NAME, LAST_NAME, PHONE_EXT, HIRE_DATE,
       DEPT_NO, JOB_CODE, JOB_GRADE, JOB_COUNTRY, SALARY
  FROM EMPLOYEE E
  JOIN DEPARTMENT D ON E.DEPT_NO = D.DEPT_NO
 WHERE D.HEAD_DEPT = 100
```

Building the DataSet

Select **EmpDataSet** from the main form, click the *Tables* property in the Object Inspector, then click the ellipsis button to open the Tables Collection Editor. Click the

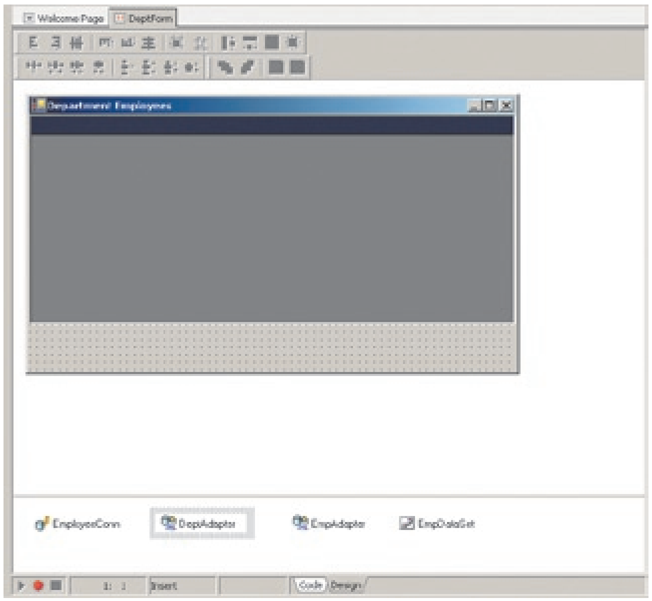
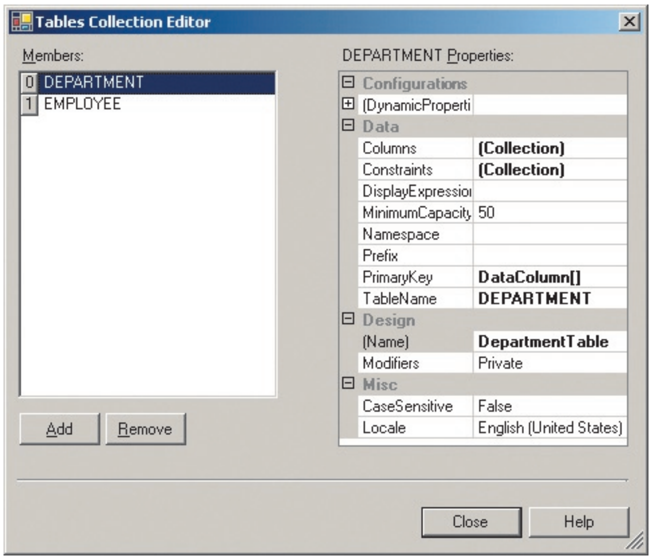


Figure 2: The main form.


Tables Collection Editor

Members:

0 DEPARTMENT
1 EMPLOYEE

DEPARTMENT Properties:

⊟ Configurations	
⊞ (DynamicProperti	
⊟ Data	
Columns	(Collection)
Constraints	(Collection)
DisplayExpressio	
MinimumCapacity	50
Namespace	
Prefix	
PrimaryKey	DataColumn[]
TableName	DEPARTMENT
⊟ Design	
(Name)	DepartmentTable
Modifiers	Private
⊟ Misc	
CaseSensitive	False
Locale	English (United States)

Add Remove Close Help

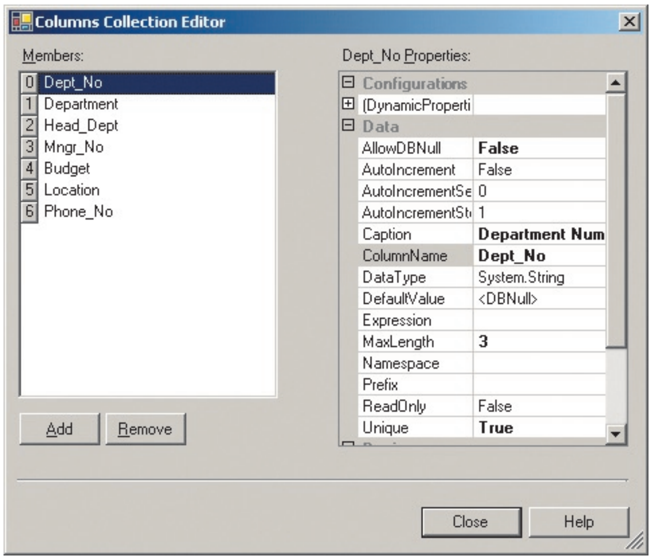
Figure 3: The Tables Collection Editor.

Add button to add a new *DataTable*, then set its *TableName* property to `DEPARTMENT` and its *Name* property to `DeptTable`. Figure 3 shows the Tables Collection Editor after the Department and Employee tables have been added.

Property	Value
AllowDBNull	*False*
Caption	*Department Number*
ColumnName	*Dept_No*
MaxLength	*3*
Name	*DeptNoColumn*

Figure 4: Dept_No column properties.

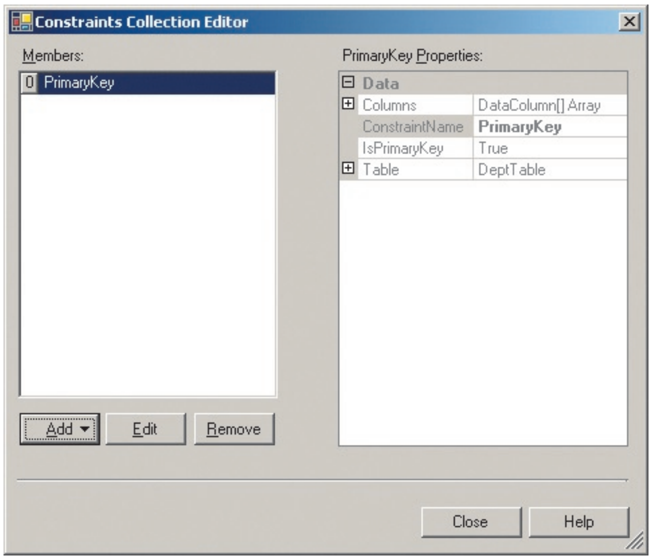
Next, select the *Columns* property and click the ellipsis button to open the Columns Collection Editor. Click the **Add** button to add a new *DataColumn* object. Set the properties as shown in Figure 4. Repeat this process for the other columns in the Department table. Figure 5 shows the Columns Collection Editor after all the columns have been



Members
0 Dept_No
1 Department
2 Head_Dept
3 Mngr_No
4 Budget
5 Location
6 Phone_No

⊟ Configurations	
⊞ (DynamicProperti	
⊟ Data	
AllowDBNull	False
AutoIncrement	False
AutoIncrementSe	0
AutoIncrementSt	1
Caption	Department Num
ColumnName	Dept_No
DataType	System.String
DefaultValue	<DBNull>
Expression	
MaxLength	3
Namespace	
Prefix	
ReadOnly	False
Unique	True

Figure 5: The Columns Collection Editor.



⊟ Data	
⊞ Columns	DataColumn[] Array
ConstraintName	PrimaryKey
IsPrimaryKey	True
⊞ Table	DeptTable

Figure 7: The Constraints Collection Editor.

Property	Description
AllowDBNull	Determines whether the column can have the null state.
AutoIncrement	Will the server automatically assign a unique number to this column?
AutoIncrementSeed	The starting value.
AutoIncrementStep	The amount the value is incremented for each new row.
Caption	The display name for the column. If you leave it blank it will be set to the same value as the *ColumnName* property.
ColumnMapping	How the column is saved using the *DataSet.WriteXML* method. This has nothing to do with the *DataAdapter.ColumnMappings* property discussed in Part III.
ColumnName	The name used in the *DataTable's Columns* collection.
DataType	The .NET Framework type for this column.
DefaultValue	The default value assigned to this column when a new row is created.
Expression	The expression used to calculate the value of this column.
MaxLength	The maximum length of a text column.
ReadOnly	Is the column read only or read/write?
Unique	Can the column have duplicate values?

Figure 6: Properties of the *DataColumn* object.

added. Figure 6 contains a brief description of the most important properties of the *DataColumn* object.

The *DataColumn* object properties should be self explanatory — with the possible exception of *AutoIncrementSeed* and *AutoIncrementStep*. Remember that the data in the *DataSet* is stored in memory on the local workstation. When a user inserts a new row in a table, the database server is unaware that a new row has been added. If the table contains an autoincrement column, a temporary unique value must be generated locally until the final value is assigned when the new row is added to the database. This process will be covered in detail in a future installment in this series.

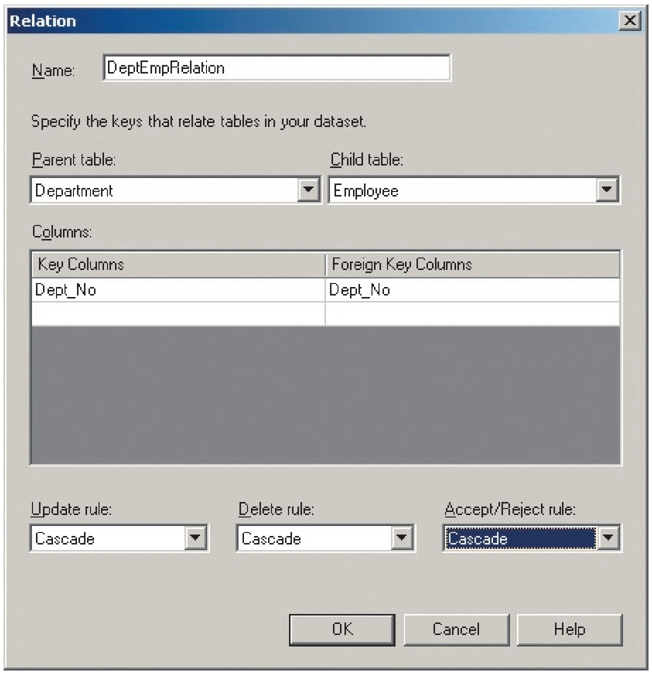
After closing the Columns Collection Editor, you'll be back in the Tables Collection Editor. The next step is to add the primary key constraint. One way to do this is to select the *PrimaryKey* property, click the drop-down button, then click the box next to each column you want to add to the primary key. The problem with this is that the constraint will be named Constraint1, which isn't very descriptive.

Another solution is to select the *Constraints* property and click the ellipsis button to open the Constraints Collection Editor, shown in Figure 7. Click the **Add** button and choose **Unique Constraint**. In the Unique Constraint dialog box enter the name for the constraint, check the fields you want to include, and check the **Primary Key** checkbox; then click **OK**. The disadvantage of this method is that you cannot control the order of the fields in the constraint.

Returning to the Tables Collection Editor, add a new table, set its *TableName* property to `EMPLOYEE` and its *Name* property to `EmployeeTable`. Next, add all the columns. When you get to the Dept_No column you'll get an error if you try to set its *Name* property to `DeptNoColumn`. The *DataColumn* object names must be unique, and the *DataColumn* object for the Dept_No column in the Department table is already named DeptNoColumn. Pick another name; for example, EmpDeptNoColumn for the *DataColumn* object in the Employee table.

You can also add calculated columns to the *Columns* collection — at least in theory. Just add another column and set the *Expression* property to give the value you want. For example, you can add a column named Full_Name and set the *Expression* property to:

```
First_Name + ' ' + Last_Name
```



Name: DeptEmpRelation

Parent table: Department

Child table: Employee

Columns:

Key Columns	Foreign Key Columns
Dept_No	Dept_No

Update rule: Cascade

Delete rule: Cascade

Accept/Reject rule: Cascade

Figure 8: The Relation dialog box.

The problem is that the value of the column is always null at run time. This is a bug in Microsoft's Columns Collection Editor. In version 1.1 of the .NET Framework the only way to add an expression column is in code. You'll see how later in this article.

Close the Columns Collection Editor and open the Constraints Collection Editor. Add a primary key constraint on the Emp_No column. Close the Constraints Collection Editor and the Tables Collection Editor. Next, make sure the EmpDataSet is selected and return to the Object Inspector. Select the *Relations* property and click the ellipsis button to open the Relations Collection Editor.

Click the Add button to open the Relation dialog box shown in Figure 8. To define the relation enter a name, then choose the parent table and child table from the drop-down lists. Then choose the parent table's primary key column, and the child table's foreign key column, from the drop-down lists in the grid. Add additional rows to the grid if the keys include more than one column. The Update rule and Delete rule drop-downs let you choose what happens when a row in the parent table is updated or deleted. The choices are Cascade, Set Null, Set Default, or None. These are the same choices offered by most relational databases, and are described in detail in the online help.

The Accept/Reject rule is unique to the ADO.NET *DataSet*. Changes to rows can be accepted using the *AcceptChanges* method or canceled using the *RejectChanges* method of the *DataSet*, *DataTable*, or *DataRow*. The Accept/Reject rule determines what happens to child rows when *AcceptChanges* or *RejectChanges* is called for the parent row. The choices are Cascade and None. If Cascade is selected the effect of *AcceptChanges* or *RejectChanges* will be cascaded to the child table. Choose Cascade for the sample application.

```
procedure DepartmentForm.DeptForm_Load(
  Sender: System.Object; e: System.EventArgs);
begin
  EmployeeTable.Columns.Add('Full_Name', typeof(string),
    'FIRST_NAME + '' '' + LAST_NAME');
  EmployeeConn.Open;
  DeptAdapter.Fill(DepartmentTable);
  EmpAdapter.Fill(EmployeeTable);
end;
```

Figure 9: The form's *Load* event handler.

Two things happen when you click the OK button. First, a foreign key constraint is created and added to the *Constraints* collection of the child *DataTable* (in this case the Employee table). You can see this by returning to the Tables Collection Editor, selecting the Employee table, and opening the Constraints Collection Editor. The constraint will be named Constraint1. To give it a more meaningful name click the Edit button and change the name. Second, the *DataRelation* object is also created and added to the EmpDataSet's *Relations* collection.

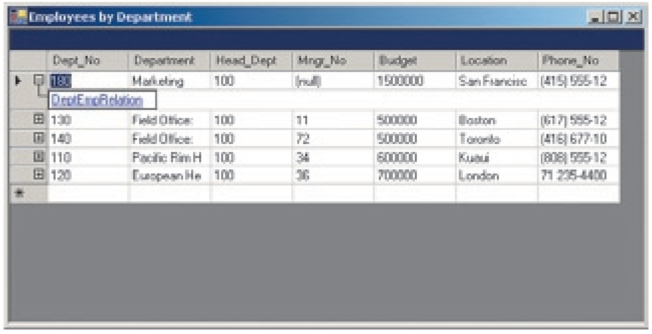
Connecting the DataGrid

Select the DataGrid, then select the *DataSource* property and click the drop-down button. Choose EmpDataSet from the list. Next, select the *DataMember* property and choose Department from the drop-down list. Because there is a *DataRelation* object connecting the Department *DataTable* to the Employee *DataTable*, the DataGrid will let users drill down from each Department row to see the Employee rows for that Department.

Select the form, then click the Events tab in the Object Inspector. Double-click the *Load* event and add the code shown in Figure 9 to the event handler. The first line adds the Full_Name column to the Employee table. The *Add* method's first parameter sets the *ColumnName* property; the second sets the DataType of the column; and the third parameter sets the *Expression* property.

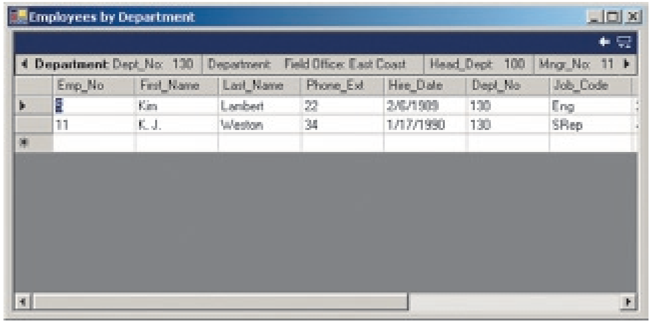
The second line opens the connection to the database. The next two lines fill the two *DataTables* with data. The *Fill* method has several overloaded versions. Note that the one called in Figure 9 takes a *DataTable* object as its only parameter. There is another version that takes two parameters. The first is the *DataSet*, and the second is the name of the *DataTable* as a string. This version assumes that the only reason you would pass a *DataTable* name instead of the *DataTable* object is that the *DataTable* doesn't exist and must be created. That won't work in this case, because the *DataTable* objects were created at design time and already exist when this code runs.

Run the application, and click the plus sign next to a row in the DataGrid. The form should look like Figure 10. Figure 11 shows what the form looks like after you click the link to display the employee records. Note that the text for the link is the name you assigned to the *DataRelation* object. Clicking the white triangle near the right end of the caption bar will return you to the parent records. You can also return to the parent rows by pressing Alt←. The button at the right end of the caption bar lets you toggle whether the parent record is displayed above the child records.


Employees by Department

Dept_No	Department	Head_Dept	Mngr_No	Budget	Location	Phone_No
180	Marketing	100	(null)	1500000	San Francisc	(415) 555-12
DeptEmpRelation						
130	Field Office:	100	11	500000	Boston	(617) 555-12
140	Field Office:	100	72	500000	Toronto	(416) 677-10
110	Pacific Rim H	100	34	600000	Kuaui	(808) 555-12
120	European He	100	36	700000	London	71 235-4400

Figure 10: The form after you run the application and click the plus sign next to a row in the DataGrid.


Employees by Department

Department Dept_No: 130 Department: Field Office: East Coast Head_Dept: 100 Mngr_No: 11

Emp_No	First_Name	Last_Name	Phone_Ext	Hire_Date	Dept_No	Job_Code
2	Kim	Lambert	22	2/6/1989	130	Eng
11	K. J.	Weston	34	1/17/1990	130	SRep

Figure 11: The form after you click the link to display the employee records.

Conclusion

Untyped *DataSets* provide all the features you need, but using them could be easier. A wizard that would automatically add the *DataTable*, *DataColumn*, and *Constraint* objects would be a big help. One of the advantages of the BDP data providers is that you can see live data a design time. Unfortunately, this doesn't work if you create your *DataTable* objects at design time, because setting the *BdpDataAdapter.Active* property to *True* tries to create the *DataTable*. Because you already created the *DataTable*, setting *Active* to *True* raises an exception.

If you don't create the *DataTable* and *DataColumn* objects, you cannot connect your user interface controls to the *DataSet* at design time. Instead, you'll have to write code to connect them at run time. The lesser of these two evils is clearly to create the *DataTable* and *DataColumn* objects at design time, and forget about viewing live data. Perhaps the next version of Delphi for .NET will be smart enough to fill the *DataTable* if it exists — or create it if it doesn't.

The files referenced in this article are available for download on the Delphi Informant Magazine Complete Works CD located in INFORM\2004\MAY\DI200405BT.

***Bill Todd** is president of The Database Group, Inc., a database consulting and development firm based near Phoenix. He is co-author of four database programming books, author of more than 100 articles, a contributing editor to* Delphi Informant Magazine, *and a member of Team B, which provides technical support on the Borland Internet newsgroups. Bill is also an internationally known trainer and frequent speaker at Borland Developer Conferences in the United States and Europe. Readers may reach him at btarticle@dbginc.com.*

By Xavier Pacheco

Exploring the *System.IO* Namespace

Part II: Streams and Serialization

In Part I of this series I discussed the *Directory* and *File* classes from the *System.IO* namespace. In this installment I'll cover the classes that deal with streaming operations, as well as serialization using formatters.

What Is a Stream?

A stream is a block of data or a sequence of bytes with a beginning and an end. At any given time you can have a reference into the stream, somewhere between its start and finish. A stream can contain any type of information. You can write data into a stream, and read data out of a stream. To manipulate the data contained within a stream, you use a descendant of the base *Stream* class (which is defined in the *System.IO* namespace). The table in Figure 1 lists the various types of streams that exist in the Framework Class Library (FCL).

Class	Description
BufferedStream	Provides a buffering layer to read/write operations on another stream.
CryptoStream	Used to define a cryptographic transform on any other data stream. This class is defined in the *System.Security.Cryptography* namespace.
FileStream	Encapsulates a file with a stream that can be accessed both synchronously and asynchronously.
MemoryStream	Encapsulates a block of memory with a stream.
NetworkStream	Provides streamed access to network resources. The *NetworkStream* class is defined in the *System.Net.Sockets* namespace.
Stream	Base class for the other stream classes.

Figure 1: *System.IO.Stream* implementers.

While streaming classes encapsulate blocks of data, readers and writers encapsulate the functionality to read and write information to and from streams. The table in Figure 2 lists the various reader and writer classes used with streams.

Writing and Reading Text Files with Streams

To illustrate a simple example of working with streams, Figure 3 shows how to use the *FileStream* and *StreamWriter* classes for writing text to a text file. Lines 4-5 declare the *FileStream* and *StreamWriter* classes, both defined in the *System.IO* namespace.

Lines 8-9 create a *FileStream* instance using one of its many overloaded constructors. This constructor takes three arguments: the name of the file to create, *FileMode*, and *FileAccess*. *FileMode* specifies how the operating system

Class	Description
BinaryReader	Provides functionality for reading primitive data types as binary values.
BinaryWriter	Provides functionality for writing primitive data types as binary values.
StreamReader	Implements the *TextReader* class for reading characters from a byte stream.
StreamWriter	Implements the *TextWriter* class for writing characters to a byte stream.
StringReader	Implements the *TextReader* class for reading characters from a string.
StringWriter	Implements the *TextWriter* class for writing characters to a string.
TextReader	An abstract class representing a reader that can read a series of sequential characters.
TextWriter	An abstract class representing a writer that can write a series of sequential characters.

Figure 2: Stream reader and writer classes.

```
 1:  procedure TWinForm.Button1_Click(
 2:    Sender: System.Object; e: System.EventArgs);
 3:  var
 4:    MyFileStream: FileStream;
 5:    MyStreamWriter: StreamWriter;
 6:  begin
 7:    // Create and write the file.
 8:    MyFileStream := FileStream.Create('c:\mydemo.txt',
 9:      FileMode.OpenOrCreate, FileAccess.Write);
10:    try
11:      MyStreamWriter :=
12:        StreamWriter.Create(MyFilestream);
13:      try
14:        MyStreamWriter.BaseStream.Seek(
15:          0, SeekOrigin.End);
16:        MyStreamWriter.WriteLine('Hello Delphi 8');
17:        MyStreamWriter.WriteLine(
18:          'Hello Delphi 8 again');
19:      finally
20:        MyStreamWriter.Close;
21:      end;
22:    finally
23:      MyFileStream.Close;
24:    end;
25:  end;
```

Figure 3: Writing text to a file using *FileStream* and *StreamWriter*.

Value	Description
Append	Opens or creates a file and seeks to the end of the file. Must be used in conjunction with *FileAccess.Write*. Requires *FileIOPermissionAccess.Append*.
Create	Creates a new file, overwriting an existing file. Requires *FileIOPermissionAccess.Write* and *FileIOPermissionAccess.Append*.
CreateNew	Creates a new file. If the file exists, it raises an exception. Requires *FileIOPermissionAccess.Write*.
Open	Opens an existing file. *FileMode.Access* must enable the opening of this file. Raises an exception if the file does not exist. Requires *FileIOPermissionAccess* indicated by the *FileMode* parameter.
OpenOrCreate	Opens the file if it exists. Creates a new file if it does not exist. Requires *FileIOPermissionAccess* indicated by the *FileMode* parameter.
Truncate	Opens an existing file and sets its size to zero. Requires *FileIOPermissionAccess.Write*.

Figure 4: *FileMode* values.

Value	Description
Read	Data can be read from the file.
Write	Data can be written to the file.
ReadWrite	Data can be read from or written to the file.

Figure 5: *FileAccess* values.

should open the file. *FileAccess* specifies the values for read, write, or read/write access to the file once it is opened.

In this instance, the file is opened if it exists, or created if it doesn't exist, as determined by *FileMode.OpenOrCreate*. Additionally, write access is granted as determined by the *FileAccess.Write* parameter. Line 11 creates a *StreamWriter*

```
 1:  procedure TWinForm.Button2_Click(Sender: System.Object;
 2:    e: System.EventArgs);
 3:  var
 4:    MyFileStream: FileStream;
 5:    MyStreamReader: StreamReader;
 6:  begin
 7:    // Create and read from the file.
 8:    MyFileStream := FileStream.Create('c:\mydemo.txt',
 9:      FileMode.OpenOrCreate, FileAccess.Read);
10:    try
11:      MyStreamReader :=
12:        StreamReader.Create(MyFilestream);
13:      try
14:        MyStreamReader.BaseStream.Seek(
15:          0, SeekOrigin.Begin);
16:        while MyStreamReader.Peek <> -1 do
17:          TextBox1.Text := TextBox1.Text +
18:            MyStreamReader.ReadLine +Environment.NewLine;
19:      finally
20:        MyStreamReader.Close;
21:      end;
22:    finally
23:      MyFileStream.Close;
24:    end;
25:  end;
```

Figure 6: Reading text from a file using *FileStream* and *StreamReader*.

Value	Description
Begin	Specifies the beginning of the stream.
Current	Specifies the stream's current position.
End	Specifies the end of the stream.

Figure 7: *SeekOrigin* enumeration values.

instance and associates it with the *FileStream* by passing the *FileStream* instance to the *StreamWriter*'s constructor. The table in Figure 4 lists the various *FileMode* values; the table in Figure 5 lists the various *FileAccess* values.

When working with streams there exists the notion of a stream position. This is the position within the stream to which data is written to, or read from. In Figure 3, this position is at the beginning of a file. This is fine if the code has created a file. However, if the file already exists, it's necessary to relocate the file handle to the end of the file so that further write operations will be appended to the file, instead of overwriting any existing data. This is done on line 14 in Figure 3 using the *Seek* method of the *StreamWriter.BaseStream* property. Finally, text is written to the file using the *StreamWriter.WriteLine* method.

I should note that another way to open the file for write access, without having to invoke the *Seek* method, is to open the file using the *Append* value of the *FileMode* parameter. I didn't do this here so that I could illustrate the use of the *Seek* method.

Figure 6 illustrates the process of reading from a text file using the *FileStream* and *StreamReader* classes. With a few exceptions, Figure 6 looks similar to Figure 3. First, Line 9 shows that the stream is opened for read access. Additionally, lines 16-18 perform a reading operation on the stream by invoking the *StreamReader.ReadLine* method, which, as the name implies, reads a line from the stream. Note the use of the *StreamReader.Peek* method. This method returns the next character to be read from the stream, or -1 to indicate that there are no more characters in the stream.

```
procedure TWinForm.Button3_Click(
  Sender: System.Object; e: System.EventArgs);
const
  cAry: array[1..8] of char =
         ('o', 'h', ' ', 'y', 'e', 'a', 'h', '!');
var
  MyFileStream: FileStream;
  MyBinWriter: BinaryWriter;
begin
  MyFileStream := FileStream.Create('bindemo.dat',
     FileMode.OpenOrCreate, FileAccess.ReadWrite);
  try
    MyBinWriter := BinaryWriter.Create(MyFileStream);
    try
      MyBinWriter.Write(True);
      MyBinWriter.Write(
        'Unten Gleeben Globbin Globin');
      MyBinWriter.Write(23);
      MyBinWriter.Write(23.23);
      MyBinWriter.Write(cAry);
    finally
      MyBinWriter.Close;
    end;
  finally
    MyFileStream.Close;
  end;
end;
```

Figure 8: Writing binary data to a file.

The code in Figures 3 and 6 employs the *SeekOrigin* enumeration with the *Seek* method. This enumeration may be one of the values listed in the table in Figure 7.

Writing and Reading Binary Files with Streams

When writing and reading to/from binary files, you must use a different stream reader and writer. Specifically, you use the *BinaryReader* and *BinaryWriter* classes. Figure 8 illustrates the process of writing different primitive data types to a binary file. To do this, it uses the *BinaryWriter* class' heavily overloaded *Write* method, with each variation writing a different primitive type.

Figure 9 illustrates how to read binary data from a file. In the example, the types Boolean, string, Integer, Double, and an array of *Char* are read from a file. The data contained in the binary file is read using the *BinaryReader* class, which also contains a matching *Read* method for each primitive data type. For illustrative purposes, the data read is used to populate a TextBox control. Note the use of the *MyBinReader.ReadChars* method, which allows you to specify a number of characters to read from the stream into an array of *Char*.

Asynchronous File Access with Streams

Often when processing a file, you want to enable the user to continue using the application. Streams give you the ability to do this, as illustrated in Listing One (on page 20). This example, which is taken from my upcoming book, *Delphi for .NET Developer's Guide*, contains a Windows Form with ListBox and TextBox controls. The example illustrates using a callback function to write data to the ListBox control, and — while doing this — allowing the user to enter text into the TextBox control.

Listing One is a partial listing of the actual example. Stream classes (*FileStream*, *MemoryStream*, etc.) have two methods for asynchronous reading and writing of the stream: *BeginRead* and

```
procedure TWinForm.Button4_Click(
  Sender: System.Object; e: System.EventArgs);
var
  MyFileStream: FileStream;
  MyBinReader: BinaryReader;
  MyCharAry: array of Char;
  i: Integer;
begin
  MyFileStream := FileStream.Create('bindemo.dat',
    FileMode.Open, FileAccess.Read);
  try
    MyBinReader := BinaryReader.Create(MyFileStream);
    try
      TextBox1.Text := TextBox1.Text +
        System.String.Format('Boolean: {0}',
        [MyBinReader.ReadBoolean]) +
        Environment.NewLine;
      TextBox1.Text := TextBox1.Text +
        System.String.Format('String: {0}',
        [MyBinReader.ReadString])+Environment.NewLine;
      TextBox1.Text := TextBox1.Text +
        System.String.Format('Integer: {0}',
        [MyBinReader.ReadByte]) + Environment.NewLine;
      TextBox1.Text := TextBox1.Text +
        System.String.Format('Double: {0}',
        [MyBinReader.ReadDouble])+Environment.NewLine;
      MyCharAry := MyBinReader.ReadChars(8);
      for i := Low(MyCharAry) to High(MyCharAry) do
        TextBox1.Text := TextBox1.Text + MyCharAry[i];
    finally
      MyBinReader.Close;
    end;
  finally
    MyFileStream.Close;
  end;
end;
```

Figure 9: Reading binary data from a file.

BeginWrite. The use of *BeginRead* is shown on lines 41 and 42 of Listing One. *BeginRead* takes the parameters listed in Figure 10.

To begin asynchronous file access, you must invoke the *BeginRead* or *BeginWrite* methods on a specified stream, passing the parameters listed in Figure 10. Notice that the example in Listing One passes the callback function *MyCallBack* to the *BeginRead* method. Also note that the *TFileState* object is passed as the last parameter. This object can be accessed within the callback function, as illustrated in lines 48-60.

The callback function is called once, after the stream has been read from. To illustrate that the user still has access to the application, *MyCallBack* simply performs a loop 10 times and writes a string to the ListBox control. It then pauses for about 1.5 seconds after each write. Also, notice how the *TFileState* instance is accessed from the *Result.AsyncState* property. The *Result* parameter is declared as an *IAsyncResult* interface. (Because this example assumes it exists, you need to make sure there is a file named demo.txt on your C: drive.)

Serialization and Deserialization

Having the ability to save and retrieve data to and from files or other mediums can be very useful. Beyond this, however, is the ability to save and retrieve your .NET objects, including their state. Serialization is the process of converting objects into a stream of bytes that can be persisted onto a medium, or transferred to another process — even across a network to another computer (this is known as remoting). Deserialization is the reverse process of reading the stream of bytes, reconstructing the serialized object and its state.

Parameter	Description
Buffer	Represents the buffer into which data will be read.
Offset	A byte offset in the buffer; the starting point to which data will be written.
Count	Maximum bytes to read from the stream.
Callback	A callback function that's invoked when the read operation is complete.
State	An object used to distinguish read requests.

Figure 10: *BeginRead* parameters.

You must perform one of two tasks to make serialization work. The class that you want to serialize can be declared with the [Serializable] attribute, or the class can implement the *ISerializable* interface. This article will discuss the former, easier method. Examine the following class declaration:

```
[Serializable]
TEmployee = class(System.Object)
public
  FirstName: string;
  LastName: string;
  HireDate: DateTime;
end;
```

The CLR (Common Language Runtime) knows how to handle the base class and any other classes (types) that are serializable, such as those declared in *TEmployee*. You can prevent a field from being serialized by defining it with the [NonSerialized] attribute, as shown here:

```
[NonSerialized]
LastName: string;
```

Because of how the CLR serializes classes, it isn't necessary for the CLR to serialize associated classes in any particular order. The reconstruction process will take care of reestablishing the original associations.

Once a class is declared as being serializable, one must determine the format to which the class will be persisted. This is done using a formatter, or a class that implements the *IFormatter* interface. There are two pre-defined formatters: *BinaryFormatter* (defined in the *System.Runtime.Serialization.Formatters.Binary* namespace) and *SoapFormatter* (defined in the *System.Runtime.Serialization.Formatters.Soap* namespace). To use *BinaryFormatter*, simply add its namespace to your **uses** clause. To use *SoapFormatter*, add the namespace and the reference to the *System.Runtime.Serialization.Formatters.Soap.dll* assembly through the Delphi 8 Add Reference dialog box.

Figure 11 illustrates the process of serializing and deserializing an array of the *TEmployee* class previously declared. A .NET Array type is already serializable; therefore, it's not necessary to declare it with the [Serializable] attribute. Lines 14-20 simply create two *TEmployee* instances, and populate the array with them. Lines 22-28 perform the process of serializing the *EmpAry* object. Note how this is done by passing the *FileStream* and the object to serialize to the *SoapFormatter.Serialize* method. After this step, you should be

```
 1:  procedure TWinForm.Button1_Click(
 2:    Sender: System.Object; e: System.EventArgs);
 3:  const
 4:    c_fmt = 'First name: {0}, Last name {1}, ' +
 5:            'Hire Date {2}';
 6:  var
 7:    fs: FileStream;
 8:    soFmt: SoapFormatter;
 9:    Emp: TEmployee;
10     EmpAry: TEmployeeArray;
11:    i: Integer;
12:  begin
13:    // Create the classes and serialize them.
14:    SetLength(EmpAry, 2);
15:    Emp := TEmployee.Create('Xavier', 'Pacheco',
16:             System.DateTime.Today);
17:    EmpAry[0] := Emp;
18:    Emp := TEmployee.Create('Frank', 'Smith',
19:             System.DateTime.Today);
20:    EmpAry[1] := Emp;
21:
22:    fs := FileStream.Create('emp.xml', FileMode.Create);
23:    try
24:      soFmt := SoapFormatter.Create;
25:      soFmt.Serialize(fs, EmpAry);
26:    finally
27:      fs.Close;
28:    end;
29:
30:    // Read Back Serialized Information.
31:    fs := System.IO.File.OpenRead('emp.xml');
32:    try
33:      soFmt := SoapFormatter.Create;
34:      EmpAry := TEmployeeArray(soFmt.Deserialize(fs));
35:      for i := Low(EmpAry) to High(EmpAry) do
36:        ListBox1.Items.Add(System.String.Format(c_fmt,
37:          [EmpAry[i].FirstName, EmpAry[i].LastName,
38:          EmpAry[i].HireDate]))
39:    finally
40:      fs.Close;
41:    end;
42:  end;
```

Figure 11: Serialization/deserialization example.

able to load the emp.xml file and view the resulting XML. Lines 31-41 in Figure 11 recover the information from the file, and reconstruct the *EmpAry* instance by invoking the *SoapFormatter.Deserialize* method, which returns a *System.Object* type. Therefore, it must be hard-cast to the type being deserialized. Although not shown here, keep in mind that it is possible for the *Deserialize* method to return **nil**. Therefore, it's recommended that a proper test is performed to avoid a *NullReferenceException* from occurring.

There is much that can be discussed about serialization that is beyond what can be covered in this article. Look for more in-depth articles about this topic in future issues of *Delphi Informant*.

The three projects referenced in this article are available for download on the Delphi Informant Magazine Complete Works CD located in INFORM\2004\MAY\DI200405XP.

Xavier Pacheco *is the president of Xapware Technologies Inc., provider of Active! Focus — a practical solution for managing software projects and requirements management. Xavier is the co-author of* Delphi 6 Developer's Guide *and the upcoming* Delphi for .NET Developer's Guide. *Xavier is available for consulting and training engagements. You may contact him at xavier@xapware.com or visit his company Web site at www.xapware.com.*

Begin Listing One — Asynchronous File Access

```
  TWinForm = class(System.Windows.Forms.Form)
  strict private
    procedure Button1_Click(Sender: System.Object;
      e: System.EventArgs);
  private
    procedure MyCallback(Result: IAsyncResult);
  end;

  TByteArray = array of Byte;

  TFileState = class(System.Object)
  public
    FFilePath: string;
    FByteArray: TByteArray;
    FFStream: FileStream;
  public
    constructor Create(aFilePath: string;
      aByteArraySize: Integer;
      aFileStream: FileStream);
    property FilePath: string read FFilePath;
    property FStream: FileStream read FFStream;
    property ByteArray: TByteArray read FByteArray;
  end;

implementation

procedure TWinForm.Button1_Click(
  Sender: System.Object; e: System.EventArgs);
var
  MyFileStream: FileStream;
  MyFileState: TFileState;
  MyFileInfo: FileInfo;
begin
  MyFileStream := FileStream.Create('c:\demo.txt',
    FileMode.Open, FileAccess.Read, FileShare.Read,
    1, True);
  try
    MyFileInfo := FileInfo.Create('c:\demo.txt');
    MyFileState := TFileState.Create('c:\demo.txt',
      MyFileInfo.Length, MyFileStream);
    MyFileStream.BeginRead(MyFileState.ByteArray, 0,
      MyFileInfo.Length, MyCallBack, MyFileState);
  finally
    MyFileStream.Close;
  end;
end;

procedure TWinForm.MyCallBack(Result: IAsyncResult);
var
  MyState: TFileState;
  i: Integer;
begin
  MyState := Result.AsyncState as TFileState;
  for i := 1 to 10 do begin
    ListBox1.Items.Add(System.String.Format(
      'In callback for File {0}',[MyState.FilePath]));
    Thread.Sleep(1500);
  end;
  MyState.FStream.Close;
end;

{ TFileState }
constructor TFileState.Create(aFilePath: string;
  aByteArraySize: Integer; aFileStream: FileStream);
begin
  inherited Create;
  FFilePath := aFilePath;
  SetLength(FByteArray, aByteArraySize);
  FFStream := aFileStream;
end;
```

End Listing One

By Fernando Vicaria

RSA Encryption

Part I: From Prime Numbers to RSA Encryption

This article initiates a two-part series where we take a look at encryption, but from a very different perspective. Starting with the simple concept of prime numbers, we'll build a solid theoretical basis that culminates with the implementation of our own encryption system — one that can potentially be made as secure as anything available in the software industry today. To get from here to there we'll use a considerable amount of math, although you should've seen most — if not all — of it in high school.

We'll wrap up this topic in Part II with an implementation for a large integer type that is limited only by the memory available in the system. With this type we'll be able to create and use integers of literally astronomical size. This new type will be used in your own implementation of the RSA encryption algorithm. (By the way, the three-letter acronym, RSA, is a result of the names of its inventors: Ron Rivest, Adi Shamir, and Leonard Adleman.)

For those of you who are not math inclined, or who are not really into the historical and theoretical aspects, feel free to jump ahead to the implementation section. This series is probably of most significance to those of you working with encryption or security in general. Having a detailed knowledge of how things really work (and why) is essential if you intend to keep your security systems one step ahead of hackers.

"Having a detailed knowledge of how things really work is essential if you intend to keep your security systems one step ahead of hackers."

What Are Prime Numbers?

Prime numbers have been of interest since the ancient Greek philosophers. However, they were not of interest for theoretical purposes until last century, when those studying cryptography began using them.

Let's start with the simplest definition of a prime number. In book VIII of *Elements*, Euclid defined a *prime* as a number that has no whole-number divisors other than 1 and the number itself. A few examples would be 2, 3, 5, 11, 13, etc. On the other hand, a number that is not a prime is said to be a *composite*.

Another important definition is that of *relative primes*. A prime number is considered a relative prime to any other number if it does not divide that number. In other words, if a prime number p is relative prime to a number n then: $n \bmod p \neq 0$.

From the above definitions we can extract the following results, or *propositions* as Euclid called them:

1) If a prime number p divides a product mn then p divides at least one of the two numbers m, n (see Figure 1).
2) Every natural number is either prime or else can be expressed as a product of primes in a way that is unique, apart from the order in which they are written.
3) There are infinitely many prime numbers.

Proposition 1
If a prime number p divides a product mn then p divides at least one of the two numbers m, n.

Proof
Let the two numbers m and n multiplied by one another make another number o, and let any prime number p divide o.

We must prove that p divides m or n.

Let's assume that p does not divide m, therefore p and m are relative primes.

Now let's make another number e be such as $e = mn$ div p (or the number of units that p divides mn).

Since p divides mn according to e, then $pe = mn$. Therefore, $p/m = n/e$.

But p is relative prime to m. Therefore, they are the least of those that have the same ration with them. And the least divides the numbers that have the same ratio. That is, p divides n (and m divides e).

Similarly, we can also prove that if p does not divide n then it must divide m.

Q.E.D.

Figure 1: Product factorization with a prime.

As promised, I won't leave you wondering (how or why) and will now prove each one of these results so that we can freely use them later. Feel free to skip the proofs if you want to take Euclid's word for granted.

Proposition 2 (see Figure 2) is also known as the Fundamental Theorem of Arithmetic. Proposition 1 immediately implies the Fundamental Theorem of Arithmetic, although Euclid never stated it explicitly. The first time it was formulated was in 1801 by Gauss in his *Disquisitiones Arithmeticae*.

We'll prove Proposition 2 in two steps. First we'll prove that a number can always be factored into primes, and then we'll show that this process is unique; that is, there is only one prime factorization for each number. Once again, feel free to skip it.

Taken together, the first two propositions form the building blocks of all natural numbers, much like the physicist's atoms. Knowing the prime number factorization of a number gives complete knowledge about all factors of that number.

Now that we've learned some of the fundamental properties of prime numbers we need to learn how to find them, and — most importantly — how to prove they are in fact primes.

Finding Primes

How do we go about finding primes? There are many ways to find primes, and their efficiency usually depends

Proposition 2
Every natural number is either prime or can be uniquely factored as a product of primes in a unique way.

Proof
Let's prove this using contradiction.

Let's assume there are numbers that cannot be written as a product of primes. Therefore, there must exist the smallest of such numbers; we'll call it n.

By definition n must be a natural number greater than 1. So $n = ab$, where a and b are natural numbers such as $a > 1$ and $b < n$.

Therefore, a and b are smaller than n, so a and b can be factored into primes.

By substitution n can be factored by primes. This contradicts the assumption that there exists composite numbers that cannot be factored into primes.

Conclusion: All natural numbers are either prime or can be written as a product of prime numbers.

Now let's prove that this product is unique, again using contradiction.

Let's assume there are numbers that can be factored into primes in at least two distinct ways. Then there must exist the smallest of such numbers, n, where n can be represented by:

$n = p_1 p_2 \ldots p_m$ or $n = q_1 q_2 \ldots q_n$

Let's now select one of the members on the left side, p_1.

We can say that p_1 divides n and therefore it divides $q_1 q_2 \ldots q_n$.

That means that $n / p_1 = q_1 q_2 \ldots q_n / p_1$.

But from Proposition 1, if p_1 divides the product $q_1 q_2 \ldots q_n$ then it must divide at least one of its members. Since all members on the right side of the equation are primes we must have that for a q_j (with $1 \leq j \leq n$) the following equality holds:

$p_1 = q_j$

We can, without loss of generality, assume $j = 1$. Therefore, $p_1 = q_j$.

The result of n/p_1 is a number smaller than n and consequently, from our initial assumption, it cannot be factorized in more than one unique way. Therefore, the sequences $p_2 \ldots p_m$ and $q_2 \ldots q_n$ contain the same primes, possibly different only in their order.

Conclusion
There is only one way to factorize a number into primes.

Q.E.D.

Figure 2: The Fundamental Theorem of Arithmetic.

Proposition 3
There are infinitely many prime numbers.

Proof
Let's start with the list: p_1, p_2, p_3 ... of all known primes. We must prove that this list continues forever.

Let's assume that we have listed all primes up to some p_m.

Now consider a number p such as:

$$p = p_1 p_2 \ldots p_m + 1$$

If p happens to be a prime, then p is a prime bigger then p_m and the list can continue (note that p might not be the next prime after p_m, in which case p cannot be taken to be p_{m+1}).

If p is not a prime, then it must be evenly divisible by a prime (Proposition 2). But none of the primes $p_1, p_2, p_3 \ldots p_m$ divides p. If you carry out such division you always end up with a remainder of 1.

Therefore, p must be divisible by a prime bigger than all primes up to p_m.

Conclusion
We can always find a bigger prime for any given list $p_1\ p_2 \ldots p_m$.

Q.E.D.

Figure 3: The infinitude of primes.

on the size of the primes in which we are interested, or on the probability of these numbers truly being primes (see Figure 3).

Eratosthenes came up with an algorithm in 280 BC, known today as the Sieve of Eratosthenes, for finding prime numbers. To verify that n is a prime, you look at all numbers from 2 to n. Then you eliminate all multiples of 2 up to n. Then you go on to 3 and do the same, and then the next number, which is still available, etc. If n is still left when you reach the square root of n, then n is a prime. Although the numbers resulting from this method are sure to be primes, this algorithm is very slow and has exponential complexity, meaning it runs exponentially in the number of digits in the number n.

Another good method for finding small primes is the Trial Division method, which, as the name suggests, is based on the trial division of a given number n by all natural numbers from 2 to $\sqrt{n}$. If we find a factor the number is a composite, otherwise it's a prime.

Figures 4 and 5 show Delphi implementations of the Sieve of Eratosthenes and Trial Division methods.

Mathematicians for centuries have been trying to find equations for generating prime numbers, but their success has been slow and valid to only a small range of natural

```
program Eratosthenes;

{$APPTYPE CONSOLE}

uses
  SysUtils, Windows;

var
  s: string;
  i, j, p: LongWord;
  Range, Count, Width: LongWord;
  Start, Finish: LongWord;
  a: array of LongWord;
begin
  // This is a simple prime-number generator that can
  // generate all the primes <= 2147107031 = 2^31 - 376617.
  // This is the maximum range you can get with the
  // provided RTL/VCL units.
  Write('Enter range (from 0 to 2147483647): ');
  ReadLn(s);
  Range := StrToInt(s);
  Width := Length(s);
  Count := 0;
  Start:= GetTickCount;
  SetLength(a, Range + 1);
  // 0 and 1 are not considered prime by definition.
  a[0] := 0;
  a[1] := 0;
  for i := 2 to Range do
    a[i] := 1;
  p := 2;
  while p < Range do begin
    j := 2 * p;
    while (j  <= Range) do begin
      a[j] := 0;
      j := j + p;
    end;
    repeat
      p := p + 1;
    until a[p] = 1;
  end;
  for i := 2 to Range do
    if a[i] <> 0 then begin
      Write(i : Width, ' ');
      Inc(Count);
    end;
  WriteLn;
  WriteLn('Total no. are ' + IntToStr(Count));
  WriteLn;
  Finish := GetTickCount - Start;
  WriteLn(FloatToStrF(
    Finish/1000, ffFixed  , 0, 2) + ' seconds');
  ReadLn;
end.
```

Figure 4: Sieve of Eratosthenes method.

numbers. Leonard Euler, the Swiss mathematician, came up with the following equation:

$f(x) = x^2 + x + 41$, which is prime for $x = 0, 1, 2, .., 39$

This quadratic was the record holder for centuries as a consecutive, distinct quadratic prime-producer for an initial range of input values. It is not, however, the current record holder. That distinction goes to this function:

$f(x) = 36x^2 - 810x + 2753$, which is prime for $x = 0, 1, \ldots, 44$

As you have probably noticed, these functions are also very limiting in their range and are of very little or no practical use.

```
program TrialDiv;

{$APPTYPE CONSOLE}

uses
  SysUtils, Windows;

var
  s: string;
  i, j: LongWord;
  Factors: Boolean;
  Range, Count, Width: LongWord;
  Start, Finish: LongWord;
begin
  Write('Enter range (from 0 to 2147483647): ');
  ReadLn(s);
  Range := StrToInt(s);
  Width := Length(s);
  Count := 0;
  Start:= GetTickCount;
  // Try all numbers from 2 (1 is not considered prime)
  // up to Range.
  for i := 2 to Range do begin
    // Reset factors count.
    Factors := False;
    // Try dividing it by 2 up to the square root of i.
    for j := 2 to Trunc(sqrt(i)) do
      if (i mod j = 0) then begin
        Factors := True;
        Break;
      end;
    // If number has no divisors then it's PRIME!
    if not Factors then begin
      Write(i : Width, ' ');
      Inc(Count);
    end;
  end;
  WriteLn;
  WriteLn('Total no. are ' + IntToStr(Count));
  WriteLn;
  Finish := GetTickCount - Start;
  WriteLn(FloatToStrF(
    Finish/1000, ffFixed  , 0, 2) + ' seconds');
  ReadLn;
end.
```

Figure 5: Trial Division method.

To be of any real use, we need a method that can produce an infinite number of primes, and of any size we want. The best way found to deal with this problem, since there are no formulas for finding all prime numbers in sequence, was to instead find numbers that are "very likely" to be primes.

Primality Test

If we are going to use our primes for industrial uses (e.g. encryption) we often do not need to prove they are prime. It may be enough to know that the probability they are composite is less than a given percentage (e.g. 0.000000 000000000000000001%). In this case we can use (strong) probable primality tests.

Most of these tests are based on what is known as Fermat's Little Theorem, which states:

If $\boldsymbol{p}$ *is a prime and if* $\boldsymbol{a}$ *is any integer, then* $a^p = a \pmod p$. *In particular, if* $\boldsymbol{p}$ *does not divide* $\boldsymbol{a}$, *then* $a^{p-1} = 1$.

Figure 6 presents a proof to Fermat's Little Theorem. It's important to note that Fermat's theorem is a composite-

Proposition
If p is a prime and if a is any integer, then $a^p = a \pmod p$. In particular, if p does not divide a, then $a^{p-1} = 1$.

Proof
Start by listing the first $p-1$ positive multiples of a:

$a, 2a, 3a, \ldots (p-1)a$

Suppose that ra and sa are the same modulo p, then we have $r = s \pmod p$, so the $p-1$ multiples of a above are distinct and nonzero; that is, they must be congruent to 1, 2, 3, ..., $p-1$ in some order. Multiply all these congruences together and we find:

$a.2a.3a.....(p-1)a = 1.2.3.....(p-1) \pmod p$

or better:

$a^{(p-1)}(p-1)! = (p-1)! \pmod p$.

Divide both sides by $(p-1)!$ to complete the proof.

Conclusion
There is only one way to factorize a number into primes.

Q.E.D.

Figure 6: Fermat's Little Theorem.

ness test, and not a primality test; that is, it tells you for sure if a number is a composite or not a prime. If the test is positive the number is guaranteed to be a composite, otherwise it might be a prime or not. However, statistically speaking, this method is good enough for many practical applications, including encryption.

Some early articles call all numbers satisfying this test pseudo-primes, but now the term pseudo-prime is properly reserved for composite probable-primes.

There are 1,091,987,405 primes less than 25,000,000,000; but only 21,853 pseudo-primes base two, which gives us an error probability less than 0.00009%. We can reduce this margin of error even further by using multiple bases. This is exactly what the Miller-Rabin method does (see Figure 7).

The Miller-Rabin primality test gives us a proved probability of error less than $1/s^{-2}$ where s is the number of bases we tried.

Factoring Primes

The dual problems of factoring integers and testing primality have surprisingly many applications for a problem long suspected of being only of mathematical interest. As we'll see below, the security of the RSA public-key cryptography system is based on the computational intractability of factoring large integers. (Note: As a more modest application, hash table performance typically improves when the table size is a prime number. To get this benefit, an initialization routine must identify a prime near the desired table size.)

```
function MillerRabin(const n: TLargeInteger;
  const s: Integer): Boolean;
var
  i: Integer;
  r: TLargeInteger;
begin
  // n must be odd.
  if n mod 2 = 0 then begin
    Result := False;
    Exit;
  end;
  for i := 1 to s do begin
    // Get a random number from the set (2, 3,...,n+1).
    r := IRandomRange(n) + 2;
    // Should use Witness here but is not correct yet.
    if not PseudoPrime(n, r) then begin
      Result := False;
      Exit;
    end;
  end;
  Result := True;
end;
```

Figure 7: The Miller-Rabin primality test.

As you would imagine, factoring and primality testing are related problems. However, they are quite different algorithmically. In the sections above, we saw that we can demonstrate that an integer is composite (i.e. not prime) without actually giving the factors. To convince yourself of the plausibility of this, note that you can demonstrate the compositeness of any nontrivial integer whose last digit is 0, 2, 4, 5, 6, or 8 without doing the actual division.

The simplest solution to factorization is also the simplest one used in primality tests; that is, brute-force trial division. However, this is extremely slow for big numbers and therefore impractical for any real application.

Another solution, a bit faster than trial-division, is the Square Root Method. Figure 8 shows one of the possible implementations for this method. It's okay for small numbers (15 digits or less), but will become too slow for anything larger than that. (Note: For encryption purposes, anything less than 150 digits is considered small.)

TLargeInteger is any valid implementation of a large integer type. *ISqrt(n)* returns the integer part of the square root of ***n*** and *ModExp(n,e,p)* raises the number ***n*** to the ***e*** power, then calculates the mod ***p*** of the result.

The fastest known algorithm used in factorization is the Number Field Sieve. It uses randomness to construct a system of congruences, the solution of which usually gives a factor of the integer. The method was developed by Pollard and was used to factor the RSA-130 number (such a feat required enormous amounts of computation).

As you can see, the strength of encryption algorithms such as RSA relies on the fact that it's really hard to decompose a large number into its factors. These algorithms use an integer ***n*** made of two or more primes, such as: ***n*** = ***p*** x ***q*** or ***n*** = ***p*** x ***q*** x ***r*** x ***s*** where ***p***, ***q***, ***r***, and ***s*** are large prime numbers. Breaking ***n*** into its factors requires an enormous amount of computation that can last for many years (even when using multiple machines).

```
// Returns the first factor found for any given number
// lying in the sequence { 2,3, 6k±1 } where k =1 to
// sqrt(n) if the number is a composite; n if the number
// is prime, 0 if the number is less than 2.
function Factor(const n: TLargeInteger): TLargeInteger;
var
  i, t: TLargeInteger;
begin
  if n < 2 then begin
    Result := 0;
    Exit;
  end;
  // If it's even and greater than 3 then it's not prime.
  if ModExp(n, 1, 2) = 0 then begin
    Result := 2;
    Exit;
  end;
  // Try dividing it by 3 up to the square root of i.
  t := ISqrt(n);
  i := 3;
  while i <= t do begin
    if ModExp(n, 1, i) = 0 then begin
      Result := i;
      Exit;
    end;
    i := i + 2;
  end;
  // Number is prime.
  Result := n;
end;
```

Figure 8: Integer factorization using the Square Root Method.

To demonstrate how secure such algorithms can be, cash prizes are usually offered to those who manage to factorize these numbers.

Message Encryption

This huge disparity between the ease of finding large prime numbers, and the difficulty of factoring large numbers, is fully exploited when devising secure forms of public key cipher systems.

Figure 9 illustrates a typical modern cipher system used for encrypting messages that have to be sent over insecure electronic communication channels. The basic components of the system are two programs, an *encryptor* and a *decryptor*. Security for the sender and the receiver is achieved by requiring a key for both encryption and decryption.

Typically the key will consist of one hundred or more digits. The security of the system depends on keeping the key secret. This method immediately raises the following question: How can we send the key to the receiver of our messages? If we decide to mail it we need to trust the carrier, and having to always physically meet the receiver may not be practical or cost efficient.

To resolve this problem, Whitfield Diffie and Martin Hellman in 1975 proposed the idea of a public key cryptosystem (PKS). In a PKS, each potential message receiver ***A*** (which would be anyone who intends to use the system) uses a program that will produce not one, but two keys: an encryption key and a decryption key. The encryption key is made public to all those interested in sending ***A*** a message, while the decryption key is kept secret.

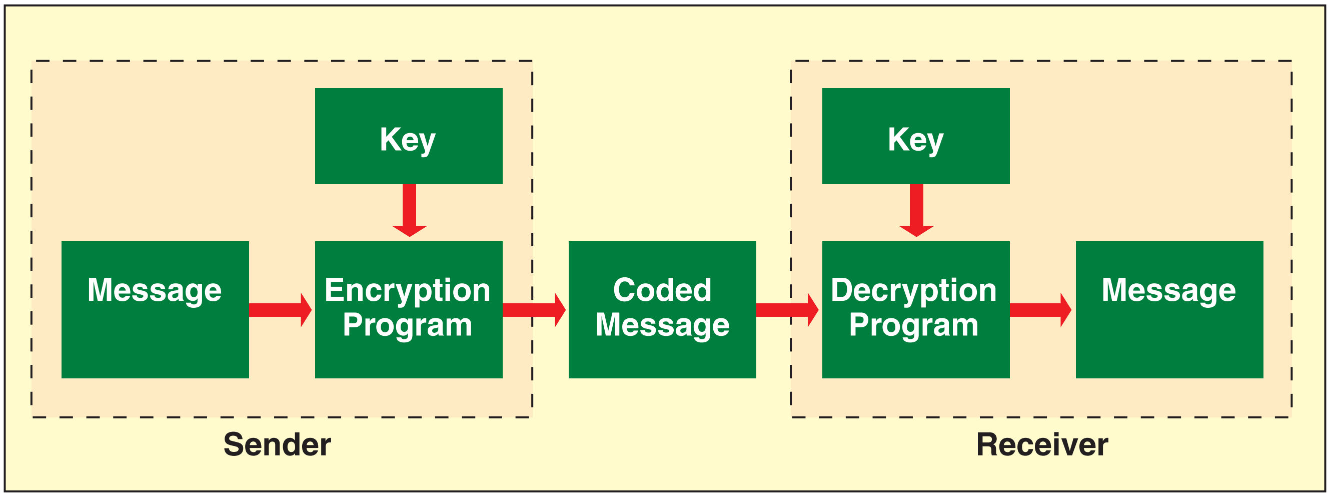


Figure 9: A typical public-key encryption system.

Although the basic idea behind such a system is simple, designing it is not. The one originally proposed by Diffie and Hellman turned out to be not as secure as they had thought, but a method devised a short time later by Rivest, Shamir, and Adleman proved to be much more robust, and the RSA system, as it's known, is now used widely in international banking, online transactions, and military and satellite communications, to name just a few.

The main difficulty for anyone designing such a system is that the encryption process should disguise the message to such an extent that it is impossible to decode it without the decryption key. But because the essence of the system, and indeed of any cipher system, is that the authorized receiver *can* decrypt the encoded message, the two keys must be mathematically related. The receiver's program and decryption key should *exactly undo* the effect of the sender's program and encryption key, so it should be theoretically possible to obtain the decryption key from the encryption key, provided one knows how the cipher programs work.

"Prime numbers have been of interest since the ancient Greek philosophers."

The trick is to ensure that, although it is *theoretically* possible to recover the decryption key from the publicly available encryption key, it is *practically* impossible. In the case of one-way ciphers such as RSA, the receiver's secret decryption key consists of two large prime numbers (at least 75 digits each), and the public key consists of their product (a 150-digit number).

With current technology, it's practically impossible to factor numbers of that size in a reasonable time. As I mentioned in the previous section, efforts to develop new algorithms to factorize such numbers are usually subjects for doctorate theses and can bring instant fame for those who succeed. The current record for factorization is around 155 digits. (Note: New algorithms designed for quantum computers, when and if they ever get to build one, may render this sort of encryption system obsolete, but until then RSA will continue to be one of the safest encryption systems available.)

The RSA Algorithm

With the facts presented up to this point we can now start a formal description of the algorithm behind RSA. The best way to describe it is using the simplest case possible: Where a sender, call it ***S***, wants to send a message to a receiver, call it ***R***. For the particular case of this example we will assume that the message is a two-digit integer.

Before it can receive secure messages from ***S*** or anyone else interested in sending messages, ***R*** will need to generate a public key. To do this ***R*** must find two large prime numbers, call them ***p*** and ***q***, having at least 75 digits, then multiply them to make a number ***n*** such as ***n*** = ***p.q***. This number will be ***R***'s public key (***p*** and ***q*** should be kept secret). Besides the generation of the public key, ***R*** should agree with ***S*** on a method to encode the actual message text.

We will now break the process of encrypting and decrypting the message into steps:

- After ***R*** selected the two primes. We'll use ***p*** = 17 and ***q*** = 31 for this example, but keep in mind that in real-life situations this number must be at least 75 digits.
- ***R*** now generates ***n*** by multiplying ***p*** and ***q***. Such as ***n*** = 17x31; that is ***n*** = 527. This will be ***R***'s public key.
- ***R*** now chooses another number ***e*** which must be relative prime to (***p*** - 1)(***q*** - 1), which is 16x30 or 480. By the definition of relative prime presented earlier in this article we can quickly find a few suitable candidates: 7, 11, 13, 14, etc. Note that ***e*** doesn't need to be a prime, or an odd number. Finding ***e*** comes down to finding a number ***e*** such as the Greatest Common Divisor between ***e*** and 480 is 1. That is *GCD*(***e***, 480) = 1. Let's pick ***e*** = 11 for this example. The number ***e*** is also part of the public key and therefore ***R*** will have to make it public together with ***n***.
- With the knowledge of ***n*** and ***e*** the sender ***S*** can now prepare to send the message to ***R***. Let's call this message ***M***. The message in our case is a single two-digit number. For example: ***M*** = 70.

- ***S*** now encrypts the message by applying a sequence of mathematical operations over ***M*** that will result in the encrypted message ***C***:

  $C = M^e \bmod n$, which for this example becomes
  $C = 70^{11} \mathit{mod}\ 527$ or $C = 8$.

- Now ***R***, after receiving the encrypted message ***C***, must decrypt it using the secret pair of prime numbers used to create the public key. ***R*** must first find the number ***d*** such that:

  $e.d\ \mathit{mod}\ (p - 1)(q - 1) = 1$ or in our case
  $11.d\ \mathit{mod}\ 480 = 1$.

  Once again we can quickly find some suitable candidates for ***d***, for example: ***d*** = 131 satisfies this condition. The problem of finding ***d*** is equivalent to finding the modular inverse of a number.

- Finally, to decode the message ***R*** applies a sequence of mathematical operations on ***C*** as follows:

  $M = C^d\ \mathit{mod}\ n$, which for this example becomes
  $M = 8^{131}\ \mathit{mod}\ 527$ or $M = 70$.

As you can see, as long as ***R*** can keep the original two primes used to generate the public key a secret he can be sure that messages addressed to him cannot be decoded and read or tampered with by anyone else.

Conclusion and Scenes from Part II

Now that we've passed all the theoretical hurdles, it's time to create our own implementation of the RSA algorithm. But first we must create a new integer type capable of holding numbers with 100 digits or more. Otherwise we would end up with an encryption system that could be broken in a matter of days, or even hours. We'll leave these and a few other practical details such as signing and verifying messages for next month.

In this article we saw how encryption algorithms based on public keys, such as RSA, rely on the fact that it's really difficult to break up a large integer into all of its factors. Although finding if a number is a composite or a prime is a relatively quick task, finding the actual factors for a composite can take a long time. The methods presented here can be easily implemented in Delphi or any other language, and require only the capability of handling large integers (something missing in the current RTL version).

“Eratosthenes came up with an algorithm in 280 BC for finding prime numbers.”

In Part II we'll see how to put into practice all the issues discussed here, and we'll see how to create our own support for large integers and RSA. We'll conclude this series by building a nice little demo application that can be used in your real-life projects.

Further Reading

- The Prime Pages: www.utm.edu/research/primes
- Prime Numbers Generator: http://bille.cudenver.edu/apps/primes
- *The Code Book* by Simon Singh (Anchor, 2000)
- *Introduction to Algorithms, 2nd Edition* by Thomas H. Cormen (MIT Press, 2001)

Fernando Vicaria *is a Senior Software Engineer currently based in Santa Cruz, CA. He was an active member of the development team for C++Builder, Delphi, and C#Builder at Borland. He is also a freelance technical author for Delphi and C++Builder issues. Fernando specializes in VCL and .NET frameworks. When he's not at work he's probably surfing at some secret spot in Northern California. He can be reached via e-mail at fernando@vicaria.com.*

INFORMANT SPOTLIGHT

ADO.NET ■ BORLAND DATA PROVIDERS ■ DELPHI 7, 8

By Glenn Stephens

Moving to ADO.NET

From *TDataSet* to *System.Data.DataSet* with Delphi 8

When you've been working with Delphi and the VCL for a while — as most of you undoubtedly have — you've probably picked up a number of tricks and techniques that make your database applications faster to build. With the move to .NET, however, you're going to need a new bag for the new tricks you're going to learn with Delphi 8 for the Microsoft .NET Framework.

For example, you may be wondering how to create calculated fields with ADO.NET, because in Delphi 7 you just *did it*. Granted, there are plenty of other tips and techniques you can use to build powerful database applications with the VCL, but by the end of this article you'll be using ADO.NET on the .NET platform with Delphi 8 for .NET to do the things you're accustomed to doing with Delphi 7.

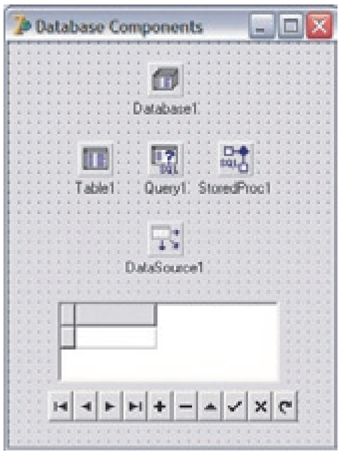


Figure 1: A typical database connection in Delphi 7.

We'll cover some of the basics of dealing with data with ADO.NET, the Borland Data Providers, accessing data, creating master-detail relationships, accessing data from code, creating calculated fields, and validating data. Let's start by looking at how database connections work in ADO.NET.

A Connectionless World

If you're familiar with Delphi's BDE, dbExpress, ADO, or IBExpress components, you'll need a connection component that connects directly with the data (see Figure 1). The big difference between this style of connection and the connection model in ADO.NET is that from the very start, ADO.NET is designed to not have the database connection component.

For those of you familiar with the MIDAS/DataSnap model, you'll find ADO.NET has similar connection styles, but works with different classes. Looking at Figure 2, you'll see that the Connection component connects to the physical database, the DataAdapter component connects to the database, and the DataAdapter is responsible for filling the contents of the dataset — as well as applying changes to the database, based on the modified contents of the dataset.

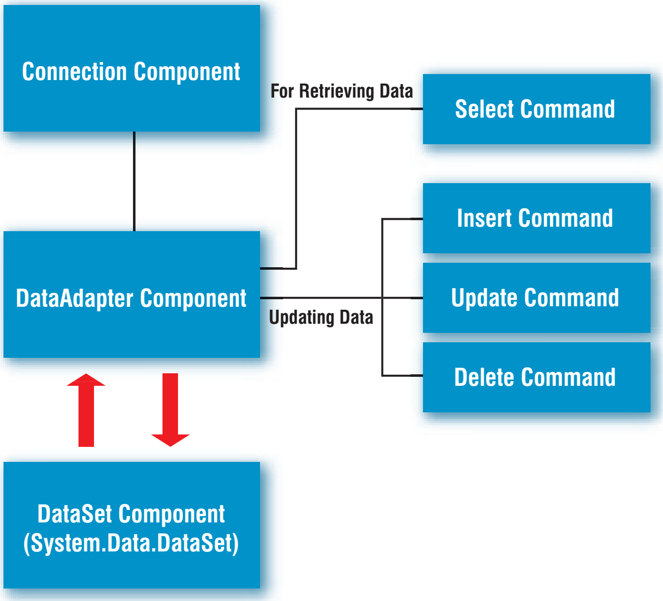


Figure 2: The connection model for ADO.NET.

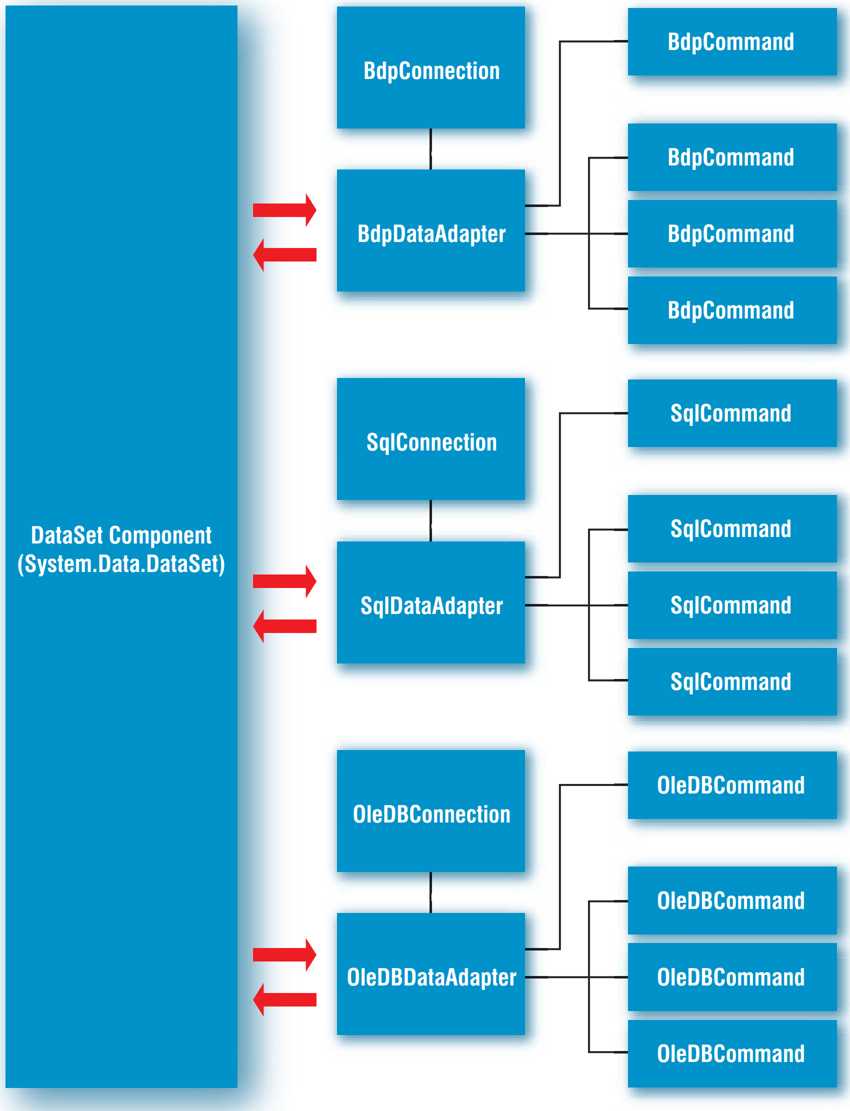


Figure 3: Various data providers for ADO.NET.

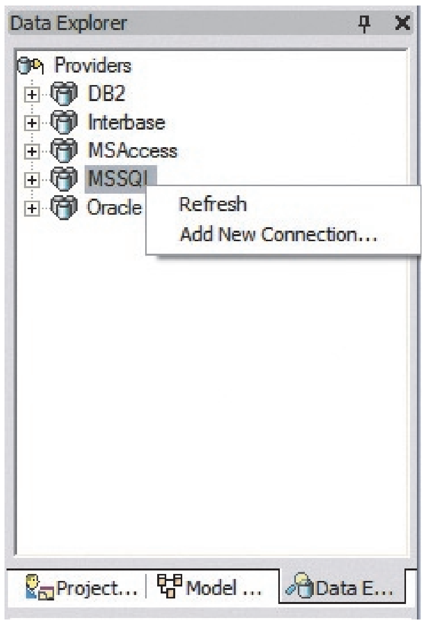
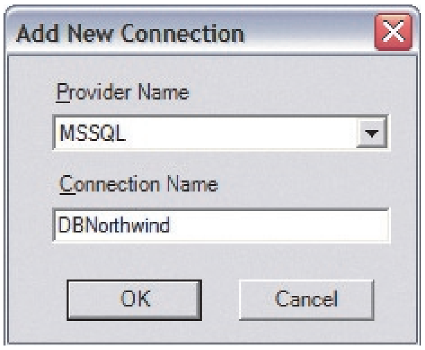


Figure 4: Creating the database connection.



Provider Name: MSSQL

Connection Name: DBNorthwind

Figure 5: Defining the provider name and the connection name.

The Main ADO.NET Classes

Figure 2 illustrates the connection model used in ADO.NET. However, the classes used to access the database will change depending on the type of database you're talking to. For example, the .NET Framework classes ship with **Connection** classes for SQL Server *and* **Connection** classes for OLE DB providers (see Figure 3). These classes can be used to access databases for SQL Server, or databases that have an OLEDB driver.

The problem with the ADO.NET providers for SQL Server and for OLEDB is that once you use them in your code, you're tied to using them. If your database changes, you must update your code to support a new database. Borland recognized this as an issue and created the Borland Data Providers (BDP). The BDP is a fast implementation of the ADO.NET connection framework used to connect to a variety of databases. This allows you to do things like develop against one database and then move the database — with no or few code changes. Currently, the BDP supports Borland InterBase, IBM DB2, Microsoft Access, Oracle, and Microsoft SQL Server.

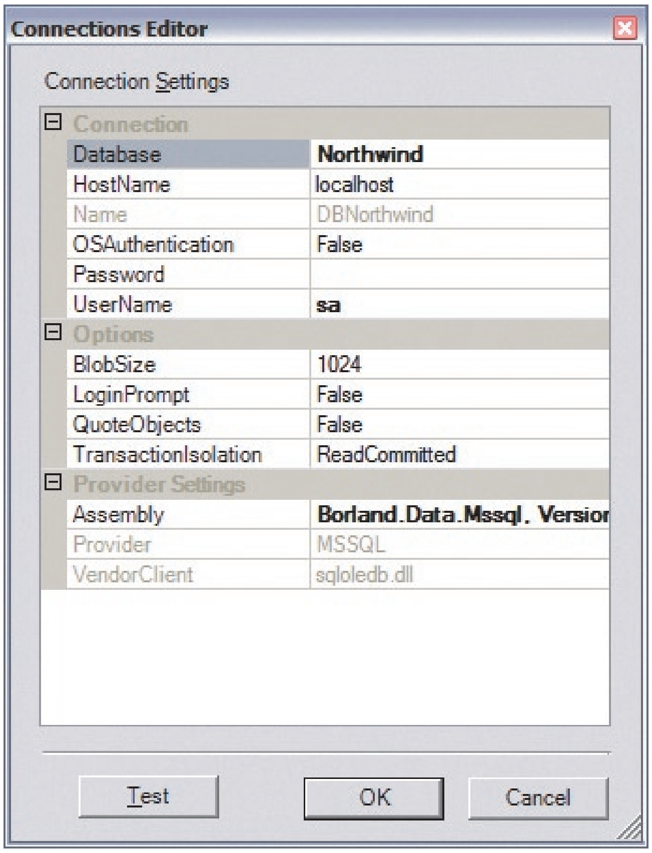
By using the BDP you can easily access your data — and be confident that you can simply change databases later if needed.

Using the Borland Data Providers

Let's get straight into using the BDP. First we must configure a connection to one of the databases. Looking at the Delphi 8 IDE you'll see the Data Explorer. Select the BDP you'll be using and select Add New Connection (see Figure 4). For our demonstrations we'll be accessing the Northwind database that comes with Microsoft SQL Server and Access.

You'll then be presented with a screen asking you to enter the BDP you'll be using, as well as the name of your new connection (see Figure 5). Like most database connection methodologies, the BDP works on the concept of a named connection.

Once you've selected your provider name and your connection name, you must provide the connection parameters to your database. You should then right-click on the connection you


Connections Editor

Connection Settings

Connection	
Database	Northwind
HostName	localhost
Name	DBNorthwind
OSAuthentication	False
Password	
UserName	sa
Options	
BlobSize	1024
LoginPrompt	False
QuoteObjects	False
TransactionIsolation	ReadCommitted
Provider Settings	
Assembly	Borland.Data.Mssql, Version
Provider	MSSQL
VendorClient	sqloledb.dll

Figure 6: Defining the connections parameters.

made in the Data Explorer, and select Modify Connection to be able to edit the parameters for the connection (see Figure 6).

You'll find that setting the dialog box where you set the parameters is very similar to the VCL for defining the parameters for a dbExpress connection. Make sure that the parameters you're using to connect to the database are correct by selecting the Test button. Click OK when your connection is defined and tested.

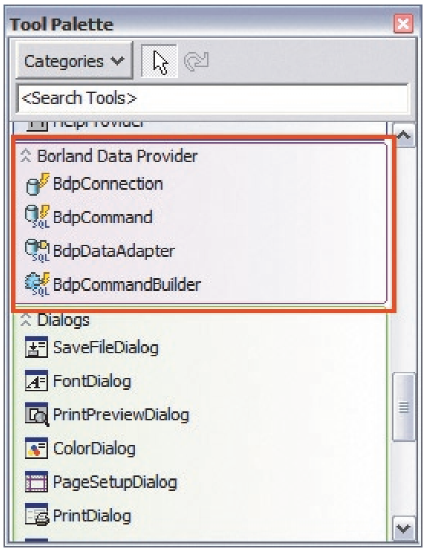
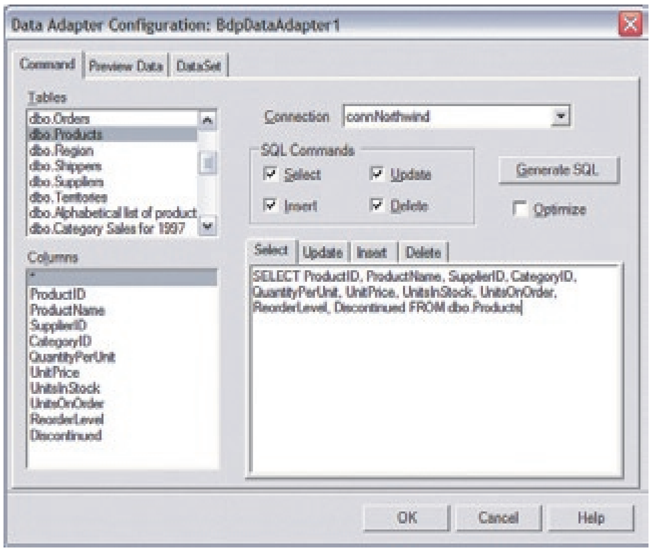


Figure 7: The Borland Data Provider components.

IDE Handy Hints for the BDP

There are many great features in the Delphi 8 IDE when it comes to working with the BDP. The first is that once your connection is defined, you can drill down to see the tables, views, and stored procedures for your database connection. By double-clicking on a table or view from


Data Adapter Configuration: BdpDataAdapter1

Command | Preview Data | DataSet

Tables: dbo.Orders, dbo.Products, dbo.Region, dbo.Shippers, dbo.Suppliers, dbo.Territories, dbo.Alphabetical list of product, dbo.Category Sales for 1997

Connection: connNorthwind

SQL Commands: [✓] Select [✓] Update [✓] Insert [✓] Delete

Generate SQL [ ] Optimize

Columns: *, ProductID, ProductName, SupplierID, CategoryID, QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued

Select | Update | Insert | Delete

```
SELECT ProductID, ProductName, SupplierID, CategoryID,
QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
ReorderLevel, Discontinued FROM dbo.Products
```

Figure 8: Configuring a BdpDataAdapter component.

this list, you can retrieve the data from this connection and have it displayed directly in the IDE — which saves you from using the SQL Explorer (in Delphi 7) and having to constantly switch back and forth to deal with data.

If you're using the tree view in the Data Explorer, you can expand tables and views to see what columns the table or view will return. Likewise for stored procedures, expanding the tree view entries allows you to see the parameters for the columns.

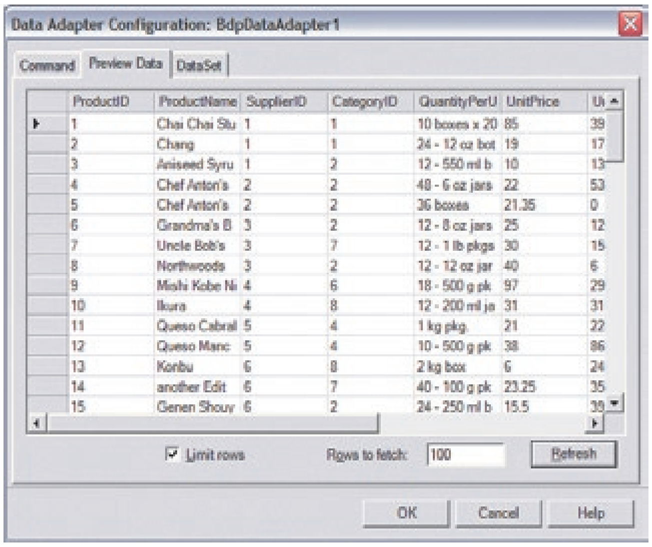
The last (and probably most useful) IDE hint for Delphi 8 is being able to select a table or view from the Data Explorer and drag it directly to your form. After you've done that, you'll see that a BdpConnection and BdpDataAdapter component have been added to the designer, and that all the correct properties have been defined for these BDP components.

Using the BDP Components in Your Application

Using the BDP components is nice and easy. For a simple SELECT statement you need to define a connection to your database by using the BdpConnection component. You'll then need a BdpDataProvider component. The BdpDataProvider component knows how to retrieve data from the BdpConnection component and populate a *System.Data.DataSet* object. The BdpDataProvider component also knows how to update data in your database based on changes made to a *System.Data.DataSet* object.

Looking at the Tool Palette you should see a category titled Borland Data Provider, wherein you'll find all the components you need to connect your database to your application (see Figure 7). Drop a BdpConnection component and a BdpDataAdapter component onto the designer. Set the *ConnectionString* property on your BdpConnection component to the name of the connection you just defined.

The next thing you'll need to set up is the Data Adapter component. Right-click on the BdpDataAdapter component and select the Configure Data Adapter option; you'll be presented with the screen shown in Figure 8. Select a table and the column(s) you want


Data Adapter Configuration: BdpDataAdapter1

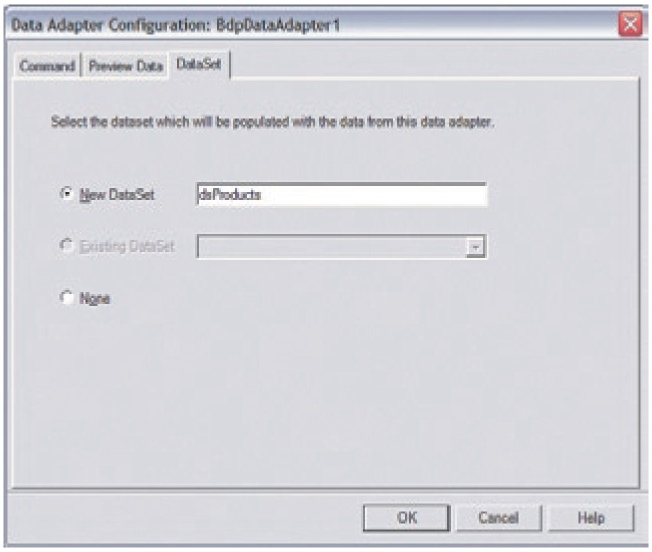
Command | Preview Data | DataSet

	ProductID	ProductName	SupplierID	CategoryID	QuantityPerU	UnitPrice	Ui
▸	1	Chai Chai Stu	1	1	10 boxes x 20	85	39
	2	Chang	1	1	24 - 12 oz bot	19	17
	3	Aniseed Syru	1	2	12 - 550 ml b	10	13
	4	Chef Anton's	2	2	48 - 6 oz jars	22	53
	5	Chef Anton's	2	2	36 boxes	21.35	0
	6	Grandma's B	3	2	12 - 8 oz jars	25	12
	7	Uncle Bob's	3	7	12 - 1 lb pkgs	30	15
	8	Northwoods	3	2	12 - 12 oz jar	40	6
	9	Mishi Kobe Ni	4	6	18 - 500 g pk	97	29
	10	Ikura	4	8	12 - 200 ml ja	31	31
	11	Queso Cabral	5	4	1 kg pkg.	21	22
	12	Queso Manc	5	4	10 - 500 g pk	38	86
	13	Konbu	6	8	2 kg box	6	24
	14	another Edit	6	7	40 - 100 g pk	23.25	35
	15	Genen Shouy	6	2	24 - 250 ml b	15.5	39

[✓] Limit rows   Rows to fetch: 100   Refresh

OK   Cancel   Help

Figure 9: Previewing the data for a BdpDataAdapter.


Data Adapter Configuration: BdpDataAdapter1

Command | Preview Data | DataSet

Select the dataset which will be populated with the data from this data adapter.

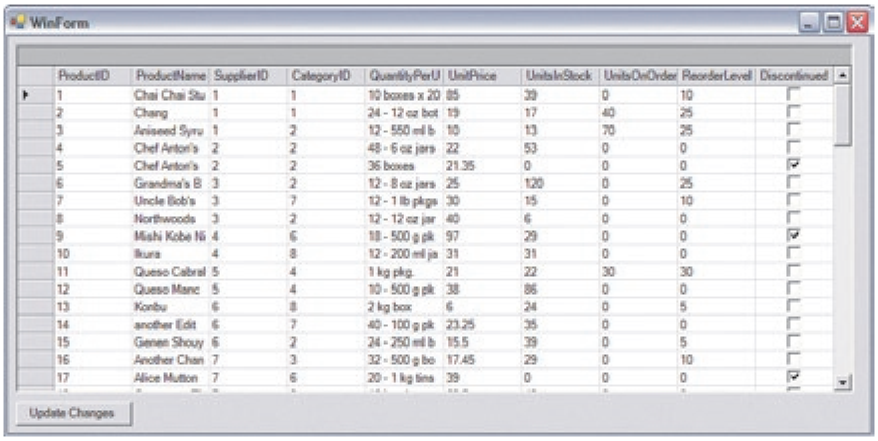
[✓ New DataSet] dsProducts

Existing DataSet

None

OK   Cancel   Help

Figure 10: Configuring the BdpDataAdapter's DataSet.


WinForm

	ProductID	ProductName	SupplierID	CategoryID	QuantityPerU	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel	Discontinued
▸	1	Chai Chai Stu	1	1	10 boxes x 20	85	39	0	10	☐
	2	Chang	1	1	24 - 12 oz bot	19	17	40	25	☐
	3	Aniseed Syru	1	2	12 - 550 ml b	10	13	70	25	☐
	4	Chef Anton's	2	2	48 - 6 oz jars	22	53	0	0	☐
	5	Chef Anton's	2	2	36 boxes	21.35	0	0	0	✓
	6	Grandma's B	3	2	12 - 8 oz jars	25	120	0	25	☐
	7	Uncle Bob's	3	7	12 - 1 lb pkgs	30	15	0	10	☐
	8	Northwoods	3	2	12 - 12 oz jar	40	6	0	0	☐
	9	Mishi Kobe Ni	4	6	18 - 500 g pk	97	29	0	0	✓
	10	Ikura	4	8	12 - 200 ml ja	31	31	0	0	☐
	11	Queso Cabral	5	4	1 kg pkg.	21	22	30	30	☐
	12	Queso Manc	5	4	10 - 500 g pk	38	86	0	0	☐
	13	Konbu	6	8	2 kg box	6	24	0	5	☐
	14	another Edit	6	7	40 - 100 g pk	23.25	35	0	0	☐
	15	Genen Shouy	6	2	24 - 250 ml b	15.5	39	0	5	☐
	16	Another Chan	7	3	32 - 500 g bo	17.45	29	0	10	☐
	17	Alice Mutton	7	6	20 - 1 kg tins	39	0	0	0	✓

Update Changes

Figure 11: A basic database application.

to include in your query. Then click the Generate SQL button. This will create the SELECT statement used to retrieve the data, as well as the UPDATE, INSERT, and DELETE SQL statements to update the database for a row value that has changed. If you have the Optimize checkbox selected when you click the Generate SQL button, the UPDATE, INSERT, and DELETE statements will only include the Primary Key columns in the WHERE clause for the table selected (instead of using every column).

Of course, you're free to enter your own SQL into the text boxes using the Select, Update, Insert, and Delete tabs. Once you have your SQL defined for the query, select the Preview Data tab to view the data (see Figure 9). This screen is handy in ensuring that the data you're retrieving from the database is correct.

The last tab on the BdpDataAdapter configuration dialog box is used to select the *System.Data.DataSet* that will be populated with the data from the DataAdapter (see Figure 10). You have the option of creating a new DataSet component that will be placed on the form, which is the option I've selected. If you had placed a DataSet component onto the designer, you could've selected the Existing DataSet option. Selecting an existing DataSet is especially useful when you want to mix the results from multiple BdpDataAdapters into one dataset for creating master-detail relationships. The other option you can select is to not populate any dataset. This last option is useful if you'll be creating and populating DataSets dynamically in code.

For our example we'll select to dynamically create a DataSet. Enter the name of the Dataset component and select the OK button (again, see Figure 10).

Select the BdpDataAdapter component again and set the *Active* property to *True*. This is useful, as later it will allow you to see the data at design time. For our example we'll need a control to bind our data to. For this we'll use the DataGrid component. Place one of these components on the form, then set the DataGrid's *DataSource* property to the DataSet that was just created. Set the DataGrid's *DataMember* property to the name of the DataTable we'll be using. This normally will be set to the name of the database table — in this case, Products. You should then have a simple form similar to that shown in Figure 11.

This shows a very simple application to display and update data. But the DataSet is only an in-memory representation of the data in your database. To update the data you need to tell the BdpDataAdapter to update the database based on the changes made in the DataSet. I added a button to the form that can be used to apply the changes to the database. Figure 12 shows the single line of code needed to update the database.

Delphi developers will probably find that this way of updating data is very similar to using the MIDAS/DataSnap *TClientDataSet* technology. In our example we are, in essence, using a two-tiered approach to updating the database. You could easily set up a three-tier system using .NET technologies such as Web services or remoting. Because the *System.Data.DataSet* object can be serialized, it's very easy to retrieve it from

```
procedure TWinForm.btnUpdateProducts_Click(
  Sender: System.Object; e: System.EventArgs);
begin
  daProducts.Update(dsProducts, 'Products');
end;
```

Figure 12: Updating changes to a *System.Data.DataSet*.

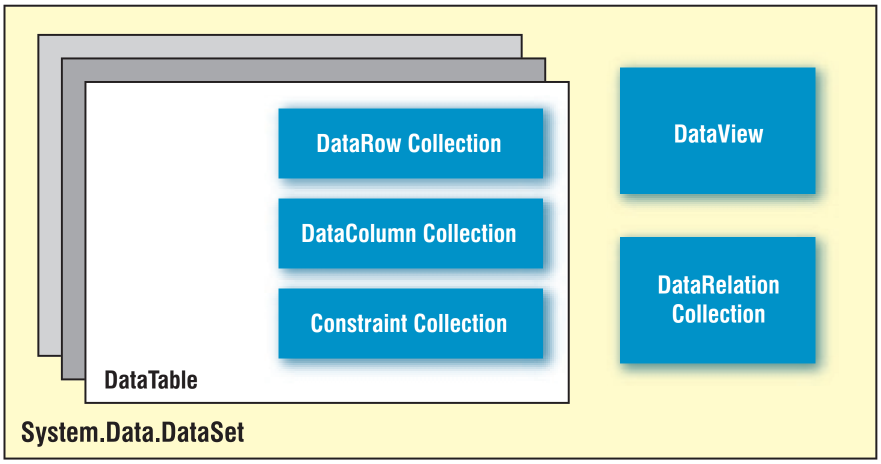
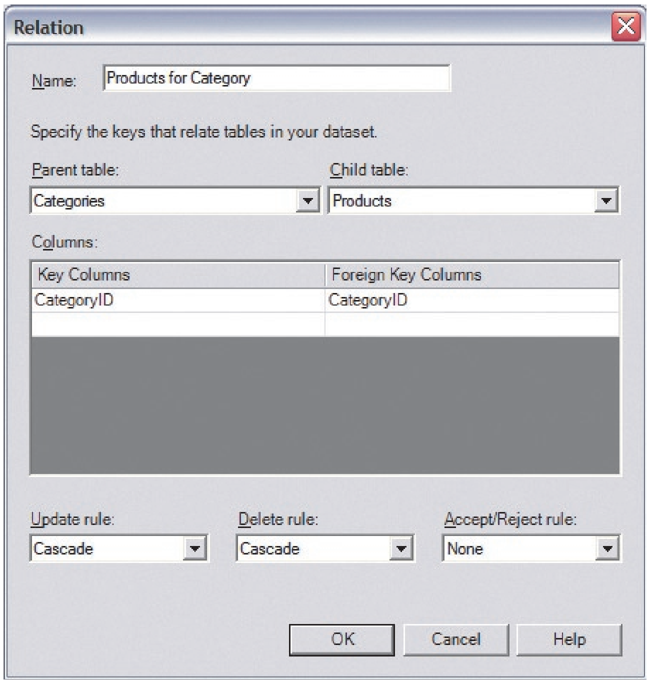


Figure 13: The layout of the *System.Data.DataSet* object.



Name: Products for Category

Specify the keys that relate tables in your dataset.

Parent table: Categories
Child table: Products

Columns:

Key Columns	Foreign Key Columns
CategoryID	CategoryID

Update rule: Cascade
Delete rule: Cascade
Accept/Reject rule: None

Figure 14: Defining a master-detail relationship on a DataSet.

a Web service or to ask a Web service to update the database based on changes to a *System.Data.DataSet*.

A Look at DataSets

With ADO.NET you'll be working with the *System.Data.DataSet* object a lot. Therefore, it's worthwhile to look at the structure of the *DataSet* object. Figure 13 demonstrates the building blocks of the *DataSet* object.

Unlike the RTL *TDataSet* class, the *System.Data.DataSet* class contains more information than just a collection of records. In ADO.NET, the *DataSet* can contain one or more DataTables. A DataTable is a collection of rows from a query. As well as the DataTables, the *DataSet* class can contain the relationships between these DataTables. In our example we populated a *DataSet* with the results from a single query, but could just have easily populated a single dataset with details from multiple BdpDataProviders.

Master-detail in an ADO.NET World

Because the DataSet maintains a list of relationships, defining a master-detail relationship is quite easy. From the Northwind database, add two BdpDataAdapter components to the designer. One BdpDataAdapter will be for the Categories table, and the other will be for the Products table. Make sure that these populate the same dataset by selecting the **Existing DataSet** option in the BdpDataAdapter configuration dialog box (again, see Figure 10). When that's done, make sure you set both the BdpDataAdapter's *Active* properties to *True*.

Select the DataSet component that you populated and select the *Relations* property. Click on the ellipsis to bring up the Relations Collection Editor. Click the **Add** button to define a new relationship, then define the relationship between the Categories and Products tables (see Figure 14).

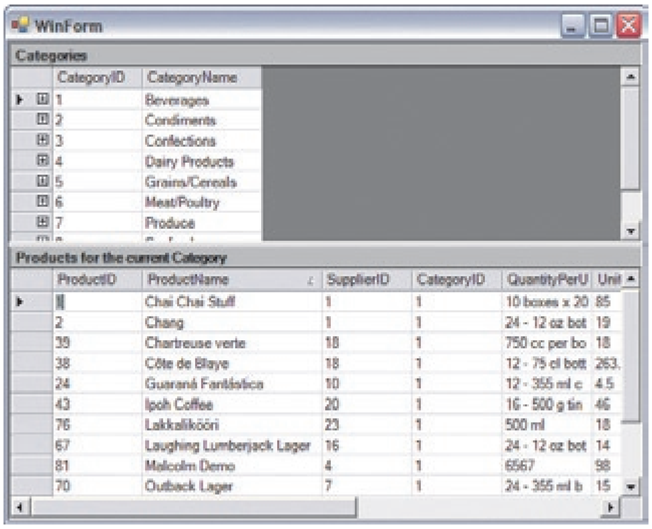
Once the relationship has been defined, you can drop two DataGrids onto your form. For the first DataGrid set the DataSource to the Categories table. For the second DataGrid, again set the *DataSource* property to the Categories table. For the *DataMember* property on the second DataGrid, select the name of the relationship you defined. By doing this you should have successfully set up a master-detail relationship between the Categories and Products tables (see Figure 15).

Accessing Data from Code

Because the DataSet is a complex piece of work, you'll need to know how to access data in code. This is where there are many differences from the VCL *TDataSet*. The two main differences are a result of the DataSet containing a collection of DataTables, and that the DataSet is contained within memory.

Because the DataSet is entirely contained in memory, there are no EOF/BOF methods to navigate throughout a DataTable. Because it's all in-memory, there is a collection of *DataRow* objects that represent the data. Instead of navigating through records the old VCL way (see Figure 16), you iterate through a collection of rows in a DataTable (see Figure 17).

Converting each column that you return to a correct data type can become a little tiresome. Later on we'll look at typed DataSets in order to expedite the writing of data access code. But before we do that, let's look at the ADO.NET way to do a *TDataSet* classic: creating calculated fields.



Categories

	CategoryID	CategoryName
▸ ⊞	1	Beverages
⊞	2	Condiments
⊞	3	Confections
⊞	4	Dairy Products
⊞	5	Grains/Cereals
⊞	6	Meat/Poultry
⊞	7	Produce

Products for the current Category

	ProductID	ProductName	SupplierID	CategoryID	QuantityPerU	Unit
▸	1	Chai Chai Stuff	1	1	10 boxes x 20	85
	2	Chang	1	1	24 - 12 oz bot	19
	39	Chartreuse verte	18	1	750 cc per bo	18
	38	Côte de Blaye	18	1	12 - 75 cl bott	263.
	24	Guaraná Fantástica	10	1	12 - 355 ml c	4.5
	43	Ipoh Coffee	20	1	16 - 500 g tin	46
	76	Lakkalikööri	23	1	500 ml	18
	67	Laughing Lumberjack Lager	16	1	24 - 12 oz bot	14
	81	Malcolm Demo	4	1	6567	98
	70	Outback Lager	7	1	24 - 355 ml b	15

Figure 15: A master-detail example.

```
MyDataSet.First;
while not MyDataSet.Eof do begin
  // Do some operation.
  MyDataSet.Next;
end;
```

Figure 16: The VCL way of navigating a *TDataSet*.

```
procedure TWinForm.btnGetValueOfStock_Click(
  Sender: System.Object; e: System.EventArgs);
var
  counter: Integer;
  TotalValueOfStock: Decimal;
begin
  TotalValueOfStock := 0;
  for counter := 0 to dsProducts.Tables[
        'Products'].Rows.Count - 1 do begin
    TotalValueOfStock := TotalValueOfStock +
      Convert.ToDecimal(dsProducts.Tables['Products'].Rows[
      counter]['UnitPrice']) * Convert.ToInt32(dsProducts.
      Tables['Products'].Rows[counter]['UnitsInStock']);
  end;
  MessageBox.Show('Total value of the current stock is ' +
    TotalValueOfStock.ToString('C'));
end;
```

Figure 17: Iterating through *DataRow* objects in a DataTable.

Calculated Fields

One of the most common things you're likely to do in a database is to create calculated fields. These aren't actual fields; instead, their value is determined as a result of a calculation on the value of another field or fields. When using a *TDataSet*, you normally invoke the Fields editor and then create a calculated field.

In ADO.NET there are two ways you can create calculated fields. The first way is to create a new DataColumn that uses an expression to obtain the value for the calculation (see Figure 18). The method shown, however, is fairly limited. There are only a limited amount of functions that you can apply to obtain the desired result. The reason for using an expression for calculated fields is because the calculated field can distribute between computers when the DataSet is serialized. By using an expression, the information can still be passed around.

```
// Create the calculated column to evaluate
// the Total including Discount.
string TotalExpression =
  "UnitPrice * Quantity * (1 - Discount)";
dsOrderItems.Tables[0].Columns.Add(
  "Total", TypeOf(Double), TotalExpression);
```

Figure 18: Adding a new DataColumn using an expression.

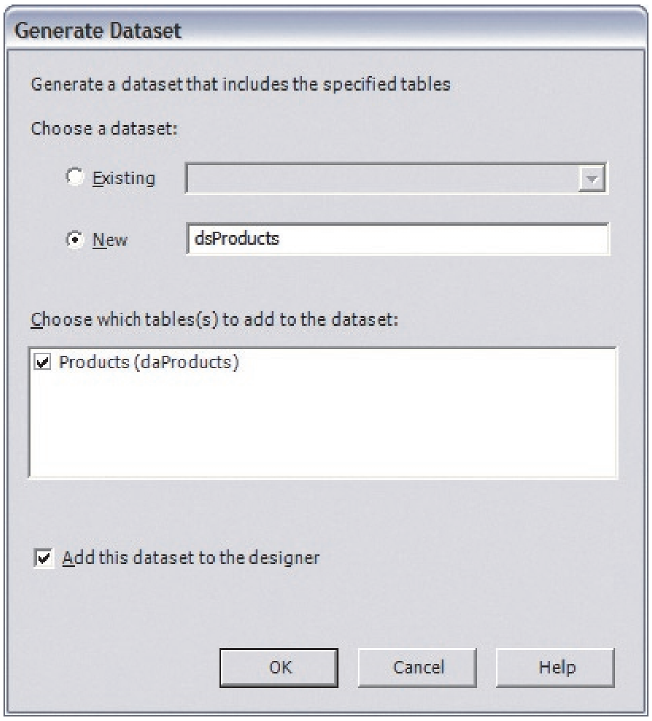
```
procedure TWinForm.SetShouldReorderForRow(Row: DataRow);
var
  ShouldReorder: Boolean;
begin
  if Convert.ToInt32(Row['UnitsInStock']) >
     Convert.ToInt32(Row['ReorderLevel']) then
    ShouldReorder := False
  else
    ShouldReorder :=
      Convert.ToInt32(Row['UnitsOnOrder']) = 0;
  Row['Should Reorder'] := System.Object(ShouldReorder);
end;

procedure TWinForm.ProductsDataColumnChanged(
  Sender: System.Object; e: DataColumnChangeEventArgs);
begin
  if (e.Column.ColumnName = 'UnitsInStock') or
     (e.Column.ColumnName = 'UnitsOnOrder') or
     (e.Column.ColumnName = 'ReorderLevel') then
    SetShouldReorderForRow(e.Row);
end;

procedure TWinForm.btnCreateCalculatedColumn_Click(
  Sender: System.Object; e: System.EventArgs);
var
  ShouldReorder: DataColumn;
  counter: Integer;
begin
  // Create the Column.
  ShouldReorder := dsProducts.Tables['Products'].Columns.
    Add('Should Reorder', TypeOf(System.Boolean));
  // Pre-calculate the values that it should be set to.
  for counter := 0 to
        dsProducts.Tables['Products'].Rows.Count - 1 do
    SetShouldReorderForRow(
      dsProducts.Tables['Products'].Rows[counter]);
  // Make sure it gets notified when any interested
  // columns change.
  Include(dsProducts.Tables['Products'].ColumnChanged,
    ProductsDataColumnChanged);
  btnCreateCalculatedColumn.Enabled := False;
end;
```

Figure 19: Creating a calculated column that isn't based on a simple expression.

A better way to simulate calculated fields would be to define a new column in a DataTable, and be notified when the certain columns change. Unlike with *TDataSet* — where you set up the calculated field, which is calculated when needed — with ADO.NET you need to pre-calculate the values for the calculated fields. You also need a method of being notified when the values change in the columns you need to calculate the values for your calculated field. For this we use the *ColumnChanged* event on the DataTable. Figure 19 demonstrates using this method of creating a calculated field.

Typed DataSets

Notice in the code in Figure 19 where we access the value for a column that I'm using the *Convert* functions to obtain the value of the column. This can become quite cumbersome in your code, because you must type the name of the field all the time. My favorite thing about the *TDataSet* class in the



Generate Dataset

Generate a dataset that includes the specified tables

Choose a dataset:

Existing

New: dsProducts

Choose which tables(s) to add to the dataset:

[✓] Products (daProducts)

[✓] Add this dataset to the designer

Figure 20: Creating a typed DataSet.

VCL is that I can create a persistent field, and then simply use code completion to get the right column without any of my many spelling mistakes. And the persistent field will be of the correct data type every time.

Luckily, typed DataSets aren't that different from using persistent fields with the *TDataSet*. In the VCL, persistent fields are created as wrappers around the field. The same happens with ADO.NET typed datasets, but what actually happens is a *System.Data.DataSet* subclass is created specifically for access to your particular dataset.

Creating the Typed DataSet

The first thing you must do to create a typed DataSet is to create your data adapters on your form. For our example you could simply drag a table from the Data Explorer onto the Form Designer. This should create a BdpDataAdapter and a BdpConnection, if needed. Right-click on the BdpDataAdapter and configure it appropriately. Once configured, right-click on the BdpDataAdapter and select Generate Typed DataSet. You'll then be presented with the Generate DataSet dialog box, where you can select which tables are to be included in the typed DataSet (see Figure 20). If you have more than one BdpDataProvider on the form, you can also include those in the typed DataSet definition.

In our example, we'll use the Products DataSet to define our typed dataset. Clicking OK creates an XML Schema Definition (XSD) file that contains the definition of the structure for your new typed DataSet. In addition, a Pascal unit is created that holds the information about the typed dataset. If you select the Project Manager in the Delphi IDE, you can double-click on the source code file located under the XSD file to view the source code unit. You'll see that there is a DataSet descendant, a DataRow

```
procedure TWinForm.btnGetValueOfStock_Click(
  Sender: System.Object; e: System.EventArgs);
var
  counter: Integer;
  TotalValueOfStock: Decimal;
begin
  TotalValueOfStock := 0;
  for counter := 0 to
        dsProducts1.Products.Rows.Count - 1 do begin
    TotalValueOfStock := TotalValueOfStock +
      dsProducts1.Products.Item[counter].UnitPrice *
      dsProducts1.Products.Item[counter].UnitsInStock;
  end;
  MessageBox.Show('Total value of the current stock is ' +
    TotalValueOfStock.ToString('C'));
end;
```

Figure 21: Accessing data using a typed DataSet.

descendant specifically for the data, and many helper functions. You should also note that your typed DataSet is added to the designer.

Putting the typed DataSet to use, we can navigate the records for all the records of products, as we did in Figure 16. Using the typed DataSet we created for the products, the code is much cleaner, and you get to use the code completion features of Delphi, enabling you to write this code in less time (see Figure 21).

Notice that we don't access the *DataTables* collection in Figure 21, but instead access the *Products* property. We can also use the *UnitPrice* and the *UnitsInStock* properties of the *ProductsDataRow* to expedite development.

We can do less coding by creating a DataRow descendant (see Figure 22). You'll notice that the DataRow descendant also creates methods to set each field to null, and to determine if a field has a null value.

Binding Data to Other Controls

You can bind data to almost any control property that you desire. In fact, you don't even need to bind to a DataSet at all; you can bind directly to anything that implements the *IListSource* interface. Even an array can be used for data binding, as shown in Figure 23.

In most cases, when you bind a data-aware component to a DataSource in VCL land, you'll set the *DataSource* and *DataField* properties and then have the data for a field created for you. For example, if you wanted a data-aware label that displayed a Product Code in the *Caption* and the Full Description in the *Hint* property, you'd have to create a custom *TLabel* subclass that allowed you to do that.

Not only can you bind to arrays and other data types in ADO.NET, you're not limited to binding to just a *DataSource*, *DataField*, etc. Instead, you can bind to most properties of components. You've seen how to hook data up to a DataGrid, but let's hook up a *Label* control, and then have several of its properties bound to our DataSet.

For this example, drop a ToolTip component so that each visible component can have a *ToolTip* property. Then drop

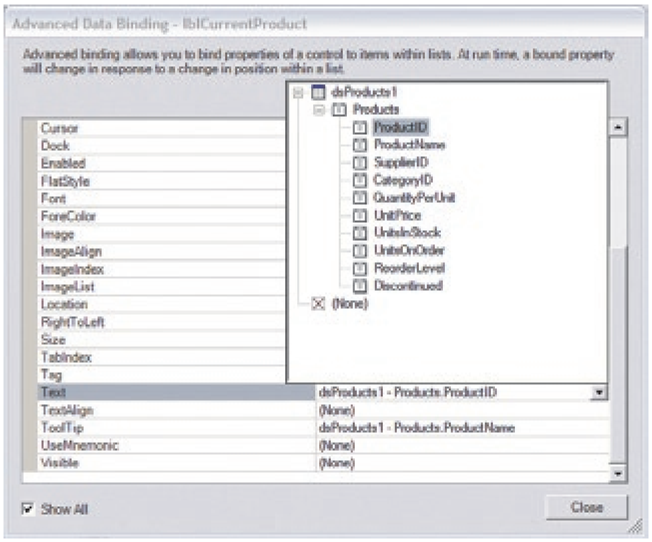
```
ProductsRow = class(DataRow)
strict private
  tableProducts: ProductsDataTable;
private
  constructor Create(rb: DataRowBuilder);
public
  function get_ProductID: Integer;
  function get_ProductName: string;
  function get_SupplierID: Integer;
  function get_CategoryID: Integer;
  function get_QuantityPerUnit: string;
  function get_UnitPrice: System.Decimal;
  function get_UnitsInStock: SmallInt;
  function get_UnitsOnOrder: SmallInt;
  function get_ReorderLevel: SmallInt;
  function get_Discontinued: Boolean;
  procedure set_ProductID(Value: Integer);
  procedure set_ProductName(Value: string);
  procedure set_SupplierID(Value: Integer);
  procedure set_CategoryID(Value: Integer);
  procedure set_QuantityPerUnit(Value: string);
  procedure set_UnitPrice(Value: System.Decimal);
  procedure set_UnitsInStock(Value: SmallInt);
  procedure set_UnitsOnOrder(Value: SmallInt);
  procedure set_ReorderLevel(Value: SmallInt);
  procedure set_Discontinued(Value: Boolean);
  property ProductID: Integer
    read get_ProductID write set_ProductID;
  property ProductName: string
    read get_ProductName write set_ProductName;
  property SupplierID: Integer
    read get_SupplierID write set_SupplierID;
  property CategoryID: Integer
    read get_CategoryID write set_CategoryID;
  property QuantityPerUnit: string
    read get_QuantityPerUnit write set_QuantityPerUnit;
  property UnitPrice: System.Decimal
    read get_UnitPrice write set_UnitPrice;
  property UnitsInStock: SmallInt
    read get_UnitsInStock write set_UnitsInStock;
  property UnitsOnOrder: SmallInt
    read get_UnitsOnOrder write set_UnitsOnOrder;
  property ReorderLevel: SmallInt
    read get_ReorderLevel write set_ReorderLevel;
  property Discontinued: Boolean
    read get_Discontinued write set_Discontinued;
  function IsSupplierIDNull: Boolean;
  procedure SetSupplierIDNull;
  function IsCategoryIDNull: Boolean;
  procedure SetCategoryIDNull;
  function IsQuantityPerUnitNull: Boolean;
  procedure SetQuantityPerUnitNull;
  function IsUnitPriceNull: Boolean;
  procedure SetUnitPriceNull;
  function IsUnitsInStockNull: Boolean;
  procedure SetUnitsInStockNull;
  function IsUnitsOnOrderNull: Boolean;
  procedure SetUnitsOnOrderNull;
  function IsReorderLevelNull: Boolean;
  procedure SetReorderLevelNull;
end;
```

Figure 22: The *ProductsDataRow* class.

a Label component onto the form. This Label component will be used to display the current product. It will display the ProductID in the caption and the ProductName in the ToolTip. Select the Label component and select its *DataBindings* property. Expand this property so the *Advanced* property is visible. Then select the ellipsis to bring up the Advanced Data Binding editor. For each property in the list, you can select which properties will be bound to which columns (see Figure 24).

```
const
  SalutationTypes: array[0..4] of string =
    ('Mr', 'Mrs', 'Miss', 'Ms', 'Dr');

procedure TWinForm.TWinForm_Load(
  Sender: System.Object; e: System.EventArgs);
begin
  cbSalutation.DataSource := SalutationTypes;
end;
```

Figure 23: Binding Items from an array of strings into a ComboBox.


Advanced Data Binding - lblCurrentProduct

Advanced binding allows you to bind properties of a control to items within lists. At run time, a bound property will change in response to a change in position within a list.

```
dsProducts1
  Products
    ProductID
    ProductName
    SupplierID
    CategoryID
    QuantityPerUnit
    UnitPrice
    UnitsInStock
    UnitsOnOrder
    ReorderLevel
    Discontinued
  (None)
```

Cursor	
Dock	
Enabled	
FlatStyle	
Font	
ForeColor	
Image	
ImageAlign	
ImageIndex	
ImageList	
Location	
RightToLeft	
Size	
TabIndex	
Tag	
Text	dsProducts1 - Products.ProductID
TextAlign	(None)
ToolTip	dsProducts1 - Products.ProductName
UseMnemonic	(None)
Visible	(None)

Figure 24: Binding to more than one property using ADO.NET data binding.

You can do this type of data binding in ADO.NET, from binding the Text of a Label to binding the Icon of a form. Data binding is one of the best features of ADO.NET, and I'm sure you'll find it extremely handy for years to come.

Data Validation

In VCL development, exception handling is used to control whether information in the current record or field is valid. You would write an *OnBeforePost* event handler on a *TDataSet* or an *OnValidate* event handler for a field. This isn't very different from how validation works with ADO.NET. Whereas with the VCL you write the event handlers on the *TDataSet*, with ADO.NET you write the event handlers for your DataTables.

Dealing with Column Changes

Usually when I have to deal with validating the values of a particular column using the VCL, I do one of two things: I'll set properties on the *TField* descendants so that I make sure the values are correct, and/or I write an *OnValidate* event handler for the specific *TField*.

Setting the properties on a data column is relatively straightforward. You'll use the appropriate column properties to set the values (see Figure 25).

In the VCL, you would write an *OnValidate* event handler for your persistent field, and raise an exception when the value wasn't acceptable. The concept is the same with ADO.NET. Each DataTable has an *OnColumnChanging* event that you use to be notified when changes in any

DataColumn Property	VCL Equivalent	Description
AllowDBNull	*Required*	Determines if null values are allowed in the column
MaxLength	*MaxLength*	The maximum length of the column
Unique	none	Determines if column must contain a unique value
ReadOnly	*ReadOnly*	Determines if the column is read only

Figure 25: Validation properties that can be set on a DataColumn.

```
procedure frmProducts.CheckTheColumns(
  Sender: System.Object;
  e: System.Data.DataColumnChangeEventArgs);
begin
  if e.Column.ColumnName = 'ProductID' then begin
    if Integer(e.ProposedValue) < 0 then begin
      MessageBox.Show(
        'The Product ID needs to be a positive number',
        'Data Entry Issue', MessageBoxButtons.OK,
        MessageBoxIcon.Warning);
      raise Exception.Create(
        'The Product ID must be a positive number');
    end;
  end;
end;

// Dynamically creating a Typed DataSet.
procedure frmProducts.TWinForm_Load(Sender: System.Object;
  e: System.EventArgs);
var
  products: ProductsDataSet;
begin
  products := ProductsDataSet.Create;
  daProducts.Fill(products, 'Products');
  DataGrid1.DataSource := products;
  DataGrid1.DataMember := 'Products';
  Include(products.Products.ColumnChanging, CheckTheColumns);
end;
```

Figure 26: Using field-level validation on a DataTable.

columns occur. Here you check the value of the column in which you're interested, and raise an exception if the value is unacceptable. It's also a good idea to display a dialog box explaining why the value is incorrect so users don't get upset when it appears that the application has changed the values for no apparent reason.

You'll need to write code to set up the *OnColumnChanging* event. As an example, Figure 26 demonstrates using the events to check that the ProductID is a positive number.

Dealing with Row Changes

Managing changes in rows isn't that different than dealing with changes with columns. The same rules apply. You set up the event handler for the change. You check the change, and if you're not happy with the values provided, raise an exception. This way you can always be sure that the data is valid.

Where you would use the VCL's *OnBeforePost* event to check for changes, with ADO.NET you would use the *RowChanging* event on the DataTable. You set up the *RowChanging* event in much the same way you set up the *ColumnChanging* event.

```
procedure frmProducts.CheckTheProductsDataRow(
  Sender: System.Object;
  e: System.Data.DataRowChangeEventArgs);
begin
  if Decimal(e.Row['UnitPrice']) < 0.00 then
    raise Exception.Create(
      'The unit price for a product cannot be negative');
  if e.Row.IsNull('SupplierID') then
    raise Exception.Create(
      'The supplier for the product must be provided');
end;

procedure frmProducts.TWinForm_Load(Sender: System.Object;
  e: System.EventArgs);
var
  products: ProductsDataSet;
begin
  products := ProductsDataSet.Create;
  ...
  Include(products.Products.RowChanging,
          CheckTheProductsDataRow);
end;
```

Figure 27: Validating a DataRow in a DataTable.

```
procedure ProductsDataSet.ProductsDataTable.OnRowChanging(
  e: DataRowChangeEventArgs);
var
  row: ProductsDataSet.ProductsRow;
begin
  inherited OnRowChanging(e);
  row := e.Row as ProductsDataSet.ProductsRow;
  if row.UnitPrice < 0.00 then
    raise Exception.Create(
      'The unit price for a product cannot be negative');
  if row.IsSupplierIDNull then
    raise Exception.Create(
      'The supplier for the product must be provided');
  if (Assigned(Self.ProductsRowChanging)) then
    Self.ProductsRowChanging(Self,
      ProductsRowChangeEvent.Create((ProductsRow(e.Row)),
      e.Action));
end;
```

Figure 28: Validating data using typed DataSets.

The major difference with the *RowChanging* event is that you don't need to display an error dialog box. Figure 27 demonstrates setting up the validation routines for a DataRow.

If you're using typed DataSets, validating rows is even easier. When the typed dataset is created, it exposes a method named *OnRowChanging* for each DataTable contained in the dataset. You can modify this method to write the validation logic for your DataRow. Not only is it easier to set up, there is also no need to typecast the column values to get the values for a column. Figure 28 demonstrates how to perform validation against the Products DataTable.

Validation isn't much of an issue when dealing with ADO.NET. Using exception handling, you implement your validation logic in a way that is simple to use, and which isn't too different from how you did the same thing using the VCL.

Conclusion

For those of you moving to ADO.NET from the VCL, you are moving to a new framework, but most of the same rules apply. You still need to manipulate data and use your bag of tricks to get your applications out the door and to your customers in the shortest amount of time. By using the techniques discussed in this article, you'll be able to convert your VCL *TDataSet* skills to ADO.NET skills in no time.

The five sample projects that accompany this article are available for download on the Delphi Informant Magazine Complete Works CD located in INFORM\2004\MAY\DI200405GS.

Glenn Stephens *designs and develops applications for various platforms, and has been programming for more than 15 years. He is a Borland Certified Consultant, a Microsoft Certified Solution Developer, and is the author of* The Tomes of Kylix: The Linux API. *Living in sunny Australia, Glenn spends his spare time training for marathon swimming races and playing the piano too loud for his neighbors. Feel free to contact Glenn at glenn@glennstephens.com.au.*

By Mike Riley

Doc-To-Help 7.0 Professional

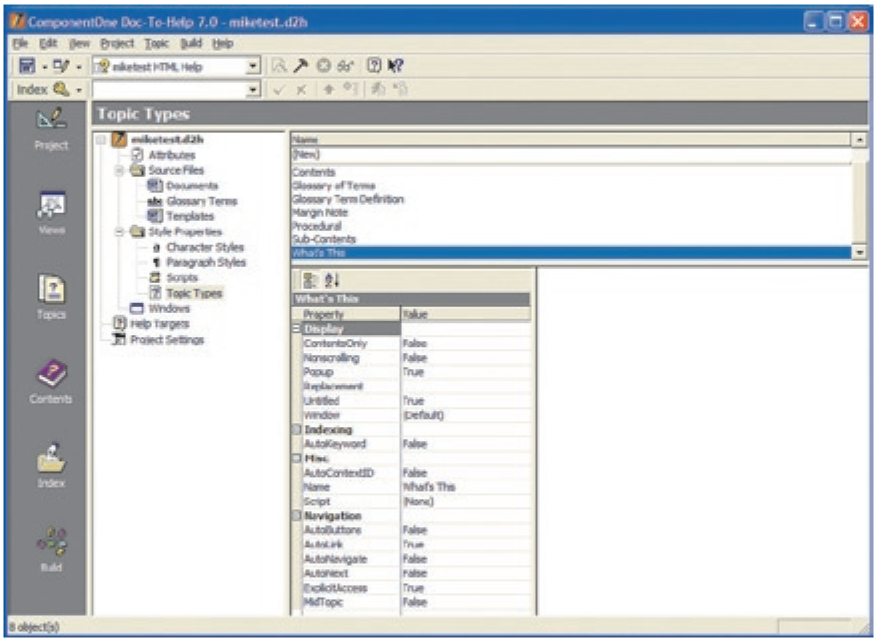
Help Never Looked So Good

Like most mature products, ComponentOne's latest release of its Help documentation construction tool has built its latest features on a bedrock of past success. When ComponentOne acquired the Doc-To-Help product line in 2001, users expected notable improvements in successive releases — and ComponentOne has delivered (see Figure 1).

New & Improved

This review will focus on the enhancements made to Doc-To-Help 7.0 Professional. In addition to the obligatory bug fixes for previous versions, five new features have been added:

- Natural Language Search
- Graphic Hot Spots
- A Modular TOC Utility
- Support for Microsoft Help 2.0 Context Strings (this may be a less interesting addition for developers because Microsoft has officially discontinued this system)
- A Theme Designer for creating and applying a customized look-and-feel for HTML Help, HTML 4.0, and Help 2.0 targets



Name
(New)
Contents
Glossary of Terms
Glossary Term Definition
Margin Note
Procedural
Sub-Contents
What's This

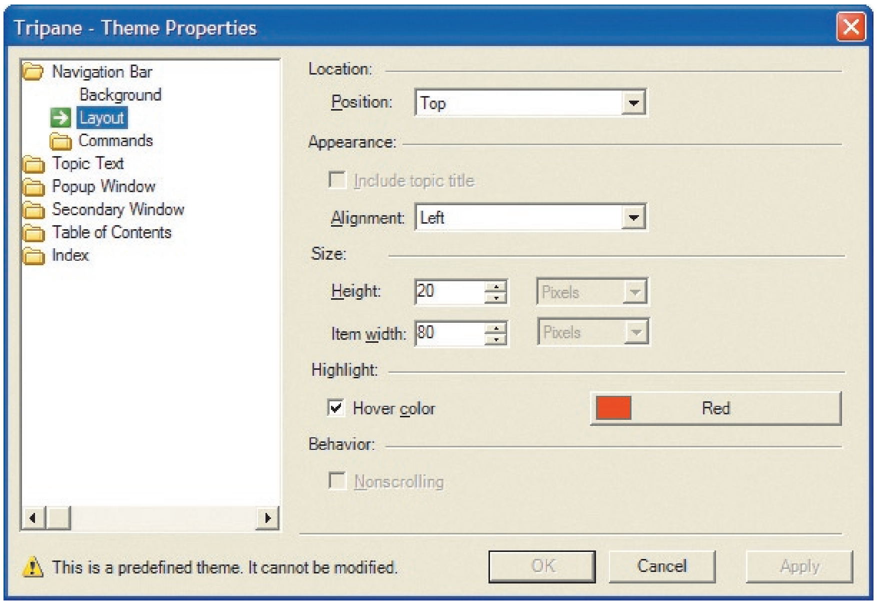
What's This

Property	Value
Display	
ContentsOnly	False
Nonscrolling	False
Popup	True
Replacement	
Untitled	True
Window	(Default)
Indexing	
AutoKeyword	False
Misc	
AutoContextID	False
Name	What's This
Script	(None)
Navigation	
AutoButtons	False
AutoLink	True
AutoNavigate	False
AutoNext	False
ExplicitAccess	True
MidTopic	False

Figure 1: The main Doc-To-Help screen provides access to the most granular details of a Help project.

The Natural Language Search feature affords users of WinHelp and compiled HTML Help files the ability to generate an index file used to retrieve Help topics via a standard question, *à la* the Microsoft Office Assistant's "What would you like to do?" natural query feature. Although the construction of this functionality into a Help project isn't as easy as the rest of the program, it does add considerable polish to a Help facility — especially if the intended consumer of the documentation prefers a more fluid search approach.

Another new feature that has finally made its way into the product is Graphic Hot Spots. This long-awaited enhancement allows Help authors to add hot-spot topic links into an embedded image via the ComponentOne Image Map Editor tool.


Tripane - Theme Properties

- Navigation Bar
  - Background
  - Layout
  - Commands
- Topic Text
- Popup Window
- Secondary Window
- Table of Contents
- Index

Location:
Position: Top

Appearance:
[ ] Include topic title
Alignment: Left

Size:
Height: 20 Pixels
Item width: 80 Pixels

Highlight:
[✓] Hover color Red

Behavior:
[ ] Nonscrolling

This is a predefined theme. It cannot be modified.

Figure 2: The Theme Designer provides the ability to customize an HTML Help file's behavior and visual appearance.

In addition to the three pre-designed themes included in Doc-To-Help, the Theme Designer is the one feature I was especially pleased to see as part of the product (see Figure 2). Although this feature will probably only be leveraged by a small percentage of Doc-To-Help users, its inclusion will make them happy campers. Navigation, background styles, and images in any window (primary, secondary, or pop-up) can provide a unique look to any Help project. Games and applications with highly stylized UIs are frequent employers of this level of customization. Although the design stage does take some planning (especially because poor background color selections can make the best documentation in the world illegible), setting the various theme properties is as simple as a few mouse clicks. More colorful Help files are sure to be a result of this feature.

The scenario most applicable for this feature is a toolbar screenshot that users can point-and-click to link directly to the corresponding topic. This feature is used extensively in Doc-To-Help's own compiled HTML Help file.

The Modular TOC Utility, an easy-to-use wizard application, adds a table of contents to each Help module in a project, allowing programmers access to a full content list for context-sensitive Help requests. Without this addition, Help files tend to become "islands," because they don't contain a master TOC to reference back to the primary Help contents file.

Just the Facts

Doc-To-Help 7.0 Professional provides developers and document specialists with a powerful online and printed user documentation construction toolkit that can adequately serve most application platform needs. It can compile and output five Help document types from a single source, its customized themes allow Help files to maintain a company's brand or approved corporate design guidelines, and it provides an extensive API for automated Help file construction needs. Its minor shortcomings include its lack of any codified examples demonstrating how the product's resulting Help files can be accessed and used within various languages, including Delphi, and the documentation is only installed as a compiled HTML Help file.

ComponentOne LLC
4516 Henry Street, Suite 500
Pittsburgh, PA 15213

Phone: (800) 858-2739
E-Mail: info@componentone.com
Web Site: www.componentone.com
Price: US$999.95

If It's Not Broken ...

Although not new to the 7.0 release, I found the most impressive feature of this product is still its powerful ability to compile well-formed Help documentation into five document platforms from a single source. Build targets are preconfigured for HTML (.htm), HTML Help (.chm), JavaHelp (.jar; this requires the JRE 1.4.1_02 and JavaHelp 1.1.3 to be installed), Printed Manual (.doc), and legacy WinHelp (.hlp) formats.

I was amazed by how well each format was optimized for its intended platform. In fact, Doc-To-Help performs this task so well that I wished it would support other document output types, such as press-ready PDFs, and even PowerPoint formats. Unfortunately, the current version doesn't provide a straightforward add-in API function for inserting new build types, thereby preventing power developers from extending the product's capabilities to support alternate output files. Nevertheless, the API that *is* exposed for scripting is still quite powerful, allowing the behavioral modification of Help documents using nearly every formatting property available through the manual Doc-To-Help interface.

Another ding against the product is that no example integration code ships with the installation, leaving it up to the developer to chase down the respective Help file instantiation code for their respective programming environment. Activating online Help in a Delphi program is remarkably easy, but it would have nevertheless been a good refresher to have a few samples demonstrating how this can be done in the various application frameworks that Doc-To-Help supports. At least two tutorials and two sample projects that ship with the product do an adequate job of introducing the program's functionality to its users.

Like previous versions, and similar to several other Help documentation tools, Doc-To-Help 7.0 Professional requires

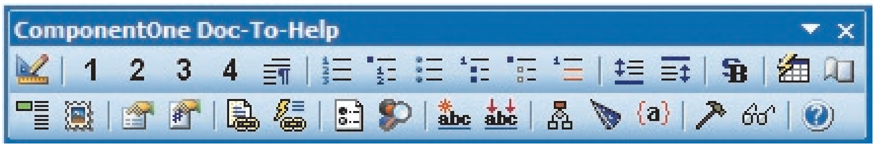


Figure 3: The Doc-To-Help formatting toolbar appears as a Microsoft Word add-in when editing Help documents.

Microsoft Word to be installed to use the product, as Word is employed as the primary document construction interface (see Figure 3). This normally isn't an issue for most technical writers; nevertheless, it adds a modestly expensive requirement to the operational use of Doc-To-Help. Developers who prefer their tools to be self-contained and don't require the multiple outputs that Doc-To-Help provides may find less expensive alternatives more aligned with their expectations. Doc-To-Help 7.0 Professional also requires the Microsoft .NET Framework 1.1. During installation, the Doc-To-Help setup will automatically check your system for the presence of this framework.

As one would expect from a Help construction tool, the online Help is excellent. It also serves as a perfect demonstration of the power of Doc-To-Help, because ComponentOne obviously "ate its own dog food" to create the program's Help files. Unfortunately, the download version doesn't provide a Microsoft Word or PDF version of the documentation, making users rely solely on the compiled HTML Help file for the program's documentation needs. This is particularly odd since Doc-To-Help provides painless output to most of these formats anyway. It would have been ideal for ComponentOne to include the precompiled Help project files for users to prune best practices from a company that obviously knows the capabilities of its tool inside and out.

Conclusion

Overall, the product is a decent addition for any application developer responsible for baking documentation into their projects. And given the variety of project types these days, whether Delphi-, Java-, or .NET-based, Doc-To-Help's output support ensures that an investment in this Help authoring tool offers enough versatility to fulfill nearly every Help documentation need.

Doc-To-Help 7.0 Professional provides developers and document specialists with a powerful online and printed user documentation construction toolkit that can adequately serve most application platform needs. Although the product does have some minor shortcomings, its flexibility, power, and ease of use make it a worthy contender in the electronic documentation space.

Mike Riley *is a chief scientist with RR Donnelley, one of North America's largest printers. He participates in the company's emerging technology strategies using a wide variety of distributed network technologies, including Delphi. Readers may reach him at mike_riley_@hotmail.com.*

Delphi Developer's Guide to XML, Second Edition

Many developers are writing Web applications to leverage XML, whether in multifunctional presentation layers, or machine-to-machine communication via SOAP, REST, or a more exotic contextually-based schema. Delphi developers faced with understanding XML will find Keith Wood's new book an educational addition to their library. Wood is a frequent contributor to *Delphi Informant*, having written a three-part series on "XML Building Blocks" as well as many other Delphi XML-related topics. In this book, he successfully combines his deep knowledge of Delphi with his clear writing style, using the same well-paced teaching approach found in his magazine articles.

The first edition of this book, published by Wordware and currently out of print, was reviewed more than two years ago by Ron Loewy (see his review in the February, 2002 issue of *Delphi Informant* or online at www.DelphiZine.com). Loewy concluded his review thus, "If you need to understand XML from a developer's point of view, this is the only Delphi-specific book. I can also recommend it without reservation. The book provides concise information, and covers parsing and producing XML from Delphi applications." I generally agree with Ron's review of the first edition, so this review will focus on evaluating the changes made in the second edition.

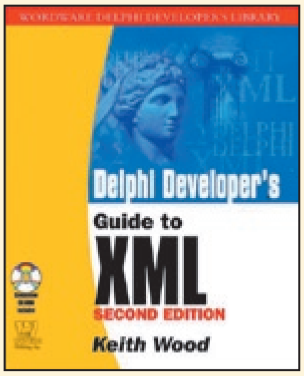
When the first edition was written, the family of XML technologies was rapidly evolving and changing — some followed the Darwinian trajectory of survival of the fittest. This edition resolves XML's early growth spurts and places the latest catalog of successful XML derivatives into meaningful context for Delphi developers to comprehend and leverage in their Delphi projects.

The most notable additions to the second edition are the discussions of XML Data Binding, XML Mapper, and SOAP support that have been added to Delphi 6 and higher releases. The book also promotes the Simple API for XML (SAX) for Pascal, a remarkable open source XML library available at http://sourceforge.net/projects/saxforpascal/. Unfortunately, the book was completed before TurboPower donated its excellent family of products to the open source community. As such, Wood missed an opportunity to promote further enhancements of the XML Partner components available for download at http://sourceforge.net/projects/tpxmlpartner/. Of all the Delphi XML components I've had the opportunity to leverage in my Delphi applications, XML Partner has been the main component library of choice. Given that this is now an XML suite freely available to Delphi developers, an appendix on using the components would have been ideal.

The price of this second edition has been reduced by US$10 from the first edition, although the relevance of the content has increased — a bargain indeed. Missing this time around is an accompanying CD-ROM; however, the code samples can be downloaded from the book's Web site at http://home.iprimus.com.au/kbwood/DelphiXML.

And for those who don't mind e-reading, BookSurge offers a non-printable digital version of the book for the remarkably low price of US$7.99. This version may be optimal for those who purchased the first edition and are seeking an inexpensive upgrade to absorb the changes in this new edition.

— *Mike Riley*



Delphi Developer's Guide to XML, Second Edition
by Keith Wood, BookSurge, www.booksurge.com.

ISBN: 1-59109-862-9
Cover Price: US$49.99
(589 pages)

The Future of Delphi

By Alan C. Moore, Ph.D.

Last month I shared my initial reactions to Delphi 8. This month I'd like to deal with an issue that seems to come up every time a new version of Delphi is released: The future of Borland and our favorite development tool.

I was inspired to write this column after reading a discussion thread on Delphi-Talk (www.elists.org/mailman/listinfo/delphi-talk) entitled "Borland's Future" (which was inspired by a previous discussion on the borland.public.delphi.non-technical newsgroup at Newsgroups.Borland.com). The Delphi-Talk thread began with this post: "Has anyone been following the newsgroups? What are we all to think when Blake Stone, Chuck J., and Eddie Churchill all jump on the Microsoft ship? Are Delphi's days coming to an end? Anyone care to comment?"

As with the Borland newsgroup, many Delphi-Talkers spoke up. The first response was almost predictable, raising the question: "Wasn't this said when Anders left four or five years ago?" The next contributor struck me as particularly thoughtful; he began with the observation that, "The Delphi jobs list gets shorter every month. There are a whole lot more people looking for jobs than jobs looking for people." But he continued with, "Borland has the best technical solution of any available." He concluded by pointing out that although every market has a leader, in this case Microsoft with its Visual Studio .NET, there is room for other players such as Borland — especially considering their recognized quality.

The next contributor took the discussion in an entirely new direction with a suggestion that surprised even me: "I think part of the problem is the lack of a visible and flamboyant CEO a la Philippe Kahn." Assessing Borland's current leader, he said that Dale Fuller seemed like a "great guy," but suggested that the CEO needed to "get out there and make a bit of a fool of himself like Kahn did, like Ellison, McNealy, and Ballmer do." I have to disagree with this a bit. I didn't attend the last Borland Conference, but at earlier conferences I found Dale to be extremely visible and accessible. And to his credit, I think he was more than willing to make something of a fool of himself, especially in the opening show. That said, perhaps he *could* be more newsworthy beyond the walls of the Land of Bor.

Another contributor to this discussion had some positive remarks about Borland's leader: "Fuller really took a risk by taking Borland under his wing early on. He stood to lose big time if he failed, but he didn't. Right now Borland is above water again and it seems things are going well..." But he did acknowledge some concern about the recent departures.

This individual made additional observations, suggesting that Delphi developers could do much to help the company on whose tool they currently depend. He mentioned the "high profile products with obvious Delphi front-ends" that fail to acknowledge the main tool with which they were built. (There are some very successful products, such as SpyBot, that have no embarrassment in acknowledging Delphi as the main tool.)

Delphi offshore, upgrades, and more. One contributor mentioned the strength of Delphi outside the United States, something those of us living in the US sometimes forget as we fall into our doom and gloom mentality. A developer from Brazil jumped in, mentioning the active Delphi communities in his country include over 4,000 subscribers. He indicated that "Delphi's and Borland's future are always popular topics," but added many avoid upgrading on a regular basis because "Delphi's expense is a concern for many Brazilian developers." But he also reported that in his country there are serious concerns with Borland's big competitor because "Microsoft is dropping support for some of its platforms" and has "changed the philosophy of its license agreements."

The thread also included some of the usual complaints about bugs. One Australian developer put complaints about Delphi bugs in an interesting context: "We all write code — who has ever written a truly bug-free large application? We beat Borland up over bugs when Microsoft releases at least a new set of patches every other day?"

The next Delphi-Talker endorsed the recommendation for a more flamboyant CEO, but focused more on a perennial complaint: The lack of effective marketing. Although this critic did not elaborate, others did. One individual whose company is still using Delphi 6 recalled a phone call from a Borland salesperson who could not convince him to upgrade. He concluded with, "... and if they can't convince me as a developer, how would they convince the bozos who make the real decisions in companies?" But another person suggested ways in which we developers could contribute to Borland marketing: "If every Delphi developer [would] put a nice splash (or something like that) in his application with text like "Made in Borland" or "Powered by Delphi", the world [might] finally say, 'Hey, I must try this Delphi stuff.'"

Resistance to upgrading must be a matter of concern for Borland executives. But when you think about it, there is a positive dimension to the tendency of some Delphi developers to stick with older versions: A product's shelf life is one indicator of its strength. For example, the Brazilian developer indicated he still uses Delphi 5 and also Delphi 7 for cross-platform applications. As the thread continued, there was much discussion of platforms — especially Win32, .NET, and Linux (I'll save that discussion for a future column).

Overall, the discussion was hardly one of doom and gloom. Based on non-U.S. testimonials of Delphi's strength, one developer added optimistically, "Maybe ... just maybe [Delphi's popularity offshore] will be enough to help keep Borland alive with Delphi sales, but that in itself seems to be a finite market." He expressed the need for "Borland to level their management team and get someone in there who knows what they are doing," criticizing the attitude he perceives as, "We're just a bunch of programmers writing software for programmers." He concluded with this view: "CEO, CIO types really don't care ... all they care about is if [the tool company] is going to be around 5, 6, even 10 years from today. Sadly, Microsoft fits that bill a lot better than Borland does and I'd like to see that changed, but apparently no one at Borland does."

The perennial issue: Upgrading. Many contributors to this discussion thread had something to say about .NET and Delphi 8. One described it as attractive but lacking the "maturity of library support that is one of Delphi's strengths," adding that he needed "to deliver now, today — not later." Another complained about "no books, ... [few] examples, no articles, and no trial version for D8." No articles? I guess this person hasn't been reading this magazine!

However, some of the other concerns are not going to disappear. There was considerable discussion about the lack of new Delphi books. One person wrote, "Software Developers are like technicians: don't consult the Owner's Manual until you hit a problem!" But another responded that, "If we are to keep Delphi alive, we need to attract new developers to Delphi. I would contend that books which are easy to read and physically easy to hold (reading in bed) are the best way to do this compared to PDF files on CDs."

One person had a suggestion that addressed multiple concerns: "Regarding getting newcomers to try Delphi, I'd love to see Borland kick start things with a 'Start Programming with Delphi package' that costs say $100 and includes a version of Delphi that is useable, has free components [he mentioned the JEDI library], useful code, and a 'Start programming' manual."

Getting back to the catalyst for this whole discussion, an earlier contributor added, "Anders and Chuck J. are the true fathers of Delphi. Anders had another shot at building a language and he came up with C#," which he assessed as "a damned good language with some very impressive strengths." But on the topic of .NET, he added that "there are things D8 can do that C# simply cannot do," mentioning the former's support for sets.

Conclusions, recommendations, and predictions. I don't have a crystal ball, but I can suggest possibilities based on current trends. Had Borland not developed tools for .NET (especially the new Delphi version), I would be much more concerned about its future. And I am well aware of the strength of Borland JBuilder and the extent to which it is propping up the company. But I continue to have concerns about Borland's marketing practices. I sincerely hope Borland will examine all its options to ensure its — and Delphi's — future.

One of the Aussie contributors encapsulated much of the discussion thread with some excellent suggestions. Because I concur with him, I will summarize his main points. Borland should:

- Target universities so students can get access to Borland products at a heavily discounted rate.
- Stop hiding — spend a few bucks to let people know there are alternatives to Microsoft development tools.
- Release service packs on a regular basis.
- Commission several good texts ASAP on Delphi 7, Delphi 8, and InterBase.
- Fund a professional mentor program to help budding developers.

We cannot know for certain the future of Borland or Delphi. As shocking as it may sound, Borland could outlive Delphi. In fact, it wouldn't surprise me to see Borland sell or license Delphi to Microsoft at some point and concentrate on other products, especially JBuilder. The one thing that keeps me from predicting this is that it would in essence come down to Borland selling its heart and soul — after all, Delphi is a descendant of Turbo Pascal, Borland's first product.

I will end with one prediction in which I have great confidence: As long as Borland and Delphi are around, we will be talking about the future of each. Until next time...

Alan Moore *is a professor at Kentucky State University, where he teaches music theory and humanities. He was named Distinguished Professor for 2001-2002. He has been named the Project JEDI Director for 2002-2004. He has developed education-related applications with the Borland languages for more than 15 years. He's the author of* The Tomes of Delphi: Win32 Multimedia API *(Wordware Publishing, 2000) and co-author (with John C. Penman) of* The Tomes of Delphi: Basic 32-Bit Communications Programming *(Wordware Publishing, 2003). He also has published a number of articles in various technical journals. Using Delphi, he specializes in writing custom components and implementing multimedia capabilities in applications, particularly sound and music. You can reach Alan at acmdoc@aol.com.*

table of CONTENTS