To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book

(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Index

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Introduction

I have a confession to make. As recently as a year ago, Visual Basic was not my preferred tool to develop client/server applications. I have been developing database applications for two decades and have used many languages and tools. While I will never claim to know it all, I have a good sense for what works in the real world and what doesn't work. As far as client/server development is concerned, Visual Basic 5 seemed to me to be right on the edge as far as being a viable tool for large applications. Visual Basic 6 has changed my attitude.

The client/server developer needs a development *tool*, not a language. The tool needs to be robust, needs to support a myriad of different backends (data sources) and needs to be able to produce a variety of different types of applications from single-user desktop solutions to multi-tiered applications deployed on the Internet. Visual Basic 6 is a superb tool adaptable to those types of projects and more.

VB 6 is not best of breed in all categories. Weaknesses remain in areas such as object-orientation and data modeling. The tools that are there sometimes lack a little in how well they are integrated. Even still, VB6 is today's best choice for rapid application development of data driven applications in a client/server environment.

In this book, my co-author Kurt Cagle and I have gleaned what works and what doesn't from months of work with VB6 betas using a variety of backends and application platforms. Between us, we have almost four decades of real-world experience. We have put together a book that is neither a rehash of the Help files nor a pie-in-the-sky Microsoft marketing brochure. We have assembled a guide to the development of client/server applications that will scale as you need and that will comfortably accommodate the rapid changes in

technology. We have kept our eye to the target audience, the experienced Visual Basic developer, while acknowledging that “experienced” means many different things in many different environments. We do not insult the reader’s intelligence with blow-by-blow details of using the Application Wizard, for instance. But, we also do not assume that the reader has previously built large-scale, multi-tiered, client/server systems. In fact, very few VB developers have built such applications. So, we lay the groundwork as we incrementally build the core knowledge from chapter to chapter.

Chapters 1 through 3 concentrate on the environment within which the Visual Basic client/server application will be running. We discuss the whys and therefores of client/server in terms of the network and the database. We discuss and summarize network issues and database platform issues. We show you how to intelligently design a database and how to use SQL.

In Chapter 4, we cover the gamut of Visual Basic 6 database access techniques, including DAO, RDO, ODBC, VBSQL, and ADO. We offer specific advice on when to use what as well as our opinions on the relative merits (and de-merits) of each approach. Chapters 5 and 6 cover development using DAO and RDO respectively. While most developers will eventually move to ADO, both DAO and RDO will be around for some time to come.

Chapter 7 introduces ADO and OLE DB. We then show you how to use these new tools, taking the time to compare and contrast with the more familiar DAO and RDO models. In Chapter 8 we guide you through the conversion of existing DAO and RDO projects to ADO. We use as scenarios example projects built in Chapters 5 and 6 and provide detailed step-by-step guidance. Again, we provide this guidance in a real-mode manner, resorting to brute-force techniques when such techniques work best.

Chapter 9 gets into more advanced and efficient techniques using ADO and OLE DB. We show you some of the newer tools with detailed examples. We discuss the **DataEnvironment** object, the **DataRepeater** object, the **Format** object, and so on. We make no bones where things are a little rough around the edges, showing you how to get around the flaws in hierarchical data presentations and similar gotchas.

Chapter 10 is where we start hammering away at solid object-oriented development techniques instead of just paying lip service to them. We detail the advantages and disadvantages of different approaches to object-oriented development using classes. We discuss the efficiencies and inefficiencies in different binding techniques. From there, we develop business objects from classes that act as data providers to different objects, optimizing each for the circumstances under which they are deployed. We show you critical gotchas when moving the local business object to a remote platform and, frankly, show you how to plain make it work.

Chapter 11 takes advantage of the knowledge assembled in the first ten chapters to explore advanced database techniques, including the creation and use of stored procedures and triggers. We deliberately placed this chapter at this point in the book both because these techniques help to solidify the traditional client/server application and because the techniques are absolutely

critical to the success of Web-based applications, which we discuss in the remainder of the book. Because so much is said about stored procedures and triggers, and because so few shops actually make effective use of them, we explore unusual uses of them both in terms of data validation and in the creation of a self-referencing data dictionary.

And that brings us to the last third of the book: VB6 and the Internet. While client/server applications are, by definition, network applications, the Internet is the mother of all networks. The Internet connects 79 million people in the United States alone. With servers in excess of 10 million and Web pages in excess of 300 million, the Internet only continues to accelerate its growth.

The astonishing thing is each of those 79 million plus people are all connected to one another and to many millions more around the globe, each one redefining what we mean by client/server programming. As cable modems and DSL lines begin to replace 28Kbps and 56Kbps analog modems, the number of people and uses for the Internet will jump astronomically as will the expectations of users.

Client/server programming is all about getting data to and from the user. Visual Basic 6 dramatically redefines the boundaries between client and server, between data and user, offering a combination of one of the best RAD development environments on the planet with all the power of Active Server Pages and Dynamic HTML. In essence, with Visual Basic 6 you can create sophisticated Web applications targeted to the widest possible audience using the same tools that you have already mastered for other client/server development.

The final four chapters of this book focus on this new technology, with an in-depth look at Internet Information Services applications and Dynamic HTML applications. Additionally, one of the hottest topics in data communications, Extensible Markup Language (also known as XML), is explored in detail, showing how you can take advantage of this new data standard in your own programs. Finally, this book looks at several useful technologies for the Internet-savvy database engineer, including remote component servers, client- and server-side scriptlets, data persistence, and more.

There is no right way or wrong way to use this book. (Well, using it as a door stop or as something to raise your monitor probably isn't the *best* use we could imagine.) The experienced client/server developer may skim or skip over the first three chapters. Those readers new to client/server may wish to spend extra time on those chapters. Those readers that are making a commitment to ADO should at least read Chapters 7 and 9 before skipping ahead to the fun stuff (Web development). Those looking to check out the viability of Web-based development may well want to skip ahead to Chapter 12 before coming back to other portions of the book. We have tried to include at the front of each chapter some keywords to highlight topics to be covered and have tried to include at the end of each chapter some suggestions for where you may want to go next.

Also be sure to check out the code samples on the enclosed CD-ROM. Several

people checked each piece of code. Although we have strived to make sure that every application works properly, the nature of the material is such that it was not possible to test on every single combination of platforms. As an example, you may need to make minor alterations in SQL syntax depending on what database you are connected to. Please pay particular attention to the fact that you will need to do some setup on your own system. Many of the examples use an ODBC data source—after creating the database (we have included the SQL statements to do so on the CD-ROM), you will need to set up the ODBC data source in the Control Panel. The paths to the data in some of the examples may need to be changed to reflect your paths. Also, pay particular attention to the fact that some samples connect to server applications. Since these sever applications need to be registered (see Chapter 10 in particular), you will need to compile them, go to the References dialog box, and reselect the references. Otherwise, you will get mysterious error messages. For the Internet chapters, please note that many of the examples were done using Internet Explorer version 5. You may wish to download that from the Microsoft Web site. We would have included it on the CD-ROM but it was in beta as this book was going to press and a more current release will surely be posted before you read this.

You can use Access for most of the examples from this book if you want to experiment at home or otherwise do not have access to a relational database. You can also download trial versions of a number of different good single-user database engines. Sybase SQL Anywhere (renamed Sybase Adaptive Server Anywhere as this book was going to press) is an easy-to-set-up and use product available from www.sybase.com/products/anywhere/index.html for a free 60-day trial.

If you like the book, please buy a couple dozen more. You can give them to your kids, your spouse, your mother (she will be impressed and thankful) and your boss (he or she will also be impressed and will give you a raise). Coriolis always invites comments, suggestions, criticisms, and so on. Visit their Web page at www.coriolis.com to find contact information or to check out any updates to the book.

Thank you,

Michael MacDonald (**mikemacd@tiac.net**)
Kurt Cagle (**cagle@olywa.net**)

Table of Contents

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

What's On The CD-ROM

The enclosed CD-ROM contains all of the source code examples from this book, plus many bonus examples. You will find them organized by chapter. Please remember to make any changes needed to reflect your own environment. See the readme file on the CD-ROM for more information.

Also included on the CD-ROM is a collection of utilities that I believe you will find helpful. Please note that this book was written while Visual Basic 6 was being beta tested; therefore some of these utilities may have been updated to take advantage of VB6 after the book went to press.

- *Advantageware VB Advantage*—A design-time add-in that gives you the power to quickly and easily customize your design environment with a broad range of powerful tools. (Trial version)
- *Apex Software: True DBGrid Pro, True DBInput, True DBList Pro, True DBWizard*—Enhanced controls to help speed up your coding.
- *Bokler Software Hashcypher*—An ActiveX control that allows you to easily provide robust data encryption using the Secure Hash Algorithm (SHA-1). (Trial version)
- *Caladonia Systems Code Print Pro*— Allows you to generate formatted, professional documentation that spans multiple projects or specific sets of routines. (Trial version)
- *Catalyst Socket Tools*—A collection of TCP/IP networking components and libraries for Windows that assist you in developing Internet and intranet applications. (Trial version)
- *CoffeeCup HTML Editor*—A top-rated, WYSIWYG HTML editor with many new and cool features to create and deploy web pages with "attitude." (Trial version)
- *DeltaPoint QuickSite*—The most productive way to create, manage,

and sell through your Web site. (Trial version)

- *Suprasoft Crystal Reports Crystal Design Component*—A custom component that encapsulates the previewing of Crystal Reports reports inside of an easy-to-use control. This is an efficient and productive building block to create a unique and customized preview application or front end for your software. (Limited version)

Requirements:

- Minimum of 486 or equivalent processor
- 16MB RAM
- Windows 95/98/NT
- Visual Basic 6

Table of Contents

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

About the Authors

Author of seven books and dozens of magazine articles, **Michael MacDonald** has nearly 20 years of experience in software development covering all major platforms. An independent consultant specializing in client/server and databases, Michael is also an instructor in the client/server certificate program for Worcester Polytechnic Institute and a courseware developer.

Kurt Cagle is an Internet developer and multimedia specialist with 18 years of experience in the field, concentrating on client-side and interface issues. He is the author of 3 books and more than 50 magazine articles on many aspects of the computer revolution, and has done work for Microsoft, AT&T, RealNetworks, Starwave, and many other major computer and technology-related corporations.

Acknowledgements

This is the area where I always get to thank everyone who so much as returned a phone call. But, there were some very special people behind this book. I am especially indebted and grateful to the publisher itself, Coriolis Group Books, for allowing the extra time to produce a book that was solid instead of a book that was early to market. It says a lot about the integrity of an organization that puts quality ahead of profits. I want to thank the Acquisitions Editor for this project, Stephanie Wall, for giving me the opportunity to do this book and for a lot of cheeriness and professionalism. Toni Zuccarini was the project editor, which is a job that does not get near enough thanks or recognition. Toni coordinated a myriad of details with grace and aplomb. John Lueders was the technical editor and painfully (to me as well as to him I am sure) reviewed every line of code in the book and on the CD-ROM, offering advice and criticisms, and did it well. Kristine Simmons was the copy editor for the project. Copy editors are those people who review manuscripts for clarity and

grammatical correctness. Kris offered many fine suggestions and untangled sentences and thoughts far better than I ever could have. I have worked with many copy editors and (I mean this in all sincerity) Kris is probably the best. I have to acknowledge the contributions of my co-author, Kurt Cagle, who stepped up to bat to handle Chapters 12 through 15 and really did a fantastic job. There are many people who go into the creation of a book who work behind the scenes without much in the way of recognition: Alan Pratt is the Technology Director and keeps, well, technical things moving between Coriolis and authors; Robert Clarfield coordinates a blizzard of details in getting that CD-ROM into the back of this book; Wendy Littley is the Production Coordinator and is in charge of getting the text laid out, getting proofs, coordinating among a zillion people, and so on; Tony Stock designed the cover of this book (and you thought I did that myself!) invisibly but professionally; and there are many others. To all of them—thanks.

I need a vacation!

Dedication

As always, to my family—my wife, Patricia, and my children, Amanda and Peter—who are the ones that
truly make the sacrifices when I work past midnight. Also, to some very special colleagues who it has been my
privilege to work with for more than a decade, including Larry Altrich, Steve Remmes, and Randy Vance.
And finally, to a person who took a neophyte under his wing almost 20 years ago, Dr. Thomas Gross.

—Michael MacDonald

To my parents, who taught me that humor and compassion will defeat arrogance
and stupidity every time.

—Kurt Cagle

Table of Contents

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Previous | Table of Contents | Next

Part I
Basics Of Client/Server Development With Visual Basic 6

Chapter 1
An Introduction To Client/Server And Networks

Key Topics:

- How we arrived at the relational data model
- Distributed computing
- The definition of client/server
- 2-tiered, 3-tiered, and n-tiered architectures
- The Microsoft Services model
- COM/DCOM and the Component model
- Application partitioning
- Network topologies and architectures

The migration to client/server development has its roots in the nature of networks, which excel in the sharing of data and devices and the cooperative processing of multiple clients, but are hampered by problems endemic to the limited amount of data that can be transmitted at the same time. Client/server solutions seek to maximize the strengths of networks while limiting their weaknesses.

In this chapter, I introduce the underpinnings of client/server and how it evolved from the distributed data model of the 1970s and 1980s to the multi-tiered model of today. For some readers, this material is old hat; skim it or move on to the next chapter. For others, however, the material provides real-world background upon which to set in context the remainder of the book. The material is as applicable to development with any language as it is to development with Visual Basic (VB).

Organizationally, this chapter is far ranging in the topics covered. I discuss the evolution of file formats and how they led to the relational model of today and the object model of tomorrow, as well as how these files evolved to support the distributed data format that was a direct ancestor of client/server today. I define basic and advanced client/server architectures, and I conclude with a review of the network topologies and architectures upon which client/server is based.

A History Of Files

Since the inception of the computer, there have really been only three file types: sequential, navigational, and relational. We are now seeing the introduction of a fourth type, the object model, along with relational/object hybrids. Concomitant with this evolution has been a shift of responsibility from the application to the database. For instance, in the sequential model, the application was responsible for the location of any given record as well as for the enforcement of all business rules. In the following sections, I review the evolution of files; in other sections of the chapter, I point out how this evolution mirrors the move to client/server development.

Sequential Model

Sequential files consist of fixed-length records, each of which typically contains fixed-length fields. The 80-column cards of old were essentially sequential files as were (and are) files stored on magnetic tape. A variation on the sequential model, text files, evolved when computer languages evolved to the point where they could use a character (such as a carriage return) as a record delimiter instead of depending on the end of a record being in a fixed location. Files such as AUTOEXEC.BAT and CONFIG.SYS are text files.

Visual Basic supports both forms of sequential files, using the **Open** keyword to open the file, **Close** to close the file, and a variety of keywords to read from and write to the file depending on the nature of the file's use. I review these in Chapter 4.

The key advantages of the sequential file model are simplicity of programming file access and the speed with which an entire file can be processed. On the other hand, it is not possible with sequential files to locate any one record without first reading all other records before it. For instance, to pull up John Smith's customer record, it is necessary for the program to read the first record in the file and determine whether it is the correct one. If it is not, the program must read the next record and so on until the desired customer is found. As a result, the sequential file model also does not support any type of ad hoc

reporting capabilities.

The application program is responsible for all file I/O, for all record searches and updates, and for all data integrity measures (such as ensuring that an order has a valid customer).

Some improvements were realized by using random access techniques. In theory, if your application knew that John Smith's record was the 318th record in the file, the program could move directly to that record without reading the first 317 records. The caveat, of course, is that the application was then responsible for creating and maintaining some sort of index into the file—not an easy piece of logic to code. Programming languages had to evolve to support this as Visual Basic does when files are opened in Random mode. Random access requires that every record be fixed length. A variation is Binary mode where, instead of accessing a record by specifying its record number, the program uses a byte offset from the beginning of the file.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Navigational Model

The navigational file model—sometimes called the hierarchical model—was a major improvement. The term stems from the fact that it was usually necessary to "navigate" to a desired record by first accessing other records. Often, the details of this navigation were hidden from the developer so that the database did the dirty work of locating a specific record. Common examples were dBase and Btrieve files. On the mainframe, the navigation model was implemented with Virtual Sequential Access Method (VSAM).

Visual Basic supports the navigational model in two ways. Using a file opened in random access mode, you can manually maintain some sort of hashing algorithm to navigate from record to record. Often, this method is implemented in a manner similar to what C programmers know as *linked lists*. A linked list essentially maintains a series of pointers into a structure or file allowing navigation by (for example) account number or customer name. Visual Basic also allows connection to navigational databases via the Microsoft Jet engine. Although the interface provided is SQL, the underlying file is still accessed navigationally. Microsoft refers to this as *Indexed Sequential Access Method* (ISAM).

Relational Model

The relational model organizes data into related *tables*. A table is a two-dimensional grid of columns and rows where each row is a record and each column is a field in that record. Each table has a unique identifier, called a *primary key*, which allows the retrieval of any one row without referring to any other row. An example of a primary key might be **customer number**. Tables are related by common information. An **Orders** table would be related to a **Customer** table via a common **customer number** column: Each table has a column containing **customer number**. The relationship is enforced via a *foreign key*. The foreign key is a special type of index on the database that

enforces a business rule such as “no order may exist without a valid customer number.” This enforcement is called *referential integrity*. The database typically sports other methods of maintaining data integrity, such as restricting the value of **gender** to male or female. These enforced rules are called *constraints*.

There are a number of vendors of *relational database management systems* (RDBMSs), including Oracle, Informix, Sybase, Microsoft, IBM, and so on. All of the RDBMS engines use SQL as a language for accessing and manipulating data.

The majority of this book is devoted to using Visual Basic in an environment with an RDBMS such as Oracle, including techniques for maximizing performance of the database as well as maximizing developer productivity. The next two chapters introduce RDBMSs and the use of SQL.

Distributed Computing

To understand the move toward client/server, it is necessary to digress briefly to the problems inherent in the centralized processing models of the mainframes of the 1960s and 1970s. All data, along with the application programs written to manipulate that data, resided within the confines of a centralized computer. For a company with a single office in Boston or Chicago, this arrangement was fine. However, for a company with offices around the country or around the globe, this created practical problems: If the home office (and the computer system) was in Chicago, office workers in Boston and Los Angeles had no practical access to data relevant to their operations.

The concept of the Distributed Computing Environment (DCE) sought to mitigate this problem by moving data closer to users on multiple machines. This solution had the added benefit of spreading processing loads to multiple machines. On the other hand, it was difficult to update information that was spread over multiple computers. It was also difficult to merge the data back together for reporting.

As networked computers evolved, so did the proliferation of DCE. Networks were (and are) a natural platform on which to distribute data. Contrary to popular belief, networks have been around almost as long as computers. In 1964, the United States government contracted with the Rand Corporation to design a network that could continue communicating even if a portion of the network was destroyed in a disaster. The result, ARPANET, came online in 1968 and eventually evolved to what is known today as the Internet.

Networks stretched the definition of DCE by adding the concept of sharing resources, such as printers and other expensive hardware. As the expense of networks decreased and capabilities increased throughout the late 1980s, the network exploded into prominence in corporate America.

With the increased use of networks, sharing data on file servers was a natural happenstance. This development, however, revealed the weakness of networks: The bottleneck on nearly any network was and is the low bandwidth—the amount of data that can be moved over the cabling. As an

example, a “high-speed” Ethernet network can theoretically move data at 100Mbps (megabits per second). In reality, the “wire” (the network cabling) is shared by many users, so even though the theoretical throughput sounds high, it dwindles as more users are added to the network. Worse, the wire can only send one piece of data at a time. Therefore, as more users are added, the potential for two users attempting to send data at the same time increases. When this happens, a data *collision* results. Any time there is a collision, the data has to be retransmitted. Thus, as network utilization increases, performance degrades very rapidly. Client/server solutions seek to take performance issues into account by minimizing the amount of data being moved.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

What The Heck Is Client/Server?

Client/server is a term much bandied, sometimes by people who don't have a clue about what it means. In fact, many industry pundits will stand on a podium and tell you that there is no single definition of client/server. Nonsense.

Client/server computing simply means that two (or more) processes run *independently* in a cooperative manner. The simplest and most common example is the classic two-tiered database application shown in Figure 1.1. Here we see a client program communicating with a database engine running on a remote server, and the two are connected via a network. The client is responsible for providing an interface to the user. Typically, the client will create an SQL request for data and send that request to the database. The database then evaluates the request, fulfills it, and sends the data back to the client.

Figure 1.1 In a two-tiered, client/server application, the client requests records from the database server, which processes the request and returns only those records that meet the criterion.

Contrast this setup with just a shared file on a server. (I discuss this concept further in Chapter 2 where I define the term *RDBMS engine*. With a shared file, there is no separate process—program—running. The file, perhaps an Access database, is merely shared by multiple users.) Let's assume that we have 10,000 customers for whom we store information. We want to find all customers who live in California. Suppose 50 customers are from the Golden State. With a shared file, the program has to read the entire file and determine

which customers are from the state of California. To do that, all 10,000 records are sent from the server to the client. The user is going to wait a very long time for a response.

In a client/server environment, the client creates a SQL statement such as **Select * From Customer Where State = 'CA'**. The request is sent to the database server. The server then evaluates the request and finds the 50 records before sending any data back to the client. Because only 50 records are sent to the client (instead of 10,000), network traffic is reduced dramatically.

The simplistic model that I have just illustrated works fairly well when the number of simultaneous users—such as in a departmental application—is not too high. However, it begins to sag when we try to *scale* (grow) it to the enterprise level. In the sections that follow, I address this issue as well.

Multitiered Architectures

With the terms *two-tiered*, *three-tiered*, or even *n-tiered architectures*, we are referring to the number of layers of communications that make up an application. Each term represents a division of duties. If you write an application program that communicates with a database server, you construct a two-tiered application. The database is primarily responsible for returning data to the client. The client is primarily responsible for providing the user interface and the business logic. This structure was shown in Figure 1.1.

There are a number of problems with this approach.

Although a goal of client/server is to minimize network traffic, the fact remains that if you add enough users, the network will still be crushed. For instance, the database itself can support only so many active, simultaneous connections before it begins to sag performance-wise, simply due to a lack of resources (memory and so on). Also, you tend to reach a point where the network traffic still becomes too high. If your multiuser FoxPro system could support 5 or 10 users, the two-tiered client/server equivalent could probably support 50 or 100 (because the client/server approach puts much less strain on network resources than does the shared-file approach used in a FoxPro application). Beyond those 50 to 100 users, however, the network still can't handle the amount of data being moved around.

Other complications come from the sheer complexity of maintaining that many clients. Assume you do manage to get 1,000 users on your client/server application. Each time the application needs to be maintained, all 1,000 PCs must be updated. That is a Herculean task for which no good administrative tools exist. Further, today's graphic-heavy environments and intensive computations require ever bigger and faster PCs, creating an endless cycle of expensive PC upgrades and replacements. Therefore, the obvious answer is to offload as much processing as possible to an *application server*. In this scenario, the client application is mainly concerned with presenting a user interface and does as little business logic as possible. The business logic is moved to a server that performs business-oriented processing. The application server manages connections to the database and performs all data requests of the database. The database also performs a limited amount of logic by

enforcing business rules on the server. This three-tiered model is illustrated in Figure 1.2.

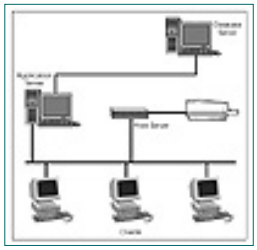

Figure 1.2 In this three-tiered system, the client connects to the application server, which in turn connects to the database server.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

A more familiar example of a three-tiered application might be what you see on the Internet. Visit a site such as www.amazon.com (an online bookstore) and search for books with “Visual Basic” in their titles. Your PC is the client in this model with the Web browser containing the user interface. It connects to a Web server at Amazon’s site. The Web server’s main job is to create Hypertext Markup Language (HTML) pages to be sent to the Web browser on the client. Extensions on the Web server perform some business logic and interact with a database server. The Web server sends a request to the database server asking for all titles that match your search criterion. The database server processes the request and sends the result back to the Web server. The Web server then formats the result set into an HTML page and sends that back to the client for presentation within the Web browser.

Beyond overcoming practical limitations imposed by the two-tiered design, a multi-tiered design offers other advantages that exploit the location and nature of each physical tier. Figure 1.3 shows a group of PCs networked together with the application server on the same ring. Notice that the client PCs can communicate with the application server, but only the application server communicates with the database server. This arrangement has the effect of reducing a certain amount of data traffic on the network because the database server isn’t even on the network. Thus, PCs on the network that are not part of the client/server application itself suffer minimal impact on network performance.

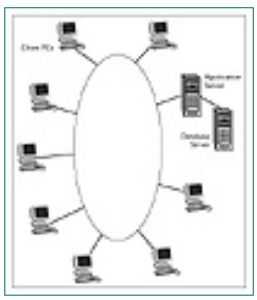

Figure 1.3 A three-tiered configuration with the client computers communicating with the application server via the network.

The Services Model

Visual Basic supports an approach to client/server in three-tiered and n-tiered designs known as *partitioning*. The application is broken into three key service areas, which typically partition the application into three discrete tiers. Together, these tiers are known as the *services model*:

- *User services*—Essentially the way the application interacts with the user.
- *Business services*—The processing necessary to accomplish business goals, such as editing a customer record or decrementing inventory levels.
- *Data services*—The maintenance of the database itself, such as updating tables, transaction management, and concurrency support (row and table locking).

Each of these areas is encapsulated into a tier of the client/server system. The client handles the user interface, the application server handles the business logic, and the database server performs data-handling chores. I expand upon these concepts throughout the book.

The Component Model

With Visual Basic, the *component model* extends into the client/server environment. Consider the word "component." Visual Basic supports the creation and use of objects developed with Visual Basic, Visual C++, and many other development tools by independent teams of developers. These objects have functionality embedded in them that can be used repeatedly in many different projects. As such, the objects become components in what is called *component-based development*. Rather than re-create a routine to calculate the sum of all line orders on an invoice, for example, we create one routine as an object that we can then use almost as a building block in other applications. Consider, as a simple example, the Common Dialog control that you add to your application when you need a File Open or Printer Setup dialog; rather than re-create it yourself every time you need that functionality, you have a prebuilt component that you can reuse from project to project.

When we look at a client/server model as a group of services (user, business, and data), then each service is a collection of assembled components.

Assume you use Visual Basic 6 to build an ActiveX object called **custUpdate**. The object is nongraphical, consisting only of code that updates customer records. Encapsulated within this object is logic that enforces business rules such as "a customer may not exceed $500 in credit" and "a customer must have a date of birth less than today's date."

Your **custUpdate** object is a component that can be deployed at the client level with no special coding because of the benefits of Microsoft's Component Object Model (COM) technology. Any tool, platform, or object that supports

COM automatically “knows” how to communicate with the object and can take advantage of its methods and properties.

More On Business Services

Client and data services tend to be easy to define, but the definition of the term “business services” can be somewhat muddled. For efficiency of design and practicality of deployment, anything that is not a client service or a data service is usually lumped into the business services bucket.

The user interface clearly belongs on the individual user PC. This includes forms, controls, screen navigation, and so on. Anything that performs physical data retrieval or maintenance—that is, the database engine—belongs in the data services area. Other items that can be shared by multiple clients (or the data services partition) then become a part of the business services partition.

An object deployed to handle multiple print chores (reports, pagination, and so on) is not, strictly speaking, a business service. However, it is not a good candidate to be deployed on each client nor should it be placed on the database server. The most convenient place to put it is on the application server as part of the business services partition.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Distributed Component Object Model (DCOM) technology extends this concept by allowing a component to be placed anywhere on a network. Any other object can communicate with it without knowing or caring where it is on the network. Assume you have a client/server application that uses the Internet as its network. Your Visual Basic application can invoke methods of **custUpdate** (without any change in coding) regardless of whether it is located on your computer or on a server on the other side of the world.

You can take advantage of DCOM by deploying all your business logic on an application server. Instead of having a **custUpdate** object on every client in the network, the object is located only at the server. The object operates independently of any client and independently of the database server. Should a business rule change (perhaps you will allow a customer credit of up to $1,000 instead of $500), you have only one place to make the change and that change is instantly made available and enforced for all clients.

By placing all your business logic on an application server, you have effectively partitioned the business services from the user services (and from the data services).

The Thin Client

One of the buzzwords of the day is “thin client” (okay, that’s two words). In the two-tiered client/server model, all of the user services and all of the business services are located on the client, whereas the database server handles the data services. This arrangement places a strain on the client because the programs tend to be big and slow and contribute to the need to continually upgrade PCs across an organization. Unfortunately, as we (client/server developers) write programs that are increasingly graphical in nature, we contribute to this bloat.

By moving the business services to an application server, the client need do

little more than present forms, menus, and the like to the user. All of a sudden, we are actually taking some of the strain off the client and *fewer resources* (such as memory, disk space, and CPU) are needed. The direction we are heading, then, is towards a *thin client*.

Visual Basic 5 introduced the capability to create applications that actually run within a Web browser. No form or menu runs on the client; it is housed in the browser. The browser then becomes the user interface. If the user wants to access a client/server order entry system, he or she launches Internet Explorer (or Netscape Communicator). If he or she wants to access the accounting client/server application, he or she launches Internet Explorer. If the PC is powerful enough to run Internet Explorer, it is powerful enough to run any of the client/server applications. This method is how you implement a thin client. It means that the application is truly network independent. If the network supports TCP/IP, it supports your application. It also means that your application is operating system independent. If the client has a Web browser that supports active clients, it can run your application. Your Visual Basic application system is portable over multiple operating systems.

Lest I sound like a walking billboard advertisement for Microsoft, let me be the first to say that the technology is not perfect. Can you do everything I just said in the preceding paragraph? Yes, sort of. You may be constrained by network limitations. (The Internet, for instance, has been known to get bogged down from time to time and that is beyond your control.) The range of database connectivity options and methodologies (a major subject in this book) is confusing to say the least. As the technology matures, it is probably wise to tread lightly and test carefully. Still, it is an exciting trend, and it will ultimately simplify development enormously.

Designing The Client/Server System

One of the keys to implementing Microsoft's DCOM in a client/server environment is proper identification of both service requirements and their logical location in the physical implementation. Put in simpler terms, what does the application have to do and where should it be done? Part of the process of ferreting out this information is traditional system analysis, part is the application of well-defined object-oriented principals, and the final part is the somewhat less clear process of properly partitioning the results.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Systems Analysis

Assume that we are building a traditional order entry system. The first step is to gather detailed requirements from users. We can use a number of well-established, formal methodologies (on which whole books have been written). Those experienced in the field often use a somewhat less formal approach. In this day and age of rapid application development (RAD), many companies turn to *joint application development* (JAD) brainstorming sessions. In these JAD meetings, users and technicians meet to hammer out first the functionality of a system and, normally, the data elements to be captured. The latter form the basis for the database design, which I review in Chapter 2. For instance, in an order entry system, typical functions will include the following:

- Add, update, and delete customer.
- Add, update, and delete order.

It is useful to compare the functions to the database design in order to verify that each database entity has at least one occurrence of add, update, and delete (to ensure that there is a mechanism to add a customer).

It is often helpful to bring senior management to at least the initial JAD meeting to clarify the organization's goals. It is not uncommon to encounter a sizable disjoint between what senior management sees as the organization's philosophy and what the rank and file sees. Management can provide a mission statement from which all functionality is derived: "Sell widgets worldwide."

The mission statement then can be viewed as the highest-level function of the organization. That function can be *decomposed* into smaller units that define how the overall mission is accomplished (see Figure 1.4).

Figure 1.4 The beginning of a simplified functional decomposition.

Each function is decomposed further until an *atomic-level* function statement is achieved. A function that has been decomposed to the atomic level generally provides one and only one output. An example of an output is a report or a table being updated. (In practical terms, you may then opt to combine two functions to make the application more functional. For instance, it is common to maintain order headers and associated line items on the same form.)

As you can see, you are not defining *how* the application will accomplish a task. Instead, at this stage, you are merely concerned with *what* the task is.

Because of space constraints in this book, I cannot explain all of the ins and outs of the different approaches to systems analysis and design. I spend a good 40 hours teaching these skills to students in what is still a compressed format. However, I end up stressing that the whole process of systems design comes down to practicing common sense. Decomposing functionality is particularly troublesome to my students who invariably are looking for the one right answer. I have to tell them that there is no one right answer. If the decomposed functions accomplish the goals of the system, then it is correct. In general, there is a one-to-one correspondence between an atomic-level function and a "program" (such as a Visual Basic form or report).

Once application functionality has been defined, you are ready to think in terms of how to make it happen. There are two aspects to this: defining the data that needs to be captured and a description of how the data will be manipulated. (In Chapter 2, I discuss how to design a database.)

Encapsulating Functionality

Another source of confusion is the concept of *encapsulating* functionality. All that this intimidating word (encapsulation) means is embedding functionality into another object such that its behavior is exposed but its integrity is protected. Okay, maybe that is not so easy to understand. Consider the database server. When you send an SQL statement to the server asking it to update the address of a customer, you are not doing the actual update, the server is. The database engine edits your SQL statement for validity, ensures that the customer number is valid, verifies the state or province and postal code, then performs the physical update. It places an entry in the transaction log and informs your program whether the update was successful. You cannot access the underlying data or functionality directly; it is encapsulated within the database server itself.

In designing your system, you seek to do the same thing—encapsulate your

application logic. If you need to perform customer maintenance, you seek to isolate the data and logic. Isolating the data allows you to guarantee the integrity of the data. For instance, in updating a customer's address, you might want to validate the state or province code as well as the postal code. Therefore, you could embed that logic into an object. By embedding the data in that object, you force all requests to update the data through the object's methods. This embedding is a recurring theme throughout this book, and I expand upon this concept in the following section.

Partitioning The Application

Hand in hand with encapsulating functionality, partitioning the application allows you to reuse objects and locate those objects at their proper place in a multi-tiered client/server application.

In partitioning an application using Microsoft's DCOM, you break each function into three discrete sets of services: user, business, and data. This partitioning can be seen in Figure 1.5. The client interface is localized to the client computer under user services. The logic to process a customer record and an associated address record is localized to the middle tier under business services. The database server handles data services.

Figure 1.5 An application broken into three tiers of services.

To build such an application, you need to create two (or more) Visual Basic applications. The client services portion is actually rather simple, needing only to provide an interface to the user and the knowledge of how to communicate with the business services tier. The business services portion is a separate program running remotely (from the client) to which multiple clients connect. Visual Basic provides a rich set of tools to build these portions. However, the hard work has already been done for you.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Previous | Table of Contents | Next

Distributed COM

When you build an ActiveX component and then use it in a VB project, COM provides a *proxy* and a *stub* for each side of the connection between your application and the component. A proxy is an object provided by COM that runs in the same memory space as the user. It packages together parameters and acts as a bridge between the two objects. If your application is communicating with an ActiveX component, the proxy runs in the memory space of your application. It collects arguments to be passed to the ActiveX control's methods and properties and communicates with the component's stub. The stub, on the other hand, runs in the address space of the receiving object, unpackages the parameters received from the proxy, and passes them to the component. This communication works both ways, as shown in Figure 1.6.

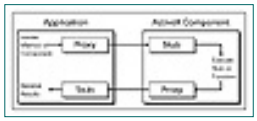

Figure 1.6 COM and DCOM automatically provide proxy and stub services, allowing two objects to communicate with one another with little to no intervention on the part of the developer.

The workings of DCOM are almost identical. The application houses a remote automation proxy, which communicates with the component's remote automation stub. DCOM also adds network transport services in a manner that is invisible to the application, making the location of the component on the network irrelevant to the client. This provides for tremendous flexibility in design.

Visual Basic (Enterprise Edition) provides tools to register components. I review step-by-step in Chapter 10 the entire procedure for implementing the DCOM.

In the remainder of this chapter, I review some of the key concepts of the underlying network architecture. Although the client/server developer does not need to be a network engineer, it is helpful to understand the basics of the physical model upon which the application is based.

Network Topologies And Architectures

The backbone of any client/server system is the network on which it is installed. It is the network that moves data from the server to the user. The network might be a 10Mbps Ethernet or it might be the Internet itself. Although network design is well beyond the scope of this book, the Visual Basic client/server developer should have a grasp of how his or her data is moved in order to properly model the application.

The expression *network topology* refers to how the network is physically laid out. In contrast, the expression *network architecture* refers to how the network is implemented on that topology. In the following sections, I briefly discuss the more common topologies and architectures.

Peer-To-Peer Networks

A peer-to-peer network is one in which the computers are connected to one another directly in series. Each PC typically contains a LAN adapter or Network Interface Card (NIC) with two jacks. Typically, these jacks accept RJ-11 pins (similar to the connectors on your phone cord). A line is run from one computer to the next serially. Although it is simple to set up, such a network is typically slow because data passes through each computer. It is vulnerable to any individual workstation being powered down or crashing, and because the network has no server, there are minimal security constraints. A Windows 95 or Windows 98 network is a peer-to-peer network.

Star Networks

The star network topology implies a network where all computers are connected to a *hub* (a hub is a box similar to a telephone switchboard containing minimal intelligence). Multiple hubs are connected to one another. Star networks are simple to maintain but offer minimal security constraints. If a hub goes down, the entire network goes down.

Ring Networks

In a ring network, all computers are connected via a continuous cable. A multiplexing unit monitors the network, allowing only one packet to circle the network at a time. The packet is continually sent from computer to computer. Each computer examines the packet to see if it is addressed to that workstation. If a computer needs to send information, it waits for the packet and attaches a “request” to transmit. The multiplexing unit gives permission to each workstation in turn in a procedure that is akin to a parliament (giving each “member” a turn to speak but also allowing for higher priority speeches such as error messages). If the packet is accidentally destroyed, the multiplexer recreates it.

IBM’s Token Ring architecture runs on a ring technology. It has the advantage of being very reliable, but because of the extra overhead of packet monitoring, it is somewhat expensive to maintain and does not scale to a large number of users.

Bus Networks

The bus topology connects computers in a single line of cable. Although each computer is typically connected to a hub, the internal wiring of the hub still connects each computer serially. When a workstation wants to send a packet of information, it “listens” for the line to be clear through a process known as *electronic sensing*. Only one packet can travel on the network at a time, and if two computers happen to transmit at the same time, a packet collision occurs. Each workstation listens for such collisions and, when they are detected, waits a random amount of time and then resends the packet.

The most common implementation of bus technology is Ethernet. Although fairly simple and inexpensive to install and maintain, the network is vulnerable to any breaks in the cable. Further, as more users are added to the network, performance can degrade drastically. Still, Ethernet is by far the most common architecture. To maintain satisfactory performance, network engineers break a large Ethernet network into smaller LANs connected via bridges.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Previous | Table of Contents | Next

Network Protocols

In the 1980s, when we logged onto a bulletin board, we said the two computers were “shaking hands”—agreeing on how to communicate with one another. Today, we more accurately refer to this process as agreeing to a protocol. A protocol is a low-level language with which two or more computers communicate. Imagine you send a 2MB file to a friend across the company or across the globe. The file is broken into packets. Each packet has various pieces of information in it:

- *Destination address*—The computer to which the packet is being sent.
- *Usually a “from” address*—The computer or person from which the file was sent.
- *Checksum*—A mechanism by which the receiving computer can verify the integrity of the data being sent.
- *Sequence number*—A number that enables the receiving computer to reassemble all the packets into one file and to tell the sending computer which packets to resend in the event of transmission errors.
- *Data*—The actual data being sent.

A number of protocols are commonly used in organizations today. Most are based on the *Open System Interconnection (OSI) Reference Model*, which defines standard network layers of communications. An in-depth discussion of these protocols is beyond the scope of this book. However, protocols that adhere to this standard can generally intercommunicate with relatively inexpensive translation hardware. In general, you can think of a network protocol as similar to an agreement at the United Nations to speak in one certain language that everyone understands.

The most common protocol in use is TCP/IP, the language of the Internet. The TCP (Transport Control Protocol) portion refers to how a packet moves from

application to application. The IP (Internet Protocol) refers to how data is moved from computer to computer. Although it is used on the Internet, TCP is widely implemented on various internal networks within organizations.

An in-depth discussion of protocols is beyond the scope of this book. For now, understand that different networks employ different protocols and often employ more than one protocol simultaneously. IPX/SPX is the native protocol of NetWare networks. IBM LAN Manager's native protocol is NetBIOS. Other protocols that you may run into range from NetBEUI (Microsoft Windows NT) to AppleTalk (Apple Macintosh).

Network Operating Systems

Like the individual client PC, the network server runs an operating system, usually called the Network Operating System (NOS). It is the NOS that is responsible for network traffic, security, shared services (such as print and file services), and group services (such as email and calendar functions). The choice of NOS can have a dramatic impact both on performance and on productivity. Throughout this section, I deliberately mix consideration of NOS with the database server's operating system. Strictly speaking, the database platform's operating system is independent of the network's NOS. However, the considerations for one are often the same as for the other. Further, I strongly urge every organization to minimize the number of different components. If you are going to use Novell NetWare as your network operating system, I highly recommend that you consider it for your database server as well.

The network itself consists of one or more servers, each providing one or more services. For instance, a given computer may be responsible for both email and print services. The best choice for overall NOS may not be the best choice for your database server's operating system.

In evaluating what NOS to use, you need to ask a few questions:

- What NOS does my organization currently use? Often, it is best to stay with what you have. A large-scale client/server migration introduces a lot of change as it is; it may not be the best time to also introduce a new network operating system.
- What expertise does my organization have on staff? NOS skills are often hard to find. If your staff's skills lie in the area of Windows NT, you might not want to even consider a Unix or NetWare platform unless you are confident that you can hire people with the necessary skill sets.
- What is the overall load on my network? Some NOSs lend themselves to certain types of environments better than others. When you consider the database server, for instance, Oracle presents a prettier face on Windows NT than it does on Unix. However, the Unix platform under many conditions will outperform the equivalent NT environment.
- How many resources can I dedicate to administering my network environment? For organizations that cannot afford a great deal of devoted resources, the highly graphical nature of the administrative tools in Windows NT may more than offset the tedious command-line

interface of a Unix platform.

In the following sections, I review the more common network operating systems that you are likely to encounter.

Novell NetWare

Once the king of the hill, NetWare has seen its market share steadily decline relative to the Microsoft Windows NT juggernaut in the past few years. Whether the declining market share is deserved is a subject for conjecture.

NetWare is known as a *dedicated* operating system, meaning that the server can run no programs other than the operating system itself. Other programs you want to run must actually be part of the operating system. You accomplish this arrangement by compiling the other program (email or database are examples) into a NetWare Loadable Module (NLM). When the NOS loads, any NLMs load as well and become part of the operating system itself. There is a clear advantage to this system in that, because the NLMs are part of the operating system, they run much faster than they otherwise might. They can take advantage of operating system services (such as file I/O) without going through intermediate drivers and link libraries. On the other hand, if the application crashes or is otherwise poorly behaved, the operating system will also crash (because the application and the operating system are one and the same).

A few years ago, Novell sought to expand its presence in the workplace with the acquisition of Quattro Pro and WordPerfect. Many in the industry believe this was a serious tactical blunder, removing Novell's focus from the network arena, where it had always been the market leader, and allowing competitors, particularly Microsoft, to make inroads. Novell eventually sold those products to Corel, but by then, the damage had been done. Regardless, NetWare has a large presence in the corporate arena and a strong suite of tools. An ironic side benefit of NetWare's market erosion is a relative abundance of talent to support the product set.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

IBM LAN Manager

Once upon a time, IBM and Microsoft codeveloped LAN Manager. Unlike NetWare, LAN Manager is not a dedicated operating system. It can run other programs independent of the operating system. Coupled with OS/2 and various DB2 tools, it forms a strong platform for shops with IBM mainframes. Never a large player in the network arena, IBM nevertheless remains a healthy company with a wide variety of tools. OS/2 itself has been unfairly maligned for its lack of software products because it is capable of running a large subset of Windows applications. Arguably, it is a more stable platform than some Windows platforms (although probably not as stable as Windows NT itself).

Banyan Vines

Banyan's product has never garnered as large a market share as it probably should have, always operating in the shadow of other players in the marketplace. Like Novell NetWare, Vines is a dedicated operating system. Although it is a solid platform for file services, I am not aware of any major database products that run on it.

Windows NT

The Microsoft juggernaut seems to roll on with Windows NT. The operating system was developed to replace Microsoft LAN Manager, which was jointly developed with IBM in the 1980s. Like Windows itself, NT did not enjoy widespread acceptance for several years. NT version 3.51 seemed to have been the right answer for organizations that began to embrace it as a viable network platform. NT version 4 is a robust operating system, although arguably not as powerful or scalable as Novell NetWare 4.x. In particular, the word among network professionals is that Windows NT 4 seems to hit a wall at about 1,100 users. Likewise, its directory services are weak in comparison to Novell

NetWare's. However, NT 5 (in beta as of this writing) seems to answer many of these complaints.

Windows NT is a nondedicated server, meaning that other programs can run along with the NOS itself. Microsoft traded some resource intensiveness (Windows NT really needs more memory and CPU speed than most other NOSs to run comfortably) against a high degree of stability. The operating system spends a fair amount of CPU cycles enforcing integrity of operations. For instance, it does not permit any program to directly address any hardware, and it also prevents applications from writing to memory outside of their own address spaces. Windows NT sports a solid set of graphical, if somewhat complicated, tuning devices to maximize performance.

Reportedly, Windows NT represented 50 percent of all network server licenses as of February 1998.

Where To Go From Here

This chapter has been a high-level overview of the nature and evolution of client/server, an introduction to Visual Basic's role in the network, and a review of the underlying network architectures. The next two chapters deal specifically with the database server portion of the network, whereas most of the remainder of the book deals with Visual Basic's role in the creation of two-tiered and three-tiered client/server applications.

The heart and soul of client/server are, in fact, the database and database access techniques. Therefore, if you are unfamiliar with database design, you will want to read Chapter 2. If you are not familiar with SQL usage for retrieving and updating the database, you will want to read Chapter 3.

Experienced database developers may want to skim Chapters 2 and 3 and begin with Chapter 4, which is a high-level overview of Visual Basic and data access. Many of the following chapters concentrate on specific techniques, and Chapter 4 attempts to steer the developer to which of those techniques, such as DAO, RDO, or ADO, is appropriate for his or her situation.

Many fine books are devoted to small subsets of the subjects discussed in this chapter. My recommendation if you do seek out these texts is to find ones written in the past few years to ensure that the most modern approaches are reviewed.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Previous | Table of Contents | Next

Chapter 2
Relational Database Management Systems

Key Topics:

- Relational databases vs. RDBMSs
- How a relational database is organized
- Database integrity constraints
- Database design
- Data Definition Language
- Data Control Language

In this chapter, I guide you through the essentials of what is the heart and soul of any client/server system: the database. Although a Visual Basic program can run against almost any back-end database, it is the Relational Database Management System (RDBMS) that brings the “server” to client/server. When training students or corporate clients in client/server development techniques, I stress that properly designing a database is 80 percent of the battle in constructing and deploying a successful client/server application. With a well-designed database, you have a good chance of deploying a successful client/server application. On the other hand, if the database is poorly designed, all the development skills in the world cannot create a robust, reliable client/server application. Therefore, no skill is more important in client/server development than properly designing a database.

A properly designed database accomplishes three essential things for the client/server developer:

- Through a properly normalized data schema, complete with intelligently designed integrity constraints, the database can assume much of the burden of validating basic business rules, such as requiring that an order have a valid customer.
- By using the referential integrity tools made available by the database, you minimize or eliminate altogether the chances of data corruption problems.
- By moving data update, retrieval, and validation processing to the server, you enhance application performance by minimizing network traffic and by moving some of the intensive processing from the client to the server (which, presumably, is a high-end machine optimized for the load).

A Note About Object-Oriented Databases

New to the database scene in the past couple of years is the emergence of object-oriented databases (OODs). An OOD does not store its data referentially; instead, it employs object-oriented techniques to store and access data as discrete objects. The technology is, as yet, immature and sales are a tiny percentage of the overall database market. None of the major database vendors offers a true object-oriented database, and it will be years, if ever, before the OOD encroaches upon the more tried and true RDBMS.

Some vendors, most notably Oracle and Informix, have introduced object-oriented extensions to their relational products. These are referred to as *hybrid* databases. With a hybrid database, you can define certain columns as containing objects and employ extensions to the database that "understand" how to manipulate those objects. For instance, Informix and third parties market Data Blades, which make it possible to store and manipulate specific types of objects in a relational table. One vendor sells a Data Blade that enables a table to store facial images and then search by those images.

Unfortunately, hybrids tend to be slow in indexing and accessing the objects that they store.

Recently, Sybase and IBM have both announced object-oriented extensions to their database products.

In the following pages, I overview some of the popular relational databases, discuss the theory of relational database design, and conclude with an overview of SQL statements, which implement intelligent database design.

Relational Databases Vs. Relational Database Management Systems

Before embarking on a survey of available relational database products, it is important to draw a distinction between relational databases and relational database management systems. The latter implies an engine that is running independent of any client program.

Any database that organizes data in related tables and that can enforce those relationships via referential integrity can lay claim to being a relational database. The database *engine* manages the data and enforces the integrity constraints. If there is no engine, the responsibility falls on the application program to perform those roles.

Consider a Microsoft Access database that I call “demo”. It is unquestionably a relational database. However, assume I put the database file on a server and then access it from multiple client programs. You will see that there are actually two files: demo.mdb is the actual database file (MDB stands for Microsoft database file), and demo.ldb is a file used to manage record locking (the LDB is for lock database file).

Now, suppose you want to see all the customers from California. Your application generates an SQL statement that looks something like:

```
SELECT * FROM CUSTOMER WHERE STATE = 'CA'
```

Assume there are 50 customers from California on a table of 5,000 customers. All 5,000 records are returned to your application, and the application itself then reads through those records to find the 50 that you need. The application receives 100 times as much data as required. It is easy to imagine how this activity bogs down network performance, especially if 10, 50, or 500 client programs all make the same types of requests.

This inefficient data processing occurs because the Access database is just a file with no engine to process the data request.

Contrast this example with one involving an RDBMS, such as Oracle or Microsoft SQL Server, which are true database engines. In this scenario, the SQL request is sent to the RDBMS, which processes the request on the server. The server sends back to your application only the 50 records that are required. Network traffic is reduced by 99 percent.

The moral? A wolf in sheep’s clothing is still a wolf. A database might be relational (such as Microsoft Access) or present an SQL front end (such as dBase), but if it is not a relational database engine, it is not an appropriate choice for client/server development.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Considerations In Selecting An RDBMS

As shown with the simple Microsoft Access illustration, your choice of RDBMS can have a dramatic impact on both the performance and the maintainability of your application. Choosing the RDBMS is then a critical decision point in your overall client/server strategy.

It is easy to jump to the conclusion that—assuming money, hardware, and so on are not an issue—you should simply purchase the most powerful engine on the market. Unfortunately, things are seldom that clear-cut. The marketing hype that passes for serious advertising from the various database vendors doesn't lend much light to the discussion either. Open any database magazine and you can see ads from Sybase and Oracle both proclaiming how each clobbers the other in side-by-side comparisons. Visit either vendor's Web site, and the same guidance is presented in more detail. In truth, the claims of all the vendors can probably be substantiated; it is the assumptions each makes about operating environments that renders one more powerful than the other in one test versus another. In any case, the most powerful engine may also be the most difficult to administer or it may simply be overkill for your organization.

In the following pages, I provide some general considerations in selecting an RDBMS for your client/server environment. These considerations include:

- Network operating system
- Typical database usage
- Number of users and volume of data
- Database administration
- Database cost
- Vendor stability and reputation

Of necessity, it is not possible to reach any definitive conclusions because the

proper choice is highly dependent on each organization's needs and environment. Further, although I have, in fact, used most of the products that I discuss in the next section ("A Survey Of Available RDBMSs"), I have not used all of them, and any specific recommendations are likely to be somewhat subjective.

Of course, your organization may have already deployed an RDBMS that is serving you quite well. In that case, get a cup of coffee and then move on to the next section.

Network Operating System

Perhaps the first consideration in choosing a database is the network operating system (NOS) upon which it is to run. Few vendors support all server operating systems, so if you are constrained to choosing from vendors whose product runs on, say, OS/2, the field of viable candidates is automatically reduced.

It is possible, of course, to run a different operating system for your database server than you do for other portions of your network (and many organizations do just that). However, my recommendation is to always keep the environment as simple as possible. That recommendation includes avoiding running multiple operating systems across the network if possible.

Along the same lines, you may have other environments with which you need to integrate. If you need to run your client/server applications with mainframe data, you might want to consider products that make this easier. IBM's DB2 running on OS/2 is a popular choice for shops with this need. XDB Systems also markets a capable database product that can run against DB2, IMS, and VSAM data on the mainframe.

An additional consideration with your NOS arises with the subject of multiple processors. Increasingly, high-end servers come configured with two, four, and even eight processors. However, if the NOS is incapable of using the processors, the database cannot use the processors. For instance, Novel NetWare 3.1x runs quite comfortably on an Intel-based machine running dual processors; it simply ignores the second CPU. NetWare 4.x does take advantage of the additional processors. Assuming your NOS supports multiple processors, your database may not (or you might need to tell the database to use those processors).

Typical Database Usage

In general, database applications tend to fall into two categories: transaction processing and decision support.

An online transaction processing (OLTP) system, such as an order-entry application or a banking system, is typically transaction oriented. Often, larger systems may need to process many transactions per second (TPS).

On the other hand, a decision support system (DSS) is typically a large database upon which sits all or a large subset of corporate data, which people analyze to make informed management decisions. The system may have many

thousands or millions of detail records that are queried and summarized but are not typically updated. Executive information systems (EIS) and data warehouses are examples of these types of applications. Either way, the needs are quite a bit different than those for an OLTP system.

If you have a busy OLTP system, you might want to consider such factors as how a given database performs record locking. Sybase, for instance, uses what is called page-level locking—all records on a database page are locked when an update is to occur. Oracle, on the other hand, performs row-level locking—only those rows that are to be updated get locked. The record locking has implications if several users are trying to update different records on the same page. With the Sybase product, the users have to wait until the first user has released the lock on that page. Microsoft SQL Server 6.0 performs page-level locking, whereas SQL Server 6.5 does pseudo row-level locking: The rows immediately before and after the row to be updated are also locked. SQL Server 7.0 performs true row-level locking.

On the other hand, the nature of your data may be such that a page contains few rows, which makes the issue of row-level versus page-level locking less important.

If you are planning a DSS, you should consider how well the database handles very large volumes of data. Most of the major vendors support some sort of variation on massively parallel processing (MPP), where a long-running query is executed in multiple threads to better handle the intensive querying and to speed up processing.

Determining which database will best handle your needs for an OLTP or a DSS is not an easy undertaking. My recommendation is to talk to peers who have used the products in environments similar to your own. In addition, check out the computer press for its independent benchmarking results. Lastly, consider searching the Internet (but avoid the vendor’s own biased claims).

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Number Of Users And Volume Of Data

Considering both the number of users and the volume of data actually goes hand-in-hand with the prior discussion of database usage patterns. If you have two users who need access to only a few thousand records, by all means, consider a product such as Microsoft Access. If, on the other hand, you have many thousands of simultaneous users chasing terabytes of data, you need to examine the beefier databases such as Oracle, Sybase, Informix, Microsoft SQL Server, and DB2.

If your needs are greater than what Access can satisfy but are perhaps less than what a database such as Oracle warrants, you might want to consider Sybase SQL Anywhere. My experiences with this product are that it can satisfy the needs of up to 50 users in a moderately heavy OLTP environment. For low volumes of data, you may be able to push the number of users to 100. For moderately sized databases, it is a viable platform for a DSS but does not offer the tools that an Oracle or Sybase SQL Server offers for massively parallel processing.

Database Administration

My own subjective experience is that Oracle's database engine is probably the most powerful in a high-transaction environment. However, this power comes at the expense of more complicated administration. To squeeze the best performance out of a complex database product such as Oracle, the database administrator (DBA) needs to monitor and refine a variety of operating parameters, such as defining the frequency of checkpoints, partitioning the data, and so on. (The particulars of database administration vary from database to database and are beyond the scope of this book.) How much administration is required also depends on the nature of the applications that the database supports in your own organization.

Although the good news is that most of the vendors (including Oracle) are simplifying administration, a poorly configured database yields poor performance. Consider Microsoft SQL Server running on a Windows NT server with 1GB of RAM. By default, SQL Server uses only the first 16MB of RAM. A low-end RDBMS product that uses most or all available memory performs much better than a high-end RDBMS that has not been configured to take advantage of the available resources.

About Database Administration

In talking to two seemingly identical organizations about their experiences with the "Brand X" RDBMS, you are likely to encounter two wildly different reports on how well the database performs. As is true in so much else in life, the database is only as good as its weakest link, and often, that weak link may well be the DBA. Under most processing loads, a properly configured environment should yield good performance with any of the major RDBMS engines.

The environment includes:

- *The server*—The computer on which the database is running should be a high-end machine loaded with plenty of memory and disk space and possibly multiple processors. Don't skimp on this critical piece of the RDBMS performance puzzle. Similarly, use common sense when configuring the database server. The RDBMS should normally be the only software running on the server other than the operating system. Use common sense when configuring the computer. For instance, spreading the database over two 10GB disk drives yields better performance than over one 20GB hard drive (because you have two heads reading and writing data, which requires less head movement).
- *The network*—Data can only be sent to and received from the RDBMS as fast as the network is capable of moving it. If you are running a 10MB Ethernet network, consider moving to a switched 100MB Ethernet architecture. Your capacity will increase at least tenfold. If the network is bogged down by too many users, performance can degrade rapidly as a result of escalating packet collisions. (Refer to my discussion of network architectures in Chapter 1.) Consider splitting the network into smaller LANs connected via bridges. Place the database server on that LAN where it will receive the most use (thus minimizing traffic that has to traverse the bridges).
- *The service model*—This can be a critical source of performance degradation, creating the illusion that the fault is with the RDBMS. I discuss the service model in Chapter 1, but for purposes of this discussion, placing the proper type of processing at the proper place on the network can mean the difference between good performance and poor performance.
- *The database design*—A poorly designed database can not only cause potential data integrity problems, but it can also yield horrific performance problems. The overuse and underuse of indexes both cause undesirable results. An over-normalized design might force the database to join together too many tables. An under-normalized design might

require the database to work too hard to locate the desired data or force it to read through too many records to return the desired result set.

- *The application data logic*—The application (or user) can easily throttle performance with poorly phrased queries that do not, for instance, correctly specify table joins. Consider a department table that has 50 rows and an employee table that has 500 rows. A list of employees showing their departments should, of course, produce 500 records. If the tables are not properly joined, you might end up with 25,000 records instead! Now, consider a program that allows the selection of an employee for maintenance. Inexperienced developers invariably seek to present a drop-down list box containing all 500 employees. That means that each time the user wants to see one employee, the database must return 500—hardly the purpose of client/server. Worse, what if the company has 5,000 employees? You can see where performance would, again, rapidly degrade through no fault of the RDBMS.
- *The RDBMS*—The RDBMS itself can easily be so badly maintained as to produce unbearable results when even minimal performance tuning might yield improvements on the order of a 90 percent reduction in response time. Study the tuning manual that comes with each database. Periodically reorganize the database—a task that is similar to defragmenting your hard drive. Ensure that the database is allowed to use all of the resources (memory, disk, and so on) available. Update the system statistics frequently (system tables maintained by the database that give the query optimizer information about the types and volumes of data that is stored on the tables).

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Database Cost

Although I personally hate to see the cost of an RDBMS become a major consideration, the reality is that some shops do have to live within certain monetary constraints. A database engine such as Oracle might run into a licensing fee of many thousands of dollars, whereas a product such as SQL Anywhere might cost only several hundred dollars. It should be noted than many vendors will wheel and deal on price negotiations.

Be careful when comparing prices. A visit to Oracle's Web site shows you that the licensing fee to run the RDBMS on a single server is $1,450 (at the time of this writing). The fine print near the bottom of the page reveals that this is for five concurrent users. Most organizations running Oracle have somewhat more than five users.

When factoring costs, you need to consider the number of servers upon which the database is to run, the maximum number of concurrent users, the maintenance fee, and quite possibly a separate telephone support fee. The maintenance fee is often 12 to 18 percent of the licensing fee annually. Telephone support can easily run into thousands of dollars per year.

A final consideration in database cost is the so-called cost-of-ownership term so often bandied about when discussing PCs. Above and beyond licensing, maintenance, and support costs, you might want to consider that the hardware requirements of the different RDBMS products differ (ignore the stated minimum requirements and concentrate on the recommended configuration) as do the internal support costs. A talented DBA can tremendously enhance the performance of the RDBMS (while commanding a salary that would make the corporate CEO blush).

Vendor Stability And Reputation

You might want to consider the financial stability of the RDBMS vendor. As of this writing, Sybase was posting large losses and laying off 10 percent of the company's employees. There were also questions regarding Informix's financial results. Such news may be of concern in terms of a company's long-term viability (will they be in business five years from now?), the organization's ability to develop and improve the product, and its ability to support the product. (Please note that although Sybase's woes cannot be construed as good news for any customers, I personally do not predict an organization that large will go under. The Sybase product set is excellent and the company continues to boast a top-notch technical development staff across a diverse product line.)

A Survey Of Available RDBMSs

In the following section, I briefly mention some of the more popular RDBMSs available, including:

- Microsoft Access
- Sybase SQL Anywhere
- Sybase Adaptive Server Enterprise (System 11)
- Oracle
- Informix
- Microsoft SQL Server
- IBM DB2
- XDB

Microsoft Access

As discussed earlier in this chapter, Microsoft Access is not truly an RDBMS. However, it is a popular development back end and offers some fairly sophisticated features, such as true database replication. On the other hand, it does not offer a complete SQL implementation (see Appendix B for a comparison of Jet SQL to ANSI SQL), and because it is not truly client/server, its performance pales against even the least expensive of the true client/server database engines. Nevertheless, because it is popular and because Visual Basic offers easy support for Access, I use it in examples throughout the book.

Access Version 8 is bundled with Microsoft Office 98 Professional and is available as a standalone product. However, you can use Visual Basic to write to and read from Access databases even if you don't own Access itself.

For more information on Microsoft Access, visit the Microsoft Web site at www.microsoft.com/access/.

Sybase SQL Anywhere

Sybase SQL Anywhere was formerly known as Watcom SQL. (The product was renamed when Sybase acquired Powersoft Corporation, which owned Watcom.) Because it is both inexpensive and widely bundled with various development tools, it is a popular RDBMS. The product is sophisticated in its

implementation, has a complete implementation of SQL (I discuss the ANSI SQL standard in Chapter 3), and offers full support for Transact-SQL. (T-SQL is a dialect of SQL used by Sybase SQL Server and Microsoft's SQL Server 6.5.) The product requires almost no administration but does not scale to the enterprise level.

The current version of SQL Anywhere (as of this writing) is 5.5. It runs as either a server or a standalone (single-user) RDBMS on Windows 95, Windows NT, Windows 3.x, OS/2, MS-DOS, and NetWare.

For more information, visit Sybase's Web site at www.sybase.com.

Sybase Adaptive Server Enterprise

Sybase Adaptive Server Enterprise (System 11) runs on Windows NT and on various Unix platforms. Adaptive Server is an enterprise-level server offering robust performance under a variety of configurations.

Despite the popular misconception that the relational database was pioneered by IBM, it was actually Sybase that was first to market in the late 1970s. For many years, Sybase was number two in sales (behind Oracle), but a reputation for customer service woes and a couple of years of poor financial results led to the company falling behind Informix in database sales volume. Still, Sybase's database server products are powerful, and the company enjoys a large installed base of users.

Like its little brother, SQL Anywhere, Sybase Adaptive Server uses a dialect of SQL known as T-SQL. Throughout various sections of the book, I present examples using T-SQL.

Sybase has joined the move toward hybrid databases with the addition of Data Stores to store and manipulate objects.

For more information, visit the Sybase Web site at www.sybase.com.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Oracle

Oracle Version 8 runs on a wide variety of platforms, including OS/2, NetWare, Windows NT, and Unix (including Sun Solaris, IBM's AIX, HP-UX, and SCO). A personal edition of Oracle runs on Windows 95/98.

Oracle has long enjoyed a reputation as the most powerful database on the market with advanced features optimizing large volume data processing, distributed data management, Web connectivity, and robust OLTP. Still, the gap in processing capabilities between Oracle and its competitors has eroded and perhaps disappeared under certain circumstances. Although Oracle was once considered a pricey product, competition has driven the costs down. Oracle has been the number one vendor for many years.

Whereas the two Sybase products offer extensions to SQL known as T-SQL, Oracle uses a robust dialect known as PL/SQL. As with T-SQL, I also present samples in this book using PL/SQL.

For more information, visit the Oracle Web site at www.oracle.com.

Informix

For years, Informix was a distant runner-up in terms of sales to the two giants, Sybase and Oracle. In recent years, the Informix product line has grown in sales until it is now solidly entrenched as the number two vendor of RDBMSs.

The Informix products scale well from the department to the enterprise level and offer well-integrated Web connectivity options.

Informix sells versions of its enterprise server that run on most types of Unix and on Windows NT.

For more information, visit the Informix site at www.infor-mix.com.

Microsoft SQL Server

Microsoft SQL Server was originally jointly developed with Sybase SQL Server (now Sybase Adaptive Server Enterprise). SQL Server 6.0 was the first large-scale departure from its Sybase roots. Version 6.5 saw major performance boosts and Version 7 looks even better.

Unlike the other database offerings, MS SQL Server runs on Windows NT only. Although that obviously might be limiting, it also offers some advantages. Historically, the RDBMS engine has had to be written with a separate layer to interact with the operating system, perform its own thread management, and so on. Because MS SQL Server is written to run on only one operating system, it can be more closely integrated with the operating system.

SQL Server has been rapidly gaining in popularity, although it probably has yet to achieve parity in performance with the most powerful of the products listed here. That performance issue might not be a consideration for your organization if you are not attempting to connect several thousand users or manage several terabytes of data.

Like the Sybase products, SQL Server uses Transact-SQL, although Microsoft has begun moving in its own direction and away from the Sybase standard. SQL Server probably offers the easiest-to-use administrative tools.

For more information, visit the Microsoft Web site at www.microsoft.com/sql/.

IBM DB2

IBM's relational database, DB2, runs on a wide variety of platforms, including most types of Unix, Windows NT, OS/2, and Windows 95/98, as well as VAX VMS and IBM MVS and VSE. Although in the past its network-level database products have not fared well in comparison to the products of other vendors, its latest version has tested well on Windows NT.

DB2 is of particular interest to those organizations that need a high degree of compatibility or interoperability with their mainframe database.

For more information, visit the IBM Web site at www.ibm.com.

XDB

XDB Systems offers a well-done RDBMS that offers a high degree of compatibility with IBM's DB2 and, like DB2 on the network, would be of interest to shops also running a mainframe. By itself, XDB is a capable RDBMS. In addition, it has extensions that enable the querying of DB2 and IMS databases and VSAM files on the mainframe (though VSAM files cannot be updated). My own experiences with XDB have been positive, although I feel it is probably not as powerful as the very top-end RDBMSs.

I am no longer nervous about the company's long-term prospects because it has announced its acquisition by Micro Focus, makers of COBOL tools for the network. This is a logical pairing because Micro Focus tools have always

integrated well with XDB. The choice of XDB is appropriate for shops interested in leveraging legacy COBOL applications by migrating them to a client/server environment.

For more information, visit XDB's site at www.xdb.com.

Other Relational Databases

Space precludes the listing of all RDBMS vendors, even though I have attempted to cover most of the major vendors. Any listing or failure to list a given product should not, of course, be construed as a positive or negative comment about a product.

Besides RDBMSs, other relational databases such as Lotus Approach might be logical alternatives to Microsoft Access. Additionally, although not organized relationally, front ends or ODBC drivers (I discuss ODBC in Chapter 3) allow the access of other file formats in a relational manner. These formats and products include Btrieve, dBase, FoxPro, Paradox, and so on. Any of these may be suitable for smaller-scale development efforts.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Database Organization

A relational database, unlike sequential files, organizes all of its data in a single file. Figure 2.1 illustrates a nonrelational and a relational design of a database with customers and orders. Both cases have four indexes: two on the customer data and two on the orders data. The nonrelational approach illustrates how you might build the database with dBase files: There are a total of six files with no true logical association between them. For instance, nothing stops me from opening the ORDERS.DBF file and adding an order even if no valid customer is associated with that order. Even worse, nothing stops me from exiting to DOS and typing “DEL CUSTOMER.DBF”. After doing so, I have two indexes serving no purpose and a whole lot of orders for which I have no clue about the customers.

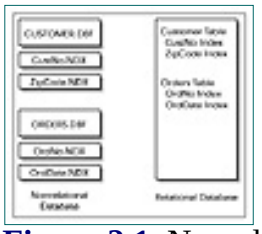

Figure 2.1 Nonrelational databases are typically a collection of un-associated files. Relational databases contain all of their tables and indexes inside a single file.

On the right side of Figure 2.1 is a single file managed by an RDBMS, such as Microsoft SQL Server, containing the two files and four indexes. These files are actually stored as tables. Tables and indexes are both stored as objects within the database.

Tables

A database table is organized into rows and columns much like a spreadsheet.

The columns correspond to fields in a sequential file. The rows correspond to records. Figure 2.2 illustrates this with a simple implementation of a **Customer** table. **Cust_No** represents a column, whereas **John Smith** represents a row in a two-dimensional grid.

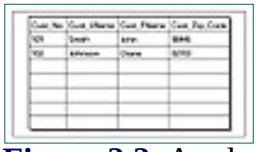

Figure 2.2 A relational database table is organized into columns and rows.

Every row in a table must be able to be uniquely identified. In other words, there must exist something on each row in a table that is different from any other row on that table. This aspect that makes a row unique is called a *primary key*. On the **Customer** table, the **Cust_No** column is the primary key; there can be one and only one customer 101, one and only one customer 102, and so on.

A Note About Primary Keys

Technically, the RDBMS does not require you to ensure that each table has a primary key. However, not having one defeats the whole purpose of relational database design for reasons that will become apparent as we continue our discussion.

A primary key is actually implemented as what is known as a *unique index* by the RDBMS. A table may actually have multiple unique indexes. (In addition to an employee number, you may also have a Social Security number, which is also unique.) However, there can be only one primary key on a table.

The primary key may actually be more than one column on the table. You can, for instance, have an inventory table where the combination of the part number and date purchased columns together form the primary key. Such a key is known as a *compound key*.

It is customary but not required that the primary key columns be the first columns on the table.

With a single table, the relational model offers no particular advantage over a sequential file. However, you can expand upon the example to include another table: **Orders**. Again, the table is organized into rows and columns as shown in Figure 2.3. Notice the column containing the customer number. You can begin to see where the “relational” in relational databases comes in. The power of the database is achieved when rows and columns on tables are logically constrained by values stored on other tables; in this case, the **Cust_No** value stored on the **Orders** table must exist within the **Customer** table.

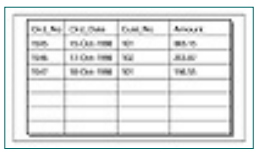

Figure 2.3 The second table, **Orders**, is related to the first table, **Customer**,

via the common column **Cust_No**.

Table Relationships

The RDBMS enforces the rule that the customer number on the order must already exist on the **Customer** table through the use of *referential integrity*. Specifically, you can create a *foreign key* on the **Orders** table that states that the **Cust_No** column must relate to a key on the **Customer** table. In general, the foreign key specifies that the values of a column (or columns) on a *dependent* table must exist uniquely on a *master* table. The **Orders** table is dependent on the **Customer** table because values on it must already exist in the master table. Sometimes this relationship is called *child-parent*.

Figure 2.4 shows the relationship between the **Customer** table and the **Orders** table with the foreign key.

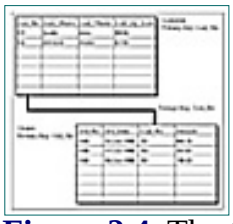

Figure 2.4 The **Customer** and **Orders** tables with primary keys and a foreign key enforcing their relationship.

TIP
Foreign Keys And Primary Keys
The foreign key on a child table is dependent on a key on the parent table. In other words, the value stored in a foreign key column on a child table—such as the **Cust_No** column on **Orders**—must exist on the parent table, in this case, the **Cust_No** column on the **Customer** table.

Some databases require that the foreign key relate to the primary key of another table. Others only require that the related key on the parent table be a unique index. Remember that a primary key is itself a unique index.

If the related key on the parent table is, itself, a compound key, then the foreign key must also be a compound key. The foreign key must be the same number of columns as the related key on the parent table and each of those columns must be the same data type.

Previous | Table of Contents | Next

Products | Contact Us | About Us | Privacy | Ad Info | Home

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Database Integrity

When designing a database, it is important to remember that the foreign key is always defined as a constraint on the child or dependent table and that table is always part of a one-to-many relationship. I cover the details of this concept under “Database Design” later in this chapter. However, for now, you should understand that related tables are always related in a one-to-many manner. For example, an examination of Figure 2.4 reveals that for any one customer, there might be many orders. However, for any one order, there can only be one customer. Wherever a one-to-many relationship exists, the table that is the “many” end of the relationship is always the child table and there should always be a foreign key defined to enforce that relationship. I review how to create foreign keys on the database in “Data Definition Language” later in this chapter.

I described a foreign key as *constraining* the values that a certain column can contain on a table. This is but one type of constraint that the database can enforce. A foreign key enforces referential integrity. Although databases differ in their capabilities, in general, integrity constraints can be placed in three categories:

- *Referential integrity*—Enforces relationships between two tables.
- *Domain integrity*—Enforces values that can occur in any column across an entire table.
- *User integrity*—Enforces values that can occur in a column based upon values on the *same* row.

Domain integrity and user integrity can be enforced using a variety of constraint types:

- The primary key, as mentioned earlier, enforces the requirement that a value uniquely identify a row on a table. This is an example of domain integrity.

- A unique key requires that any value in a given row be unique within that column. For instance, a Social Security number might be defined as unique. A primary key is implemented as a unique key.
- A check constraint validates the contents of a column on a row at either the domain or the user level. A row may be defined as **Not Null**, meaning that the column is not allowed to be null. (*Null* is the absence of any value at all and is not the same as an empty string. I discuss null more thoroughly in Chapter 3.) You can also employ a check constraint, which validates that a column's values are absolutely restricted. For instance, you can create a check constraint that requires salary to be greater than zero. Because this rule applies to any row on the table, it is also a domain integrity constraint. On the other hand, you might define a check constraint, such as date of hire must be greater than date of birth (to avoid problems with child labor laws, of course), that is dependent on another column on the same row. This is an example of a user integrity constraint.

I show the syntax for creating each type of constraint later in this chapter.

Data Types

Relational databases define each column to be a certain data type. Although a number is really just a number regardless of what database is storing it, the RDBMS vendors tend to differ somewhat in what they call those types. For instance, what Oracle calls *number* Sybase and Microsoft call *numeric*.

Data types tend to fall into four categories:

- Character data
- Numeric data
- Date data
- Binary data

Character Data

Character data types can hold any alphanumeric values. These values are usually referred to as *strings* or *string literals*. Even if the string happens to contain a number, you cannot perform arithmetic operations on it.

The two main types of character data types are **CHAR** and **VAR-CHAR**.

A **CHAR** data type is a fixed-length field defined as follows:

```
Cust_LName   CHAR (21)
```

If **Cust_LName** holds the value **'Jones'**, it right-pads the field with 16 spaces to fill up the field. Fortunately, most RDBMSs do not require you to account for this when searching for values. Typically, the database automatically trims the longer of two search conditions:

```
SELECT * FROM Customer
WHERE Cust_LName = 'Jones'
```

In this example, the database automatically takes into account that the search term **'Jones'** is shorter than **Cust_LName**.

The **VARCHAR** data type (called **VARCHAR2** by Oracle), as its name implies, is a variable-length data type and looks like the following:

```
Cust_LName VARCHAR (21)
```

TIP
CHAR Vs. VARCHAR
There is some debate about which is the better data type to use. Certainly, the **CHAR** data type can waste a lot of space in a database. This was an important decision when disk space cost many dollars per megabyte. However, disk space is cheap now and the database has to work harder to find columns where variable-length data is used. It is difficult to say what the performance is exactly, but I would speculate that the average database takes a 10 percent performance hit if it employs all variable-length character types.

My recommendation is to use fixed-length **CHAR** data types for all but the biggest columns (such as a column to record comments). If you do end up using some **VARCHAR**s, put those columns at the end of each row where the performance penalty is minimized.

In this case, the column holds up to 21 characters, but if the name **'Jones'** is stored, it is only 5 characters long.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Numeric Data

Unlike character data, numeric data can take many forms and vary more from RDBMS to RDBMS than other data types. In general, you will run into the following data types (refer to your vendor's documentation for any specifics or differences in implementation):

- **NUMERIC (*S*, *P*)**—***S*** stands for *scope* and ***P*** stands for *precision*. Scope refers to how many digits the number is allowed to hold, and precision refers to how many of those numbers are to the right of the decimal place, as in the following example:

```
Ord_Amount   NUMERIC (11,2)
```

  In this example, the largest number that can be stored is 999,999,999.99. Even if your number has no digits to the right of the decimal place, you can use only nine to the left of the decimal point. Thus, a column defined as **NUMERIC (4,4)** can be no larger than 0.9999.

- **INTEGER** is a data type that holds a whole number from –32,768 to +32,767. An **UNSIGNED INTEGER** is capable of holding positive whole numbers from 0 to 65,536.

- **LONG** numbers can be any whole number of approximately negative 2.1 billion to positive 2.1 billion. As with the **INTEGER** data type, your database may also support an unsigned variation, **UNSIGNED LONG**, which holds a positive whole number of 0 to approximately 4.2 billion.

- **FLOATING POINT**, **SINGLE**, **SINGLE PRECISION**, **DOUBLE**, and **DOUBLE PRECISION** are all variations on floating-point numbers. A floating-point number can handle either real numbers or irrational numbers (as well as whole numbers, of course). A real number is any number that can be represented in a fixed number of decimal

places, such as 1.25. An irrational number cannot be represented in a fixed number of decimal points. An example is 1/3, which is .333333… (to infinity). The database stores these numbers in scientific format. Single precision numbers (**SINGLE** or **SINGLE PRECISION**, depending on your database) are usually 4 bytes long. Double precision numbers are usually 8 bytes long, effectively making them able to handle a much larger range of numbers.

Number Storage On The Database

Although you and I represent numbers decimally—that is, using what is called base 10—computers store numbers in binary format using base 2. If a number is stored in 2 bytes, there are a total of 16 bits with possible values of 0 or 1. Therefore, the largest possible number of values that can be represented without some sort of encoding algorithm is 2^{16} or 65,536. With signed numbers, one bit is reserved for the *mantissa* to indicate a negative or positive number, reducing the potential number of values to 2^{15}. Thus, a signed integer field stored in 2 bytes has a range of –32,768 to + 32,767. Even in an 8-byte field, there can only be 2^{64} possible values.

Computers use what is the known as the IEEE format to store floating-point numbers in scientific format to get around this limitation. Although this does not allow the storage of an infinite range of numbers, it does allow the computer to approximate very large and very small numbers with a high degree of accuracy.

Scientific formats for numbers consist of three parts: the root, the letter E, and the exponent: 144.34267E+4. In this example, the exponent +4 indicates that the decimal point should be moved four places to the right: 1443426.7. An exponent of –4 would indicate that the decimal point should be moved four places to the left, yielding the number .014434267. The **DOUBLE** data type generally supports an exponent of plus or minus 308, which accommodates the storage of a very large or very small number in a compact manner.

Date Data

Dates are encoded on the database as numbers and are, therefore, limited in their range. Oracle's **DATE** data type, for instance, has a range of 1 January, 4714 BC to 1 January, 4712 AD. The major RDBMSs are generally Year 2000 compliant. However, you should consider that your front-end development language might not support the same range of dates. Visual Basic, for instance, has a range of 1 January, 100 to 31 December, 9999.

Inputting dates into the database can sometimes be a chore. The format of the date that the database expects is usually a startup option (that is, it is specified as an optional parameter when starting the database engine on the server). Usually, you can override this setting in an individual session by using the **SET** command. Oracle's default date format is:

```
INSERT INTO…
```

```
VALUES ('18-Dec-98')
```

Sybase, on the other hand, allows dates to be entered as either '1998-12-18' (December 18, 1998) or '1998/12/18'. To alter the default format, you can specify **SET OPTION Date_Order 'DMY'**, which then causes dates to be interpreted in day, month, and year order.

Should You Store Objects In The Database?

Assume you are building a database for a legal firm handling many legal documents, such as contracts and wills. You might want to store those documents on the database where it is easy to associate them with specific clients. The down side, of course, is that it makes each row on the database very large, which could impact performance.

An alternative is to simply store a string with the path and name of the document, such as "\DOCUMENTS\CONTRACTS\J819980347.DOC". The down side of this approach, however, is that there is no longer any integrity constraint preventing that document from being moved from the \DOCUMENTS\CONTRACTS directory or even from being deleted.

The choice depends on your needs, and you need to carefully evaluate the pros and cons of either approach. However, if you do decide to store the object directly in the database, make it the last column on your table where it will have the least impact on performance.

Binary Data

Generally, the only time you will need to store binary data is when you want to embed objects such as bitmaps or word processing documents into the database. You accomplish such a task with the **BLOB** (Binary Large Object) data type. Although some RDBMSs are adding object-oriented extensions to automatically handle these objects (refer to "A Note About Object-Oriented Databases" earlier in this chapter), you generally need to plan in advance how you will display and manipulate this type of data.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Database Design

The key to a solid client/server application is a solid database design. Two approaches have a considerable amount of overlap. *Entity-relationship modeling* seeks to identify entities, provide a unique identifier for each, determine relationships between those entities, provide a method for enforcing those relationships, and then assign attributes to those entities. This approach provides a logical design of the database where entities map to tables and attributes map to columns. The other approach is *database normalization*. This approach seeks to take an existing design and validate it against a number of design guideline "rules" to eliminate processing anomalies. You accomplish this task by reassigning columns to existing or new tables.

Although entire books are written on the theory and practice of database design, I provide an overview of the essentials in the next few pages.

Entity-Relationship Modeling

With entity-relationship modeling, you seek to identify those entities that represent a business and eventually map them into a physical database design. Figure 2.5 shows a graphical representation of the process.

Figure 2.5 Summary of the entity-relationship modeling process.

Let's assume that we are modeling a simple college. The first thing to do is

identify the business entities. An *entity* is something physical with a unique identifier. If we were modeling a mail-order company, typical entities would be Customers, Orders, Items, and so on. For our example, we have Students and Courses. Because an entity must be uniquely identifiable, we assign a primary key to each one and draw them on a piece of paper, as shown in Figure 2.6.

Figure 2.6 Step 1 of the modeling process.

Next, for each pair of entities, we determine whether there is a relationship between the two. In this case, Students and Courses are indeed related. Therefore, we draw a line to represent that relationship, as shown in Figure 2.7. In our mail-order company, we would have determined that Customers are related to Orders (because customers place orders) and that Orders are related to Items (because items appear on orders) but that Customers are not related to Items. It might be helpful to place some wording on the drawing to describe how the two entities are related.

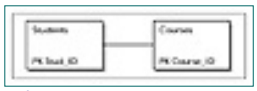

Figure 2.7 Step 2 of the modeling process.

The next step is to determine the nature of the relationship. To do this, you need to ask yourself two questions for each related pair of entities: For each occurrence of the first entity, how many occurrences might there be of the second entity? In addition, for each occurrence of the second entity, how many occurrences might there be of the first entity? Using our example, for each occurrence of Students, how many occurrences of Courses might there be? If there can be only one occurrence of Courses per occurrence of Students, draw a single arrowhead next to the Courses entity. If there can be more than one, draw two arrowheads. If there can be zero occurrences of Courses for each occurrence of Students, also draw a 0 on the line just before the arrowheads. In our example, a student can take zero, one, or more than one course. For any given occurrence of Courses, how many occurrences of Students might there be? For purposes of this exercise, we will assume that it can be one or many but not zero. The result is shown in Figure 2.8. When you have double arrows at each end, the relationship is called a *many-to-many* relationship. The 0 drawn at the Courses end of the relationship is called a *conditional* relationship.

Figure 2.8 Step 3 of the modeling process identifies the nature of the relationship between the two entities.

You are not permitted to have many-to-many relationships in a referential relationship. Therefore, you need to resolve this problem by creating a logical entity called an *assignment* entity or a *relationship* entity. Instead of relating the students and courses to each other, relate them each to the new entity, which we call Student_Courses. Because every entity must have a primary

key, we assign one to the Student_Courses entity. Whenever you create a relationship entity, its primary key is always a combination of the primary keys of the two other entities—in this case, Stud_ID and Course_ID. A primary key that consists of more than one item is called a compound key. The relationship entity always has a one-to-many relationship with both of the original entities, and the new entity is always at the many end of the relationship (it is, therefore, the child entity). The revised diagram is shown in Figure 2.9.

Figure 2.9 Step 4 of the modeling process resolves any many-to-many relationships by creating relationship entities.

Next, you need to assign foreign keys to enforce the relationships between the tables. Remember that a foreign key means that for each occurrence of a child entity, there must be a corresponding value in the parent entity. In entity-relationship modeling, the foreign key of the child table always corresponds to the primary key of the parent table. Also, remember that wherever a one-to-many relationship occurs, there is a foreign key relationship. Figure 2.10 shows two foreign keys in the Student_Courses entity. The Stud_ID that appears in the Student_Courses entity is a foreign key referencing the Students entity.

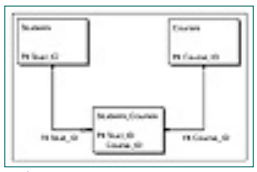

Figure 2.10 Step 5 of the modeling process assigns foreign keys to enforce referential integrity rules.

The last step is to assign attributes, which are things that describe the entities. The attributes always include the primary key and any foreign keys. Sometimes, there is room to assign attributes on the same drawing as the relationship diagram, but typically, it is done on a separate piece of paper using a matrix as shown in Figure 2.11.

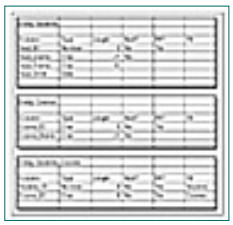

Figure 2.11 Step 6 of the modeling process assigns attributes to each of the entities.

With the output of the entity-relationship modeling process, you have the information that you need to create a physical database design from the logical one (as represented by the final model). Entities become tables, attributes become columns, and primary and foreign keys are identified. However, you

should first validate your database design by normalizing it. This process is described in the next section.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Database Normalization

There is a lot of overlap between the entity-relationship modeling process discussed in the last section and the database normalization process discussed in this section. It is quite conceivable that you can properly design a database using either approach alone. However, I don't recommend it because each approach has advantages that the other doesn't. The entity-relationship modeling process is particularly well suited for identifying database integrity constraints, although it offers little help to ensure that attributes are assigned to the proper entities. Database normalization seeks to do just this—verify attribute assignment by the application of standardized "rules."

We do this by comparing our design to a series of normal forms. We say that our database is fully normalized when it is in third normal form. Each normal form seeks to eliminate data redundancy and to insert, update, and delete anomalies. In terms of database operations, an anomaly is an action that either produces an undesirable effect or prevents us from performing a desired action in the database. I discuss these anomalies throughout this section.

For purposes of illustration, I use a simplified order-entry system. The order-entry system has the following pieces of information to be stored in the database:

```
ORDERS
Ord_No
Ord_Date
Cust_No
Cust_Name
Cust_Address
Cust_City
Cust_State
```

```
Cust_Postal_Code
Item_No
Item_Desc
Item_Qty
Item_Price
```

Further, I make the assumptions that a customer can place any number of orders, that any given order has only one customer, and that an order can be placed for any number of items. (In other words, a customer can call and purchase a number of different items all on the same order.) However, any one item can appear on the order only once.

To begin, we need to assign a primary key to our present design. We will call the primary key **Ord_No**.

First Normal Form

First normal form, sometimes abbreviated as FNF or 1NF, states that there can be no repeating groups of data. A repeating group of data is an item that can appear more than once in a row of data. For instance, because an order can be placed for multiple items, the columns **Item_No**, **Item_Desc**, and so on form a repeating group. The insert and update anomalies here are intuitively obvious: With our present design, there is no way for a customer to order more than one item.

To resolve a first normal form violation, you must move the repeating group to a new table. The primary key of the new table is a combination of the primary key of the first table (in this case, **Ord_No**) and a key that uniquely identifies the repeating group (in this case, **Item_No**). Our revised design looks like the following:

```
ORDERS           ORDER_ITEMS
Ord_No (PK)      Ord_No (PK)
Ord_Date         Item_No (PK)
Cust_No          Item_Desc
Cust_Name        Item_Qty
Cust_Address     Item_Price
Cust_City
Cust_State
Cust_Postal_Code
```

Unfortunately, we cannot stop here because we have some anomalies. First, we have an update anomaly in that we cannot add an item to the database unless there is an order for that item (because the item number and item description are not recorded until there is a row in the **ORDER_ITEMS** table). Second is an update anomaly: If an item's description changes, it must be updated on every single order for that item. Third is a delete anomaly: If an order is deleted, the item's description is deleted as well.

Therefore, we need to move on to second normal form.

Second Normal Form

Second normal form, usually abbreviated as SNF or 2NF, states that we can have no partial-key dependencies. A partial-key dependency is a column that is dependent on only part of the primary key. To determine whether we have any second normal form violations, we examine all tables with compound keys (in this case, the **ORDER_ITEMS** table). An examination of the table shows that **Item_Desc** is dependent on **Item_No** in the primary key but not on **Ord_No**. In other words, an item's description changes if the **Item_No** is different but not if the **Ord_No** is different.

To resolve this problem, we move the partial-key dependencies to another table whose primary key is the partial key from the original table (in this case, **Item_No**). Our new design looks like the following:

```
ORDERS           ORDER_ITEMS      ITEMS
Ord_No (PK)      Ord_No (PK)      Item_No (PK)
Ord_Date         Item_No (PK)     Item_Desc
Cust_No          Item_Qty
Cust_Name        Item_Price
Cust_Address
Cust_City
Cust_State
Cust_Postal_Code
```

This resolves our insert anomaly from the first normal form because we can now insert a new item into the database even if there are no orders for it. It also resolves our update anomaly because we now have only one row to update if an item's description changes. Finally, we have resolved our delete anomaly because deleting an order no longer deletes the associated items.

However, we still have anomalies. We have an insert anomaly in that we can't add a customer to the database unless that customer places an order. We have an update anomaly in that if we need to change the customer's name or city, we have to update every order that customer has placed. We have a delete anomaly in that if we delete an order, we also delete the customer.

To resolve this, we move on to third normal form.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Third Normal Form

Third normal form, usually abbreviated TNF or 3NF, states that there can be no non-key (or intra-data) dependencies. In other words, every column must be dependent on the primary key and not on another column that is not the primary key. In our example, customer name, address, city, and postal code are all dependent on the customer number and not the order number.

To resolve this, we have to move the non-key dependencies to a new table whose primary key is that non-key column on which the columns were dependent (in other words, the **Cust_No** column). We also leave that non-key column in the original table.

Our new design now looks like the following:

```
ORDERS          ORDER_ITEMS     ITEMS           CUSTOMERS
Ord_No (PK)     Ord_No (PK)     Item_No (PK)    Cust_No (PK)
Ord_Date        Item_No (PK)    Item_Desc       Cust_Name
Cust_No         Item_Qty                        Cust_Address
                Item_Price                      Cust_City
                                                Cust_State
                                                Cust_Postal_Code
```

We are now in third normal form.

Normally, you can stop at third normal form and conclude that your database is fully normalized. There are actually other normal forms: Boyce-Code normal form, fourth normal form, and fifth normal form. These normal forms are used for rare circumstances, such as resolving circular references, and nearly always indicate a totally inappropriate data model. In my 15 years of working with relational databases, I have never needed to consider anything beyond third normal form.

Sometimes, you need to step back and examine the design with a critical eye toward performance. If you see that you are going to end up having to join too many tables, you

might want to consider selectively *denormalizing* back to second or first normal form. My recommendation is to build the database in third normal form, load it with some sample data, and then run some benchmark tests against it to see if you have any performance bottlenecks. If you do, examine those bottlenecks and decide if they warrant denormalization.

Database Design Summary

Database design is part science, part art, and part experience. I provided you with two tools that, although either will work alone, I urge you to use in tandem to double-check your design. A lot of the design work is simply common sense, and you may well find that as you put together a database design, it is already in first, second, or third normal form. However, go through the steps of both entity-relationship modeling and database normalization anyway to confirm your design assumptions. Then, compare the design to the output of your system, the reports and screens. Make sure that you have identified the correct data elements and that you have not introduced a performance nightmare. Regarding the former, if you find that you have missed data elements that need to be captured and stored, add them into the design and repeat the entity-relationship modeling and the database normalization processes. If you find that you need to join four or five tables frequently, especially in the online environment, consider denormalizing your database.

Once the database design is in place, it is time to create the database, which is the subject of the next section.

Data Definition Language

Data Definition Language (DDL) is one of three parts of SQL. I discuss Data Control Language (DCL) later in this chapter and Data Manipulation Language (DML) in the next chapter.

DDL is the language we use to create and modify objects in the database. Objects are tables, indexes, keys, and so on.

Before proceeding, it is important to understand the concept of ownership. When you log in to the database, you have a user ID. If you create a table, your user ID is considered the owner of that table. Every object's name consists of two parts—the owner name followed by the table name:

```
Mike.EMPLOYEE
```

In this example, **Mike** is the user ID that created the table **EMPLOYEE**. If you issue any kind of SQL statement against an object such as a table and omit the owner name, the database assumes your ID is the owner. Thus, if you perform a **SELECT** against the **EMPLOYEE** table without prefixing it with the owner, and your ID is **Jane**, SQL translates your statement to:

```
SELECT...
FROM Jane.EMPLOYEE
```

If there is no **Jane.EMPLOYEE** or if you do not have permission to access the table, SQL returns an error message similar to "object not found."

DDL consists of three statements: **CREATE**, **ALTER**, and **DROP**. For purposes of this discussion, the next three sections deal with using these statements to work with tables. Following that, I discuss other database objects such as indexes and keys.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

CREATE TABLE

The **CREATE TABLE** statement has the syntax:

```
CREATE TABLE TABLE_NAME
(col_name DATATYPE [NOT NULL],
 col_name ...)
```

Following **CREATE TABLE**, specify the name of the table. Inside parentheses, list the columns separated by commas. After each column name, list the data type and, optionally, **NOT NULL** if the column is not allowed to have a null value.

The syntax to create the **EMPLOYEE** table that I use throughout the remainder of this book is as follows:

```
CREATE TABLE EMPLOYEE
(Emp_No              SMALLINT NOT NULL,
 Emp_LName           VARCHAR (21)   NOT NULL,
 Emp_FName           VARCHAR (15),
 Emp_SSN             CHAR (9),
 Emp_DOB             DATE,
 Emp_Hire_Date       DATE NOT NULL,
 Emp_Term_Date       DATE,
 Emp_Salary          NUMERIC (9,2),
 Emp_Dept_No         SMALLINT,
 Emp_Mgr_ID          SMALLINT,
 Emp_Gender          CHAR (1),
 Emp_Health_Ins      CHAR (1),
 Emp_Dental_Ins      CHAR (1),
 Emp_Comments        VARCHAR (255) )
```

TIP
About The Tables
Throughout this book, I use a series of tables, including the **EMPLOYEE** table here. Appendix A contains the syntax to create all of these tables along with primary keys, foreign keys, and so on. Because there are small differences between different databases, the CD-ROM contains these statements in RDBMS-specific directories. For instance, the Oracle **CREATE** statements are in the directory \SAMPDATA\ORACLE. If your database is not included, it is highly likely that one of the supplied syntax variations will work just fine. Otherwise, the modifications should be simple. Additionally, the CD-ROM includes the **INSERT** statements needed to create the data.

You can optionally specify various table constraints such as primary keys at the same time you create the table. The following syntax is for Sybase SQL Anywhere, but if you change the line **Emp_Salary NUMERIC (9,2)** to read **Emp_Salary NUMBER (9,2)**, it will work with Oracle as well:

```
CREATE TABLE EMPLOYEE
(Emp_No             SMALLINT NOT NULL,
 Emp_LName          VARCHAR (21) NOT NUll,
 Emp_FName          VARCHAR (15),
 Emp_SSN            CHAR (9),
 Emp_DOB            DATE,
 Emp_Hire_Date      DATE NOT NULL,
 Emp_Term_Date      DATE,
 Emp_Salary         NUMERIC (9,2),
 Emp_Dept_No        SMALLINT,
 Emp_Mgr_ID         SMALLINT,
 Emp_Gender         CHAR (1),
 Emp_Health_Ins     CHAR (1),
 Emp_Dental_Ins     CHAR (1),
 Emp_Comments       VARCHAR (255),
CONSTRAINT Pk_Emp_Id PRIMARY KEY (Emp_No),
CONSTRAINT Fk_Emp_Dept FOREIGN KEY (Emp_Dept_No)
 REFERENCES DEPARTMENT (Dept_No) )
```

This syntax presupposes that the table **DEPARTMENT** already exists. If it doesn’t, you need to create that table first. When you create tables and add constraints with the same code, figuring out which table needs to be built first can get complicated. I recommend that you create all your tables first and then add the constraints later with **ALTER** statements.

ALTER TABLE

The **ALTER TABLE** command changes a table’s structure in some way. Exactly what you can change and when you can change it is somewhat dependent on the exact RDBMS. However, you can generally add a column at any time using the following syntax:

```
ALTER TABLE EMPLOYEE
ADD Emp_Maiden CHAR (21) NOT NULL
```

This example adds a column named **Emp_Maiden** with a data type of **CHAR (21)**. It is specified **NOT NULL**. The column is always added to the end of the table. You cannot, unfortunately, delete a column.

You can add constraints to a table using syntax similar to the following:

```
ALTER TABLE EMPLOYEE
ADD CONSTRAINT Pk_Emp_Id PRIMARY KEY (Emp_No)

ALTER TABLE EMPLOYEE
ADD CONSTRAINT Fk_Emp_Dept FOREIGN KEY (Emp_Dept_No)
REFERENCES DEPARTMENT (Dept_No)
```

This example adds a primary key to the **EMPLOYEE** table and a foreign key that references the **Dept_No** column in the **DEPARTMENT** table. Note that many databases assume the primary key when you don't specify the column name on the parent table. In other words, the following statement causes SQL to assume that the primary key of the **DEPARTMENT** table is being referenced:

```
ALTER TABLE EMPLOYEE
ADD CONSTRAINT Fk_Emp_Dept FOREIGN KEY (Emp_Dept_No)
REFERENCES DEPARTMENT
```

Some databases allow you to modify the definition of an existing column as long as the currently stored data in that column does not violate the new column definition. For instance, you can specify a column as **NOT NULL** even if it was not originally defined that way, as long as there are no null values already stored in the column:

```
ALTER TABLE EMPLOYEE
MODIFY Emp_Gender NOT NULL
```

DROP TABLE

The **DROP TABLE** command deletes a table and all of its data from the database. Also dropped are all integrity constraints and permissions (I discuss permissions under "Data Control Languages" later in this chapter). The syntax is:

```
DROP TABLE EMPLOYEE
```

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Indexes

The database allows you to add indexes on one or more columns of any table. The indexes may be unique, meaning that the index does not permit duplicate, or non-unique, values. You can also use a unique index as a constraint to restrict the values in a column on a table. For instance, you might want to add a unique index on the Social Security number of an employee table.

The syntax to create an index is as follows:

```
CREATE [UNIQUE] INDEX INDEX_NAME ON TABLE_NAME
   (col_name, col_name …)
```

The ***TABLE_NAME*** must be any valid table name. The ***col_name*** must be a valid column on that table. If more than one is listed, the column names must be comma-delimited. The following examples add a unique index on the **Emp_SSN** column and a non-unique index on the **Emp_LName** and **Emp_FName** columns:

```
CREATE UNIQUE INDEX IDX_SSN ON EMPLOYEE (Emp_SSN)

CREATE INDEX IDX_NAME ON EMPLOYEE (Emp_LName, Emp_FName)
```

Most databases allow you to create one *clustered* index per table. With a clustered index, the rows are arranged in the same order as the index, which speeds up retrieval. Because the syntax for creating clustered indexes varies greatly from database to database, check your RDBMS documentation.

You will want to use indexes judiciously. They can speed up database operations tremendously when you often access data in a certain manner. For instance, if you need to list employees in last name, first name order, an index tremendously speeds up retrieval time. On the other hand, indexes need to be maintained. If you update a table on the database, you must update all the indexes on that table as well. Thus, you want to

only use indexes where they are needed.

TIP
Indexes And The Query Optimizer
All of the RDBMSs have a built-in query optimizer. When you issue a query against the database, the optimizer attempts to determine the most efficient way of fulfilling the request. If indexes are available, it attempts to use them. However, if a column has only three or four discrete values (for example, the **Emp_Gender** column has only two values: F and M) it is actually more expensive to use the index than to ignore it. Fortunately, most modern query optimizers usually see this and don't use the index. Even so, you should consider the nature of the data in the columns that you are thinking about indexing.

Integrity Constraints

In the previous discussion of **CREATE TABLE** and **ALTER TABLE**, I presented some sample syntax on creating primary key and foreign key constraints. Different RDBMSs have some variance in how to create these constraints, and most allow you to create constraints in a variety of different manners. The syntax that I presented is fairly generic and works on most RDBMSs. For other options, consult your RDBMS documentation.

A couple of notes are in order. Consider the following statements, which add three constraints to the **EMPLOYEE** table:

```
ALTER TABLE EMPLOYEE
ADD CONSTRAINT Pk_Emp_Id PRIMARY KEY (Emp_No)

ALTER TABLE EMPLOYEE
ADD CONSTRAINT Fk_Emp_Dept FOREIGN KEY (Emp_Dept_No)
REFERENCES DEPARTMENT (Dept_No)

ALTER TABLE EMPLOYEE
ADD CONSTRAINT Ch_Gender CHECK (Emp_Gender IN ('M','F')
```

The first example adds a primary key constraint to the table, forcing the **Emp_No** column to be a unique identifier for the table. The second adds a foreign key constraint, requiring that the value stored in **Emp_Dept_No** be a valid value in the **Dept_No** column on the **DEPARTMENT** table. Because the **Emp_Dept_No** column was not defined as **NOT NULL**, the column may also contain null values. (This is an example of a conditional foreign key constraint; it actually says that if the value is not null, it must be a valid value from the parent table.)

The third example is an example of a check constraint, which is not supported by all RDBMSs. A check constraint restricts the values of the columns by providing an SQL function. In this case, it specifies that gender must be **'M'** or **'F'**. The constraint could also reference another column on the same row, as in the following example:

```
ALTER TABLE EMPLOYEE
ADD CONSTRAINT Ch_Hire_Date CHECK (Emp_Hire_Date > Emp_DOB)
```

This example requires that hire date be greater than date of birth. You cannot use an SQL select in a check constraint. When a check constraint references another column on the same row, it is called a *user integrity constraint*. When it provides an absolute check

(such as being only **'M'** or **'F'** without regard for other columns on the row), it is called a *domain integrity constraint.*

Generally, you do not need to provide a constraint name. If you do not, the database generates one. My recommendation is that you do provide a name so that any messages from the database are meaningful. For instance, if you attempt to set **Emp_Gender = 'X'**, the database responds with a message similar to "Update Failed - Integrity Constraint: CH_GENDER".

If you elect not to name the constraint, you should also omit the **CONSTRAINT** keyword, as in the following example:

```
ALTER TABLE EMPLOYEE
ADD CHECK (Emp_Gender IN ('M','F'))
```

Views

A view is an SQL **SELECT** permanently stored on the database. A view can be convenient for your users because it helps them avoid typing complicated queries. It can also be used to show a user a portion of the table without revealing the entire underlying table. The following example creates a view on the **EMPLOYEE** table that allows users to see information such as name but not salary:

```
CREATE VIEW EMPNOSAL AS
SELECT Emp_No, Emp_LName, Emp_FName, Emp_Dept_No, Emp_Mgr_ID
FROM EMPLOYEE
```

To use the view, select from it as though it were a table:

```
SELECT *
FROM EMPNOSAL
ORDER BY Emp_LName, Emp_FName
```

A view is not a copy of the data—just a stored query. Also note that most databases allow you to update through a view, as in the following example:

```
UPDATE EMPNOSAL
SET Emp_Dept_No = 200
WHERE Emp_No = 100
```

In the example, the employee's department number was updated (I discuss the syntax for the **UPDATE** command in Chapter 3). Because the view **EMPNOSAL** does not contain the **Emp_Salary** column, the user can't update salary information.

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Triggers And Stored Procedures

A trigger is a small program that is stored on the database and defined to run automatically as a result of a database action, such as an insert or update of a certain table. The program consists of SQL statements plus RDBMS-specific extensions. For instance, a trigger written for an Oracle database is created using PL/SQL.

A stored procedure is similar to a trigger except that it runs when called by a trigger, another stored procedure, or your program.

The advantage of triggers and stored procedures is that they can move a good deal of business logic onto the server. For example, you might want the ZIP code to be validated every time an address is updated. This is an ideal candidate for a trigger that is tied to the address table and defined to run any time a row is updated or inserted. Implementing this rule as a trigger both reduces network traffic and relieves the client program of the burden of performing this edit.

Because each RDBMS uses different SQL dialects, the syntax for triggers and stored procedures is highly variable. Nevertheless, later in the book I introduce some examples to show how triggers and stored procedures are used by your Visual Basic programs.

Data Control Language

Data Control Language (DCL) consists of two commands that control who has access to what objects on the database. By definition, the database administrator (DBA) has authorization to all objects. Beyond that, if you want to give another user access to an object that you created, you must specifically grant him or her permission.

The two commands used are **GRANT** and **REVOKE**. For a user to have access to the database, he or she must first be granted connect privileges:

```
GRANT CONNECT TO user_ID IDENTIFIED BY password
```

This syntax creates a new user and password. Assuming the user ID is **Jane**, to alter the password, use the following syntax:

```
GRANT CONNECT TO Jane IDENTIFIED BY new_password
```

To grant access to individual objects (such as a table), you have to grant a comma-delimited list of permissions to a comma-delimited list of users on one table at a time:

```
GRANT SELECT, UPDATE, DELETE
ON EMPLOYEE
TO Tom, Dick, AND Harry
```

To grant all permissions, use **GRANT ALL PERMISSIONS**. To grant permissions to all users, specify **PUBLIC** instead of a user list:

```
GRANT SELECT
ON DEPARTMENT
TO PUBLIC
```

You can selectively revoke permissions, but you cannot revoke access from yourself. The syntax to revoke privileges is similar to the following:

```
REVOKE UPDATE, DELETE
ON EMPLOYEE
FROM Dick, Harry
```

In this example, we have taken away update and delete privileges from users **Dick** and **Harry**. Their select privileges remain.

Most databases allow you to create groups and assign users to groups. You can then grant permissions to and revoke permissions from the groups. This technique helps simplify administration.

Where To Go From Here

This chapter covered the theory of relational databases from their history through the theory of database design to actually implementing that design. From here, you will want to read the next chapter, “An Introduction To SQL Data Manipulation Language.” Data Manipulation Language (DML) is where you will spend most of your SQL time, so it is a critical concept to learn. Also, you might want to look at Appendix A, where I provide the syntax used to create the sample databases in this book as well as the syntax for the constraints. In Appendix B, I discuss the key differences between Jet SQL and ANSI SQL, which is a concern both for developers using Access databases and VB developers using Microsoft Jet in conjunction with DAO.

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Previous | Table of Contents | Next

Chapter 3
An Introduction To SQL Data Manipulation Language

Key Topics:

- The definition and purpose of SQL
- Data Manipulation Language
- Scalar and aggregate functions
- Grouping data and the **HAVING** clause
- Joining tables, including self-joins
- SQL subqueries
- Performance considerations in tuning an SQL statement

In Chapter 2, I discussed the concept of the relational database, mentioning at the same time that SQL is used to access and manipulate the database. I also discussed the theory of relational database design and showed you the SQL syntax used to implement that design.

In this chapter, I guide you through the essential concepts of using *Structured Query Language* (SQL) to access and manipulate your database. I need to stress again that SQL is not a database nor does it necessarily imply the existence of a back-end relational database engine. All RDBMSs use SQL as an interface (to extract data or to create tables, for example), but other, non-relational data models such as dBase may offer SQL as an access language. Further, although a product such as Microsoft Access is a fully relational database, it is not a separate database engine; the application program is responsible both for creating the data request and for creating the

query result. (I expanded upon this in Chapter 2 when I defined the term RDBMS.)

What Is SQL?

The American National Standards Institute (ANSI) defines various computer industry standards. We are all familiar with the ANSI character set, for instance, which defines each character to be 1-byte long with the decimal value 65 representing the letter A. SQL-92 is an ANSI definition of what features must be supported in a full implementation of SQL. For example, the language needs to support the **SELECT** statement and the **SUM** function. If a vendor's database product is in full compliance with these standards, it is ANSI-92 compliant. All of the major relational database products, such as Oracle and Microsoft SQL Server, are 100 percent ANSI-92 compliant. (Microsoft Access uses Jet SQL. Although Jet SQL is close, it is not ANSI-92 compliant. I discuss the differences in Appendix B.) Additionally, most vendors add *extensions* (additional features) implemented as what are commonly called *dialects* of SQL. Oracle uses PL/SQL, whereas Sybase and Microsoft use Transact-SQL (usually called T-SQL). Microsoft has been moving away from the Sybase model of T-SQL. Mostly, the dialects are close enough in syntax that moving from one to the other is relatively easy.

When application programs access a relational database, they create SQL statements and send them to the RDBMS. The RDBMS processes the statements and sends the results back to the application program. Each RDBMS exposes its functionality via an *Application Programming Interface (API)*. To shelter the developer from the intricacies of learning different APIs for each RDBMS, three categories of drivers were developed to interface with Visual Basic:

- *VBSQL*—A Visual-Basic-specific set of drivers that allow "native" interface to Microsoft SQL Server.
- *ODBC (Open Database Connectivity)*—An API that applications can address to further expose the underlying functionality of a wide variety of databases. The many variations on how this is done is the subject of much of this book. Appendix C overviews the implementation of ODBC.
- *OLE DB*—A new, still evolving, API that exposes the underlying functionality of a wide variety of databases in an object-oriented manner. OLE DB also allows the joining of disparate data sources.

A Note About ODBC

ODBC is itself an API that application programs can use to gain access to a variety of database back ends using standard SQL. Visual Basic provides varying levels of intermediate drivers (such as Microsoft Jet) that shelter you from the need to write API calls. The back end might be an RDBMS, such as Oracle, or it might be an Indexed Sequential Access Method (ISAM) file, such as a FoxPro database. In theory, the developer can write code to access virtually any back-end database while being shielded from the complexities of the different access methods. This shielding is accomplished by providing a

single API, which communicates with ODBC drivers written for each of the different back-end databases. Figure 3.1 illustrates this communication process.

Figure 3.1 Client programs communicate with the ODBC API, which in turn communicates with the individual database-specific drivers. These drivers interact with the back-end databases.

In reality, not all ODBC drivers are created equal. There are actually three levels of ODBC compliance: Level 0, sometimes called *Minimum ODBC*, is a set of minimal functionalities that must be supplied by an ODBC driver, including the ability to perform basic SQL **SELECT**s and so on. Level 1, sometimes called *Core ODBC*, is a more thorough implementation of SQL, including statements such as **ALTER TABLE** as well as more advanced **SELECT** functionality, such as the **MAX** and **MIN** functions. Level 2, or *Extended ODBC*, is an implementation of SQL that is at least ANSI-92 compliant, including outer joins, cursor-positioned **UPDATE**s, and so on. The exact definition of each of these compliance levels is discussed in Appendix C.

It is not always easy to determine the level of conformance of a given ODBC driver, and performances of the drivers vary from vendor to vendor. Microsoft builds a variety of drivers for both Microsoft and non-Microsoft back-end database products. Intersolv also markets a wide variety of drivers that you can use in your programs. Finally, some vendors create drivers for their own database products; the most notable example is the very well implemented Level 2-compliant Sybase SQL Anywhere database.

For technical information about ODBC standards and driver development, refer to Appendix C in this book, as well as to the *ODBC 3.0 Programmer's Reference And SDK Guide* available from Microsoft Press.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Data Manipulation Language (DML)

Most client/server developers will spend the majority of their time using DML—that portion of SQL used to manipulate the actual data stored in SQL tables. There are only four DML commands: **INSERT**, **UPDATE**, **DELETE**, and **SELECT**. Of these four statements, most developers will spend most of their time coding SQL **SELECT** statements. In the sections that follow, I discuss the use of each of these statements. In Chapter 2, I discussed Data Control Language (DCL) and Data Definition Language (DDL).

SELECT

The **SELECT** statement is used to retrieve rows from the database. The basic syntax of the **SELECT** statement is:

```
SELECT [DISTINCT] column-list| *
FROM table-listT
[WHERE condition1 AND condition2…]
[GROUP BY column-list
   [HAVING condition1 AND condition2…]]
[ORDER BY column-list]
```

All clauses except **FROM** are optional.

Use the **SELECT** clause to list those columns that are to appear in the result set. Each column name must be separated by a comma. To select all columns, you can use the asterisk instead of listing each column separately. Use the **FROM** clause to list the tables that contain the columns being referenced. For more than one table, separate the table names with commas.

The following example selects three columns from the **EMPLOYEE** table:

```
SELECT EMP_ID, EMP_FNAME, EMP_LNAME
FROM EMPLOYEE
```

The next example selects all the columns from the **EMPLOYEE** table:

```
SELECT *
FROM EMPLOYEE
```

Derived Columns

SQL allows you to "select" columns that are not part of the database by creating derived columns. These columns might be string literals, system functions, or the results of operations on table columns.

TIP
About The Examples
Except where specifically noted, all of the examples in this chapter use the sample data provided on the CD-ROM. See Appendix A for details on loading this information into your database.

The following **SELECT** creates a derived column that is the result of multiplying **EMP_SALARY** by 1.08:

```
SELECT EMP_SALARY * 1.08
FROM EMPLOYEE
```

The next example creates a derived column that is actually a string literal:

```
SELECT EMP_FNAME, EMP_LNAME, 'IS AN EMPLOYEE'
FROM EMPLOYEE
```

As shown in the following result set, the string literal is returned for every row:

```
EMP_FNAME  EMP_LNAME  'IS AN EMPLOYEE'
---------  ---------  ----------------
ANN        CALLAHAN   IS AN EMPLOYEE
BOB        JOHNSON    IS AN EMPLOYEE
CAROL      DEMORA     IS AN EMPLOYEE
```

Some system functions are not truly related to any data at all. For instance, **SYSDATE** returns the current date and time from the database server. Even still, the **SELECT** statement must have a **FROM** clause. Some RDBMSs provide a "dummy" table from which to select such non-data-related functions:

```
SELECT SYSDATE
FROM DUAL
```

This example is from Oracle, which provides a table named **DUAL** with a single row so that you can select against system-level functions. If you select **SYSDATE** from the **EMPLOYEE** table, and if the table has 100 rows, you get a result set of 100 lines of the current date and time.

If your database does not provide a dummy table, you can easily create one:

```
CREATE TABLE DUMMY
(DUM_COL CHAR(10) )

INSERT INTO DUMMY VALUES ('DUMMY')

SELECT SYSDATE
FROM DUMMY
```

When the result set is displayed, the column headings default to the column names. You can rename a column heading using the **AS** keyword:

```
SELECT EMP_NO AS 'EMPLOYEE NUMBER'
FROM EMPLOYEE
```

Some databases allow you to omit the **AS** keyword. If the new column name contains a space, you must surround the column name with double quotes. Some databases require that you use double quotes all the time (regardless of whether there is a space in the name).

The WHERE Clause

The **WHERE** clause is used to restrict the rows returned in the result set by specifying search conditions. For specifying more than one search condition, use the **AND** keyword. The following **SELECT** returns all rows in the **EMPLOYEE** table where gender is female and where salary is greater than 50,000:

```
SELECT *
FROM EMPLOYEE
WHERE EMP_GENDER = 'F'
AND   EMP_SALARY > 50000
```

The **WHERE** clause allows Boolean logic to determine which rows are to be returned. **NOT** negates the search condition:

```
WHERE NOT EMP_GENDER = 'F'
```

OR allows for the selection of rows based upon multiple criteria:

```
WHERE EMP_GENDER = 'F'
 OR   EMP_SALARY > 50000
```

Use **NOT** to negate a search condition:

```
WHERE NOT (EMP_SALARY > 50000 AND EMP_GENDER = 'F')
```

Use parentheses to perform algebraic evaluations of search conditions. The following **WHERE** clause specifies that the result set should include females who make more than $50,000 as well as those employees who make more than

$75,000 (regardless of gender):

```
WHERE (EMP_GENDER = 'F' AND EMP_SALARY > 50000)
 OR    EMP_SALARY > 75000
```

You may use a scalar function in a **WHERE** clause but not an aggregate function. I discuss SQL functions later in this chapter, but for now, understand that a *scalar function* is one that operates on a single value at a time (such as returning the length of a last name), whereas an *aggregate function* operates on a range of values (such as returning the sum of all salaries). The following example returns rows where the length of the last name is greater than five:

```
WHERE LENGTH(EMP_LNAME) > 5
```

A column used in the **WHERE** clause does not have to appear in the **SELECT** statement, as shown in the following example:

```
SELECT EMP_ID, EMP_FNAME, EMP_LNAME
FROM EMPLOYEE
WHERE EMP_GENDER = 'F'
```

You can perform wild-card searches using the **LIKE** keyword. To do so, use the underscore (_) character to represent a single character and the percent sign (%) to represent any number of characters. The following statement searches for all last names beginning with the letter B:

```
WHERE EMP_LNAME LIKE 'B%'
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The next example locates all those rows where the last names have the letter B in the second position and the letter D in the fourth position:

```
WHERE EMP_LNAME LIKE '_B_D%'
```

The **WHERE** clause also supports the use of the **BETWEEN** and **IN** modifiers. With **BETWEEN**, you can specify an inclusive range of values as in the following example, which returns rows where the salary is from $50,000 to $60,000:

```
WHERE EMP_SALARY BETWEEN 50000 AND 60000
```

The **IN** keyword allows you to search a list of discrete values. You separate each value with commas and place the entire list inside parentheses. If the values are strings, they must be in single quotes:

```
WHERE EMP_LNAME IN ('BROWN', 'SMITH', 'JOHNSON')
```

TIP
About Databases And Case Sensitivity
When dealing with character data (strings), the database is case sensitive just as Visual Basic is case sensitive when comparing two strings. Later in this chapter, I will introduce case conversion functions to help deal with this problem.

Though the SQL code samples in this book have all keywords in uppercase, databases are generally not case sensitive with column names unless you surround the column names with double quotes.

Testing for the existence of null values requires special functions. Because any operation involving null is automatically null (I discuss this fully later in this chapter), SQL provides the **IS NULL** and **IS NOT NULL** functions. The following statement locates rows where **YEARS** is null:

```
WHERE YEARS IS NULL
```

Joining Tables

When you select data from more than one table, you need to tell SQL how the tables are related. You do this with the **WHERE** clause. Consider the **DEPARTMENT** and **EMPLOYEE** tables (see Appendix A for how the tables are defined). If I want to produce a list of departments and all the employees in each of those departments, I need to relate the **EMPLOYEE** table to the **DEPARTMENT** table. Each row on the **EMPLOYEE** table has an **EMP_DEPT_NO** column, and the **DEPARTMENT** table has a **DEPT_NO** column. Therefore, given a row in the **DEPARTMENT** table, I want to list all rows in the **EMPLOYEE** table where **DEPT_NO** is equal to **EMP_DEPT_NO**. This forms a join condition:

```
SELECT DEPT_NO, DEPT_NAME,
       EMP_NO, EMP_FNAME, EMP_LNAME
FROM EMPLOYEE, DEPARTMENT
WHERE DEPT_NO = EMP_DEPT_NO
```

Although I did not do so, many developers use common column names between tables. Whereas I used **EMP_DEPT_NO** in the **EMPLOYEE** table to store the department number, others might have simply used **DEPT_NO**. Obviously then, the two tables both have a column with the same name (**DEPT_NO**). There is nothing wrong with this at all. However, if an SQL query references a column name that appears in more than one table in the query, you have to qualify the column name with the table name to avoid ambiguity:

```
SELECT DEPARTMENT.DEPT_NO, DEPT_NAME,
       EMP_NO, EMP_FNAME, EMP_LNAME
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.DEPT_NO = DEPARTMENT.DEPT_NO
```

This example shows how to qualify the column names where ambiguity exists. The example assumes, of course, that the two column names are identical.

Table Aliasing

Typing in all those table names can be tiring, especially when qualifying column names. SQL allows you to *alias* table names in the **FROM** clause:

```
FROM EMPLOYEE EMP, DEPARTMENT DEP
```

Once you have aliased a column, you use the alias instead of the table name elsewhere in the query:

```
SELECT EMP.EMP_NO, DEP.DEPT_NO
```

Aliasing also has implications in correlated subqueries and self-joins, both of which I discuss later in this chapter. Aliased names are often referred to as *correlation names*, particularly when dealing with subqueries.

You can join together more than two tables. If in the previous example I want to add the location name (from the **LOCATION** table), I have to join that table to one of the other two tables. The **DEPARTMENT** table has a column **DEPT_LOC_ID** that corresponds to the **LOC_ID** column in the **LOCATION** table. Therefore, my query might look like this:

```
SELECT DEPT_NO, DEPT_NAME,
       LOC_NAME, EMP_NO, EMP_FNAME, EMP_LNAME
FROM EMPLOYEE, DEPARTMENT, LOCATION
WHERE DEPT_NO = EMP_DEPT_NO
 AND DEPT_LOC_ID = LOC_ID
```

Sample output based on the above query is shown here:

```
DEPT_NO  DEPT_NAME   LOC_NAME   EMP_NO  EMP_FNAME  EMP_LNAME
-------  ---------   --------   ------  ---------  ---------
100      ACCOUNTING  NORTHEAST  133     MAUREEN    PODANSKI
100      ACCOUNTING  NORTHEAST  101     ANN        CALLAHAN
100      ACCOUNTING  NORTHEAST  105     EUNICE     BROWN
100      ACCOUNTING  NORTHEAST  113     MICHAEL    ANDERSON
```

Self-Joins

A confusing concept for many developers is that you can join a table to itself. Consider the **EMPLOYEE** table: One of the columns is **EMP_MGR_ID**, which represents the manager of the employee. If you select the contents of the **EMPLOYEE** table, you will see that employee number 126's (David Madison) manager is employee 110 (John Smith). Assume I want a list of all employees and the names of their managers. In Figure 3.2, you can see where I need to join a column from one row (**EMP_MGR_ID**) with a column in another row (**EMP_NO**). Essentially, I need to "pretend" that there are two copies of the **EMPLOYEE** table.

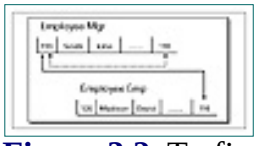

Figure 3.2 To find the name of David Madison's manager, I need to reference the **EMP_MGR_ID** column on David's row and use it to find John Smith's row.

The query that I put together uses two aliases for the **EMPLOYEE** table. I use the first alias (**EMP**) to find the employee information and the second (**MGR**) to find the manager information:

```
SELECT EMP.EMP_NO AS 'EMP NO',
       EMP.EMP_FNAME AS 'EMP FIRST',
       EMP.EMP_LNAME AS 'EMP LAST',
       MGR.EMP_NO AS 'MGR ID',
       MGR.EMP_FNAME AS 'MGR FIRST',
       MGR.EMP_LNAME AS 'MGR LAST'
FROM EMPLOYEE EMP, EMPLOYEE MGR
WHERE EMP.EMP_MGR_ID = MGR.EMP_NO
```

```
ORDER BY EMP.EMP_LNAME, EMP.EMP_FNAME
```

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The join is on the **EMP_NO** column from the **MGR** copy of the **EMPLOYEE** table to the **EMP_MGR_ID** column of the **EMP** copy of the table. Sample output based on the query is presented here. Note that Ann Callahan has no manager:

```
EMP NO   EMP FIRST   EMP LAST    MGR ID   MGR FIRST   MGR LAST
------   ---------   --------    ------   ---------   --------
111      KATHRYN     AMES        119      URSULA      SMITHSONIAN
113      MICHAEL     ANDERSON    101      ANN         CALLAHAN
106      FRANK       BENSON      110      JOHN        SMITH
105      EUNICE      BROWN       101      ANN         CALLAHAN
101      ANN         CALLAHAN    101      ANN         CALLAHAN
```

Self-joins are not terribly difficult once you get used to the idea of joining a table to itself; they can often be useful to find information in one row that is related to information in another row in the same table.

The ORDER BY Clause

You can ask the database to sort your result set by using the **ORDER BY** clause. Specify the column or columns by which you want to sort your result set. If ordering by more than one column, separate the column names with commas. The default sort sequence is ascending. If you want to sort in descending order, specify the **DESC** option. The following query lists employees and their salaries sorted by department number. Within each department number, the employees are sorted by salary in descending order:

```
SELECT EMP_DEPT_NO, EMP_FNAME, EMP_LNAME, EMP_SALARY
FROM EMPLOYEE
ORDER BY EMP_DEPT_NO, EMP_SALARY DESC
```

Sample output based on the query is presented here:

```
EMP_DEPT  EMP_FNAME  EMP_LNAME  EMP_SALARY
--------  ---------  ---------  ----------
100       ANN        CALLAHAN   127500.00
100       MICHAEL    ANDERSON    72563.34
100       ROSE       DANIELS     67572.42
100       EUNICE     BROWN       61426.08
100       CHARLES    GERRFON     53496.87
100       WINIFRED   VANCE       37712.10
100       MAUREEN    PODANSKI    34591.90
100       GEORGE     DE NORVA    21313.97
200       DAVID      MADISON     77492.47
200       HENRIETTA  KEOUGH      72789.20
200       BRIAN      JOHNSON     72750.62
```

The columns that you order by do not have to appear in the **SELECT** list, although it makes no sense not to include them. (How would the reader know that data was sorted by department if you did not display the department?)

You can order *positionally* by specifying the column number instead of the column name, as shown in this example:

```
ORDER BY 1, 4 DESC
```

However, it is likely that the next ANSI SQL standard will remove this option as part of the SQL definition. You may, therefore, be better off ordering by the column names themselves. If you have renamed the columns (see the discussion of **SELECT** earlier in this chapter), you can order by the "new" column name as shown:

```
SELECT EMP_FNAME AS 'FIRST', EMP_LNAME AS 'LAST'
FROM EMPLOYEE
ORDER BY 'LAST', 'FIRST'
```

Under certain circumstances, you *must* order by the new column names. An example is performing a **UNION** (which I discuss next).

UNION And UNION ALL

The **UNION** statement allows you to combine the results of two queries. The only restrictions are that the queries must include the same number of columns and that the columns must be the same general data type. (For example, it is okay that one is a **CHAR** and the other is a **VARCHAR**.) You can combine as many queries as you want.

Whereas **UNION ALL** combines all rows, **UNION** sorts the result set and eliminates any duplicate rows.

Ordering a **UNION** query can be tricky because the column names from one query to another may be different (as in the following example). Positional ordering is most convenient. Some databases require that you order by the column names of the first query (**EMP_NO** and **EMP_LNAME** in the following example) or that you rename the columns and order by those new names. Check your RDBMS documentation or just try the query and experiment with the **ORDER BY** clause:

```
SELECT EMP_NO AS 'NUMBER', EMP_LNAME AS 'NAME'
FROM EMPLOYEE
UNION
SELECT DEPT_NO, DEPT_NAME
FROM DEPARTMENT
ORDER BY 2, 1
```

SQL Functions

SQL provides a number of functions to manipulate information from the database. Functions fall into two categories: *Scalar* functions act on one item at a time and *aggregate* functions act on a range of values at a time.

Deciding whether a function is scalar or aggregate is more than an academic exercise. Consider the **WHERE** and **HAVING** clauses (I discuss **HAVING** later in this chapter):

- ***WHERE** clause*—You can use a scalar function as a search condition in a **WHERE** clause. For instance, **WHERE SQRT (EMP_SALARY) > 5000** (**SQRT** returns the square root of an expression) is valid, but **WHERE SUM (EMP_SALARY) > 250000** is not.
- ***HAVING** clause*—You can use aggregate functions in a **HAVING** clause to restrict a result set. For instance, **HAVING SUM (SALARY) > 250000** is valid, but **HAVING SQRT (EMP_SALARY) > 5000** is not.

Whether a function is scalar or aggregate also has an impact on handling of null values.

Null Handling

A value of **NULL** stored in a database creates special problems. Consider the following:

```
100 + NULL = ?
```

Nonintuitively, the answer to this equation is **NULL**. Any operation performed on **NULL** is automatically **NULL**. Assume that you want to give every employee a $100 a week raise, and you want to see what the new weekly salary for each employee would be. You might code the following query:

```
SELECT EMP_FNAME, EMP_LNAME, EMP_SALARY/52 + 100
FROM EMPLOYEE
```

The **SELECT** works fine except for those employees whose salary happens to be **NULL**. The result for those employees is still **NULL**. Different databases handle this problem with various functions. With Oracle, you can use the **NVL** function:

```
SELECT EMP_FNAME, EMP_LNAME, NVL(EMP_SALARY, 0)/52 + 100
FROM EMPLOYEE
```

In this case, the **NVL** function specifies that if **EMP_SALARY** is **NULL**, then the value 0 should be substituted.

Some other RDBMSs implement the similar function **ISNULL**:

```
SELECT EMP_FNAME, EMP_LNAME,
   ISNULL(EMP_SALARY, 0, EMP_SALARY)/52 + 100
FROM EMPLOYEE
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

In this case, the **ISNULL** function says that if **EMP_SALARY** is equal to **NULL**, the value 0 should be substituted. If it is not equal to **NULL**, the **EMP_SALARY** value should be used. (The **ISNULL** function is similar to the **IF** function in most spreadsheets.)

Selected SQL Functions

SQL functions are one of those areas with a fairly high amount of variability between RDBMSs. Some functions, such as **SUM**, are common to all, whereas others may have no equivalent in another RDBMS. For instance, Oracle has a flexible **TO_CHAR** function to convert a noncharacter data type (such as **DATE** or **NUMBER**) to a character type and optionally format the output. Sybase uses either the **CAST** or **CONVERT** function to accomplish the same thing. In general, if you are comfortable with using functions in one RDBMS, you will have little problem moving to another RDBMS.

In the following sections, I discuss some of the functions you are most likely to use on a regular basis.

DISTINCT

Arguably, **DISTINCT** is not as much a function as it is a **SELECT** statement qualifier. However, because it operates on a single value, I include it here. Its purpose is to return only unique values for a row:

```
SELECT DISTINCT (EMP_DEPT_NO) FROM EMPLOYEE
```

The result is:

```
EMP_DEPT_NO
-----------
100
```

```
200
300
400
```

Most databases do not require that the column name be surrounded by parentheses when using **DISTINCT** (however, it does not hurt to use them).

COUNT

COUNT returns a count of all rows that meet a condition. For instance, **COUNT (EMP_DEPT_NO)** returns 32 because there are 32 values stored in that column. However, **COUNT (DISTINCT (EMP_DEPT_NO))** returns 4 because there are only 4 unique values. Often, **COUNT (*)** is used to find the number of rows on a table:

```
SELECT COUNT(*) FROM EMPLOYEE
```

Also, you can use **COUNT(*)** in conjunction with the **GROUP BY** clause to determine counts in categories (I discuss **GROUP BY** later in this chapter):

```
SELECT EMP_GENDER AS 'GENDER', COUNT(*) AS 'COUNT'
FROM EMPLOYEE
GROUP BY EMP_GENDER
```

The following result set is returned:

```
GENDER  COUNT
------  -----
F       18
M       14
```

SUM

SUM adds values in a column. To obtain a total of the salaries from the **EMPLOYEE** table, run the following query:

```
SELECT SUM(EMP_SALARY)
FROM EMPLOYEE
```

The following is the result:

```
SUM(EMP_SALARY)
---------------
1861241.06
```

MIN, MAX, And AVG

MIN calculates the minimum value in a column. Similarly, **MAX** calculates the maximum value, and **AVG** calculates the average value:

```
SELECT MIN(EMP_SALARY), MAX(EMP_SALARY), AVG(EMP_SALARY)
```

```
FROM EMPLOYEE
```

The result is:

```
MIN(EMP_SALARY)  MAX(EMP_SALARY)  AVG(EMP_SALARY)
---------------  ---------------  ---------------
18331.50         127500           58163.78
```

LENGTH

As with the Visual Basic **LEN** function, **LENGTH** returns the length of a column or expression. **LENGTH ("Smith")** returns **5**.

SOUNDEX

SOUNDEX searches for strings based upon their similarity (in sound) to the supplied argument. Although this should be a powerful function, its usefulness can be iffy. RDBMSs tend to have different levels of tolerance for what two words sound alike and may need some tuning. The following query and result comes from Sybase SQL Anywhere. Refer to your RDBMS's documentation regarding the use of **SOUNDEX**.

```
SELECT EMP_NO, EMP_FNAME, EMP_LNAME
FROM EMPLOYEE
WHERE SOUNDEX('SMYTHE') = SOUNDEX(EMP_LNAME)
```

The result of the query is:

```
EMP_NO  EMP_FNAME  EMP_LNAME
------  ---------  ---------
110     JOHN       SMITH
118     STEVEN     SMITH
103     CAROL      SMITH
```

SUBSTR, LEFT, And RIGHT

The functions **SUBSTR**, **LEFT**, and **RIGHT** each return a portion of a string from character data types.

You use **SUBSTR** to return a specific number of characters beginning anywhere within the string:

```
SELECT SUBSTR (EMP_FNAME, 1, 4), SUBSTR (LOC_NAME, 3, 8),
  SUBSTR (DEPT_NAME, 2)…
```

In this example, the first **SUBSTR** is returning 4 characters of **EMP_FNAME** beginning with position 1. The second **SUBSTR** is returning 8 characters from **LOC_NAME** beginning with position 3. In the last **SUBSTR**, the number of characters to return has been omitted. In this case, SQL simply returns all remaining characters (in this case, beginning with position 2 of **DEPT_NAME**).

The **LEFT** and **RIGHT** functions are similar in operation. **LEFT** returns the

leftmost *n* characters from a column, whereas **RIGHT** returns the rightmost *n* characters from a column:

```
SELECT LEFT (EMP_LNAME, 1), RIGHT (EMP_LNAME, 4)
```

Not all databases support the **RIGHT** and **LEFT** functions, but if not, the **SUBSTR** function is adequate.

Concatenation

Some databases provide specific functions to concatenate two strings. For instance, Oracle offers the **CONCAT** function, which accepts two arguments (two strings to be concatenated). Almost all RDBMSs support the concatenation operator, which is simply two pipe characters (the pipe character is usually above the backslash on your keyboard and looks like one vertical dash on top of another). The advantage of the concatenation operator is that you can concatenate as many strings as needed without resorting to nesting functions. The first example in the next function shows the concatenation operator in use.

TRIM, RTRIM, And LTRIM

The similar functions **TRIM**, **RTRIM**, and **LTRIM** take a string and trim any leading or trailing spaces from it. **RTRIM** is useful when you want to concatenate two fixed-length columns:

```
SELECT RTRIM(EMP_FNAME) || ' ' || EMP_LNAME AS 'NAME'
FROM EMPLOYEE
WHERE EMP_NO < 110
ORDER BY EMP_LNAME
```

The result is:

```
NAME
----------------
FRANK BENSON
EUNICE BROWN
ANN CALLAHAN
BRIAN JOHNSON
CAROL SMITH
GENEVIEVE WESLEY
```

Without trimming the rightmost characters from the first name, you get a result that looks like the following:

```
NAME
----------------------
FRANK         BENSON
EUNICE        BROWN
ANN           CALLAHAN
BRIAN         JOHNSON
CAROL         SMITH
```

```
GENEVIEVE     WESLEY
```

The **LTRIM** function removes leading spaces from a string if it has any. Some RDBMSs also provide a **TRIM** function, which combines the functionality of **LTRIM** and **RTRIM**.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

ROUND And TRUNCATE

ROUND and **TRUNCATE** are similar but not identical. The first rounds a number to the specified position, whereas the latter merely truncates a number at a specified position. Observe the difference when working on **EMP_SALARY/52** in the following example:

```
SELECT EMP_NO AS 'EMP NO',
       EMP_SALARY/52 AS 'WEEKLY',
       ROUND(EMP_SALARY/52, 3) AS 'ROUNDED',
       TRUNCATE (EMP_SALARY/52,3) AS 'TRUNCATED'
FROM EMPLOYEE
WHERE EMP_NO BETWEEN 110 AND 115
ORDER BY EMP_NO
```

The result follows. Note that the third and last rows have differences between the rounded and the truncated numbers. Any result ending in a 5 or above is rounded up.

```
EMP NO  WEEKLY     ROUNDED   TRUNCATED
------  ---------  --------  ---------
110      645.10212   645.102    645.102
111     1876.92308  1876.923   1876.923
112      696.18808   696.188    696.188
113     1395.44885  1395.449   1395.448
114      900.75038   900.750    900.750
115      678.30673   678.307    678.306
```

MOD Or REMAINDER

The **MOD** function returns the remainder after dividing two numbers. **MOD**

(15, 2) returns 1 because 15 divided by 2 equals 7 with a remainder of 1. This can be a useful function when performing a lot of math at the database server. Some RDBMSs call this function **REMAINDER**, but the purpose and syntax is otherwise identical.

POWER

Similar to the Visual Basic exponentiation operator, **POWER** raises a number to the power of another number: **POWER (2, 4)** returns 16 (2 raised to the 4th power). Not all RDBMSs support this function. You can use fractional exponents to return the root of a number. **POWER (8, 3)** returns two, which is the third root of eight.

Date Functions

The different RDBMSs have a wide range in what functions they provide to support operations on dates. For instance, Oracle has a **DAYS_AFTER** function, which returns the date that falls *n* days after the supplied date:

```
SELECT DAYS_AFTER ('02-FEB-99', 365)
```

This function returns '**02-Feb-00**'. Sybase uses **DAYS**:

```
SELECT DAYS ('02-FEB-99', 365)
```

Check your RDBMS documentation for specific date functions.

GROUP BY

The SQL **GROUP BY** clause, which allows you to group data, is particularly useful when taking breaks on aggregate functions (such as showing subtotals). When a **SELECT** has a column with an aggregate function, you must **GROUP BY** all columns where there is no aggregate function. The following query performs a sum on **EMP_SALARY** and also selects **EMP_DEPT_NO** and **EMP_GENDER**. Therefore, you must group by the latter two columns, as shown in this example:

```
SELECT EMP_DEPT_NO, EMP_GENDER, SUM(EMP_SALARY)
FROM EMPLOYEE
GROUP BY EMP_DEPT_NO, EMP_GENDER
ORDER BY EMP_DEPT_NO, EMP_GENDER
```

It sometimes confuses developers that they need to **GROUP BY** all non-aggregate function columns, but the example shows it only makes sense that if you are summing salary information, you need to take breaks (show subtotals) on all the other columns. The result follows:

```
EMP_DEPT_NO  EMP_GENDER  SUM(EMP_SALARY)
-----------  ----------  ---------------
100          F                 328802.50
100          M                 147374.18
```

```
200          F                147115.66
200          M                311425.32
300          F                525056.55
300          M                 77568.70
400          F                 75759.54
400          M                248138.61
```

You can use multiple aggregate functions as shown in the following example, which compares minimum, maximum, and average salaries for males and females in each department and shows a count for each:

```
SELECT EMP_DEPT_NO AS 'DEPT',
       EMP_GENDER AS 'SEX',
       COUNT(*) AS 'COUNT',
       MIN(EMP_SALARY) AS 'MIN SALARY',
       MAX(EMP_SALARY) AS 'MAX SALARY',
       ROUND(AVG(EMP_SALARY),2) AS 'AVG SALARY'
FROM EMPLOYEE
GROUP BY EMP_DEPT_NO, EMP_GENDER
ORDER BY EMP_DEPT_NO, EMP_GENDER
```

The result of the query follows:

```
DEPT  SEX  COUNT  MIN SALARY  MAX SALARY  AVG SALARY
----  ---  -----  ----------  ----------  ----------
100   F    5      34591.90    127500.00   65760.50
100   M    3      21313.97     72563.34   49124.73
200   F    3      27487.44     72789.20   49038.55
200   M    6      33545.31     77492.47   51904.22
300   F    8      18331.50    104800.00   65632.07
300   M    1      77568.70     77568.70   77568.70
400   F    2      33543.13     42216.41   37879.77
400   M    4      36201.78     73643.82   62034.65
```

As shown in the result, females in this company tend to make less than their male counterparts in similar jobs except in Department 100. (That was entirely unintentional. All data in the supplied sample tables was generated randomly.)

You can also **GROUP BY** columns that do not appear in the **SELECT** list. This option mostly has applications in subselects, which I discuss later in this chapter. Sometimes, however, this option is useful even without subselects but in conjunction with the **HAVING** clause, which I discuss next.

HAVING

The last clause in the SQL **SELECT** statement is the **HAVING** clause. The **HAVING** clause allows you to restrict groups of data based on the results of aggregate functions and thus is usually associated with the **GROUP BY** clause. You will most likely read in some SQL texts that **HAVING** *must* be used in conjunction with the **GROUP BY** clause, but that is not actually true. The following statement, although not very meaningful, executes just fine:

```
SELECT SUM(EMP_SALARY)
FROM EMPLOYEE
HAVING SUM(EMP_SALARY) > 1
```

The example says to select the sum of all salaries if that sum is greater than one. When used in conjunction with **GROUP BY**, the **HAVING** clause is more meaningful. I refine the earlier example of salaries by department and gender to show only those departments and gender combinations where the average salary is less than $50,000:

```
SELECT EMP_DEPT_NO AS 'DEPT',
       EMP_GENDER AS 'SEX',
       COUNT(*) AS 'COUNT',
       MIN(EMP_SALARY) AS 'MIN SALARY',
       MAX(EMP_SALARY) AS 'MAX SALARY',
       ROUND(AVG(EMP_SALARY),2) AS 'AVG SALARY'
FROM EMPLOYEE
GROUP BY EMP_DEPT_NO, EMP_GENDER
HAVING AVG(EMP_SALARY) < 50000
ORDER BY EMP_DEPT_NO, EMP_GENDER
```

The new report follows:

```
DEPT  SEX  COUNT  MIN SALARY  MAX SALARY  AVG SALARY
----  ---  -----  ----------  ----------  ----------
100   M    3      21313.97    72563.34    49124.73
200   F    3      27487.44    72789.20    49038.55
400   F    2      33543.13    42216.41    37879.77
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Sometimes it is useful to combine the **GROUP BY** and **HAVING** clauses even where the aggregate function does not appear in the **SELECT** list:

```
SELECT EMP_DEPT_NO
FROM EMPLOYEE
GROUP BY EMP_DEPT_NO
HAVING MIN(EMP_SALARY) < 25000
```

This query returns two rows (Departments 100 and 300) where the minimum salary is less than $25,000.

You can also use **HAVING** in conjunction with a **GROUP BY** column that does not appear in the **SELECT** list, as you will see in the next section.

Subqueries

A subquery can be considered a way to join the result of two queries. Suppose you want to find all employees who work in a department whose average salary is greater than $55,000. You really need to first know what departments have an average salary greater than $55,000:

```
SELECT EMP_DEPT_NO, AVG(EMP_SALARY)
FROM EMPLOYEE
GROUP BY EMP_DEPT_NO
```

As shown in the following result, you need to list those employees in Departments 100 and 300:

```
EMP_DEPT_NO  EMP_SALARY
-----------  ----------
100          59522.08
200          50948.99
```

```
300          66958.36
400          53983.02
```

You can now run a second query:

```
SELECT EMP_NO, EMP_DEPT_NO, EMP_FNAME, EMP_LNAME, EMP_SALARY
FROM EMPLOYEE
WHERE EMP_DEPT_NO IN (100, 300)
```

The trick, however, is to ask *both* questions at once. To do this, you need a subquery:

```
SELECT EMP_NO, EMP_DEPT_NO, EMP_FNAME, EMP_LNAME, EMP_SALARY
FROM EMPLOYEE
WHERE EMP_DEPT_NO IN
 (SELECT EMP_DEPT_NO
  FROM EMPLOYEE
  GROUP BY EMP_DEPT_NO
  HAVING AVG(EMP_SALARY) > 55000)
```

The subquery returns a list of those departments where the department's average salary is greater than 55,000 and produces the following result set:

```
EMP_NO  EMP_DEPT_NO  EMP_FNAME  EMP_LNAME    EMP_SALARY
------  -----------  ---------  ---------    ----------
101     100          ANN        CALLAHAN     127500.00
105     100          EUNICE     BROWN         61426.08
113     100          MICHAEL    ANDERSON      72563.34
117     100          ROSE       DANIELS       67572.42
119     300          URSULA     SMITHSONIAN  104800.00
121     100          WINIFRED   VANCE         37712.10
123     300          ALAN       LAWRENCE      77568.70
125     100          CHARLES    GERRFON       53496.87
127     300          EUGENE     FITZ          58120.56
129     100          GEORGE     DE NORVA      21313.97
131     300          JOAN       WILLIAMS      74586.61
133     100          MAUREEN    PODANSKI      34591.90
135     300          PATRICIA   NORTON        61597.15
103     300          CAROL      SMITH         74748.78
107     300          GENEVIEVE  WESLEY        18331.50
111     300          KATHRYN    AMES          97600.00
115     300          OLIVERA    MAHARAMBA     35271.95
```

If I want to know which employee in each department has the highest salary, I need to use what is called a *correlated subquery*. A correlated subquery is one where a search condition in the inner query, specified in the **WHERE** clause, is correlated to the **WHERE** clause of the outer query.

Subqueries And Database Performance

A subquery does not significantly diminish performance because the *inner* query (the subquery) is performed only once. It returns its result set to the outer query, which then executes like any other query.

However, when you run a correlated subquery, the inner query runs once for each occurrence of the outer query. For example, in the next example where I search for which employee makes the highest salary in each department, the outer query runs first, returning every row in the **EMPLOYEE** table. As each row is returned by the database, the inner query runs. In this case, the database checks whether the employee returned in the outer row has a salary matching the row returned by the inner query. Imagine that your company has 10,000 employees; the inner query then runs 10,000 times. Correlated subqueries can have a devastating impact on performance, so you should use them only when absolutely necessary.

Often, you can rephrase the query to yield better performance. If you are forced into writing a correlated subquery, you might be able to rewrite it so that the outer query returns the most exclusive result set (the least number of rows).

Although I generally recommend moving as much of the processing to the database server as possible, correlated subqueries are often better handled partially within your Visual Basic program. For instance, you can run the inner query from your VB application and have the database return just four rows with department number and salary:

```
SELECT EMP_DEPT_NO, MAX(SALARY)
FROM EMPLOYEE
GROUP BY EMP_DEPT_NO
```

Then, your VB program would dynamically construct another query, passing the department numbers and salaries as part of the **WHERE** clause.

Assume you are running this correlated subquery in an organization with 10,000 employees and 50 departments. Further, assume that the inner query takes three seconds to run and that a **SELECT** of employees matching a specific department and salary condition takes one second to run. The correlated subquery might actually run for more than 10 hours before it completes the work. (Database-caching techniques and so on could actually improve this number tremendously, but expect to go to lunch, dinner, and then to the unemployment office.) Using the technique I suggest, the query would run in less than a minute (much less than a minute if the database is properly tuned).

The following query finds the person making the highest salary in each department:

```
SELECT EMP_DEPT_NO, EMP_FNAME, EMP_LNAME, EMP_SALARY
FROM EMPLOYEE EMP
WHERE EMP_SALARY IN
  (SELECT MAX(EMP_SALARY)
   FROM EMPLOYEE
   WHERE EMP_DEPT_NO = EMP.EMP_DEPT_NO
   GROUP BY EMP_DEPT_NO)
```

```
ORDER BY EMP_DEPT_NO
```

The output of the query looks like the following:

```
EMP_DEPT_NO  EMP_FNAME  EMP_LNAME    EMP_SALARY
-----------  ---------  ---------    ----------
100          ANN        CALLAHAN     127500.00
200          DAVID      MADISON       77492.47
300          URSULA     SMITHSONIAN  104800.00
400          PAUL       JAMESSON      73643.82
```

The **WHERE** clause in the outer query creates a *correlated variable*, **EMP**, allowing the match of **EMP_DEPT_NO** from the outer query to the same column from the inner query.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

You use the **EXISTS** keyword in subqueries to test for the existence of occurrences in an outer query. Suppose you want to find all departments that have one or more employees making less than $25,000 and show the number of those employees. You can run a query and test whether the results of the inner query exist in the outer query. The following example selects **EMP_DEPT_NO** from **EMPLOYEE** and then restricts the results to those rows where the department number is also returned by the inner query:

```
SELECT EMP_DEPT_NO
FROM EMPLOYEE EMP
WHERE EXISTS
  (SELECT EMP_DEPT_NO
   FROM EMPLOYEE
   WHERE EMP.EMP_NO = EMP_NO
   AND EMP_SALARY < 30000 )
ORDER BY EMP_DEPT_NO
```

The result is:

```
EMP_DEPT_NO
-----------
100
200
300
```

You have a variety of other ways to join the results of an inner query to an outer query, and you can nest the queries as deeply as necessary. Two keywords that you will find in most databases are **ANY** and **ALL**. The **ANY** keyword allows you to compare a column from the outer query to any of the values returned by the inner query. Assuming you want to know all the male employees who have a female manager, you can code the query as follows:

```
SELECT EMP_NO, EMP_FNAME, EMP_LNAME
FROM EMPLOYEE
WHERE EMP_GENDER = 'M'
AND EMP_MGR_ID = ANY
   (SELECT EMP_NO
    FROM EMPLOYEE
    WHERE EMP_GENDER = 'F')
ORDER BY EMP_NO
```

The results are:

```
EMP_NO  EMP_FNAME  EMP_LNAME
------  ---------  ---------
112     LARRY      JOHANSSEN
113     MICHAEL    ANDERSON
116     PAUL       JAMESSON
120     VINCENT    LINCOLN
123     ALAN       LAWRENCE
125     CHARLES    GERRFON
129     GEORGE     DE NORVA
132     KEVIN      KERRIGAN
```

TIP
A Few Words About COMMIT And ROLLBACK
Chapter 11 delves into the concepts of transactions, sometimes called logical units of work (LUW). Every transaction ends with either a **COMMIT** or a **ROLLBACK. COMMIT** makes permanent all changes to the database in the current transaction. **ROLLBACK** puts the database back to where it was. In other words, it undoes all the changes made during the current transaction.

In the following sections, I show you examples of commands that alter data. So that your results remain the same as mine, I highly urge you to create a backup of the database so that you can restore it to its original condition. Alternatively, you can issue a **ROLLBACK** command after each **INSERT, DELETE,** or **UPDATE** to undo all of the changes.

INSERT

It comes as no surprise that the **INSERT** statement is used to add new records to a table. The syntax of the command is:

```
INSERT INTO tablename [(column-list)]
VALUES (value1, value2, … valuen)
```

The ***tablename*** is an existing table on which the developer has insert authority. The ***column-list*** lists those columns into which data is to be inserted. If all columns are used, the column list is not necessary. The following statement adds a new row to the **DEPARTMENT** table. Because all columns are used, the column list is omitted:

```
INSERT INTO DEPARTMENT
```

```
VALUES (500, 'NE', 'New Department')
```

Notice that all character values must be enclosed in single quotes. If you use double quotes, SQL assumes you are referencing another column. If you are inserting a row where a column will be set to **NULL**, you must specify the **NULL** keyword:

```
INSERT INTO DEPARTMENT
VALUES (500, 'NE', NULL)
```

DELETE

The **DELETE** statement removes one or more rows from a table. The syntax is:

```
DELETE FROM tablename
[WHERE condition]
```

The ***tablename*** is any valid table. If you omit the **WHERE** clause, all rows are deleted. The following statement deletes all employees whose department is 100:

```
DELETE FROM EMPLOYEE
WHERE EMP_DEPT_NO = 100
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

UPDATE

The **UPDATE** command updates one or more rows in a single table and takes the following syntax:

```
UPDATE tablename
SET column_name= value
    [, column_name= value…]
[WHERE condition]
```

You can update any one or more columns, although most RDBMSs won't allow you to update the primary key columns. If you leave out the **WHERE** clause, all rows are updated. In the following code snippets, the first example gives all employees an 8 percent raise. The second command then gives all female employees in departments 200, 300, and 400 an additional 10 percent raise (obviously to make up for the inequities noted in the **GROUP BY** discussion earlier in this chapter):

```
UPDATE EMPLOYEE
SET EMP_SALARY = EMP_SALARY * 1.08

UPDATE EMPLOYEE
SET EMP_SALARY = EMP_SALARY * 1.1
WHERE EMP_GENDER = 'F'
AND EMP_DEPT_NO IN (200, 300, 400)
```

Where To Go From Here

In this chapter, I discussed SQL Data Manipulation Language, which is where the average developer will spend the vast majority of his or her time. Although the commands (**INSERT**, **DELETE**, and **UPDATE**) are simple enough, I

really only scratched the surface of what can be done with the **SELECT** statement.

A point of confusion for any developer is where to perform processing. For instance, you can do a simple **SELECT** from the database and then sort the result set at the client using Visual Basic. Alternatively, you can use the **ORDER BY** clause to do the sorting at the database level. Although there is no hard and fast rule, common sense usually dictates the correct answer. In general, it is better to let the database perform the work unless doing so would generate more network traffic. The most obvious example of that is letting the database format numbers (adding thousand and decimal separators and currency symbols). That is probably best left to Visual Basic to add when it displays the data. I discuss these issues in future chapters.

If you feel comfortable with SQL, particularly with the **SELECT** statement, then read on. If not, you might want to consult one of the many books on the subject. If you will be developing for more than one RDBMS, you might want to consider books that are not RDBMS-specific. Otherwise, you probably want to find a book that is specific to your database.

If you are developing and Microsoft Jet will be involved, you might want to consult Appendix B, where I discuss some of the vagaries of that dialect of SQL and how it differs from ANSI SQL. If you are using the ODBC API, you will also want to refer to Appendix C.

If you haven't yet read Chapter 2, you might want to do so. I discuss the other two portions of SQL (DCL and DDL) there. For advanced topics, sneak a peek at Chapter 11. Otherwise, read the next chapter (Chapter 4) where I overview data access with Visual Basic.

Finally, don't forget that the Internet provides a wealth of information. Specifically, each of the major database vendors has its own Web site, and a search of the knowledge base at Microsoft's Web site using the terms "database" and "SQL" yields many tricks and tips.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Chapter 4
Visual Basic 6 Data Access

Key Topics:

- Evolution of Data Access with Visual Basic
- Overview of flat (sequential) file processing
- Overview of DAO, RDO, and ADO
- Overview of VBSQL
- Overview of the ODBC API
- Considerations in choosing a data model

The nature of nearly all business (and most nonbusiness) programs is to access and manipulate data. In client/server development, this data is nearly (but not always) stored in a database. Visual Basic provides a dizzying array of options to access the database. The correct one to use is often difficult to ascertain. In this chapter, I discuss each of these access methods in turn and attempt to provide some guidance as to when to use each. I wish I could say that there is one right solution, but there isn't. The best general guidance I can offer is that you should base your decision on the current and anticipated skill levels of your staff as well as the current size and complexities of your application.

The good news in all of this is Microsoft's migration to *Active Data Objects* (ADO), which provides a one-stop interface to nearly all data sources, a high degree of compatibility with what you may have already developed using DAO or RDO, and performance metrics at least as robust as RDO.

Visual Basic Data Access Trends

Visual Basic 1.0 was released about the time Windows 3.0 came along. It was the much-ballyhooed successor to the DOS-based QuickBasic. There was some resentment at the time that the good folks at Microsoft had forsaken the DOS-based development community. Microsoft responded with the release of Visual Basic For DOS, which was essentially a bridge for those developers who, for whatever reason, were not yet ready to make the jump to Windows. It behaved much like VB for Windows with features such as event-driven programming and Windows-like controls, but the code itself still looked like the top-down Basic code of old. I believe that I might have been the only developer in the world to actually buy a copy.

QuickBasic, Microsoft Basic Professional, Visual Basic For DOS, and Visual Basic 1.0 For Windows all had one thing in common—no real method of doing database programming. The explosion in popularity of the Internet left Microsoft scrambling to catch up with the likes of Netscape. Plus, the success of Visual Basic as a development tool left Mr. Gates and company scrambling to meet developers' demands for industrial-strength enterprise development tools.

Visual Basic 3 was the first real attempt to give Visual Basic real-world database development tools when Microsoft added Data Access Objects (DAO) version 1.0. DAO was a beginning but was hardly a panacea for large-scale development efforts and, in fact, was really only well suited for small-scale projects aimed at ISAM files such as FoxPro or Paradox.

Version 4 of Visual Basic saw the introduction of Remote Data Objects (RDO), which removed much of the overhead involved in DAO, allowing direct access to ODBC rather than forcing everything through Microsoft Jet. Although laudable, RDO 1.0 was immature and placed a lot of restrictions on developers about what data sources they could connect to. At the same time, DAO saw some improvements in the underlying Jet engine that made it somewhat more practical to attach to ODBC data sources. VBSQL was added to allow native access to Microsoft SQL Server via DB-Lib (DB-Library).

It is worth noting that the Microsoft development team essentially rewrote Visual Basic from the ground up from versions 3 through 5 to support the concept of add-in components, objects, and so on. Although it looks much the same on the outside, on the inside, VB4 (and now VB5 and VB6) are very different on the inside. Many projects developed in VB3 needed a fair amount of recoding.

Version 5 of Visual Basic was really the first release where robust client/server development became a practical reality. DAO was refined with a new technology: ODBCDirect, which permitted the bypassing of the Jet engine, became a practical if not particularly scalable data model. Jet itself was further refined to increase performance and to offload some query processing on remote data sources that supported that processing. RDO 2.0 was a major enhancement over RDO 1.0, but careful testing of the Remote Data Control (RDC) revealed some serious flaws in its architecture. Although technically the ODBC API was always available, Microsoft talked more openly of using

the ODBC API as a means of database development. During VB5's development, Microsoft discussed the new Active Data Object model. However, the technology was not ready by the time VB5 was shipping (and, in fact, it will be evolving for some time to come), and most VB developers therefore really did not understand that using the ADO model was another option to development.

When the purchaser of Visual Basic 6 opens the box, he or she will again see something that looks basically familiar. However, the client/server developer will quickly lose that illusion when poking around under the covers. Wizards that once generated DAO- or RDO-based forms don't even offer those data models as an option. The days of ADO have arrived, and ADO 1.5 is a welcome, robust, and scalable data model upon which to develop. The ADO Data control, although not "backwards compatible" with the RDC or the DAO Data control, is similar enough that most VB developers will have little problem adjusting. Best of all, ADO combines the best of DAO (access to ISAM databases) and RDO (efficient access to ODBC data sources) with a high-performance, low-overhead OLE DB engine.

Although this book discusses all of the data models, the unmistakable trend is toward Active Data Objects. In the pages that follow, I discuss each of these data models with a brief overview and some examples. In the chapters that follow this one, each model is examined more in depth. For some developers, it will be worth the effort to convert existing projects to ADO, but for others, there will not be enough payback at this point in time to do so.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Previous | Table of Contents | Next

What's Behind Door Number One?

I teach client/server development using Visual Basic (as well as other tools) to students with a wide variety of backgrounds. Even the most experienced developers are confused by Visual Basic's approach to database development, and with good reason. VB really offered little in the way of database tools until version 3.0 when DAO and Microsoft Jet were added. Even the "simple" interface of DAO and its hierarchy of objects is enough to strike fear in the hearts of the object-purists among us. Still, once mastered, each of the approaches that I discuss offers amazing flexibility and, if approached methodically, can be conquered.

Table 4.1 lists the 11 flavors of data access with Visual Basic and my own subjective opinion of relative power, stability, learning curve, and ease of use. Each is scored on a scale of 1 to 10; in all cases, a higher score is better. By power, I judge the robustness of the end product in terms of processing a lot of data efficiently and flexibly. A score of 10 indicates that the method is very powerful. As to stability, I am referring to the likelihood of your program crashing in the event of a program bug. A score of 1 means that the method listed is not very stable. Learning curve refers to my opinion of how difficult the technology is to master. A score of 10 means that the technology is easy to learn. Finally, when I use the expression ease of use, I am referring to how much programming is required to use the method. A score of 10 indicates that the method is very easy to use.

Table 4.1 The varieties of data access methods with Visual Basic 6.0.

Method	Power	Stability	Learning	Ease Of Use
Flat File I/O	2	10	10	6
DAO/Jet Data controls	3	9	9	10

DAO/Jet*	4	7	5	6
DAO/ODBCDirect Data controls	5	8	8	9
DAO/ODBCDirect*	6	7	5	5
RDO Data controls	6	9	8	9
RDO*	7	8	5	6
VBSQL	9	7	5	4
ODBC API	10	4	1	1
ADO Data controls	9	10	8	9
ADO*	10	9	5	7

**Writing all code instead of using the Data control.*

Table 4.1 lists 11 separate methods of data access that all overlap at some point or other. Table 4.2 shows which types of databases each method can access. Flat file I/O (using VB's **Open** and **Get** keywords) is, of course, the roll-your-own approach and offers little to no realistic likelihood of accessing a true, relational database. Under some circumstances, flat file I/O may be appropriate for ISAM files. DAO/Jet (with or without the Data control) can process some sequential file formats but only to the extent that they are structured. DAO/Jet can also access most major relational databases. DAO/ODBCDirect (with or without the Data control) can access almost any relational database for which an ODBC driver is supplied and can access many ISAM formats as long as an ODBC driver has been defined. RDO is strictly bound to ODBC data sources, so it can connect to virtually all relational databases and many ISAM files. VBSQL is used with Microsoft SQL Server using DB-Lib. Older versions of Sybase SQL Server also support the DB-Lib interface and should, in theory at least, be accessible via VBSQL. The ODBC API method, like RDO, can access any ODBC data source. ADO is meant to cover the entire spectrum of data sources, including those that may be defined at some point in the future.

Table 4.2 The types of databases and which data access methods can access each.

Method	**Sequential**	**ISAM**	**Relational**
Flat File I/O	Yes	Limited	No
DAO/Jet Data controls	Limited	Yes	Yes
DAO/Jet*	Limited	Yes	Yes
DAO/ODBCDirect Data controls	No	Limited	Yes
DAO/ODBCDirect*	No	Limited	Yes
RDO Data controls	No	Limited	Yes
RDO*	No	Limited	Yes
VBSQL	No	No	DB-Lib only

ODBC API	No	Limited	Yes
ADO Data controls	Yes	Yes	Yes
ADO*	Yes	Yes	Yes

**Writing all code instead of using the Data control.*

In the following sections, I discuss each of the 11 database access methods.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Flat File I/O

The Visual Basic commands to process flat (nonrelational) files, more commonly called *sequential files*, have been around for many years, dating back to the days of QuickBasic (and in some cases, before that). Although using flat file I/O will not be your primary means of processing data, it does have a place even in modern client/server applications.

Consider the possibility that you are updating many thousands of records currently stored in a relational database. One of the down sides of RDBMSs is that there is a good deal of overhead involved in processing. I have worked on some applications where it was more efficient to “drop” a table into a sequential file (using an export utility of the database), update it the old-fashioned way, and then reload it to the database. I would never encourage this approach where not absolutely necessary because you lose the advantages of the database’s built-in integrity constraints and so on. Flat file I/O is also well suited when you need to process unstructured data. Perhaps you have received a file of records that you need to add to the database. It may well be advantageous to read the file sequentially and add the records programmatically.

Most Visual Basic developers who are reading this book are already fairly accomplished in the use of the VB language, so I have not devoted a separate chapter to flat file I/O. Instead, I outline the basics in this section. For additional guidance, refer to the Visual Basic documentation.

Opening And Closing Files

Visual Basic uses the **Open** command to open a file regardless of the mode in which the file is accessed. The syntax of the **Open** command is as follows:

```
Open filename For open_mode [Access access_mode] [lock_mode] _
   As [#]file_number [Len=record_length]
```

The ***filename*** includes a fully qualified path to the file. If the file name is omitted, Visual Basic attempts to open the file in the current directory.

The five ***open_mode*** options are **Append**, **Output**, **Input**, **Random**, and **Binary**. The first three options are used where a file is to be accessed sequentially (as opposed to randomly). In **Append** mode, any data that you write is added to the end of the file. Conversely, in **Output** mode, any data overwrites the file from the beginning. I discuss **Random** and **Binary** modes in a moment. The default is **Random**.

The ***access_mode*** determines whether the file can be read to or written from. The options are **Read**, **Write**, and **ReadWrite**. The default is **ReadWrite**.

The ***lock_mode*** option specifies how the file locking is handled in a multiuser environment. The options are **Shared**, **Lock Read**, **Lock Write**, and **Lock Read Write**. If you open a file in **Shared** mode, any other process has full access to the file. In **Lock Read** mode, the file is locked against others reading it. Likewise, in **Lock Write** mode, other processes may not write to the file. The most restrictive (and default) mode is **Lock Read Write**, which prevents any other process from opening the file while your program has it open. If you have opened a file as **Lock Write** and another program attempts to open it in **Output** mode (for example), a runtime error in that program will be generated.

You must assign an unused ***file_number*** when opening a file. All subsequent operations against the file reference the assigned file number. The number must be in the range of 1 to 511. Use numbers of 255 and below if the file will not be used by other processes. Use numbers above 255 if the file can be opened by other processes. Use the **FreeFile** function to return the next available file number:

```
' Return a file number from 1 to 255
Dim iFile1 As Integer
iFile1 = FreeFile

' Return a file number from 256 to 511
Dim iFile2 As Integer
iFile2 = FreeFile 1
```

The pound sign (#) before the file number is optional, and it is retained by Visual Basic for backward compatibility with Basic versions that did require it.

The ***record_length*** argument is used with files opened in **Random** mode and is ignored if the file is opened in **Binary** mode. With files opened randomly, all records in the file are normally a fixed length. This provides the ability to locate a record quickly by simply supplying a record number. For example, if all records in a file are 100 bytes long, the operating system can use that to locate record number 4 by moving the “file pointer” to byte 301 in the file (assuming the first record begins at byte 1, the second record at byte 101, and so on). Of course, your program is responsible for maintaining the logic necessary to know where in the file a given record is.

When a file is opened in **Binary** mode, you can move around the file by specifying an absolute byte offset from the beginning of the file or from the current location in the file.

Once you have finished using a file, use the **Close** statement followed by the file number to close it. If you do not supply a file number, all currently opened files are closed. When a file is closed, the file number is disassociated from that file and can be used again for another file. You should be sure to close all open files, especially those opened in **Output** or **Append** modes, because that forces the changes to be written to disk. Failure to properly close the file can result in your changes being lost.

Listing 4.1 opens three files (in **Append**, **Input**, and **Random** modes). Notice that a user type **EmpRec** is created and used as an argument to the **Len** clause on the **Random** file; the file will be used to store employee records.

Listing 4.1 Demonstration of several flat file access techniques.

```
' Variables to store file numbers
Dim iFile1 As Integer
Dim iFile2 As Integer
Dim iFile3 As Integer

' Open a file in Append mode
iFile1 = FreeFile
Open "c:\program.log" For Append Lock Read Write As iFile1

' Open a file in Input mode, read only
iFile2 = FreeFile
Open "c:\autoexec.log" For Input Read As iFile2

' Create a user-defined data type for the next file
Type EmpRec
   Emp_No As Integer
   Emp_Name As String * 30
   Salary As Currency
End Type

' Create a variable of type EmpRec
Dim strEmp As EmpRec

' Open a file randomly using a record length
iFile3 = FreeFile
Open "c:\employee.dat" For Random As iFile3 Len = Len(strEmp)

' Read the record number 8 and display
Get #iFile3, 8, strEmp
MsgBox strEmp.Emp_Name & "'s salary is " & _
   Format$(strEmp.Salary, "$###,##0.00"

' Close the first file
Close iFile1

' Close all other open files
Close
```

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Reading Files

Use the **Input #** function to read a file opened in **Binary** or **Input** modes using the syntax **Input #** ***file_number***, ***var1***, ***var2****…*. You use this function to read a comma-separated list of variables from a file:

```
' Read three variables from a file
Dim myVar1, myVar2, myVar3
Open "c:\myfile.dat" For Input As 1
Input #1, myVar1, myVar2, myVar3
Close 1
```

About File Handles

The operating system is actually responsible for all file I/O, including opening and closing the files. When a file is opened, it has a *handle*, which is a **Long** "address" to the file. The file number you assign (or obtain using **FreeFile**) is not the same as the file handle. In 16-bit Windows, you could use the **FileAttr** function to determine in what mode a file has been opened and to return the file handle: **Handle = FileAttr (1, 2)** where "1" is the file number that you assigned when you opened the file and "2" is an action code telling Visual Basic you want to return the operating system's file handle. Unfortunately, this option is not supported in 32-bit Windows.

You can, however, use **FileAttr** to obtain information about what mode a file was opened in. In this case, use an action code of 1: **Mode = FileAttr(1, 1)** where the first "1" is the file number you assigned when you opened the file and the second "1" is the action code. Visual Basic returns one of the following values:

- **Input**—1
- **Output**—2

- **Random**—4
- **Append**—8
- **Binary**—32

The **Input** function (without the pound sign), also used with files opened in **Binary** or **Input** modes, differs from the **Input #** function in that it reads in a fixed number of characters from the file, including commas, carriage returns, and line feeds, as shown in the following example:

```
Dim sBuffer As String
Open "c:\autoexec.bat" For Input As 1
' Read 100 characters
sBuffer = Input (100, 1)
```

Line Input is used to read an entire line of text, up to (but not including) the character-return and line-feed characters. The following code reads a file and displays each line in a ListBox control. The result is shown in Figure 4.1:

```
Dim sBuffer As String
' Open the file
Open "c:\autoexec.bat" For Input As 1
Do While Not EOF(1)
   Line Input #1, sBuffer
   ' List1 is an existing ListBox control
   List1.AddItem sBuffer
Loop
Close 1
```

Figure 4.1 Example of using **Line Input** to populate a ListBox control with the contents of a file.

Notice that **Line Input** requires the use of the pound sign in front of the file number. Consistency is not Microsoft’s middle name.

You use the **Get** function to read a file opened in **Random** or **Binary** mode. With **Random** mode, you supply a record number, whereas with **Binary** mode, you supply a byte offset. If the record number or byte offset is omitted, reading begins at the next record or byte in the file. The following statement reads a file that was previously opened randomly:

```
' Read the 55th record into the strEmpRec variable
Get #iFile3, 55, strEmpRec
```

The following statement reads the next 512 bytes into a variable from a file previously opened as **Binary**:

```
Dim sBuffer As String * 512
Get #iFile3, , sBuffer
```

Navigating Files

The current position in the file is said to be the *file pointer*. Therefore, if the current record is number 55 in the file, the file pointer is 55. The **Seek** statement moves the file pointer (that is, it sets the location in the file for the next read or write), whereas the **Seek** function returns the current file pointer:

```
' Move to the 30th record
Seek #iFile3, 30
' Read the next record
Get #iFile3, , strEmpRec
' Display the current record number (31)
MsgBox Seek (#iFile3)
```

The **Loc** is the counterpart to **Seek**, returning the location of the last read or write.

The counterpart to **Get** is **Put**, which writes a record (in **Random** mode) or the value of a variable (in **Binary** mode):

```
Put #iFile3,, strEmpRec
```

Writing To Files

Put has additional functionality that varies somewhat depending on whether the file was opened for **Binary** or **Random**. For instance, you can write the contents of an array to disk. See the Visual Basic help file for additional information.

The **Write #** statement writes data to a file in the same format that it is read with the **Input #** function. **Write #** places quotes around strings and commas after each variable. A carriage-return and line-feed sequence is written after each **Write #** operation. Dates are surrounded by pound signs ("**#December 25, 1998#**"). Boolean data is written as **#TRUE#** or **#FALSE#**.

Print # writes formatted data to a sequential file. Strings do not have quotes and variables are not separated by commas. However, a carriage-return and line-feed sequence is written after each **Print #** operation unless the last character is a semicolon or comma. **Print #** is useful for writing formatted reports to a disk file for printing later. See the Visual Basic help file for more details on some of the ways that you can use **Print #**.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Other File Support Functions

The **EOF** and **LOF** functions are very useful when you perform flat file access. **EOF** returns a Boolean, indicating whether the end of the file has been reached. **LOF** returns the length of an open file. (You can use **FileLen** to determine the length of an unopened file.) The example for **Line Input** earlier in this chapter (see Figure 4.1) used **EOF**. Listing 4.2 provides an example of using **LOF** to perform a buffered read of a file.

Listing 4.2 Using LOF and a method to process very large files with buffered reads.

```
Dim sBuffer As String
Dim lSize As Long
Dim lRemaining As Long
Dim iBufSize As String
' Open the file in binary mode
Open "c:\windows\modem.log" For Binary As 1
' Determine the length of the file
lSize = LOF(1)
' Initialize the value of lRemaining
lRemaining = lSize
' Loop through the file
   ' Pad the input buffer
   If lRemaining < 4096 Then
      ' If less than 4K characters remaining
      sBuffer = Space$(lRemaining)
   Else
      ' Otherwise, read a 4K chunk
      sBuffer = Space$(4096)
   End If
```

```
    ' Read the data
    Get #1,, sBuffer
    lRemaining = lRemaining - Len(sBuffer)
    If lRemaining < 1 Then Exit Do
Loop
```

The Visual Basic help file contains information on other functions that may be useful, such as **FileDateTime** (which returns the date and time when a file was last modified), **GetAttr** (which returns the attributes of a file such as **Hidden** or **System**), and **SetAttr** (which sets file attributes).

Data Access Objects

Data Access Objects (DAO) is the original model of the relational data access models for Visual Basic. The current version is DAO 4.0. In this section, I discuss the capabilities of DAO and its object models, and I review the DAO hierarchy. I go into much more depth on these subjects in Chapter 5.

Capabilities

With the exception of ADO, DAO is the most flexible of the data access methods available to the VB developer. It offers the ability to manipulate ISAM databases as well as relational databases, shielding the developer from many complexities and creating a layer where the developer can use a common code base to interact with multiple back ends. In other words, if the developer uses a reasonable amount of planning, he or she can use the same code to interact with both FoxPro and Oracle. The actual implementation of DAO is shown in Figure 4.2.

Figure 4.2 The relationship of DAO to the application and to the database.

The original implementation of DAO was with Microsoft Jet, which then interacts with the ODBC Driver Manager. The ODBC Driver Manager handles all low-level interactions with the database itself. Unfortunately, what Jet gains in flexibility (because it can interface to virtually any data source), it loses in performance. Jet is robust in its capabilities but is a “thick” layer between your applications and the data, slowing down all database access. Also, Jet leaves a large footprint in memory; it occupies more than a megabyte of RAM.

The seasoned (or should I say grizzled?) Visual Basic developer who wrote database applications with Visual Basic 3 used Jet version 1.1. This was a primitive implementation. Jet’s performance enhancements are derived in large part by allowing the back-end database to do what it does best: process queries. Jet 1.1 pulled all of the records from a table in a back-end database such as Oracle and then performed the query locally.

ODBCDirect was added to DAO with version 5 of Visual Basic. It is not as flexible as Jet in that it can only communicate with defined ODBC data sources. However, it bypasses Jet and communicates with RDO. It also offers some functionality improvements because it can access ODBC features not available when going through Jet. Because ODBCDirect is a thin layer leaving a minimal footprint in memory, it offers performance nearly as good as RDO itself. ODBCDirect essentially maps DAO functionality to RDO functionality. The RDO layer interacts with the ODBC Manager.

In Chapter 5, I discuss the DAO object hierarchy in depth.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Using DAO

DAO has a fairly complex object hierarchy. (See Figures 5.1 and 5.2 in Chapter 5 for a graphical representation of this hierarchy). In this model, all data access objects exist in the context of a single **DBEngine** object.

Are DAO And Jet Dead?

One might easily jump to the conclusion that Jet, and by extension, DAO, are both in the coffin, if not actually buried. Microsoft SQL Server 7 is making a bold attempt to cover all portions of the enterprise from clustered servers down to a desktop database. (You can purchase a Windows 95 version of SQL Server that supports up to five users, although it is primarily aimed at the desktop user.) If you are predicting the demise of Jet in particular based upon Microsoft's announcements regarding SQL Server 7, don't bet a lot of money. Although SQL Server 7 may, and probably will, eventually be a viable desktop platform, it is far from that today. Although Microsoft may have refined the product some by the time you read this, it appears that the memory requirements are about two to three times greater for SQL Server 7 than for Jet. SQL Server 7 for the desktop is about a 200MB install, and although Microsoft specifies a Pentium platform, my recommendation is to not run it with anything less than a 200MHz CPU with at least 64MB of RAM. This is not your typical desktop configuration.

Microsoft has continued to refine and improve Jet. Even more, informal testing shows that not only is Jet 4.0 faster than its 3.0 counterpart, but it is also much faster than SQL Server 7 on a typical desktop PC. That is not to say that a future release of SQL Server will not change that discrepancy, but that day has not arrived yet.

If anything is going to knock Jet off its desktop throne, it is ADO and OLE DB (which is part and parcel of SQL Server 7, by the way), which I discuss

later in this chapter. For new relational database development efforts, I see no compelling reason to use DAO over ADO because OLE DB seems to be a much more robust technology than Jet and especially because Microsoft is clearly going to concentrate its development bucks there.

For the typical desktop development effort, I see no reason not to use Jet against ISAM-type databases, including MS Access. If more robustness is needed (such as logging), then I recommend a product such as Personal Oracle or Sybase SQL Anywhere, both of which offer good performance with modest resource requirements. Otherwise, Jet is a proven data engine technology for file-oriented databases.

The **DBEngine** object contains the **Workspaces** collection, which consists of individual **Workspace** objects. Each **Workspace** object describes a current "session" with the database and includes the **Databases** collection. Each **Database** object includes several collections including **QueryDefs** and **RecordSets**. A **QueryDef** object describes an SQL select. Each **RecordSet** object is an active result set from the database. The actual implementation of the object hierarchy varies a little based on whether ODBCDirect is used. Because the objects themselves vary, there is also some variation in methods and events available to the VB developer. For instance, ODBCDirect offers the **Cancel** method to cancel a pending asynchronous query.

DAO gives the developer the opportunity to use data-aware controls, greatly simplifying the task of DAO-based development. This opportunity comes at the expense of some flexibility in design but can turn 200 lines of code into 20 or fewer. For instance, the Data control automatically connects to the database, builds a **RecordSet** object, handles the chore of scrolling through the **RecordSet**, and performs all updates behind the scenes. Without the Data control, the VB developer has to code all of this functionality. Likewise, by having a Data control, the user can bind other data-aware controls such as the TextBox and the ListBox to individual columns in the Data control's **RecordSet** object. Without the Data control, the developer has to manually populate each TextBox (or other control) as the user scrolls through a RecordSet.

Figure 4.3 shows a form module that uses the Data control and several TextBox controls to display records from the **Employee** table in the Access version of the sample database provided with this book.

Figure 4.3 The Employee application created using the Data control and very little coding.

TIP
Using The Sample Code
Because the application in Figure 4.3 uses an Access database without ODBC, the path to the database file is "hard-coded" in the **Database** name of the Data control. In addition, because the database is on your CD-ROM, the database cannot be updated. To run this code on your own, copy the project file and the form file to your hard drive along with the Access database.

Open the project and change the **Database** property of the Data control to reflect the new path.

I actually wrote only one line of code (shown below) to display the record number. The rest of the code was all generated as a result of drawing controls on the form and setting their properties as needed.

```
Private Sub Data1_Reposition()

Data1.Caption = "Record " & _
   Str(Data1.Recordset.AbsolutePosition + 1) & _
  " of " & Str(Data1.Recordset.RecordCount)

End Sub
```

In Chapter 5, I expand on the use of DAO, adding functionality such as error handling, data validation, and so on using the sample data provided with this book (see Appendix A). I will also walk through the process of coding a DAO-based application without using the Data control, as a means of illustrating the DAO object hierarchy and to provide additional flexibility to the application.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Remote Data Objects

Introduced with Visual Basic version 4.0, Remote Data Objects (RDO) introduced a major performance boost for client/server applications. You can use RDO with any backend database defined as an ODBC data source name (DSN) that supports the **SQLNumParams** ODBC function (see sidebar). Although some of the names of properties and objects have changed, RDO has a lot in common with DAO. Most VB developers should have little problem converting existing code to use RDO and even less problem grasping its usage. Where RDO will work (when working with any ODBC-defined database), it makes little sense to continue using DAO. However, before converting an application from DAO to RDO, you might want to carefully consider the benefits of converting to ADO instead. I discuss ADO later in this chapter.

RDO Capabilities

RDO puts a smaller layer between your application and the data than does DAO. This is shown in Figure 4.4. RDO basically places a thin wrapper around the ODBC API, allowing the developer most of the benefits of the ODBC API with few drawbacks. The chief advantages of RDO are:

- No need for a local query processor (Jet); query processing is done remotely.
- Smaller memory footprint.
- Event-driven asynchronous queries.

Figure 4.4 The Remote Data Objects model.

The Visual Basic 5.0 implementation of RDO (RDO 2.0) and the Remote Data control (RDC) was a significant improvement over the original Visual Basic 4.0 implementation (RDO 1.0). For example, under Visual Basic 4.0, you had to continuously poll for the completion of a database operation. RDO 6.0, introduced with Visual Basic 6 (don't ask me what happened to versions 3, 4, and 5), is another improvement. Unlike DAO, in which data access is usually performed synchronously (meaning that the program has to wait for the result set to be returned before processing can continue), RDO permits asynchronous queries of the database. For example, if you set the RDC's **Options** property to **rdAsyncEnable**, the ResultSet is populated as a background task. When the query is complete (and the ResultSet completely populated), the **QueryCompleted** event is fired, providing an event-driven means for your application to know that the query has completed.

The RDC's functionality is similar to the Data control, which most VB developers have used. Likewise, the Remote Data Object hierarchy is similar (though simplified) to the Data Access Object hierarchy.

Using RDO

Figure 4.5 shows an application created using the RDC that is almost identical to the DAO example in Figure 4.3. The snippet places the current row number into the **Caption** property of the RDC. The properties of the RDC are almost identical to those in the Data control. A few differences mostly reflect the SQL row orientation of the Remote Data control versus the file record orientation of the Data control. For instance, DAO's **RecordSet** property is equivalent to RDO's **ResultSet** property. DAO's **RecordCount** property is equivalent to RDO's **RowCount** property as shown in the following code snippet:

```
Private Sub MSRDC1_Reposition()

MSRDC1.Caption = "Record " & _
   Str(MSRDC1.Resultset.AbsolutePosition) & _
  " of " & Str(MSRDC1.Resultset.RowCount)
```

```
End Sub
```

Figure 4.5 The Employee application coded using the Remote Data control.

RDO And ODBC Compliance

The ODBC (Open Database Connectivity) standard is broken into three levels. *Base level* or *Core level* defines a minimal set of functionality that an ODBC driver must support. *Level 1* defines additional functionality, whereas *Level 2* defines yet another layer of more advanced functionality. A Level 2-compliant ODBC driver essentially supports the full set of SQL functionality defined by ANSI SQL.

Appendix C lists the different functions required by each level along with the syntax for calling each from Visual Basic. RDO in Visual Basic 4 required a Level 2-compliant implementation, whereas RDO in VB5 and now VB6 has relaxed this requirement considerably to require only the **SQLNumParams** function.

VBSQL

VBSQL is a library for accessing Microsoft SQL Server via DB-Lib. Because Microsoft is heading away from DB-Lib and more toward an ODBC- and OLE DB-oriented access method to SQL Server, the use of VBSQL is not recommended except for existing projects already utilizing it.

VBSQL Capabilities

VBSQL is written to provide an interface to Microsoft SQL Server only. Specifically, any functionality exposed by SQL Server's DB-Lib is implemented by VBSQL. Connections tend to be cursor-oriented, and you are pretty much on your own to manually populate controls with information retrieved from the database (as opposed to binding controls to a data source). On the other hand, you have an unusual degree of control (relative to other Visual Basic data models) over the efficiency of your database processing. For instance, you can set the packet size for communicating between the database server and your application using the **SQLSetPacket** function with the **SQLOpen** function. (Note that you can determine the current packet size using **SQLGetPacket** but that once a connection is established, the packet size cannot be changed.) The packet size can be any size up to 64K and, if not specified, is determined by the current SQL Server default. If the default is set to 4,092 bytes but your application is mostly updating and retrieving single records, the large packet size slows down your application. On the other hand, if your application is processing a lot of data, a larger packet size is preferable to reduce the number of disk reads.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Using VBSQL

To use VBSQL, you need to visit the Microsoft Web site and download the VBSQL ActiveX control (look for VBSQL.OCX). The version I used in this book was a prerelease version, but a shipping version should be available by the time you read this.

VBSQL requires the use of the VBSQL control, which is really more like a DLL than a Visual Basic control. You need to add to your project the VBSQL.BAS module, which is also available from the Microsoft Web site and included on the CD-ROM (be sure to check the Web site for a more recent release). The VBSQL.BAS module has a number of function calls defined, such as establishing a connection with the database. The declarations all reference the OCX file as shown in this code snippet:

```
Declare Function SqlNextRow Lib "VBSQL.OCX" _
   (ByVal SqlConn As Long) As Long
```

Figure 4.6 shows a VBSQL application running in front of the VB6 development environment. A quick look at the two open code windows reveals that this coding, although not terrifically difficult, is tedious. Because Microsoft is in the process of eliminating the DB-Lib interface, and because the Active Data Object model is available, you have little reason to use VBSQL code. Because of the length of the code, and the fact that I don't recommend VBSQL when clearly better data access models are available, the entire 20 or so pages of code is not printed in the book.

Figure 4.6 The Employee application using VBSQL.

To open a connection to the database, a sample code snippet might look like the

following:

```
Dim lSQLConn As Long
Dim lRtn As Long
Dim sRtn As String
' Share with other procedures
Private myLogIn As Long
' Initialize
sRtn = SQLInit ()
' Establish login record
myLogIn = SQLLogIn ()
' Set login parameters
lRtn = SQLSetLUser (myLogIn, "Coriolis")
lRtn = SQLSetLPwd (myLogIn, "Coriolis")
' Workstation
lRtn = SQLSetLHost (myLogIn, "W243A")
' Can add other parameters
lRtn = SQLSetLApp (myLogIn, "Coriolis VBSQL Sample App")
' Establish connection
lSQLConn = SQLOpen (myLogIn, "Home")
```

The ODBC API

Perhaps the ugliest approach to data access using Visual Basic, at least in terms of complexity of code, is via the ODBC API. It is not for the faint of heart. At one time, particularly before the release of Visual Basic 5 (with its credible RDO version 2), the developer seeking to expose the robustness of the underlying ODBC had little choice but to drop down to the ODBC API. As with VBSQL, I see little reason to do so now, especially with ADO obviating more complex data models. Even still, just as the Visual Basic developer still finds legitimate need to drop to the Windows API for specialized functions, the VB client/server developer may well need to drop to the ODBC API at some point.

ODBC API Capabilities

The capabilities provided by the ODBC API are a function of both the backend database and the driver with which the ODBC manager communicates. For instance, I have used the Microsoft ODBC driver to access Oracle databases. I have also used Oracle's own driver. As odd as it may sound, Microsoft's is more functional under many circumstances. For instance, in a DAO environment, if you set the Data control's **Options** property to **vbSQLPassThru** (in order to tell Jet to let the back-end database handle the query), the Oracle ODBC driver can only process the tables as read-only.

Appendix C documents some of the technical details of the API.

Using The ODBC API

Listing 4.3 provides a brief look at some of the steps involved in connecting to and manipulating an ODBC data source. In this case, I have coded the steps necessary to list the available ODBC data source names (DSNs). The application is shown in Figure 4.7.

Listing 4.3 Code from frmDSNList.

```
Private Sub cmdClose_Click()

End

End Sub

Private Sub cmdGetSources_Click()

Dim iRtn As Integer, iDSNLen As Integer, iDescLen As Integer
Dim hEnv As Long
Dim sDSN As String * 32, sDesc As String * 128

' Clear the list box
List1.Clear

' Allocate env handle
iRtn = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, _
   hEnv)

' Set environmental variables
iRtn = SQLSetEnvAttr(hEnv, SQL_ATTR_ODBC_VERSION, _
   SQL_OV_ODBC3, SQL_IS_INTEGER)

' Get first data source
iRtn = SQLDataSources(hEnv, SQL_FETCH_FIRST, sDSN, _
   Len(sDSN), iDSNLen, sDesc, Len(sDesc), iDescLen)

Do While iRtn = SQL_SUCCESS
   ' Add DSN to listbox
   List1.AddItem Left$(sDSN, iDSNLen)
   ' See if there are any more
   iRtn = SQLDataSources(hEnv, SQL_FETCH_NEXT, sDSN, _
      Len(sDSN), iDSNLen, sDesc, Len(sDesc), iDescLen)
Loop

' Free the handle
iRtn = SQLFreeHandle(SQL_HANDLE_ENV, hEnv)

' Report the results
Text1.Text = Str$(List1.ListCount)

End Sub
```

Figure 4.7 Listing the ODBC data sources using the ODBC API.

The project, included on the CD-ROM, consists of a general code module (ODBC.BAS) containing API declarations and constants and a form module, **frmDSNList**.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Active Data Objects—The Future Is Now

One giant leap for Microsoft is an even bigger leap for us mere mortals who don't happen to own a lot of Microsoft stock. For those of us who have careers based at least in part on the use of Visual Basic to develop client/server applications (indeed, *any* database applications), Active Data Objects and its concomitant technologies (such as OLE DB) are a huge leap forward. Not only will most applications show anything from a modest to a drastic performance improvement, but also the implementation of the data model is simpler than either DAO or RDO. Perhaps even better, Microsoft has adopted a common data model to be used across all its development products, making objects as applicable in Visual C++ as they are in Visual Basic.

ADO Capabilities

ADO is actually an interface to OLE DB, which provides a common access point for both relational and nonrelational data structures as well as such disparate and nonstructured data sources as text, graphics, and email. In fact, with OLE DB, you can relate two entirely separate data sources to each other as long as there is an OLE DB driver written for each data source. For example, you could read an **Employee** table in an Oracle, Access, or DB2 database. You could then join the **Emp_Email** column in that table to the **Pop3_Account_Name** "column" in a Microsoft Outlook data source (when and if an OLE DB driver is written to access Outlook). Whereas ADO provides a high-level interface to the data, OLE DB is a low-level interface. By and large, VB developers can ignore the dirty business of bits and bytes in which OLE DB deals and concentrate on the streamlined interface provided by ADO. ADO encapsulates OLE DB functionality (and, by extension, data) and exposes it as objects.

When dealing with OLE DB data sources, OLE DB is considered the *data provider*, whereas your application (and the Active Data Objects) is considered

the *data consumer*.

Using ADO And OLE DB

With ADO and OLE DB come new ways to look at and process data. Data-aware controls such as the TextBox have a new property, **DataMember**. You can still treat data sources in the "old way": Assign to the **DataSource** property the name of an existing Data control (or Remote Data control or Active Data control) and assign to the **DataField** property a column that corresponds to one of the columns returned by the Data control. This is perfectly acceptable.

However, the new data environment allows you to encapsulate the logic in a relational database's structure. Consider the sample database included with this book. An **Orders** table is related to the **LineItem** table. In turn, the **Item** table is related to the **LineItem** table. The relationship is shown graphically in Figure 4.8.

Figure 4.8 The relationship between the **Orders**, **LineItem**, and **Item** tables.

Now, consider a selection from the **Orders** table along with all related rows in the **LineItem** and **Item** tables. This *data hierarchy* can be modeled using *hierarchical cursors* in the new Data Environment Designer. You can build a **DataEnvironment** object that encapsulates this data hierarchy. Then, you can assign to a data-aware control's **DataSource** property a **DataEnvironment** object instead of assigning a Data control to it. When you assign a **DataEnvironment** object to a data-aware control, the **DataEnvironment** object has certain properties that are exposed as *data members*.

In Figure 4.9, I am creating a **DataEnvironment** object using the Data Environment Designer. After creating a **DataEnvironment** object, you can insert stored procedures previously created on the database, or you can add new **Connection** objects or **Command** objects. A **Connection** object represents a connection to the database, including various parameters such as user ID, password, and timeout. A **Command** object can be a table similar to a DAO **RecordSource** type, an existing SQL view or stored procedure, or an SQL statement. In Figure 4.9, I am creating an SQL statement type **Command**.

Figure 4.9 Creating the **deADO-Example DataEnvironment** object using the Data Environment Designer.

When placing a data-aware control on a form, you can set its **DataSource** property to any valid **DataEnvironment** object. Because the **DataEnvironment** object is a project-level object (as opposed to a form-level object, such as a Data control), any control on any form may access it. Once you have selected a **DataEnvironment** object as a control's **DataSource** property, you can use the control's **DataMember** to select from any of the **Command** objects of the **DataEnvironment** object. In Figure 4.10, I have just selected my **deADOExample DataEnvironment** object as the **DataSource** for the DataGrid control. I then selected the **Ord_Rpt Command** object as the **DataMember** property.

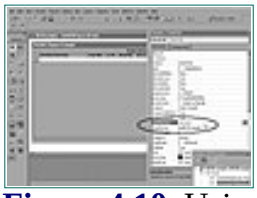

Figure 4.10 Using a **DataEnvironment** object as a **DataSource** of a data-bound control.

Of course, none of this is to say that you can't use the ADO Active Data control (ADC) as you would the DAO Data control or the RDO Remote Data control. In Figure 4.11, I do exactly that. The figure shows two forms open inside an MDIForm. The top form shows the form that was designed in Figure 4.10, displaying rows of data from the three tables represented in Figure 4.8. The bottom form shows the same Employee Maintenance form that I created using DAO and RDO earlier in this chapter.

Figure 4.11 An MDIForm housing two other forms using a DataGrid control bound to a **DataEnvironment** object and some TextBox controls bound to an ADO Data control.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The code for this application is included on the CD-ROM. To build it yourself, first create an MDI form named **mdiADOExamples**. Add two forms named **frmADOEmployee** and **frmADOReport**. Set both of their **MDIChild** properties to True. On the MDI form, add a File menu with three items: Report, Employee, and Exit. For the **mnuFile-Report_Click** event, code the following:

```
frmADOReport.Show
```

In the **mnuFileEmployee_Click** event, code the following:

```
frmADOEmployee.Show
```

On **frmADOEmployee**, add an ActiveData control. To do so, select Microsoft Active Data Objects from the Projects|Components menu to put the control on the toolbox. Set its properties as follows: **Align=2 'Align Bottom**; **Connect="DSN=Coriolis VB Example"**; **DataSource Name="Coriolis VB Example"**; **RecordSource="employee"**.

Next, create an array of five textbox controls named **txtEmpDat**. The **DataSource** property for each should be **Adodoc1**. Set their **DataField** properties to: **emp_no**; **emp_fname**; **emp_lname**; **emp_hire_date**; and **emp_salary**. You can set the **DataFormat** properties for the date and salary fields as you see fit.

Add label controls as shown in Figure 4.11 and then add the following code:

```
Private Sub Adodoc1_MoveComplete (ByVal adReason As _
  EventReasonEnum, ByVal pError As Error, adStatus _
  As EventStatusEnum, ByVal pRecordSet As RecordSet)

Adodc1.Caption = "Record " & _
  Str$(Adodc1.RecordSet.AbsolutePosition) & " of " & _
```

```
  Str$(Adodc1.RecordSet.RecordCount)
End Sub
```

On **frmADOReport**, add a DataGrid control named **DataGrid1**. Set the DataGrid control's properties as follows: **AllowArrows=True**; **AllowSizing=True**; **Caption="Order Details"**; **DataSource="deADOExample"**; **DataMember="Ord_Rpt"**; **WrapText=True**. The **DataField** property for the columns should be set to: **ord_no**; **ord_date**; **ord_cust_no**; **line_no**; **line_item_no**; **item_desc**; **line_qty**; **line_price**; **line_total**. The **Caption** and **Format** properties should be set appropriately, such as those shown in Figure 4.11.

deADOExample has one **Connection** named **CorVBExample**. Its **SourceOfData** property should be set to **"3 - deUseOLEDBConnect-String"**. The **ConnectionSource** property should be set to **" Provider=MSDASQL.1; Password=coriolis; User ID=coriolis; DataSource=Coriolis VB Example"**.

The **Connection** object has two **Command** objects named **Orders_Only** and **Ord_Rpt**.

For **Orders_Only**, set the **CommandType** property to **"2 - AdCmdTable"**. Set other properties as follows: **CommandText="Coriolis.Orders"**; **CursorLocation="3 - adUseClient"**; **CursorType="3 - adUseStatic"**; and **LockType="1 - adLockReadOnly"**.

For the **Ord_Rpt Command** object, set properties as follows: **CommandType="1 - adCmdText"**; **CommandText="SELECT orders.ord_no, orders.ord_date, orders.ord_cust_no, line_item.line_no, line_item.line_item_no, item.item_desc, line_item.line_qty, line_item.line_price, line_item.line_total FROM item, line_item, orders WHERE item.item_no=line_item.line_item_no AND line_item.line_ord_no=orders.ord_no ORDER BY orders.ord_no, line_item.line_no"; CursorLocation="3 - adUseClient"; Cursor-Type= "3 - adUseStatic"; and LockType="1 - adLockReadOnly"**.

In the CD-ROM code listing, you will see other **Command** objects as well, though they are not used in the sample application.

Each **Command** object exists in the context of a **Connection** object. The **Connection** object must first exist, and it must be specified as the "parent" of the **Command** object. This brings us to the simplified (and non-hierarchical) object model of ADO relative to DAO and RDO, as shown in Figure 4.12. (Note that the figure itself is somewhat simplified. For instance, **Errors** is actually a collection of Error objects.)

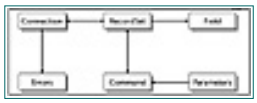

Figure 4.12 The ADO data model.

One of the most impressive and useful aspects of ADO and VB6 is that you can create a **DataEnvironment** object and use it on multiple forms (indeed, in any module). Contrast that to the Data control or the Remote Data control, which must be re-created for each form that will use them. This change is a leap

forward in the continued evolution of Visual Basic into an object-oriented development tool.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Data Access As An Object—Introducing Data-Aware Classes

To use a Data control in your application, you paste the control onto a form and define properties that enable the control to communicate with the database. When the application is running, the Data control is essentially an instance of a class—in other words, an object.

In Chapter 5, I discuss creating data access objects and some of the theory behind it. For now, understand that a class is a definition of an object. It is *not* an object until instantiated. When you design a form in the development environment, you attach controls, set properties, and add event procedures and general procedures. (An event procedure is a predefined procedure that reacts to an event such as **Command1_Click ()**. A general procedure is one that you create yourself as either a **Sub** or a **Function**.) All the properties, code, and so on of that form represent the *class*. They are the *definition* of the form. The form does not become an object until it is loaded into memory. The process of loading it into memory is called *instantiation*—literally, creating an instance of.

All objects have *data*, *state*, and *behavior*. An object's data may include information from a database, but it also includes the simple variables that you use within your procedures. An object's state is really those things that describe the object. For example, a form has a **Caption**. An object's behavior describes what the object can do. A form can make itself visible by invoking its **Show** method.

When a form includes a Data control, the Data control is part of the Form object's definition. It comprises a part of the class. Assume that you have developed a form named **frmCustomer**, which consists of a Data control, some TextBox controls bound to the Data control, and perhaps a menu and some buttons. If you, the reader, are an object purist, you might want to

suspend belief as you read the next statement: In a sense, **frmCustomer** is the result of a loose form of multiple inheritance.

Under inheritance, we can create a class (which, when instantiated, becomes an object) by inheriting the attributes of another class and then adding (or customizing) to create an entirely new class. For instance, in the real world, we might say that cat and person are both inherited from the class called mammal. The mammal class has certain attributes such as “has fur” and “bears its young live.” The classes cat and person both inherit those attributes, but cat has additional attributes such as “nocturnal,” “walks on four legs,” and “scratches my furniture.” If an attribute is inappropriate for a new class, it is overridden. For instance, for the class “platypus,” the behavior “bears it young live” is overridden to “lays eggs.”

Unfortunately, Visual Basic does not truly support inheritance. You cannot create a Form object and then inherit its properties to create a new Form object. Each Form object is created from scratch.

Pure object-oriented programming (OOP) languages such as Smalltalk and C++ support an extension to inheritance called *multiple inheritance*. With multiple inheritance, you can inherit from more than one object to create an entirely new object. A real-world example of that might be a motorcycle class. We inherit from the “engine” class and the “bicycle” class to create a new class, motorcycle, which uses attributes of both of the other classes.

If you stop and consider that the Data control and the TextBox control are both classes, we can combine them with the Form class to create an entirely new class, perhaps **frmCustomer**. Unfortunately, this falls apart when you seek to reuse the customized behaviors of the Data control or indeed the **frmCustomer** class. You can certainly copy the form, save it under a new name, and then modify it for specific functionality (perhaps a new Form object that will also display address information). However, if you alter the attributes of the original **frmCustomer** class, the changes are not reflected in the new form. Under inheritance, any changes to an *ancestor* class are propagated to its descendants. In other words, I should be able to change the background color of **frmCustomer** and then see that change reflected in all forms that inherit from **frmCustomer**. VB does not do that.

The purpose of inheritance is ultimately something called *reuse*. I should be able to create a class, such as a form, and then take all that work and reuse it on other forms. The same is true for any objects that I create to interact with the database or to enforce data validation rules. For instance, if I have a rule that says an employee’s date of hire must be greater than his or her date of birth, I should not have to code that in every place that accesses employee records. I should be able to create an object that encapsulates that rule and reuse that object wherever I manipulate employee data. Likewise, if I create a Data control that connects to the database and retrieves employee records, I should not have to re-create that Data control on every form object that handles employee data.

The closest that Visual Basic comes to inheritance is a form of reuse known as *delegation*. Visual Basic 4 introduced class modules. Class modules are

similar to general modules in that you can write procedures that can then be accessed from anywhere in a VB project. However, the class module is more like an object in that you can define properties, methods, and events. Whereas the interface of a Data control is those properties and methods that another object can access, the interface of a class is those properties and methods (**Sub**s and **Function**s) that you define as **Public**.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

You can also combine existing classes into a new class. The new class can then take advantage of the interface of the other classes. Although this is a simplified explanation, it is essentially what is meant by the term delegation. Ironically, because the new class can combine the attributes of several classes, Visual Basic actually does a better job of multiple inheritance than it does with simple inheritance.

The inability to reuse objects in Visual Basic is partly solved with class modules, but this did not (and still does not) provide a total solution. When considering client/server development and its data-centric philosophy, the problem is especially acute with data handling. Sure, you can manually code database access from within a class module, but you lose the simplicity of the Data control and you also lose the ability to bind controls to a data source.

Visual Basic 6 answers that problem with data-aware classes. When you add a class module to your Visual Basic 6 project, it now has three “built-in” properties instead of one. Besides the **Name** property, it now has the **DataBindingBehavior** and **DataSourceBehavior** properties.

Further, Visual Basic now includes the **BindingCollection** object, which can be used to bind classes as data sources to data-aware controls. You can even bind data-aware classes to each other.

This newfound capability of the class module goes hand-in-hand with ADO and OLE DB. A principal of ADO and OLE DB is that an object can be a *data source* or a *data consumer*. This is true for data-aware class modules as well. (Note that although the theory of data-aware class behavior is based on the premise of OLE DB, the class is not restricted to using ADO as a data model.)

A class that is a data source interacts with the database and provides data for other data consumer objects. It is analogous to having a Data control operating independently of any form. By setting the class module’s **DataSourceBehavior** property to **vbDataSource**, the class can act as a data

source to other objects, including other classes.

A class can also be a data consumer. By setting the **DataBinding-Behavior** to **vbSimpleBound**, the class is bound to a single column or field from your external database. If you set the property to **vbBound-Complex**, the class is bound to an entire row or record from the external data source.

The **BindingCollection** object consists of bindings between a data source and one or more consumers of that data. To access the **Binding-Collection** object, you add a reference to it as you would with other ActiveX libraries (on the Project menu, select References), as shown in Figure 4.13.

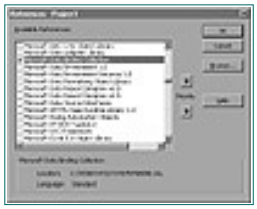

Figure 4.13 To use the **BindingCollection** object, first add it as a reference.

As shown in Figure 4.14, the **BindingCollection** object has the usual property (**Count**) and methods (**Add**, **Clear**, and **Remove**) of collections as well as several that are specific to data manipulation (such as **DataSource** and **DataMember**).

Figure 4.14 You use the Object Browser to explore the various attributes of the new **BindingCollection** object.

Using data-aware classes allows the encapsulation of data manipulation and business logic into a reusable object that can not only be accessed from any Form object, but also can be shared from project to project. The instantiated object can even be deployed at an application server, fulfilling the goal of DCOM as I began discussing in Chapter 1. In Chapter 10, I expand on these concepts to create practical, reusable data and business components to drive the enterprise-level client/server development effort.

Other Approaches To Database Development With Visual Basic

From a strictly technical point of view, the term client/server refers to an application involving two or more discrete processes acting in cooperation. I discuss this in Chapter 1. However, as a practical matter, the term has come to imply a number of other approaches to development to include the graphical user interface (which therefore also implies event-driven development) and rapid application development techniques.

Some of the most welcome changes in VB, even though they may not add a lot of additional functionality, have been the new wizards and templates.

Templates are prebuilt, fill-in-the-blank objects (such as the About form) that you can modify to your purpose rather than create from scratch. The wizards walk you through a series of dialogs to create an object or an entire application. The resulting object or application is not complete—you still need to customize it (such as adding data validation edits), but the wizard eliminates a lot of the repetitive drudgery. Further, because the wizard creates the same "shell" over and over again, you gain a lot in terms of the reliability of the code and the consistency of the user interface. Although the templates and wizards will not fulfill all of your needs, where they do help, I highly urge their use.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The Visual Basic Application Wizard

When you create a new project in Visual Basic, you are prompted with a choice of a number of templates and wizards, as shown in Figure 4.15. Of most interest are the VB Application Wizard and the Data Project. (In Chapters 13 and 14, we delve into IIS and DHTML applications.)

Figure 4.15 Visual Basic will help automate the creation of a variety of different project types.

The Application Wizard generates a remarkably complete application with a simple fill-in-the-blanks approach. Unlike the wizard bundled with VB5, the wizard in VB6 offers a lot of flexibility in how you complete the data access portion. Whereas VB5 restricted you to the Data or Remote Data controls, VB6 offers the option for a Data control approach, an all-code approach, or a data-aware control approach. VB6 wizards also offer more presentation styles than were offered in VB5, as shown in Figure 4.16.

Figure 4.16 Configuring the Application Wizard on how to access the database and how to present the resulting data.

Figure 4.17 shows an application generated by the VB Application Wizard. The chart reflects average salaries by department and gender, whereas the other form is a

master/detail presentation of employees by department. The form with the chart was not built correctly by the Application Wizard (in fact, it did not even place a chart on the form), but that problem should be rectified by the time you receive your copy of VB6.

Figure 4.17 An application created by the VB Application Wizard.

Admittedly, the data entry forms are not attractive; it us up to you to move the controls on the forms to suit your needs. The chart form is probably even less attractive, a reflection of my being “artistically challenged.”

An examination of the code generated will show that VB adds comments where you should add your own customized code. Listing 4.4 shows where VB inserted comments prompting me to add validation code. Because the class is the data provider, it provides public methods to move through displayed records and to edit those records. These actions (and others) call the private **adoPrimaryRS_WillChange-Record** procedure. A look at this reveals something very similar to the validation code placeholder that VB5 added in its Data and Remote Data control wizards. Because any form (or other module) can create an instance of the class, the validation code you write here can be used over and over. That’s encapsulation and reuse.

Listing 4.4 Code generated by the Wizard.

```
Private Sub adoPrimaryRS_WillChangeRecord _
  (ByVal adReason As ADODB.EventReasonEnum, _
   ByVal cRecords As Long, adStatus As _
   ADODB.EventStatusEnum,
   ByVal pRecordset As ADODB.Recordset)

  'This is where you put validation code
  'This event gets called when the following actions occur
  Dim bCancel As Boolean
  Select Case adReason
    Case adRsnAddNew
    Case adRsnClose
    Case adRsnDelete
    Case adRsnFirstChange
    Case adRsnMove
    Case adRsnRequery
    Case adRsnResynch
    Case adRsnUndoAddNew
    Case adRsnUndoDelete
    Case adRsnUndoUpdate
    Case adRsnUpdate
  End Select
```

```
  If bCancel Then adStatus = adStatusCancel
End Sub
```

The Visual Basic Data Form Wizard

I, for one, see no reason to craft an application from scratch when the VB Application Wizard will do so much of the work for you. On the other hand, the Application Wizard is not that much help when you are adding functionality to an existing project. This is where the Data Form Wizard can be a big help. The Data Form Wizard runs from the Add-Ins menu and offers essentially the same functionality on a form-by-form basis as the Application Wizard. In Figure 4.18, I used the Data Form Wizard to create a customer maintenance form. I then spent a few minutes customizing it to make it more usable and then added a line to the MDIForm's menu to open it. Time invested? Ten minutes, tops.

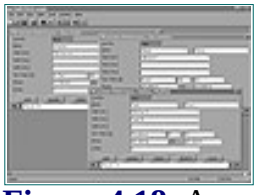

Figure 4.18 A customized customer maintenance form originally generated by the Data Form Wizard.

The Visual Data Manager

Visual Basic provides the Visual Data Manager utility to help you with many of the more mundane tasks involved in building a database-centric client/server application. From the Add-Ins menu, I invoked the Visual Data Manager and then opened the Coriolis VB Example data source. In Figure 4.19, you can see that the Visual Data Manager opens a window with all of the tables in the database. I expanded the four tables that I was interested in: **Customer**, **Orders**, **LineItem**, and **Item**. I then used the Query Builder utility (from the Utility menu) and generated the query shown in the SQL Statement window. From there, you can ask VB to develop a data form, run the query to test it, and so forth.

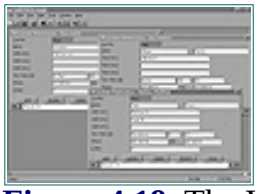

Figure 4.19 The Visual Data Manager lets you invoke a suite of tools to generate forms, queries, and so on.

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Previous | Table of Contents | Next

So…What Flavor Tastes Best?

I am partial to chocolate fudge, but when it comes to data access, I tend to lean toward whatever seems to have the most flexibility, the most power, and is easiest to use. In the past, that meant using three different and mutually exclusive choices. In my mind, the introduction of ADO changes all that. In a word, ADO is slick. Some might even say it is sexy, but I try not to get that involved.

If you are developing desktop applications, you are already using DAO with Jet, and if you are happy with it, there is no compelling reason not to continue doing so. However, if you are developing beyond the desktop, and especially if you are using ODBCDirect, you will probably want to convert to ADO. Chapter 9 will help you do that.

Likewise, if you are using RDO and it suits your needs for the time being, I see no reason to move to ADO either. However, that should not preclude you from doing any additional development with ADO. If RDO is not giving you what you need, then you probably should consider moving to ADO. Again, Chapter 8 should be of assistance in that regard.

If you are using the ODBC API in your present applications, things get a little murkier because you have a large code base to change. My advice for the time being is to do any new development using ADO but leave your existing code base alone. You can always convert piece-by-piece (or form-by-form) as the need arises.

Such is not the case with VBSQL. Although you definitely have a rather large code base to convert, Microsoft is moving away from DB-Lib, which makes me wonder how long VBSQL will be viable. Can you say “rewrite”? I am concerned that you might be trapped technologically and urge you to consider migrating to the ADO model. Again, the conversion does not have to occur all

at once; but you should move carefully to ensure that you don't break what ain't broke already.

Finally, you face the question of the model to choose for new development. For simple desktop development, DAO is still viable, and it is a more robust platform than in the past. In particular, Jet keeps showing improvements. How long Microsoft will continue to improve Jet remains to be seen, but it is certainly not disappearing in the short term. Still, you might want to develop a test form using both methodologies to see whether you prefer ADO. For all other development efforts, I think ADO is the way to go. Not only does it provide a solid, reasonably high-powered path to the database, but it also encompasses a wide array of data sources. Further, it is Microsoft's chosen technology of the future and will only get better.

Where To Go From Here

I have taken some time in this chapter to explore in a reasonable amount of depth each of the data access options available to the VB developer. I spent a little more time with the newer technologies. For more information on DAO-based development, move on to Chapter 5, which covers the subject in depth. Likewise, if you are proceeding with RDO or you need to enhance current RDO-based applications, you will want to move on to Chapter 6.

Because there is little need to do large-scale development with the ODBC API, I have not included a separate chapter for that. However, some additional information on ODBC can be found in Appendix C.

All readers should look at Chapter 7, which explores ADO and OLE DB in depth. Chapter 9 uses that technology to build scalable, robust database client/server applications.

Readers looking for even more information are urged to check out the Microsoft Web site at www.microsoft. com or to access the Microsoft Developers Network (MSDN), either online (www.microsoft.com/msdn/) or on CD-ROM (with a yearly subscription fee).

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Part II
Visual Basic 6 Database Programming

Chapter 5
Data Access Objects (DAO)

Key Topics:

- The DAO hierarchy
- Jet vs. ODBC workspaces
- Using DAO objects
- Using the Data control
- Transaction and concurrency management
- Database replication with Jet

As discussed in Chapter 4, Visual Basic provides a dizzying array of options to access the database. Which one is appropriate is largely a function of what it is you are doing. DAO is probably the simplest option, although even it can get complicated if you get away from using data-aware controls. In this chapter, I guide you through an in-depth examination of DAO, beginning with the object hierarchy and then discussing the Data control. Throughout the chapter, I provide numerous code examples (which you can also find on the enclosed CD-ROM). At the end of the chapter, I also overview Visual Basic's built-in Data control, with which most VB developers are familiar.

DAO Object Models

With Visual Basic 5, DAO had two object models (or workspaces): Microsoft Jet and ODBCDirect. With the Data control, DAO defaults to using Jet. To use ODBCDirect with the Data control, set the **DefaultType** property to 1 (**useODBC**). When using DAO objects in code, after creating the **Workspace** object, create a **Database** object for Jet data models or create a **Connection** object to use ODBCDirect.

Microsoft Jet

Version 3.0 of Visual Basic was the first with realistic tools to access and manipulate a database. Microsoft Jet in particular provided an interface to which the developer could write and access almost any relational database, as well as a wide array of Indexed Sequential Access Method (ISAM) file formats, such as Btrieve or dBASE. Even better, the interface was for the most part transparent to the developer, allowing the same program to run against data sources ranging from Microsoft SQL Server to FoxPro using Jet SQL. (See Appendix B for a discussion of the elements of Jet SQL; see Chapters 2 and 3 for a discussion of the elements of ANSI SQL.)

Microsoft generally mentions three types of Jet-accessible back-end databases: Microsoft databases (MDB files—in other words, Access databases), ISAM files, and ODBC data sources (not to be confused with access via ODBCDirect, which bypasses Jet). A Microsoft database is a hybrid between an ISAM database (which tends to have its “tables” in separate files whereas MS databases have all of their tables in one file) and a true relational database management system with its own database engine.

Jet offered developers a reliable if not particularly robust method to access and update databases. Although the Jet engine has improved in performance, it is not nearly as efficient as a native or ODBC interface to, say, Microsoft SQL Server. Even more, Jet’s footprint (the amount of space it takes) in memory is much larger than that of ODBCDirect, RDO, or other comparable access models.

The clear advantage to developers is the simplicity of the programs. Jet handles much of the dirty work for you. If you further restrict your development efforts to the Data control and data-bound controls, such as the textbox, building a database application is almost a no-brainer. Indeed, we’ll start with simple examples that do just that a little later in this chapter.

Although the performance of the Jet engine has improved with subsequent releases, it is still hardly a barn-burner. With Visual Basic 6, the current release of DAO is 4. Whether Microsoft will continue to develop the Jet engine (given the introduction of Active Data Objects [ADO]) is a point of conjecture.

Writing a client/server application for 4 or 5 users using Jet is fine, but you probably will not be happy if you scale it to 20 or 30 users, let alone 100 or 200 users. Even more, to be a one-stop solution, Jet implements a lowest common denominator approach to SQL. It is not a full implementation of

Structured Query Language, which limits you, the developer, to a core set of statements. For instance, common SQL statements such as **FETCH NEXT** and **FETCH PRIOR** are not supported. Jet relies on the application to scroll through a result set already loaded into memory. Likewise, Jet relies on DAO to perform transaction management. Jet itself does not directly support **COMMIT** and **ROLLBACK**. (The **RecordSet** object needs to provide this service.)

ODBCDirect

When you use ODBCDirect, Jet is bypassed. Instead, ODBCDirect creates a small layer that actually communicates with RDO (making it questionable whether a project using ODBC should even consider DAO). Still, there is a large DAO code base in existence, and ODBCDirect offers DAO applications almost the same performance as RDO. It also offers most of the same benefits, including asynchronous executions. You cannot use stored queries, but you can save precompiled queries as **QueryDef** objects. As with a stored procedure, you can input or output (or both) parameters on a stored query by adding **Parameter** objects to the **QueryDef**.

An ODBC **Connection** object can have one of four types of **RecordSet** objects: Dynamic, Dynaset, Forward-Only, and Snapshot. These objects are analogous to the ODBC cursor types of Dynamic, Keyset, Forward-Only, and Static.

The **OpenConnection**, **OpenRecordSet**, **Execute**, and **MoveLast** methods all allow asynchronous operations via the **dbRunAsync** option. You can cancel a running operation using the **Cancel** method and determine the status of an operation via the **StillExecuting** property.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

An Overview Of DAO Programming

In the next section, I discuss in depth each of the Data Access Objects. Whether you are using the Data control or creating all of the objects directly in code, the principles of DAO development remain the same; only the level of how dirty your hands get is different. Although using the Data control greatly simplifies development, an entirely code-based solution offers some additional flexibility. Use whichever approach makes sense for you.

Most developers tend to be intimidated by the complexity of the DAO hierarchy. However, a methodical review reveals that the model merely implements into objects the interface to the database. As a developer, you manipulate those objects using their properties and methods to read data from and write data to the database.

The hierarchy of the DAO objects in a Jet workspace is shown in Figure 5.1, whereas Figure 5.2 shows the DAO object hierarchy for ODBCDirect workspaces. Note that ODBCDirect is a “flattened” model that is intuitively simpler to understand. The Jet workspace data model is more involved because the Jet engine itself needs to do so much more work than the equivalent ODBCDirect data model. Put in another way, ODBC takes care of many of the details that Jet normally needs to take care of. That is why, for the RDO and ADO models in subsequent chapters, the hierarchies are also similarly simpler: Each relies on the tools provided by the back-end database to manage such things as users accounts, indexes, and so on.

Figure 5.1 DAO objects in the Jet workspace model.

Figure 5.2 DAO objects in the ODBCDirect workspace model.

The highest level object in each hierarchy is the **DBEngine** object. There is only one, even if you open both a Jet and an ODBCDirect workspace. When you use the Data control, the **DBEngine** object is automatically created. Likewise, any time you create a **Workspace** object in code, the **DBEngine** object is automatically created (almost any reference to a DAO object causes the **DBEngine** object to be created). Note, however, that if you take a code-based approach to DAO development, you need to add a reference to the DAO library: From the menu in Visual Basic, select Project|References. Scroll until you find the Microsoft DAO Library 4.0 and select it.

The two entities belonging to the **DBEngine** object are the **Errors** collection and the **Workspaces** collection. You will use the **Errors** collection to monitor database errors as they occur. With the Data control, you can place code in an **Error** event to handle database-related errors. The **Workspaces** collection contains all of the **Workspace** objects in your application. Each **Workspace** object represents a session with the database. If you are manipulating a FoxPro file, you have a **Workspace** object that contains all of the objects, methods, and properties needed to manage that file. If you are also manipulating an Oracle database, you have another **Workspace** object.

In the Jet model, a **Workspace** object has a **Users** collection and a **Groups** collection representing all of the authorized user and group accounts with permissions to the database represented within the **Workspace** object. In Figure 5.1, you will notice that a **User** object contains a **Groups** collection and that a **Group** object contains a **Users** collection. This seems like a circular reference, but it isn't. It represents the fact that any group (permissions are often administered at the group level in a database) can have multiple users in it and that any user may be a member of multiple groups.

An ODBCDirect **Workspace** object has a **Connections** collection and a **Databases** collection. When you create a **Connection** object, a corresponding **Database** object is also created. The same is true in reverse. The two objects are similar except that the **Connection** object has more capabilities than does the **Database** object. Both have a **RecordSets** collection, and the **Connection** object also has a **QueryDefs** collection.

The Jet **Workspace** object has only the **Databases** collection (in addition to

the **Users** and **Groups** collections).

When working with either ODBCDirect or Jet **Workspace** objects, you will spend most of your time with **RecordSet** objects. The **RecordSet** retrieves from the database one or more records and makes them available to be maintained. You also use the **RecordSet** object to add or delete records. The **RecordSet** object includes methods to update the database, handle concurrency issues, and so on. The **RecordSet** object includes a **Fields** collection, which represents all of the fields in each record of the record set. Each **Field** object contains information such as the value of the field and its data type.

TIP
Types Of Workspaces
Note that a workspace connecting directly to an ODBC data source is called an ODBCDirect workspace. All others are called Jet workspaces even if they connect to an ODBC data source via Jet.

The **QueryDef** object represents a stored or predefined query (on non-ODBC databases, the query is actually stored within the database). The query can be executed directly or it can be the source of the **RecordSet** object. The **Fields** collection represents all of the fields of the **QueryDef** object, whereas the **Parameters** collection allows for the maintenance of query parameters that are not known until runtime.

A Jet workspace also has a **TableDefs** collection, which represents the layout of tables stored on the database. The **Relations** collection stores relationships between tables (foreign key constraints).

The **Containers** and **Documents** collections represent information about objects within the workspace. Each **Container** object contains a **Documents** collection. Three **Container** objects are predefined by DAO. The **Databases** container contains **Documents** that store information about all saved databases (whereas the **Databases** collection contains information about all open databases). Similarly, the **Tables** and **Relations** containers contain **Document** objects that describe information about saved tables and saved relationships. The **Tables Container** object also contains information about saved queries.

Finally, most DAO objects (except **Connection** and **Error**) have a **Properties** collection that contains all properties of the containing object. In Figures 5.1 and 5.2, the **Properties** collections were omitted for clarity.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

DAO Objects

The following section examines in detail each of the objects in the DAO hierarchy. I have included a lot of code examples; you should review each one carefully, especially observing the differences between the Jet and the ODBCDirect workspaces.

In the interest of space, I do not repeat most of the methods and properties that these object have in common. For instance, each object except **DBEngine** has a **Name** property by which you reference an object in a collection: **RecordSet ("Employee")**. Likewise, every object except **Connection** and **Error** has a **Parameters** collection, so I do not discuss this separately for each object.

DBEngine

The top-level object is **DBEngine**—that is to say, **DBEngine** contains all other DAO collections and objects. Whenever any DAO object is instantiated (such as when connecting to the database with the Data control), **DBEngine** is automatically created.

The **DBEngine** object contains three collections: **Errors** is a collection of **Error** objects; **Workspaces** is a collection of **Workspace** objects; and **Properties** is a collection of **Property** objects. **Workspaces** is the default collection of the **DBEngine** object.

DBEngine has several properties:

- **DefaultPassword** is a string with a maximum length of 14 characters in Jet workspaces and no limit in ODBCDirect workspaces. It is used to store a default password in case no password is supplied at connection time. If the password is not specified, the value defaults to an empty string. Typically, passwords are case-sensitive, although that is not always true with an ODBCDirect data source (where it is normally a function of the back-end database or the ODBC driver).
- **DefaultUser** is a string with a maximum length of 20 characters in Jet workspaces and no limit in ODBCDirect workspaces. It is used to store a default user ID in case no ID is supplied at connection time. If the ID is not specified, the **DefaultUser** property defaults to "Admin". Unlike the **DefaultPassword** property, user IDs are not case-sensitive. However, there are restrictions on valid characters—the ID cannot contain control characters (ASCII values 0 to 31); slashes; brackets; colons or semicolons; commas; question marks; plus or equals symbols; asterisks; or pipe characters ("|"). Also, the ID cannot contain any leading spaces.

- **DefaultType** sets or returns the default workspace type. Specify **dbUseJet** to create a Jet workspace or **dbUseODBC** to create an ODBCDirect workspace.
- **IniPath** (Jet workspace only) is a string that sets or returns the Registry key where settings such as installable ISAM DLLs for the Jet engine are maintained. The **IniPath**, if used, must be set prior to the execution of any other DAO code and remains in effect as long as the **DBEngine** object is instantiated. You may specify a root key of HKEY_LOCAL_MACHINE (the default) or HKEY_LOCAL_USER in the Registry.
- **LogInTimeOut** is an integer that sets or returns the number of seconds that will elapse before an error occurs when attempting to connect to an ODBC data source (Jet or ODBCDirect workspaces). The default is 20 seconds. A value of 0 specifies no limit.
- **SystemDB** (Jet workspace only) is a string that sets or returns the path of the workgroup information file. The default is **System.MDW** with no path. The workgroup information file is most often used with Microsoft databases (Access files), although you can use it with any Jet data source. It contains settings that permit or deny users and groups access to secured data objects in the database. You must create the setting prior to instantiating the **DBEngine** object, although it can follow setting the **IniPath** property. Once **DBEngine** is created, **SystemDB** remains in effect until the application ends.
- **Version** returns the current version of DAO.

The **DBEngine** object also has a number of methods:

- **BeginTrans** is used to begin a transaction on the database. **CommitTrans** tells the database to end the current transaction and make permanent all of the changes to the database. You may optionally specify the **dbFlushOSCacheWrites** constant to force the operating system to immediately write all changes to disk. Although this could adversely affect performance, it also prevents the changes from being lost if the user turns off the PC before the changes are written. I highly recommend using this setting, as shown in the first example. **Rollback** is the opposite of **CommitTrans** ; it discards all changes made to the database since the last **BeginTrans** and ends the current transaction or logical unit of work (LUW). You will normally use this method when an error occurs during one or more of the updates. Listing 5.3, later in this chapter, illustrates the use of these methods.
- **CompactDatabase** (Jet workspace only) takes an existing database and compacts it, optionally creating a new collating sequence. You can also use this method to create a copy of an existing database. The syntax is shown in the next code example. Databases are much like the hard drive on your computer. As you add and delete records, the database quickly becomes fragmented, resulting in loss of performance. Compacting the database rearranges the rows in each table contiguously, resulting in less head movement to read records. Use **old_db** to specify the fully qualified path and name of the database file. Use **new_db** to specify the new path and name of the database file. **dest_locale** is where you specify a new collation (sorting) sequence. If omitted, the locale will be the same as for **old_db**. The **options** property allows the use of one or more of the settings summarized in Table 5.1. To use more than one setting, add the values. **src_locale** is the current collation sequence and can normally be omitted. Listing 5.3, later in this chapter, illustrates the repairing and compacting of a database.

```
DBEngine.CompactDatabase old_db, new_db, dest_locale, _
   options, password, src_locale
```

Table 5.1 Valid option settings.

Value	Purpose
dbEncrypt	Encrypts the database while compacting.
dbDecrypt	Decrypts the database while compacting.
dbVersion10	**new_db** will be the Jet version 1.0 file format.

dbVersion11	**new_db** will be the Jet version 1.1 file format.
dbVersion20	**new_db** will be the Jet version 2.0 file format.
dbVersion30	**new_db** will be the Jet version 3.0 file format.

- **CreateDatabase** (Jet workspace only) creates and opens a new Microsoft database object. The syntax is shown in the following code segment. **workspace** is a reference to an existing **Workspace** object. **db_name** is a string containing the name of the new database file. **locale** refers to the collation sequence to be used with the database, and **options** sets various options for the database, such as encryption and file format (see Table 5.1). To merely copy an existing database, use the **CompactDatabase** method.

```
Set dbs = CreateDatabase = workspace.CreateDatabase _
   (db_name, locale, options)
```

- **CreateWorkspace** is used to create a new **Workspace** object. The new **Workspace** object is not automatically appended to the **Workspaces** collection nor do you need to do so before using it. (For more information, see "The **Workspaces** Collection And **Workspace** Object" later in this chapter.) **workspace_name** is an object variable by which you will reference the **Workspace** object. **user_name** is a string identifying the owner of the workspace. **password** is a string containing the password for the **Workspace** object, and **type** is either **dbUseJet** or **dbUseODBC**. If **type** is omitted, the default is to create a Jet workspace. If the **DBEngine** object's **DefaultType** property has been set, it will govern the default type of workspace. Listing 5.1 creates two **Workspace** objects and appends them to the **Workspaces** collection. Note that because the first workspace uses ODBCDirect, the Jet engine is not loaded until the second workspace object is created. At the end of Listing 5.1, the two **Workspace** objects are closed, which automatically removes them from the **Workspaces** collection.

Listing 5.1 Creating Workspace objects and appending them to Workspaces.

```
' Create object variables
Dim wrkJet As Workspace
Dim wrkODBC As Workspace

' Create the ODBCDirect Workspace
Set wrkODBC = CreateWorkspace("ODBC Workspace", "Coriolis", _
   "Coriolis", dbUseODBC)
' Append to collection
Workspaces.Append wrkODBC

' Create the Jet Workspace
Set wrkJet = CreateWorkspace("Jet Workspace", "Coriolis", _
   "Coriolis", dbUseJet)
Workspaces.Append wrkJet

' Close the Workspace objects
wrkODBC.Close
wrkJet.Close
```

- **Idle** (Jet workspace only) can be considered similar to the Visual Basic **DoEvents** function. The **Idle** method suspends processing, allowing pending tasks to be completed. In a high volume, multiuser environment, **DBEngine** might not have the chance to release locks. Using the **Idle** method allows these locks to be released, thus potentially enhancing performance. You may optionally specify the **dbRefreshCache** argument, which forces any pending writes to a Microsoft database file to be written to disk. It also refreshes the records

that are currently buffered in memory.

- **OpenConnection** (ODBCDirect workspace only) is similar to the **OpenDatabase** method. The syntax for the method appears in the following code example. **connection** is a valid **Connection** variable by which you reference the **Connection** object. **Workspace** is a valid existing **Workspace** object. If omitted, the default **Workspace** object is used. The **Name** argument must specify either a valid data source name (DSN)— if not specified in the argument property—or it may contain any string. Either way, the **Name** argument then becomes the **Name** property of the **Connection** object. You may also omit both the name and the connect arguments depending on how you set the **Options** argument. If you allow the ODBC driver to prompt for missing information, the DSN chosen by the user will become the **Name** property. The **Options** argument determines if and how the ODBC driver will prompt the user for DSN information. The valid values are listed in Table 5.2. **Read_Only** specifies whether the **Connection** object will be read-only. If omitted, the default is **False** (the **Connection** object is read-write). The **Connect** argument is an optional string specifying how to connect to the database. If supplied, it must begin with "**ODBC;**". All parameters must be delimited by semicolons. If omitted, the password and user ID are taken from the **Password** and **UserName** properties of the **Connection** object. The syntax of the **Connect** property is fairly flexible, and its exact requirements will vary according to the back-end database. A typical connect string is **ODBC; DSN=Coriolis VB Example; UID=Coriolis; PWD=Coriolis;**. Listing 5.2 shows an example of using the **OpenConnection** method for connecting to an ODBC data source.

```
Set connection = Workspace.OpenConnection (Name, Options, _
  Read_Only, Connect)
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Table 5.2 Valid options when connecting to an ODBC data source.

Value	Meaning
dbDriverComplete	(Default) Prompt only for missing connect information.
dbDriverCompleteRequired	Prompt only for required missing information.
dbDriverNoPrompt	Use settings from the **Connect** string.
dbDriverPrompt	Display the ODBC Data Sources dialog box.
dbRunAsync	Connect asynchronously. May be combined with any of the preceding options.

Listing 5.2 The OpenConnection method.

```
Dim wrkODBC as Workspace
Dim conCoriolis As Connection

' Create ODBCDirect Workspace object
Set wrkODBC = CreateWorkspace("Coriolis", _
   "Coriolis", "Coriolis", dbUseODBC)

' Open Connection ' object
' Connect asynchronously and prompt only
' for missing information
Set conCoriolis = wrkODBC.OpenConnection _
  ("Coriolis", dbDriverCompleteRequired + dbRunAsycn, False, _
  "ODBC;DSN=Coriolis VB Example;UID=Coriolis;PWD=Coriolis;")
```

- **OpenDatabase** is similar to **OpenConnection**, but it can also be used in Jet workspaces. When **OpenDatabase** is used in an ODBCDirect workspace, a **Connection** object is automatically created. The syntax is shown in the next code segment. **database** is a valid object variable of type **Database** that is used to reference the **Database** object. **workspace** is a valid, existing **Workspace** object. If not supplied, the current **Workspace** object is used. **db_name** is a string containing the name of the database. For a Jet workspace, **db_name** is a qualified path and file name of a Jet database file. For ODBCDirect workspaces, it can be a connect string as shown in the **OpenConnection** method. If it is

left blank and the connect property begins with "**ODBC;**", the ODBC driver manager will prompt the user for other information. If the connect argument is complete, then **db_name** can be any string, and you can then reference the **Database** object using that name. For a Jet workspace, the **options** argument may be either **True** or **False** (default). If **True**, the database is opened in exclusive mode (no other process may use it). If another process attempts to open it, an error results. If the database is already opened and you attempt to open it in exclusive mode, an error results. For ODBCDirect workspaces, the value may be any of those specified in Table 5.2. The **connect** argument specifies what database type (such as Access or Paradox) to open and may also specify a password and other information. Separate each of these parameters with semicolons. To access an ODBC data source (regardless of whether you use Jet or ODBCDirect), begin the string with "**ODBC;**".

```
Set database = workspace.OpenDatabase (db_name, options, _
   read_only, connect)
```

• **RegisterDatabase** enters connection information about the particular database into the Windows Registry. I strongly urge that you instead use the ODBC Administrator applet in the Control Panel. The requirements vary by database type, and you should consult the help file for the database to determine exact requirements.

• **RepairDatabase** (Jet workspace only) is used to attempt to repair a corrupt database. The database must be closed; no application may be using it when you attempt to repair it. The method will validate all data and indexes and will delete any data that can't be repaired. A runtime error occurs if the repair fails. The CD-ROM contains an application that lets you select a database for repair (see Figure 5.3) and optionally compact it. (Modify the application as needed to add additional database types. Be sure to add a reference to Microsoft DAO objects.)

Figure 5.3 The Database Repair application included on the CD-ROM.

• **SetOption** (Jet workspace only) is a method that allows you to temporarily override Jet engine values stored in the Windows Registry. The syntax is shown in the next code example. **parameter** is the Registry key to override, and **new_value** is the temporary new setting. The valid parameters are **dbPageTimeout**, **dbMaxBufferSize**, **dbSharedAsyncDelay**, **dbMaxLocksPerFile**, **dbExclusiveAsyncDelay**, **dbLockDelay**, **dbLockRetry**, **dbRecycleLVs**, **dbUserCommitSync**, **dbFlushTransactionTimeout**, and **dbImplicitCommitSync**. See your database documentation for valid values and meanings of these parameters:

```
SetOption parameter, new_value
```

The Errors Collection And Error Object

The **Errors** collection consists of **Error** objects. When errors occur during a DAO operation, they are added to the **Errors** collection. The collection is cleared when a new DAO operation generates an error. This means that all **Error** objects in the collection relate to one error condition. Typically, the first object in the collection represents the lowest-level error. For instance, if you select from a table that does not exist, the ODBC driver will return the first error. The ODBC driver manager will likely then return an error, and finally, DAO will return an error indicating that the data wasn't found. You should check the **Count** property of the **Errors**

collection after database operations to ensure that no errors have taken place. If using the Data control, you can also place code in the **Error** event to intercept any errors. If you use the **New** keyword to create a new instance of a DAO object and the operation results in an error, the error will not be appended to the **Errors** collection because the object does not get created and thus is not yet part of DAO. VB's **Err** object will contain the pertinent error information instead.

You can iterate through the **Errors** collection to examine all **Error** objects. Because the collection is zero based, the **Count** property is always one greater than the highest index. However, by using the **For Each** construct, you do not need to take this into account as you would with a **For Next** loop. The following code iterates through the **Errors** collection, displaying each error description:

```
Dim vError As Variant
For Each vError in Errors
   MsgBox vError.Description
Next
```

The **Errors** collection has only a **Refresh** method. Because **Error** objects are automatically appended, there are no **Append** or **Delete** methods.

The **Error** object has properties similar to those of VB's **Err** object. The **Description** property is a string that contains a textual description of the error, and the **Number** property is a **Long** containing the error number. For a list of these errors, consult "Trappable DAO Errors" in the VB help file. The **Source** property is a string containing the programmatic class ID of the object where the error occurred. The **HelpFile** property is a string containing the path and name of the help file to present to the user for more information about the error. The **HelpContext** property is a string containing the appropriate help context ID.

Figure 5.4 shows an application that you can copy from the CD-ROM. The application allows you to type an SQL **SELECT** statement and then run it. Alternatively, you can enter an employee number or employee last name and press the retrieve button, and the application will dynamically create a **SELECT** statement and run it. The statement generated is displayed in a listbox along with the results, including records found or errors generated.

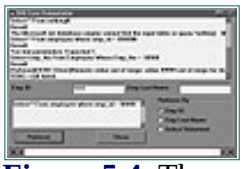

Figure 5.4 The sample DAO Errors application allows you to dynamically generate SQL statements and see the results. The application code is on the CD-ROM.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The Properties Collection And Property Object

Every DAO object except **Error** and **Connection** has a **Properties** collection. You can add your own properties to **Database**, **Field**, **Index**, **QueryDef**, **TableDef**, or **Document** properties on Jet workspaces only using the **Properties** collection's **Append** method. (The most common use occurs in maintaining partial replicas, as I discuss at various points in this chapter.) You must first define its characteristics with the **CreateProperty** method of the object to which you want to add the **Property** object. The following syntax creates a new **Property** object:

```
Set prop_var = object.CreateProperty (prop_name, prop_type, _
  prop_val, prop_DDL)
```

Prop_name is a string containing the name by which you will reference the **Property** object. **Prop_type** specifies the data type of the object, as listed in Table 5.3. **Prop_val** is the initial value and **prop_DDL** is a boolean which, if **True**, indicates that this is a DDL object. There is an example later in this chapter where we discuss the **Replicable** property of the **Database** object.

Table 5.3 Valid Property types.

Type	Description
dbBigInt	Big integer (whole number with precision of 20 [unsigned] or 19 [signed])
dbBinary	Binary (fixed-length binary up to 255 characters)
dbBoolean	Boolean
dbByte	Byte
dbChar	Char (fixed-length string)
dbCurrency	Currency
dbDate	Date/Time

dbDecimal	Decimal
dbDouble	Double
dbFloat	Float
dbGUID	GUID (Global Unique Identifier used with RPCs [remote procedure calls])
dbInteger	Integer
dbLong	Long
dbLongBinary	Long binary (OLE object)
dbMemo	Memo (variable length up to 1.2MB)
dbNumeric	Numeric
dbSingle	Single
dbText	Text (fixed-length string up to 255 characters)
dbTime	Time
dbTimeStamp	Time stamp
dbVarBinary	VarBinary (variable-length binary data up to 255 characters)

User-defined properties can be inherited in Jet workspaces. If you add a user-defined property to a **QueryDef** object and then create a new **RecordSet** object using the **QueryDef** object as a record source, the **RecordSource** will inherit the new user-defined property.

You can determine whether a property is inherited by examining the **Property** object's **Inherited** property. The property is a Boolean, and **True** indicates that the property is inherited. Other properties of the **Property** object include **Name**, **Type**, and **Value**, which correspond to the **prop_name**, **prop_type**, and **prop_val** arguments in the **CreateProperty** method discussed previously.

I have included an application on the CD-ROM called DAOHierarchy that iterates through all DAO objects and their **Properties** collections. The application has a Jet workspace and an ODBCDirect workspace. The **Property** objects are displayed in a listbox, as shown in Figure 5.5.

Figure 5.5 This application iterates through all DAO objects and lists the **Property** objects in each.

The Workspaces Collection And Workspace Object

Workspaces is the default collection of the **DBEngine** object, and it is the collection with which you will interact the most. You will create a **Workspace** object for each database session that you require. The VB documentation defines a session as "…a sequence of operations performed by the Microsoft Jet database engine. A session begins when a user logs on and ends when a user logs off. All operations performed during a session form one transaction scope…." The definition is fine for Jet workspaces, but ODBCDirect

workspaces often have multiple transactions within a current session. Thus, I will use the term a little more loosely than does the VB documentation.

The first time you create or refer to a **Workspace** object, the default **Workspace** object is created as **DBEngine.Workspaces (0)**. You create a new **Workspace** object with the **CreateWorkspace** method of **DBEngine**. You must append the object to the **Workspaces** collection using the collection's **Append** method. You can create a "hidden" workspace by not appending it to the collection. I discussed the creation of a workspace and appending it to the collection under **DBEngine** earlier in this chapter. You can reference any of the **Workspace** objects within **Workspaces** by using its ordinal index or by its name: **Workspaces (1)** or **Workspaces ("Coriolis")**.

The **Workspaces** collection has the usual **Append**, **Delete**, and **Refresh** methods as well as the **Count** property.

The **Workspace** object is comprised of different collections itself (refer to Figures 5.1 and 5.2), depending on whether it is a Jet workspace or an ODBCDirect workspace. Jet workspaces have these collections: **Databases**, **Groups**, **Users**, and **Properties**. ODBCDirect workspaces have **Connections**, **Databases**, and **Properties**. In both cases, **Databases** is the default collection.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The **Workspace** object has the following properties:

- **DefaultCursorDriver** (ODBCDirect workspace only) is a long equal to one of the values shown in Table 5.4. It sets or returns the type of cursor driver used when a new **Database** or **Connection** object is created. (I discuss the concept of cursors, cursor libraries, cursor stability, and so on, at some length in Chapter 11.)

Table 5.4 Valid cursor driver types.

Type	Description
dbUseDefaultCursor	(Default) If the server supports cursors, use them. Otherwise, use clientside cursors.
dbUseODBCCursor	Use client-side cursors.
dbUseServerCursor	Use server-side cursors.
dbUseClientBatchCursor	Use the client batch cursor library.
dbUseNoCursor	Do not use cursors. Record set is open read-only, forward-only with a rowset size of 1.

- **IsolateODBCTrans** (Jet workspace only) is a Boolean that indicates whether transactions on the same Jet ODBC data source are isolated. For example, assume you have two record sets open on the same ODBC database; perhaps one is scrolling through the **Employee** table while another is executing a report on the **Customer** table. If **IsolateODBCTrans** is set to **True**, then both of the **RecordSet** objects represent distinct transactions. If set to **False**, then the two **RecordSet** objects comprise a single transaction and a **Rollback** or **CommitTrans** on either **RecordSet** affects both objects. Having multiple transactions is not supported by the **Workspace** object, so you have to open a separate **Workspace** object in order to have multiple transactions on the same ODBC data source (which is why this is a property of the **Workspace** object and not the **RecordSet** object, which would be more

intuitive). A further consequence of setting **IsolateODBCTrans** to **True** is that each workspace involving the same ODBC data source creates a distinct connection. Therefore, for three **Workspace** objects connected to a given ODBC data source, there are three separate, simultaneous connections to the server. This can be expensive in terms of server resources, and I recommend that you consider carefully the wisdom of having more than one operation against the same database occur simultaneously.

- **LogInTimeOut** (ODBCDirect workspace only) is an integer representing how many seconds will elapse while trying to connect to a database before an error occurs. A value of 0 indicates there is no limit. This value overrides the **LogInTimeOut** value set at the **DBEngine** level.

- **Type** is an integer that sets or returns the workspace type. You can set the type only when creating the **Workspace** object. The valid values are **dbUseJet** and **dbUseODBC**. For true back-end RDBMSs, we recommend using the ODBCDirect workspace because it is more efficient than Jet. Even better would be to consider RDO or ADO. With an ISAM database such as Access or Paradox, you are probably better off using the Jet model.

- **UserName** is a string representing the owner of the **Workspace** object. If you are going to use this property, you must set it before appending the object to the **Workspaces** collection. The most common use of the property is in verifying or altering security privileges. We discuss this option in more detail when discussing the **Container** object later in this chapter.

The **Workspace** object supports several methods:

- **BeginTrans** creates a new transaction. **CommitTrans** commits the current transaction. **Rollback** rolls back the current transaction. I discussed these three methods under the **DBEngine** topic earlier in this chapter. When you use **BeginTrans** with the **Workspace** object, you face additional considerations. If you close a **Workspace** object after issuing a **BeginTrans**, all changes are rolled back. If you issue a **CommitTrans** or **Rollback** without first issuing a **BeginTrans**, a runtime error occurs. If you change a record outside of a transaction (without first issuing a **BeginTrans**), the change is automatically committed. Not all ISAM databases (or ODBC drivers) support transactions. If the database does not support transactions, the **Database** object's **Transaction** property will be **False**. Depending on how the record set is opened, the **RecordSet** object may not support transactions. In either case, issuing **BeginTrans**, **CommitTrans**, or **Rollback** is ignored and no error is generated. If you use a Jet workspace, a "log file" of transactions applied is maintained in the Temp directory. If the disk runs out of space, an error occurs and data may be lost. Issuing a **Rollback** empties the log. Issuing a **CommitTrans** outside of a nested transaction also flushes the log. You can nest transactions as shown in the next code example. Even if an inner transaction commits changes to the database, if an outer transaction issues a **Rollback**, the inner transactions are rolled back as well. Listing 5.3 illustrates the use of

nested transactions in an ODBCDirect workspace. The six textbox controls are an array named **txtFields**. The two CommandButton controls are an array named **Command1**. The running application is shown in Figure 5.6. It creates **Workspace**, **Connection**, and **RecordSet** objects in code, displays the results in the textboxes, allows the user to alter data, and then rolls back the changes. Note that the inner transaction may commit the work but that the outer will roll back all changes anyway.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Listing 5.3 The DAO transaction demonstration program.

```
Option Explicit

Private Sub Command1_Click(Index As Integer)

Dim iRtn As Integer
Dim sRaise As String
Dim sMsg As String
Dim sConn As String
Dim cSalary As Currency
Dim cSaveSal() As Currency
Dim wrkEmp As Workspace
Dim conCoriolis As Connection
Dim rsEmp As RecordSet

Select Case Index
    Case 0
      ' Create Workspace
      Set wrkEmp = CreateWorkspace _
         ("emp", "admin", "", dbUseODBC)
      ' Append to collection
      DBEngine.Workspaces.Append wrkEmp
      ' Create connection object
      sConn = "ODBC;DSN=Coriolis VB Example;UID=Coriolis;" & _
         "PWD=Coriolis;Database=Coriolis;"
      Set conCoriolis = wrkEmp.OpenConnection _
          ("", , , sConn)
      ' Create RecordSet object
      Set rsEmp = conCoriolis.OpenRecordSet _
          ("Select * from Employee", dbOpenDynamic)
      ' Start of outer transaction.
```

```
wrkEmp.BeginTrans
' Start of inner transaction.
wrkEmp.BeginTrans
With rsEmp
    ' Make sure row count is accurate
    .MoveLast
    .MoveFirst
    ' Set up array to save old salaries
    ReDim cSaveSal(rsEmp.RecordCount)

    ' Prompt for changes to salary
    Do Until .EOF
      txtFields(0) = !emp_no
      txtFields(1) = !emp_fname
      txtFields(2) = !emp_lname
      txtFields(3) = Format$(!emp_Salary, "###,##0.00")
      cSaveSal(rsEmp.AbsolutePosition) = !emp_Salary
      sMsg = "What percent raise for " & !emp_fname & _
          " " & !emp_lname & " making $" & _
          Format$(!emp_Salary, "###,##0.00") & "?"
          sRaise = InputBox$(sMsg, "Raise", "0")
      cSalary = !emp_Salary * (1 + Val(sRaise) / 100)
      If cSalary <> !emp_Salary Then
              ' If changed, edit the record
              .Edit
              !emp_Salary = cSalary
              ' Save the change
              .Update
      End If
      ' Move to next record
      .MoveNext
    Loop
    ' Commit the changes?
    ' This is the inner transaction!!!
    If MsgBox("Save all changes?", vbYesNo + _
       vbQuestion,"Commit or Rollback") = vbYes Then
    wrkEmp.CommitTrans
    Else
        wrkEmp.Rollback
    End If
    ' Display changes (if any)
    .MoveFirst

    Do While Not .EOF
        txtFields(0) = !emp_no
        txtFields(1) = !emp_fname
        txtFields(2) = !emp_lname
        txtFields(3) = cSaveSal(rsEmp.AbsolutePosition)
        txtFields(4) = !emp_Salary
        If MsgBox("Show next record?", vbOKCancel + _
            vbQuestion, "Display Salary Changes") _
```

```
                = vbCancel Then
                Exit Do
            End If
            .MoveNext
        Loop
        ' Roll back all updates
        ' This is the outer transaction
        wrkEmp.Rollback
        .Close
    End With
    conCoriolis.Close
  Case 1
    End
End Select

End Sub
```

Figure 5.6 The DAO transaction application.

TIP
About Workspaces And Transactions
Transactions always have workspace scope. That is, a transaction affects all objects within the **Workspace** object, even if the object contains multiple **Database** objects. If you issue a **CommitTrans** for one of multiple **Database** objects, all the **Database** objects are affected. If you need to manage transactions separately for different **Database** objects, create additional **Workspace** objects. Note that you can nest transactions within a **Workspace** object. See the **BeginTrans** method later in the list for more information.

- The **Close** method closes the **Workspace** object. Its use is illustrated near the end of Listing 5.3 where **wrkEmp** is closed. You cannot close the default workspace. (Attempting to close it is ignored; no error is generated.)
- The **CreateDatabase** method (Jet workspace only) creates a new **Database** object. The ODBCDirect **Database** object is retained for backward compatibility with Jet workspaces and is created automatically when you use the **OpenConnection** method. However, although you can use the **Database** object much as you would use it in a Jet workspace, the **Connection** object is richer in functionality. Use of the method was discussed under the **DBEngine** topic earlier in this chapter.
- The **CreateGroup** method (Jet workspace only) creates a new **Group** object. The syntax is shown in the next code segment. **object** is either a **Workspace** or **User** that will own the **Group** object. The **name** may have any combination of letters, numbers, and underscores but must begin with a letter. **pid** (personal identifier) is a string of 4 to 20 characters, which Jet uses in conjunction with an account name to identify a user or group in a **Workspace** object.

```
Set Group = object.CreateGroup (name, pid)
```

- The **CreateUser** method (Jet workspace only) creates a new **User** object. The syntax is shown in the following code example. **object** is either a **Workspace** or **Group**. **pid** is the personal identifier and **password** is a string of up to 14 characters containing the **User** object's password.

```
Set User = object.CreateUser (name, pid, password)
```

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The Users Collection And User Object

The **Users** collection is a Jet-workspace-only construct belonging to the **Workspace** and **Group** objects (see Figure 5.1). The **User** object is an account with a certain set of permissions to the underlying data within a **Workspace** object. Specifically, **User** objects enforce access permissions and restrictions on **Document** objects within the **Workspace** that in turn represent the databases, tables, and queries. (Note that permissions are actually administered at the document level by the **Permissions** property of the **Document** object; see "The **Documents** Collection And **Document** Object" later in this chapter.) Given a **User** object, you can create a **Workspace** object with the same set of permissions as the **User** object. You can append a **User** object to a **Group** object's **User** collections, having the effect of giving that **User** object the same permissions as the **Group** object. Conversely, you can append a **Group** object to the **Groups** collection of a **User** object, having the effect of making that **User** object a member of that group.

Note that because of the way the **Group** and **User** objects tend to reference each other, it is a good idea to invoke the **Refresh** method before referencing a collection: **wrkEmp.Users.Refresh**.

DAO automatically creates two **User** objects, one named **Admin** and the other **Guest**. **Admin** is added to the **Group** objects known as **Admin** and **Users**, whereas **Guest** is made a member of the **Group** object named **Guests**. See "The **Groups** Collection And **Group** Object" later in this chapter for more information about these groups.

To create a new **User** object, use the **Workspace** or **Group** object's **CreateUser** method. I discussed this method earlier in the chapter in the discussion on **Workspaces**.

The **User** object has two collections: **Groups** and **Properties**. The object has

three properties: **Name**, **Password**, and **PID**. I discussed each of these under the **CreateUser** method.

The **User** object has two methods. I discussed the **CreateGroup** method earlier in the chapter. You use the **NewPassword** method to change the password of an existing **User** object. The syntax is shown in the next code segment. **old_password** is the object's current password. **new_password** is the object's revised password. Note that you must have the proper permissions to alter the **Password** property of a **User** object. To clear the password, use a zero-length string for **new_password**. Passwords are case-sensitive. I also discussed this method earlier in the chapter under the **DBEngine** topic.

```
object.NewPassword old_password, new_password
```

The Groups Collection And Group Object

The **Groups** collection is a Jet-workspace-only construct belonging to either a **Workspace** object or a **User** object (see Figure 5.1). To create a new **Group**, use the **CreateGroup** method, which we discussed earlier in the chapter. You can append a **Group** object to a **User** object's **Groups** collection, which grants that user membership in the group (also see the discussion of the **User** object earlier in this chapter). Because of the way users and groups tend to reference each other, it is good practice to use the **Refresh** method before referencing the collection.

The **Group** object represents a group of users who have common access permissions and restrictions to **Document** objects within a **Workgroup** object. Note that the actual security is maintained at the **Document** object level via the **Permissions** property.

The **Group** object has two collections (see Figure 5.1), **Users** and **Properties**; two properties, **Name** and **PID**; and one method, **CreateUser**. All of these were discussed earlier in the chapter.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The Databases Collection And Database Object

The **Databases** collection contains all **Database** objects created within a **Workspace** object. The **Database** objects are automatically appended to the collection and are created by the **Workspace** object's **CreateDatabase** method, which I discussed earlier in this chapter. The **Databases** collection has only the **Refresh** method and the **Count** property. The **Database** object is automatically removed from the collection when you close it with the **Close** method. You should be sure to close all **RecordSet** objects within a **Database** before closing the **Database** object.

The **Database** object consists of a number of collections: **Containers**, **QueryDefs**, **Relations**, and **TableDefs** are Jet-workspace-only collections. **Properties** and **RecordSets** are collections common to both Jet and ODBCDirect workspaces. **TableDefs** is the default for Jet workspaces, whereas **RecordSets** is the default for ODBCDirect workspaces (see Figures 5.1 and 5.2).

The **Database** object is used to manipulate an open database. However, note that although you can use it for ODBCDirect workspaces, the **Connection** object has more functionality. The **Database** object has a number of properties:

- **CollatingOrder** (Jet workspace only) affects the sort sequence by setting which alphabet to use for string comparisons.
- The **Connect** property sets or returns a string describing connection parameters to a database. The syntax for the **Connect** property is shown in the next code example. **object** is a valid **Database**, **TableDef**, **QueryDef**, or **Connection** object. (For **QueryDef** objects, the property is read-only.) **db_type** is a valid database type, as listed in Table 5.5. If under a Jet workspace, you are performing a connection to a table linked via a Microsoft database (called a *linked table*), the **db_type** argument must be a valid ODBC connection string. **parameters** is an optional argument with requirements that vary by database (see the database documentation for more information). If supplied, the **db_type** must end in a semicolon and all parameters must be semicolon delimited. If using ODBCDirect, **db_type** must be **ODBC;**. If you do specify this as a **db_type** and supply no parameters, then the ODBC driver displays a dialog box of all available data sources. If a password is required but not supplied in the parameters argument, a logon dialog is displayed. For more information, see the **OpenDatabase** method of the **Workspace** object discussed earlier in this chapter.

```
object.Connect db_type; parameters;
```

Table 5.5 Connect string settings.

Database Type	Specifier	Example
Microsoft Jet Database	[*database*];	*drive:\path\filename.mdb*
dBASE III	dBASE III;	*drive:\path*
dBASE IV	dBASE IV;	*drive:\path*
dBASE 5	dBASE 5.0;	*drive:\path*
HTML Import	HTML Import;	*drive:\path\filename*
HTML Export	HTML Export;	*drive:\path*
Lotus 1-2-3 WKS	Lotus WK1;	*drive:\path\filename.wks*
Lotus 1-2-3 WK1	Lotus WK1;	*drive:\path\filename.wk1*
Lotus 1-2-3 WK3	Lotus WK3;	*drive:\path\filename.wk3*
Lotus 1-2-3 WK4	Lotus WK4;	*drive:\path\filename.wk4*
Microsoft Excel 3.0	Excel 3.0;	*drive:\path\filename.xls*
Microsoft Excel 4.0	Excel 4.0;	*drive:\path\filename.xls*
Microsoft Excel 5.0	Excel 5.0;	*drive:\path\filename.xls*
Microsoft Excel 95	Excel 5.0;	*drive:\path\filename.xls*
Microsoft Excel 97	Excel 8.0;	*drive:\path\filename.xls*
Microsoft Exchange	Exchange 4.0; MAPILEVEL=*folderpath*; [TABLETYPE={ 0 | 1 }]; [PROFILE=*profile*;] [PWD=*password*;] [DATABASE=*database*;]	*drive*:*path\filename.mdb*
Microsoft FoxPro 2.0	FoxPro 2.0;	*drive:\path*
Microsoft FoxPro 2.5	FoxPro 2.5;	*drive:\path*
Microsoft FoxPro 2.6	FoxPro 2.6;	*drive:\path*
Microsoft Visual FoxPro 3.0	FoxPro 3.0;	*drive:\path*
Paradox 3.x	Paradox 3.x;	*drive:\path*
Paradox 4.x	Paradox 4.x;	*drive:\path*
Paradox 5.x	Paradox 5.x;	*drive:\path*
ODBC	ODBC; DATABASE=*database*; UID=*user*; PWD=*password*; DSN=*datasourcename*; [LOGINTIMEOUT=*seconds*;]	None
Text	Text;	*drive:\path*

- The **Connection** property (ODBCDirect workspace only) returns the **Connection** object that corresponds to the **Database** object. Note that the **Connection** object and **Database** object are two different variables that correspond to the same open database.
- The **DesignMasterID** property (Jet workspace only) returns a 16-byte value that corresponds to a *Design Master* in a replica set. A Design Master is a database to which system tables, system fields, and replication properties have been added, and it represents the first replica in a replica set. I discuss replicas throughout the chapter.
- The **QueryTimeOut** property is an integer that sets how long a query will run before timing out. A value of 0 specifies no limit.
- The **RecordsAffected** property returns the number of records that were affected by the most recent invocation of the **Execute** method. For example, if you were to run the following

query on the sample **Employee** table, the **RecordsAffected** property would return 35 because there are 35 rows on the **Employee** table:

```
Update Employee Set Emp_Sal = Emp_Sal * 1.08
```

• The **Replicable** property (Jet workspace only) must first be created by using the **CreateProperty** method (discussed under the **Properties** topic earlier in this chapter). It must be a **Text** data type. If the value is “T”, the database becomes replicable. Once the property is set, it cannot be changed. The following code illustrates this process:

```
Dim prpReplica
Set prpReplica = .CreateProperty ("Replicable", dbText, "T")
dbEmp.Properties.Append prpReplica
dbEmp.Properties ("Replicable") = "T"
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Microsoft Jet And Replication

One of the neater features of Microsoft databases in general and Microsoft Jet in particular is the ability to perform database replication. *Replication* refers to the process of being able to distribute portions or all of a database to distributed locations and then merge back all the changes to the master database. For instance, a traveling salesperson might have a copy of the database on his or her laptop computer and then be able to add and maintain customers and take new orders. The changes to the replicated database can be merged back into the main database at a later time.

A *replica* is a copy of a database, including all of its objects (tables, queries, and so on) and is a member of a *replica set*. The replica set is all of the replicated copies of the database. Changes in one replica are synchronized and applied to all other replicas in the replica set.

When you set the **Replicable** property of a Jet database, you should first make a backup of the database because a lot of changes will be done immediately. When you set this property on a **Database** object, Jet adds fields, tables, and properties to all database objects. All tables, as an example, add three new columns that help Jet determine what rows have changed so they can then be merged back into other copies of the database.

If you inherit an object and that object has a **Replicable** property, you might assume that the inherited object will have the same **Replicable** value. This is not accurate. You still need to explicitly set the value. I discussed the concept of inheriting properties earlier in the chapter in the discussion of **Properties** collections and the **Property** object.

- **ReplicaID** (Jet workspace only) returns a 128-bit Globally Unique Identifier (GUID). In terms of a replicable database, the GUID on both the server and the client must match for the two to bind. The **ReplicaID**, then, is a unique identifier of the replica itself. It returns a reference to the Design Master, which is the first replica in a replica set. Different replicas can serve as the Design Master, but there can only be one Design Master at a time. This value is stored in the **MSysReplicas** system table in the database.

- **Updatable** is a Boolean that returns **True** if the underlying object is updatable. There are a number of reasons an object might not be updatable, from the mode in which it was opened (such as read-only) to the nature of the query itself (not having a primary key or a join on several tables) to the nature of the record set type (such as snapshot).
- **V1XNullBehavior** (Jet workspace only) is applicable to Jet version 1 databases converted to later versions. It is a Boolean that, if set to **True**, indicates zero-length text fields will be converted to **Null**.
- **Version** returns the ODBC driver version on ODBCDirect work-spaces, but it returns the version of Jet on Jet workspaces.

The **Database** object has the following methods:

- **Close** closes the object. You should be sure to first close all **RecordSet** objects.
- **CreateProperty** (Jet workspace only) adds a property to the **Properties** collection. I discussed this method earlier.
- **CreateQueryDef** creates a **QueryDef** object and appends it to the **QueryDefs** collection. The syntax is shown in the next code example. **object** is a valid **Database** or **Connection** object. **sql** is a string containing an SQL statement. In an ODBCDirect workspace, it can also contain the name of a stored procedure on a Microsoft SQL Server database. You can leave **sql** blank and set the **SQL** property of the object after you create it. We discuss the **QueryDef** object later in this chapter.

```
Set QueryDef = object.CreateQueryDef (name, sql)
```

- **CreateRelation** (Jet workspace only) creates a **Relation** object that specifies the relationship between two objects in **TableDef** or **QueryDef** objects. I discuss the **Relation** object later in this chapter. The syntax of the statement is shown in the following code segment. **db** is a valid **Database** object. **table** and **foreign_table** are **Variant**s of type **String** that identify the two tables involved in the relationship. **attributes** is an optional argument that specifies how two tables are related. See the discussion of the **Relationship** object for more information.

```
Set relation = db.CreateRelation (name, table, _
  foreign_table, attributes)
```

- **CreateTableDef** (Jet workspace only) creates a **TableDef**, which I discuss later in this chapter. The syntax of the statement is shown in the next code segment. **db** is a valid **Database** object. **attributes** is one or more constants that describe attributes of the object. **source** is the name of the table in the underlying database that the object refers to. **connect** is a string that describes the connection. See the discussion of the **Connect** property of the **Database** object for more information.

```
Set tabledef = db.CreateTableDef (name, attributes,  _
  source,  connect)
```

- **Execute** runs an SQL statement such as **SQL SELECT** or **UPDATE**. Two variations on the statement appear in the following code example. In the first example, **object** is a valid **Database** or **Connection** object. **source** is a string that contains either an SQL statement or the name of a **QueryDef** object. **options** is one or more constants (as listed in Table 5.6) that specify the data integrity characteristics of the query. The second form of the statement specifies the

QueryDef object and thus omits the **name** argument. **Execute** does not return a record set. For best performance, I recommend issuing a **BeginTrans** before the **Execute** method. Immediately following, perform a **Rollback** or **CommitTrans** method to end the transaction.

```
object.Execute source, options
querydef.Execute options
```

Table 5.6 Valid options constants for the **Execute** method.

Value	Description
dbDenyWrite	Other users cannot write to the object. Jet only.
dbInconsistent	(Default) Perform inconsistent updates. Jet only.
dbConsistent	Perform consistent updates. Jet only.
dbSQLPassThrough	Perform an SQL pass-through query. Jet only.
dbFailOnError	Perform rollback if an error occurs. Jet only.
dbSeeChanges	Generate runtime error if another user changes data. Jet only.
dbRunAsync	Perform asynchronous query. ODBCDirect only (**Connection** and **QueryDef**).
dbExecDirect	Do not first call ODBC **SQLPrepare** function. ODBCDirect only (**Connection** and **QueryDef**).

• **MakeReplica** (Jet workspace only) makes a new replica from another database replica. (See the discussion of **Replicable** and **ReplicaID** properties earlier in this section.) The syntax for the method is shown in the next code segment. **database** is a valid **Database** object. **replica** is a string containing the full path and name of the new replica. It cannot already exist. **description** is a string describing the replica. This user-defined argument might be something such as “Replica of California Customers”. **options** is one or both of the constants **dbRepMakePartial**, which specifies the replica is a partial replica (not all records are included). You need to set the **ReplicateFilter** properties of the **TableDef** object to determine what data will actually be replicated. Initially, all properties are set to **False**, meaning no data will be replicated. **dbRepMakeReadOnly** prevents users from changing the data in the replica. However, when synchronized with other replicas, the changes in those replicas will be applied to the new replica. This is a useful option when replicating data that is normally static (such as a ZIP code table).

```
database.MakeReplica replica, description, options
```

• **NewPassword** (Jet workspace only) creates a new password for the object. See the discussion of this method earlier in the chapter under the **User** object.

• **OpenRecordSet** opens a new **RecordSet** object and appends it to the **RecordSets** collection. I discuss the **RecordSet** object later in this chapter. Two variations on this method are shown in the next code example. **recordset** is the name of a valid object variable of type **RecordSet**. **object** is a valid **Database** or **Connection** object in the first example or a valid **QueryDef**, **TableDef**, or **RecordSet** object in the second example. **source** is a string that specifies the source of the data. For Jet workspace table-type record sets, this can only be a table name. For ODBCDirect workspaces, this can be a table name, a query name, or an SQL statement. **type** is the record set type, as outlined in Table 5.7. Under ODBCDirect, the default is

dbOpenForwardOnly. Under Jet, the default is **dbOpenTable**. If Jet cannot open this type or if you are opening an ODBC data source, the default is **dbOpenDynaset**. Valid **options** and **lockedits** are listed in Tables 5.8 and 5.9. I discuss these settings in more detail later in this chapter under "The **RecordSets** Collection And **RecordSet** Object."

```
' Database and Connection objects
Set recordset = object.OpenRecordSet (source, type, _
   options, lockedits)
' QueryDef, TableDef and RecordSet objects
Set recordset = object.OpenRecordSet (type, options, _
   lockedits)
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Table 5.7 Valid RecordSet **type** constants.

Constant	Jet	ODBCDirect	Description
dbOpenTable	Yes	No	Opens a table-type record set.
dbOpenDynamic	No	Yes	Opens a dynamic-type record set.
dbOpenDynaset	Yes	Yes	Opens a dynaset-type record set.
dbOpenSnapshot	Yes	Yes	Opens a snapshot-type record set.
dbOpenForwardOnly	Yes	Yes	Opens a forward-only-type record set object.

Table 5.8 Valid RecordSet option constants.

Constant	Jet	ODBCDirect	Description
dbAppendOnly	Yes	No	User can only append new records. Dynaset only.
dbSQLPassThrough	Yes	No	Passes an SQL statement to ODBC for processing. Snapshot only.
dbSeeChanges	Yes	No	Generates an error if another user changes data. Dynaset only.
dbDenyWrite	Yes	No	Prevents other users from modifying or adding records.

dbDenyRead	Yes	No	Prevents other users from reading data in a table.
dbForwardOnly	Yes	No	Creates a forward-only record set. Snapshot only.
dbReadOnly	Yes	No	Prevents changes to the record set.
dbRunAsync	Yes	No	Runs query asynchronously.
dbExecDirect	No	Yes	Does not first call ODBC **SQLPrepare** function.
dbInconsistent	Yes	No	Allows inconsistent updates. Dynaset, snapshot only.
dbConsistent	Yes	No	Allows only consistent updates. Dynaset, snapshot only.

Table 5.9 Valid RecordSet lockedit constants.

Constant	Jet	ODBCDirect	Description
dbReadOnly	Yes	Default	Prevents users from making changes.
dbPessimistic	Default	Yes	Uses pessimistic locking.
dbOptimistic	Yes	Yes	Uses optimistic locking.
dbOptimisticValue	No	Yes	Uses optimistic concurrency based on row values.
dbOptimisticBatch	No	Yes	Enables batch optimistic updating.

- **PopulatePartial** (Jet workspace only) populates a partial replica, synchronizing it with the Design Master. It takes only the argument of the replica to synchronize as a string containing a path and name of the replica (see **MakeReplica**). Based on the values of the **Filter** properties of the **TableDef** objects, the replica is populated with data based on the current **DataBase** object.
- **Synchronize** (Jet workspace only) synchronizes two replicas. The syntax is shown in the next code example. The **database** argument is a valid **Database** object. **pathname** is a **String** containing the path and name of the replica. **exchange** is a constant indicating which way to replicate. **dbRepExportChanges** sends changes from the **Database** object to the replica. **dbRepImpChanges** sends changes from the replica to the **Database** object. **dbRepImpExpChanges**, which is the default, sends changes both ways. **dbRepSyncInternet** replicates changes between two files connected by an Internet pathway. To use

this, specify a URL instead of a file name and path in the **pathname** argument. Note that when you replicate changes, changes to the design of the database are always replicated first. This occurs even if replication is going only one way. The two replicas cannot have the same **ReplicaID**. If synchronizing partial replicas with other partial replicas, you must use the **PopulatePartial** method instead.

```
database.Synchronize pathname, exchange
```

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Previous | Table of Contents | Next

The Connections Collection And Connection Object

The **Connections** collection is an ODBCDirect workspace construct that encompasses the functionality of the **Databases** collection and adds additional functionality. If you create a **Database** object in an ODBCDirect workspace, a corresponding **Connection** object is also created and added to the **Connections** collection. Conversely, if you create a new **Connection** object, a corresponding **Database** object is also created and added to the **Databases** collection. To create a new **Connection** object, see the **CreateConnection** method of the **Workspace** object discussed earlier in this chapter. **Connection** objects are automatically appended to the collection. Because you can open the same **Connection** object more than once, its **Name** may appear more than once in the collection.

The **Connection** object itself represents a connection to an ODBC data source. It contains two collections (see Figure 5.2): **QueryDefs** (the default) and **RecordSets**. The **Connection** object has a number of properties, several of which are in common with the **Database** object and were discussed earlier in this chapter: **Connect**, **QueryTimeOut**, **RecordsAffected**, and **Updatable**. Other properties of the **Connection** object include the following:

- **Database** returns an object variable that is a reference to the corresponding **Database** object.
- **StillExecuting** is a Boolean that indicates whether a query run asynchronously is still executing. It is available when invoking the **Execute** or **CreateConnection** methods.
- **Transactions** is a Boolean that returns **True** if the underlying data source supports transactions.

The **Connection** object also has a number of methods in common with the **Database** object: **Close**, **CreateQueryDef**, **Execute**, and **OpenRecordSet**. I discussed these methods earlier in this chapter under the **Database** object

topic. **Connection** also has a **Cancel** method that you can use to cancel a currently executing asynchronous query: **myConnection.Cancel**.

The Containers Collection And Container Object

The **Containers** collection is a Jet-workspace-only construct representing all of the **Container** objects in a **Database** object. Some of these **Container** objects are defined by the Jet engine, whereas others are defined by other applications. Because **Container** objects are automatically appended to the **Containers** collection, the collection has only a **Refresh** method and a **Count** property.

The purpose of the **Container** object is to group similar types of **Document** objects. For the **Database** object, the **Container** contains information about saved databases. For the **Table** object, the **Container** maintains information about tables and queries. For the **Relationship** object, **Container** maintains information about saved relationships.

The **Container** object has two collections: **Properties** and **Documents** (which is the default). It has the following properties:

- **AllPermissions** returns all of the permissions of the current **UserName** property, as listed in Table 5.10. Some of these permissions may be specific to the user; others may be inherited from the group to which he or she belongs. The **Document** object also has this property, although the possible permissions are slightly different. See also the **Permissions** property.

Table 5.10 Container and Document permission constants.

Constant	Container	Document	Description
dbSecDBAdmin	No	Yes	User can replicate the database and change the password.
dbSecDBCreate	No	Yes	User can create new databases. This setting is valid only on the **Databases** container in the workgroup information file (System.MDW).
dbSecDBExclusive	No	Yes	User has exclusive access to the database.
dbSecDBOpen	No	Yes	User can open the database.
dbSecDeleteData	Yes	Yes	User can delete records.
dbSecInsertData	Yes	Yes	User can add records.
dbSecReadDef	Yes	Yes	User can read the table definition, including column and index information.

dbSecReplaceData	Yes	Yes	User can modify records.
dbSecRetrieveData	Yes	Yes	User can retrieve data from the **Document** object.
dbSecWriteDef	Yes	Yes	User can modify or delete the table definition, including column and index information.

- The **Inherit** property indicates whether new **Document** objects will inherit the **Permissions** property from this object. Set it to **True** to cause new objects to inherit the **Permissions** property.
- The **Owner** property is a string that is the name of an object in either the **Groups** or **Users** collection and represents the owner of this object. As such, he or she has the privilege of changing the **Owner** property, assuming other permissions permit this.
- The **Permissions** property is a long containing constants indicating which permissions have been assigned to the **UserName** property. This does not include permissions inherited by membership in a group (see **AllPermissions**). These permissions are summarized in Table 5.11.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Table 5.11 Valid Permissions constants.

Constant	Table	Database	Other	Description
dbSecCreate	Yes	No	No	User can create new documents.*
dbSecDBAdmin	No	Yes	No	User can replicate database and change password.*
dbSecDBCreate	No	Yes	No	User can create new databases.*
dbSecDBExclusive	No	Yes	No	User has exclusive access to the database.
dbSecDBOpen	No	Yes	No	User can open the database.
dbSecDeleteData	Yes	No	No	User can delete records.
dbSecDelete	No	No	Yes	User can delete the object.
dbSecFullAccess	No	No	Yes	User has full access to the object.
dbSecInsertData	Yes	No	No	User can add records.
dbSecNoAccess	No	No	Yes	User doesn't have access to the object.*
dbSecReadSec	No	No	Yes	User can read the object's security-related information.
dbSecReadDef	Yes	No	No	User can read the table definition.
dbSecReplaceData	Yes	No	No	User can modify records.
dbSecRetrieveData	Yes	No	No	User can retrieve data from the **Document** object.
dbSecWriteOwner	No	No	Yes	User can change the **Owner** property setting.

dbSecWriteSec	No	No	Yes	User can alter access permissions.
dbSecWriteDef	Yes	No	No	User can modify or delete the table definition.

*Not valid for **Document** objects.*

- **UserName** is a string representing the user of the object.

The **Container** object has no methods.

The Documents Collection And Document Object

The **Documents** collection is a Jet-workspace-only construct belonging to the **Container** object. **Document** objects are automatically appended to the **Documents** collection. **Document** objects describe instances of objects specified by the owner **Container** object. With the properties discussed in this section, you can obtain information about saved databases (in a database **Container**), tables (in a table **Container**), and relationships (in a relationship **Container**).

The **Document** object has one collection: **Properties**.

The **Document** object has a number of properties in common with its **Container** object: **AllPermissions**, **Name**, **Owner**, **Permissions**, and **UserName**. See the discussion of the **Container** object earlier in this chapter for explanations of these properties. Other properties include the following:

- **Container** returns an object variable that is a reference to the parent **Container** object.
- **DateCreated** returns the date and time that a table was created for table-type record sets (Jet workspace only) or that the object was created in all other cases. **LastUpdated** returns the date and time the table or object was last updated. Both values are of type **Variant** with an underlying data type of **Date**.
- **KeepLocal** is similar to the **Replicable** property of the **Database**, **Document**, **TableDef**, and **QueryDef** objects. It is a user-defined property and must therefore be created using the **CreateProperty** method discussed earlier in this chapter for the **Property** object. You should create this property before making an object replicable. Then, this **Document** object will not be replicated with the other objects. The following code segment is adapted from the Microsoft VB help file and shows **KeepLocal** being used on the Northwind database:

```
Dim dbNorthwind As Database
Dim docTemp As Document
Dim prpTemp As Property
Set dbNorthwind = OpenDatabase("Northwind.mdb")
Set docTemp = dbNorthwind.Containers("Modules"). _
   Documents("Utility Functions")
Set prpTemp = docTemp.CreateProperty("KeepLocal", _
   dbText, "T")
' Set an existing document
docEmp.Properties("KeepLocal") = "T"
```

```
    docTemp.Properties.Append prpTemp
    dbNorthwind.Close
```

- **Replicable** is a user-defined property that determines whether the object can be replicated. I discussed this method under the **Database** topic earlier in this chapter.

The **Document** object has no methods.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The TableDefs Collection And TableDef Object

The **TableDefs** collection is a Jet workspace construct belonging to the **Database** object. With the individual **TableDef** objects, you can manipulate a table definition, including validation rules. When you create a **TableDef** object, you must append it to the **TableDefs** collection. The **TableDef** object has three collections: **Properties**, **Fields**, and **Indexes** (see Figure 5.1).

To create a **TableDef** object, you invoke the **Database** object's **CreateTableDef** method, which I discussed earlier in this chapter.

The **TableDef** object has some properties in common with the **Document** object, discussed earlier in the chapter: **DateCreated**, **KeepLocal**, **LastUpdated**, **Name**, and **Replicable**. It also has the following properties:

- **Attributes** is a long consisting of one or more constants describing various attributes of the object, as listed in Table 5.12. You can write to this property even if the object has not been appended to the collection. Note that attributes of the individual columns are stored in the **Attributes** property of the individual **Field** objects. Attributes of the relationships are stored in the **Attributes** property of the individual **Relationship** objects.

Table 5.12 Valid Attributes constants of the TableDef object.

Constant	Description
dbAttachExclusive	The table is a linked table opened for exclusive use.[1]
dbAttachSavePWD	Save the user ID and password with the connection information.[1]
dbSystemObject	Table is a system table provided by Jet.[2]
dbHiddenObject	Table is a hidden table provided by Jet.[2]
dbAttachedTable	Table is a linked table from a non-ODBC data source. Read-only.
dbAttachedODBC	Table is a linked table from an ODBC data source. Read-only.

[1] *Not valid for local tables.*
[2] *You can set this constant on an appended **TableDef** object.*

- **ConflictTable** is a string containing the name of the table that caused an error during a replica synchronization process. If this is a zero-length string, there was no conflict. This

property is useful when two users make changes to the same record. The update of one record will be replicated to the other, but the second user's changes will not be replicated. This property tells you the name of the table that contains the conflict information so that the problem can be resolved. The table name will be suffixed with "_conflict". For instance, if a conflict occurs on the **Employee** table, this value will be **Employee_conflict**.

- **Connect** is a string containing connect information. I discussed this property earlier in the chapter in the **Database** topic.
- **RecordCount** is a long containing the total number of records in the object. If the table is a linked table, then the **RecordCount** is always -1.
- The **ReplicaFilter** is used with partial replicas (I discussed this earlier in the chapter when I discussed the **Database** object). The three settings are **True** (replicate all records), **False** (don't replicate any records), or a string specifying a filter to use to determine which records to replicate. It is specified similar to the **RecordSet** object's **Find** criteria argument. You cannot use any aggregate functions or user-defined functions. Before changing the filter, be sure to execute the **Synchronize** method. After changing the filter, invoke the **ReplicatePartial** method. Listing 5.4 replicates only the employees from the state of California to a partial replica contained in the "California.MDB" file.

Listing 5.4 Using ReplicaFilter and performing a partial replication.

```
Dim tdfEmp As TableDef
Dim sFilter As String
Dim dbCoriolis As Database

' Open the database
Set dbCoriolis = OpenDatabase("Coriolis.mdb")

' Define the TableDef as the employees table
Set tdfEmp = dbCoriolis.TableDefs("Employees")
' Synchronize first! This is a full replica
dbCoriolis.Synchronize "California.mdb"

' Set the replica filter
sFilter = "Emp_St = 'CA'"

' Now do the partial replication
tdfEmp.ReplicaFilter = _
   sFilter dbCoriolis.PopulatePartial "California.mdb"
```

- The **SourceNameTable** property is a string that specifies the name of a linked table or a base table in a Jet workspace. In other words, it is the name of the underlying table of the **TableDef** object.
- The **ValidationRule** for a **TableDef** object differs from that for a **Field** object in that it can apply to multiple fields in the table. The **ValidationRule** property allows you to validate values entered or maintained by the user on the underlying object (the table, in this case) by creating a rule similar to an SQL **WHERE** clause without the **WHERE** keyword. The data entered by the user must conform to the rule set in the **ValidationRule** or the change is rejected and the message entered into the **ValidationText** property is displayed. For example, the following code enforces the rule for the **Item** table that price must be greater than cost and that the cost must be greater than zero:

```
' tdfItem is an existing TableDef
tdfItem.ValidationRule = Str$(Item_Price) & " > " & _
```

```
Str$(Item_Cost) & " AND " & Str$(Item_Cost) & " > 0"
```

- The **ValidationText** property sets the text of a message that will be displayed if an edit does not pass the **ValidationRule**. The following code sets the property:

```
tdfItem.ValidationText = "Price must be greater than" & _
   "cost and cost must be greater than 0!"
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The **TableDef** object has a number of methods:

- The **CreateField** method creates a new **Field** object, which you must append to the **Fields** collection manually. I discuss the **Field** object later in this chapter. You cannot delete a **Field** object from the collection if an **Index** object is referencing it. The syntax for the **CreateField** method is shown in the next code example. **object** is an existing **Field**, **TableDef**, or **Relation** object. **type** is the data type of the new **Field** object. Table 5.3 earlier in the chapter listed the valid data types. **size** is a **Variant** (subtype **Integer**) that is used to set the maximum size of a text field.

```
Set field = object.CreateField (name, type, size)
```

- The **CreateIndex** method creates a new **Index** object. The syntax is shown in the next code example. You must manually append the **Index** object to the **Indexes** collection. I discuss **Index** later in this chapter.

```
Set Index = TableDef.CreateIndex (name)
```

- **OpenRecordSet** opens a new **RecordSet** object.
- The **RefreshLink** method allows you to refresh connection information. For instance, you could change the **Connect** property to attach to a different data source. When you do, you should issue the **RefreshLink** method to physically attach to the new data source. Note that once you change a data source, you need to use the **Refresh** method on all collections (**Fields** and **Indexes**) belonging to the **TableDef** object so that they will also be in sync.

The Fields Collection And Field Object

The **Fields** collection is a construct belonging to a **TableDef**, **QueryDef**, **RecordSet**, **Relation**, or **Index** object (see Figures 5.1 and 5.2). It represents all of the fields associated with that object. Whereas a **TableDef** object represents a table within a database (represented by the **Database** object), the **Fields** collection of

Field objects represents all of the fields or columns on that table. For an **Index** object, the **Fields** collection is the database fields represented in that index.

Within a **RecordSet** object, you use the **Fields** collection to read and set values from the current record. For other objects, such as **TableDef**, the **Field** objects are the specifications of the field (such as data type).

When referring to a **Field** object in the collection, you can reference it by its **Name** property or by its ordinal number (as in any other collection). However, the ordinal number represents the actual order in which the fields appear within the owning object. For instance, if you were to append to the **Fields** collection objects representing **Emp_Salary**, **Emp_LName**, and then **Emp_FName**, **Emp_LName** would be ordinal number 1 (the collection is zero-based) and would appear to be the second column on the **Employee** table.

The **Field** object has a number of properties that you use depending on its context. For example, **FieldSize** is an important property in terms of the **TableDef** object, whereas **Value** is a frequently used property when dealing with **Field** objects belonging to the **RecordSet** object. As such, different properties are available depending on the owner of the **Field** object. These are summarized in Table 5.13.

- **AllowZeroLength** is a Boolean that sets whether a text or memo field can contain a zero-length string.
- **Attributes** is a **Long** containing one or more constants with which you can specify various attributes of the **Field** object, such as whether it can be updated (**dbUpdatableField**). **dbAutoIncrField** causes the value of the field in new records to increment to a new **Long** value; this is useful for primary keys, such as a customer number. **dbDescending** specifies that a **Field** object in an **Index** will sort in descending order. **dbFixedField** specifies that the field is fixed length. This is the default for numeric values. **dbVariableField** specifies that the field is variable length. **dbHyperLinkField** is used with a memo field to specify that it contains a hyperlink. **dbSystemField** specifies that the field contains replica information.
- **CollatingOrder** is a **Long** containing a constant that specifies a collation sequence (sort order).
- **DataUpdatable** is a Boolean indicating whether the field can be updated. **True** indicates the field is updatable.
- **DefaultValue** is a string of up to 255 characters specifying a default value for the field. The string can contain the special value **GenUniqueID()**, which causes a unique random number to be assigned as a value of the **Field** object. The string can also contain any expression except a user-defined function.
- **FieldSize** returns a **Long** indicating the size of the **Field** object. For a memo field, this is the number of characters. Note that the number of characters and number of bytes may not be the same, depending on whether ANSI or multibyte characters are used. For a numeric field, it is the number of bytes.
- **ForeignName** is used in referential integrity relationships. **ForeignName** contains the name of a **Relation** object. If used, the **Relation** object's **TableName** property contains the name of the table on whose primary key this **Field** object is dependent. For example, if the **Field** object is part of the **Orders** table containing the definition of the **Cust_No** field, then the

ForeignName property is set to the name of that **Relation** object whose **TableName** property is set to **Customer**. Thus, the **Value** property of the **Field** object is constrained to already existing **Cust_No** values on the **Customer** table.

- **OrdinalPosition** is an integer that sets or returns the relative position of the **Field** object within the **Fields** collection.
- **OriginalValue** contains the original value of the **Field** object when first retrieved. This is useful during batch updates so that you can determine whether another process has altered a record in between the time that you retrieve it and the time you update it. If a change occurred, then the new value of the field is shown in the **VisibleValue** property. When the **VisibleValue** is different from the **OriginalValue**, a *collision* is said to have occurred. I talk at more length of transaction and concurrency issues in Chapter 12. Listing 5.8 illustrates some techniques to handle collisions.
- **Required** is a Boolean indicating whether the field must contain a non-null value. Setting this property to **True** is the equivalent of specifying **Not Null** in a table **Create** statement.
- **Size** is used with the text data type to set a maximum number of characters (up to 255 characters). For most data types, the **Type** value determines the maximum size and this property is set automatically. See also the **FieldSize** property.
- **SourceField** and **SourceTable** set or return the name of the field (or column) and file (or table) to which this **Field** object corresponds. If the object is used for a customer's last name, then **SourceField** would be **Cust_LName** and the **SourceTable** would be **Customer**.
- **Type** sets or returns the data type of the **Field** object. It is a constant as listed in Table 5.3 earlier in this chapter.
- **ValidateOnSet** is a Boolean that dictates when the data will be validated. If **True**, the data is validated as soon as it is set. If **False**, it is validated only when being updated to the table.
- **ValidationRule** was discussed earlier in this chapter under the **TableDef** object. However, unlike the **ValidationRule** property of the **TableDef** object, the **ValidationRule** can only reference the **Field** object to which it applies. In other words, the **ValidationRule** property of the "**Cust_LName**" **Field** object cannot reference the "**Cust_FName**" **Field** object.
- **ValidationText** sets or returns the text that will be displayed if data does not pass validation. See the **ValidationRule** example earlier in this chapter under the **TableDef** topic.
- **Value** contains the value of the **Field** object. For instance, the "**Cust_LName**" **Field** object might have a **Value** of "**Smith**".
- **VisibleValue** is used in conjunction with the **OriginalValue** property discussed in concurrency management in batch updates.

Table 5.13 Properties of the Field object and when they are available.

Property	Index	QueryDef	RecordSet	Relation	TableDef
AllowZeroLength	No	No	Jet only	Jet only	Jet only
Attributes	Yes	Yes	Yes	No	Yes

CollatingOrder	No	Jet only	Jet only	No	No
DataUpdatable	Yes	Yes	Yes	Yes	Yes
DefaultValue	No	Jet only	Jet only	No	Jet only
FieldSize	No	No	Yes	No	No
ForeignName	No	No	No	Yes	No
Name	Yes	Yes	Yes	Yes	Yes
OrdinalPosition	No	Yes	Yes	No	Jet only
OriginalValue	No	No	ODBCDirect*	No	No
Required	Yes	Yes	Yes	No	Yes
Size	No	Yes	Yes	No	Yes
SourceField	No	Yes	Yes	No	Yes
SourceTable	No	Yes	Yes	No	Yes
Type	Yes	Yes	Yes	No	Yes
ValidateOnSet	No	No	Jet only	No	No
ValidationRule	No	Jet only	Jet only	No	Yes
ValidationText	No	Jet only	Jet only	No	Yes
Value	Yes	No	Yes	No	No
VisibleValue	No	No	ODBCDirect*	No	No

Only if **DefaultCursorDriver is **dbUseClientBatchCursor**.*

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The **Field** object has three methods. I discussed the **CreateProperty** method earlier in this chapter in the **Properties** collection section. The other two methods help in dealing with large memo or long binary fields. **GetChunk** retrieves all or a portion of such a field. **AppendChunk** adds data to the end of a memo or long binary field. Note that the first time you use **AppendChunk** on a field, the value of the field is replaced. Subsequent calls to **AppendChunk** then add to the end of the field. **GetChunk** uses the syntax shown in the following code segment. **variable** is a **Variant** of type **String**. **RecordSet** is a valid **RecordSet** object. **Field** is a valid **Field** object. **offset** is a **Long** specifying from which byte to start copying, and **length** is how many bytes to copy. The syntax for **AppendChunk** is also shown. **source** is a **Variant** of type **String** from which data is to be copied to the **Field** object.

```
Set variable = RecordSet ! Field.GetChunk (offset, length)
Recordset!Field.AppendChunk, source
```

The Indexes Collection And Index Object

The **Indexes** collection contains all of the **Index** objects of a **TableDef** object in a Jet workspace (see Figure 5.1). You create an **Index** object using the **TableDef** object's **CreateIndex** method. You should then use the **Append** method to append it to the collection. However, you can only do that if the **TableDef** is updatable (as denoted by the **Updatable** property of the **TableDef**).

An index is used on a database either to speed up access or to act as an integrity constraint. I discussed indexes in Chapter 2. You can place an index on one or more columns in a table, such as **Cust_LName** and **Cust_FName**. When ordering by **Cust_LName** or by both **Cust_LName** and **Cust_FName**, the database will use the indexes to tremendously improve performance. An index can be unique, in which case it also acts as an integrity constraint by preventing any duplicate values in the column or columns on which the index is defined. For instance, if a unique index were created on the last and first name fields of the **Customer** table, the database would allow

duplicate values of "**Smith**" in **Cust_LName**, but it would allow only one combination of "**Smith**" in **Cust_LName** and "**John**" in **Cust_FName**.

An index becomes "active" only when it is appended to the **Indexes** collection. If the index is unique and there is a duplicate value in the column or columns covered, then a trappable error occurs. Because the index becomes active when appended to the collection, you must set all of its properties before appending it.

An **Index** object has two collections: I discussed **Properties** earlier in this chapter and **Fields** right before this topic. The **Fields** collection is used, of course, to specify what columns comprise the index.

The **Index** object has the following properties, which you use to maintain and manipulate the index. Listing 5.5 demonstrates the use of most of the properties.

- **Clustered** is a Boolean indicating whether the index is clustered. Some ISAM files and most relational databases support clustering of data. A table may have only one clustered index on it. The clustering specifies in what order the data is physically stored on the table. Microsoft databases (MDB files) do not support clustering. You should consult your database documentation for whether your database does support clustering and what restrictions are enforced. For example, most (but not all) databases require that the clustered index be specified when the table is created or before any data is added to the table. ODBC data sources always return **False** for this property; the clustered index is not detected. By and large, you are better off maintaining the clustering information at the database level and not within your Visual Basic program. Because of performance considerations, you should also consult with your DBA.
- **DistinctCount** is a **Long** that returns the number of unique values in the index. If the index is on one column, then the property is equivalent to the number of unique values in that column. If the index covers more than one column (called a *compound index* or *composite index*), then the property is the number of unique combinations of those columns. Note that this property may not always reflect the actual number of distinct properties. This information is typically maintained in the database's system tables. Some databases have an **Update Statistics** command that should be run periodically to update this type of information (which is helpful to the query optimizer). From Visual Basic, you should use the **CreateIndex** method if you need this property to be exact.
- **Foreign** is a Boolean indicating whether this index references a column in another table in a referential integrity relationship. In other words, if this property is **True**, the index represents a foreign key to a parent table.
- The **IgnoreNulls** property is a Boolean that determines how null values are handled. If a field allows null values, you can set this property to **False** to ignore those null values and thus speed searches. You use this property in association with the **Required** property, which is a Boolean that determines if null values are allowed. **True** is the equivalent of the **Not Null** clause in the **CREATE TABLE** statement. If **IgnoreNulls** is **True** and **Required** is **False**, then null values are allowed but are not reflected in the index. If **IgnoreNulls** and **Required** are both **False**, then null values are allowed and are also reflected in the index. If **Required** is **True**, then the **IgnoreNulls** property is irrelevant because there can be no null values.

- The **Primary** property is a Boolean indicating whether the index is the primary key of the table. A table can have only one primary key, and by definition, it is unique.
- The **Unique** property is a Boolean indicating whether the index is unique.

NOTE
An **Index** object does not specify in what order the data is *stored*, only in what order the data will be returned when the index is used in an **Order By** clause.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Listing 5.5 demonstrates the use of the **Index** object using the sample Access database supplied on the CD-ROM.

Listing 5.5 Using the **Index** object.

```
' Declare variables
Dim dbCoriolis As Database
Dim tdfEmployee As TableDef
Dim idxName As Index

' Open the database object
Set dbCoriolis = OpenDatabase("Coriolis.mdb")
' Open the Employee table
Set tdfEmployee = dbCoriolis!Employee

' Create a new index object
Set idxName = tdfEmployee.CreateIndex("Name")
' Set properties as needed
With idxName
  ' Create a non-unique index on the employee
  ' last name and first name fields
  .Fields.Append .CreateField("Emp_LName")
  .Fields.Append .CreateField("Emp_FName")
  ' Not unique
  .Unique = False
  ' Do not ignore null values
  .IgnoreNulls = False
End With

' Make the index active (builds the index)
tdfEmployee.Indexes.Append idxName
```

```
' Display how many unique values there are
MsgBox "There are " & _
  tdfEmployee.Indexes.Index ("Name").DistinctCount & _
  " unique values."
```

The **Index** object also has two methods: **CreateProperty** and **CreateField**. Both were discussed earlier in the chapter, in the **Properties** collection and **TableDefs** collection sections, respectively.

The Relations Collection And The Relation Object

The **Relations** collection contains all of the **Relation** objects of a **Database** object and is created using the **CreateRelation** method, which I discussed earlier in this chapter in the **Database** object section. After creating a **Relation** object, you should append it to the **Relations** collection.

A **Relation** object sets or returns how columns in a database relate to one another. You can determine if the relationship is one-to-one or one-to-many. You can also specify how columns are to be used when joining two tables. The default is a *natural* or *inner* join, which simply specifies that data is returned from both tables if the join condition is satisfied. However, you can also specify the use of an outer join. An outer join states that even if the join condition is not satisfied, some data is to be returned. Consider the following selection:

```
SELECT DEPT_NO, DEPT_NAME, EMP_NO, EMP_FNAME, EMP_LNAME
FROM DEPARTMENT, EMPLOYEE
WHERE DEPT_NO = EMP_DEPT_NO
```

If a department did not happen to have any employees in it, it would not be listed because the join condition would not be satisfied. We can solve that by creating an outer join. A left outer join is how we indicate that it is okay if the information on the left-hand side of the comparison is missing; list the other information anyway. In this example, a left outer join states that even if a department number is missing on the department table, list all employees. A right outer join is just the opposite. It indicates that even if no employees are in a particular department, the department information should be returned. We can perform outer joins using the **Attributes** property of the **Relation** object.

The **Relation** object has two collections (see Figure 5.1)—**Fields** and **Properties**—which were discussed earlier in this chapter.

The **Relation** object has these properties:

- The **Attributes** property is one or more constants and describes the relationship between the **Field** objects in the **Fields** collection associated with this **Relation** object. **dbRelationUnique** specifies the relationship is one-to-one. **dbRelationDontEnforce** means there is no referential integrity enforced. **dbRelationInherited** means that the relationship is enforced in some other database than the current database (this is a Microsoft Access concept only). **dbRelation-UpdateCascade** specifies that any updates to a value will cascade down to dependent values. For instance, if a customer number is changed on the **Customer** table, then all of the customer

numbers on the referenced **Orders** table will also be updated to reflect the change. **dbRelationCascade** is similar, specifying that if a parent row is deleted, all child rows will also be deleted. For example, if a customer is deleted, all of the customer's orders will also be deleted. **dbRelationLeft** and **dbRelationRight** are used with Microsoft Access to specify a left outer join or a right outer join as the default join type.

- The **Table** and **ForeignTable** properties specify the parent and child tables in a relationship. The **Attributes** property describes how to relate them. Use the **Fields** collection to set or determine what fields are involved in the relationship.
- The **PartialReplica** is a Boolean that, if **True**, specifies that the **Relation** should be considered when populating a partial replica from a full replica. This has implications if a row in a partial replica has its parent or child link broken because the partial replica does not include the parent or child information. If set to **True**, associated information will be included in the partial replica, even if it's not defined to be a part of it in the **TableDef** object's **PartialReplica** property.

The **Field** object has one method, **CreateField**, which I discussed earlier in the chapter in the **TableDefs** topic.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The QueryDefs Collection And QueryDef Object

The **QueryDefs** collection maintains all **QueryDef** objects of a **Database** object (see Figure 5.1) or a **Connection** object (see Figure 5.2). The **QueryDef** object represents a saved query. To create a **QueryDef** object, use the **CreateQueryDef** method, which I discussed earlier this chapter when I discussed the **Databases** collection. In a Jet workspace, the **QueryDef** object is permanent (that is, it is saved in the database). For ODBCDirect workspaces, it is a temporary object and is destroyed when the application ends or the object is unloaded from memory. **QueryDef** objects are manually appended to the collection.

The **QueryDef** object consists of the **Fields** collection (which I discussed earlier in the chapter) and the **Parameters** collection (which I discuss later in the chapter). I illustrate the use of the **QueryDef** object in Listing 5.6, later in this chapter, under the **Parameters** section.

The **QueryDef** object has a number of properties, which you can use to specify how the query will run and behave:

- The **CacheSize** property (ODBCDirect workspace only) is a **Long** that sets or returns how many rows will be cached locally. In a Jet workspace, all records are cached locally. A setting of 0 turns off caching. Otherwise, the value must be between 5 and 1,200 but cannot exceed the memory available. Also, see the **CacheStart** property of the **RecordSet** object. You can fill the cache by invoking the **FillCache** method or simply by moving through the record set. Note that the **CacheSize** property is a tuning parameter. By caching records locally, you can improve performance by eliminating how many times rows have to be retrieved from the server. However, you also don't want to set this too high because the user may not be able to interact with the data until the cache fills. A good compromise setting is 100 rows.

- The **Connect** property specifies connection parameters to the database and was discussed earlier in this chapter in the **Database** section.
- The **DateCreated** and **LastUpdated** properties (Jet workspace only) return when the object was created and when it was last updated.
- The **KeepLocal** property (Jet workspace only) applies to partial replicas. Before using this property, it must first be created, which I illustrated under **KeepLocal** earlier in the chapter in our discussion of the **Document** object. Setting this property causes the object not to be distributed to other replicas.
- **LogMessages** (Jet workspace only) causes messages returned from an ODBC database to be written to a log file. The property is a Boolean where **True** causes messages to be logged. The messages are contained in a table within the database. The name of the table is the user name appended with a sequential number such as “Admin-01” or “Admin-02”.
- **MaxRecords** is used to limit the number of records that will be returned from an ODBC data source. It is a **Long** where 0 specifies no limit. As a practical matter, you will want to limit the number of records returned by making your SQL **SELECT** statements as restrictive as possible. However, this setting is useful in limiting potentially “runaway” selects, such as when testing. Once the maximum number of records is reached, no more will be returned, even if more fit the selection criteria.
- The **ODBCTimeOut** property indicates the number of seconds that will occur before an executing query will time out. Like the **MaxRecords** property, this is useful in reining in a potential runaway query. A value of 0 specifies no limit. A value of -1 indicates that the default **Database** or **Connection QueryTimeOut** value should be used.
- The **Prepare** property (ODBCDirect workspace only) is a **Long** that indicates whether a query should be prepared before being executed. You can ask the database to prepare a statement, which is roughly equivalent to compiling it. The database analyzes the query and determines the optimum access path to the data. It is then stored temporarily on the database until run. In general, a query that is going to run in less than three seconds does not need to be prepared. Where a query is more complex or will take more time to execute, you may get somewhat better performance by preparing it before execution. Also, if you are going to run the same query repeatedly, it may make sense to prepare it first. The value **dbQPrepare** causes the query to be prepared, whereas **dbQUnPrepare** causes the query to run dynamically (without being prepared). You can override **dbQPrepare** by setting the **Execute** method’s **Options** argument to **dbExecDirect**.
- The **RecordsAffected** property is a **Long** that returns the number of records affected by the most recent **Execute**. This is the number of records added, modified, or deleted. In ODBCDirect workspaces, the value is meaningless when performing a **Drop Table** statement (the value will not reflect the number of records deleted).
- The **Replicable** property sets or returns a value that determines if the

current object should be replicated. I discussed this property's usage earlier in the chapter when I discussed the **Database** object.

- The **ReturnsRecords** property (Jet workspace only) sets or returns a Boolean that indicates whether an SQL passthrough query returns records. An SQL **UPDATE** command, for instance, does not return any records. If the query does return records, you should set this property to **True**; otherwise, set it to **False**.
- **SQL** is a string containing the query to be executed by the **QueryDef** object. In Jet workspaces, you should use Jet SQL (see Appendix B) unless you are performing a passthrough query, in which case you should use your database's SQL dialect (such as Oracle or MS SQL Server). In ODBCDirect workspaces, you use the database's dialect. (Note that the ODBC driver for your database may not necessarily implement 100 percent of your database's functionality. Refer to your ODBC driver's documentation.) The SQL statement can contain parameters. The parameters (see the discussion of the **Parameters** collection later in this chapter) must be set prior to execution of the query. You can also call a stored procedure with or without parameters in ODBCDirect workspaces.
- The **StillExecuting** property (ODBCDirect workspace only) returns a Boolean indicating whether a query being run asynchronously is still executing. You cannot manipulate data retrieved until this property is **False**.
- The **Type** property sets or returns the query type, as listed in Table 5.14. In general, this is a read-only property, meaning that DAO reports back to you the type of query stored in the **QueryDef** object.

Table 5.14 Valid QueryDef object Type values.

Constant	Jet	ODBCDirect	Description
dbQAction	Yes	Yes	A query that changes data such as **Update**.
dbQAppend	Yes	Yes	Adds records to the end of a table.
dbQCompound	Yes	No	A compound query.
dbQCrosstab	Yes	Yes	A cross-tab query.
dbQDDL	Yes	Yes	Data definition language (that is, **Create** and **Drop**).
dbQDelete	Yes	Yes	A query that deletes rows.
dbQMakeTable	Yes	Yes	A query that creates a new table from an existing record set.
dbQProcedure	No	Yes	A stored procedure.
dbQSelect	Yes	Yes	A **Select** statement.
dbQSetOperation	Yes	Yes	A **Union** query (combines two or more **Select** queries).

dbQSPTBulk	Yes	No	With **dbQSQLPassThrough**, a query that doesn't return records.
dbQSQLPassThrough	Yes	No	A passthrough query.
dbQUpdate	Yes	Yes	An action query that changes a range of records.

- The **Updatable** property returns **True** if the query definition itself can be updated, even if the resulting record set is read-only.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The **QueryDef** object has a number of useful methods:

- The **Cancel** method (ODBCDirect workspace only) cancels a currently running asynchronous query.
- The **Close** method closes the **QueryDef** object and removes it from the collection and from memory.
- The **CreateProperty** method creates a new **Property** object. We discussed this method earlier in the chapter in the **Properties** section.
- The **Execute** method runs the query and takes the syntax shown in the next code example. The query must be an action query (see the **Type** property). The **options** argument is one or more constants, as shown in Table 5.8 earlier in the chapter. After the query has been run, you can use the **RecordsAffected** property to determine how many records were modified. In Jet workspaces, you will get better performance if you place the **Execute** statement after an explicit **BeginTrans** statement. Listing 5.3 demonstrated the **BeginTrans** method.

```
querydef.Execute options
```

- The **OpenRecordSet** method, demonstrated in Listing 5.3 and discussed earlier in the chapter under the **Databases** topic, creates a new **RecordSet** object based on the **QueryDef** object and appends it to the **RecordSets** collection. The syntax is somewhat different from when used within a **Database** or **Connection** object and is shown in the next code segment. The object is a valid object variable of type **QueryDef**, **RecordSet**, or **TableDef**. **type** is the type of record set to open, as listed in Table 5.7 earlier in the chapter. **options** is one or more constants, as listed in Table 5.8, and **lock** is a constant specifying what locking strategy to use, as listed in Table 5.9 earlier in the chapter. I expand on locking and options strategies later in this chapter when I discuss the **RecordSet** object.

```
Set recordset = object.OpenRecordset (type, options, lock)
```

The Parameters Collection And Parameter Object

The **Parameters** collection contains all of the **Parameter** objects of a **QueryDef** object. A parameter is a value that is supplied to a query at runtime and can be thought of much like a

variable in your program where a value is supplied during execution. You might create a query with a placeholder similar to "Where Cust_No = ?". Here, the question mark represents a value that will be supplied before the query is actually run. You supply the value via a **Parameter** object. This allows you to create a stored query where you do not know in advance what some of the values will be. You supply the value via the **Value** property of the **Parameter** object. Additionally, you can use the **Direction** property to determine whether the parameter is input, output, or both, much as you would with a stored procedure on a database.

Imagine you have this query stored about your employees:

```
SELECT EMP_DEPT_NO, EMP_GENDER, EMP_NO, EMP_FNAME,
       EMP_LNAME, EMP_SALARY
FROM EMPLOYEE
WHERE EMP_DEPT = ?
  AND EMP_GENDER = ?
ORDER BY EMP_DEPT, EMP_LNAME, EMP_FNAME
```

You want to be able to run this query, supplying the department number and gender at runtime. You first create a **QueryDef** object to store the query definition and then two **Parameter** objects to supply the missing information.

The code in Listing 5.6 creates a **QueryDef** object with two parameters for department number and gender, creates a **RecordSet** based on the **QueryDef**, and then displays the results on the screen shown in Figure 5.7. The application (which is also on the CD-ROM) prompts the user to supply a department number and a gender and then displays all the records that fit that criteria. The query itself creates the two parameter objects, as shown on the first highlighted line. After executing the statement, you can examine the **Parameters** collection in the Immediate window. If you type "? dbCoriolis.CreateQueryDef.Parameters.Count", the answer will be 2. Type "? dbCoriolis.CreateQueryDef.Parameters (0).Name", and the answer is Dept. Notice in the statement that you use SQL data types when "declaring" the parameter type instead of Visual Basic data types.

Listing 5.6 Demonstration of the QueryDef, Parameter, and RecordSet objects.

```
Private Sub Command1_Click()
Dim dbCoriolis As Database
Dim qdfDeptGenEmpSal As QueryDef
Dim rsReport As RecordSet
Dim sCoriolis As String
Dim sGender As String
Dim iCtr As Integer
Dim iDept As Integer
' Change this to reflect your path
sCoriolis = "c:\coriolis.mdb"

Set dbCoriolis = OpenDatabase(sCoriolis)

' Create a temp QueryDef object
' Use 2 parameters

Set qdfDeptGenEmpSal = _
  dbCoriolis.CreateQueryDef("", _
    "PARAMETERS Dept Integer, Gender Char ; " & _
    "SELECT EMP_DEPT_NO, EMP_GENDER, EMP_NO, EMP_FNAME, " & _
```

```
    "EMP_LNAME, EMP_SALARY FROM EMPLOYEE WHERE " & _
    "EMP_DEPT_NO = [Dept] AND EMP_GENDER = [Gender] " & _
    "ORDER BY EMP_DEPT_NO, EMP_LNAME, EMP_FNAME")

' Prompt the user for the values to search for
iDept = Val(InputBox("Enter Department to Search For", _
  "Department", "100"))
GenderBad:
sGender = UCase$(InputBox("Enter Gender to Search For",
   "Gender", "F"))
If sGender <> "F" And sGender <> "M" Then
    MsgBox "Gender must be M or F!", vbOKOnly + _
        vbExclamation, "Gender Crisis"
    GoTo GenderBad
End If

' Set the parameters
qdfDeptGenEmpSal.Parameters("Dept") = iDept
qdfDeptGenEmpSal.Parameters("Gender") = sGender

' Open the RecordSet
Set rsReport = _
  qdfDeptGenEmpSal.OpenRecordSet(dbOpenSnapshot)

' Scroll to last record to ensure accurate record count
rsReport.MoveLast
rsReport.MoveFirst

' Display the results
For iCtr = 0 To rsReport.RecordCount - 1
    If iCtr > 0 Then
        ' Display More?
        If MsgBox("Display More?", vbYesNo + vbQuestion) _
               = vbNo Then
            Exit For
        Else
            rsReport.MoveNext
        End If
    End If
    txtFields(0).Text = rsReport!EMP_DEPT_NO
    txtFields(1).Text = rsReport!Emp_Gender
    txtFields(2).Text = rsReport!Emp_No
    txtFields(3).Text = rsReport!Emp_FName
    txtFields(4).Text = rsReport!Emp_LName
    txtFields(5).Text = rsReport!Emp_Salary
Next

' Close everything
rsReport.Close
qdfDeptGenEmpSal.Close
dbCoriolis.Close

End Sub
```

Figure 5.7 This application returns all records that match the department and gender entered by the user.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The **Parameter** object has no methods. It does have these properties:

- **Direction** (ODBCDirect workspace only) determines whether the parameter is input, output, or both. This is most useful with stored procedures (see Chapter 11). Some RDBMSs, such as Microsoft SQL Server, can determine the direction of parameters. Others (depending especially on the ODBC driver being used) cannot determine the direction. Therefore, it is safest to set the direction property prior to executing the procedure. The possible values are **dbParamInput**, which is the default and indicates an input parameter, such as in the example in Listing 5.6; **dbParamOutput**, which indicates an output parameter (a parameter that the database will supply, such as a column in a query result); **dbParamInputOutput**, which passes information to the procedure and receives information from the server; and **dbParamReturnValue**, which is used to receive the return value from the procedure.
- The **Type** property specifies the data type of the **Parameter**. If you examine Listing 5.6, you see that you use SQL data types and not the equivalent Visual Basic data types. If you make the assignment directly, and not as part of the **CreateQueryDef** method, then you must use one of the Visual Basic constants listed in Table 5.3 earlier in this chapter.
- The **Value** property is the actual value stored in the parameter. Because it is the default property of the object, you can omit it in assignments and references, as I did in Listing 5.6. Alternatively, I could have coded the assignments as shown here:

```
qdfDeptGenEmpSal.Parameters("Dept").Value = iDept
qdfDeptGenEmpSal.Parameters("Gender").Value = sGender
```

The RecordSets Collection And RecordSet Object

The DAO object with which you will most often interact is the **RecordSet** object. The **RecordSets** collection contains all of the **RecordSet** objects in a **Database** object or, in an ODBCDirect workspace, a **Connection** object. When you open a **RecordSet** object, it

is automatically appended to the collection. When you close it, it is automatically removed. You can open the same **RecordSet** object more than once, meaning that you can have duplicate named **RecordSet** objects within the same collection. To refer to them, you must create an object variable to reference them uniquely, as shown in this code:

```
' Create two object variables
Dim rsSet1 As RecordSet
Dim rsSet2 As RecordSet

' qdfSample is an existing QueryDef object
' First instance of the RecordSet
Set rsSet1 = qdfSample.OpenRecordSet(dbOpenSnapshot)
' Second instance of the RecordSet
Set rsSet2 = qdfSample.OpenRecordSet(dbOpenSnapshot)
```

The **RecordSet** object itself is the program's near-exclusive interface to the database. It represents either all of the records in a base table (in a table-type record set) or the rows generated from a query.

Depending on whether you use a Data control to access the records, you can generate up to five types of record sets. You should use the type that provides the functionality you need but uses the least resources. In other words, if you need to look up records for a report but you don't need to scroll forward and backward through the records, you should use a forward-only-type record set instead of the snapshot-type. The latter provides more flexible access to the records (you can move forward and backward through the records) but also takes more resources.

The types of record sets that you can generate follow:

- *A table-type record set* (Jet workspace only) represents one entire base table. You can add, delete, and modify records and move through the records in any direction.
- *A dynaset-type record set* is also fully updatable and you also can move freely through the records. However, the rows can consist of one or more fields from multiple tables. Depending on what fields are actually represented, you may be restricted in whether you can update. This is a function of the database and not of the record set. For instance, you generally cannot update a record if the record set does not contain the primary key of that record. Use the **Updatable** property to determine if the record set can be updated.
- *A snapshot-type record set* is not updatable. If other users add or change records on the database, they are not included in the snapshot until it is closed and reopened. You can move freely through the records. This is most useful for data that you need to reference but not change, such as a postal code lookup table.
- *A forward-only-type record set* is identical to a snapshot except that you can only move forward through the records.
- *A dynamic-type record set* (ODBCDirect workspace only) is the result of a query against one or more tables. You can add, update, or delete records. Depending on the nature of the data returned, the database may not allow you to update, but this is not a restriction of the record set. For instance, you generally cannot update a record if the record set does not contain the primary key of that record. Use the **Updatable** property to determine whether the record set can be updated. The record

set automatically reflects the changes, additions, and deletions made by other users to the database that affect your records.

To specify the type of record set, use one of the constants listed in Table 5.8 earlier in this chapter. This is specified in the **type** argument of the **OpenRecordSet** method, which I discussed earlier in this chapter with the **Database** topic. Most of the listings in this chapter show examples of opening record sets, closing them, and referencing them. Listing 5.3 makes extensive use of a **RecordSet** object in an ODBCDirect workspace, whereas Listing 5.6 illustrates the use of a **RecordSet** object in a Jet workspace. As shown, the use is nearly identical.

When you open a record set, the current record is always the first record (denoted by the property **AbsolutePosition** =0). If there are no records, the **BOF** and **EOF** properties are both true and the **RecordCount** property is equal to 0.

The **RecordSet** object has two collections, each of which I discussed earlier in the chapter. The **Properties** collection contains all of the **Property** objects of the **RecordSet** object. The **Fields** collection denotes all of the fields within the **RecordSet** object. You can examine each **Field** object to determine the data type of each field in the record set using the **Field** object's **Type** property. Use the **Value** property to examine the value of each field.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The properties that are available when using the **RecordSet** property depend both on the type of workspace (Jet or ODBCDirect) and on the type of record set. The properties are discussed in turn in the following list:

- **AbsolutePosition** is a **Long** that returns the current record number. Alternatively, you can set the property to move to a specific record number within the **RecordSet** (except on a forward-only-type record set). The **AbsolutePosition** is zero-based so the highest record number will always be one less than the number of records. Specifying an **AbsolutePosition** of less than zero or greater than the number of records causes a runtime error. If no record is current, this property is equal to -1.
- **BatchCollisionCount** (ODBCDirect workspace only) returns the number of records that did not complete the last batch update. A collision in a batch update occurs when you attempt to update a record that has been altered because you retrieved it. This can only happen in a multiuser environment or when your application opens multiple record sets on the same data. Use this property along with the **BatchCollisions** property, which is an array of bookmarks of those records on which collisions occurred. This allows your application to then move to each record and correct the problem. Once the problem is rectified, you can invoke the **Update** method again. If there are any more collisions, then these two properties (**BatchCollisionCount** and **BatchCollisions**) are repopulated. Visual Basic knows not to re-update those records for which no collision occurred because their **RecordStatus** is changed to **dbRecordUn-Modified** once successfully updated.
- The **BatchSize** (ODBCDirect workspace only) property sets and returns how many statements are sent to the server at a time during a batch update. Note that not all ODBC drivers support more than one statement being sent at a time. The default value is 15.
- The **BOF** and **LOF** properties both return Boolean values that determine whether the current record position is before the first record (**BOF**) or after the last record (**LOF**). If both values are **True**, then there are no valid records. If **BOF** is **True** but **LOF** is **False**, there are valid records, but the current record position is before the first record. If you are positioned on the last record and you delete it, the **LOF** property may not accurately reflect the fact that you are now beyond the last record. In either case, if you do delete the last record in a record set, you should perform a **MoveLast** to reposition to the new last record in the record set. If there are valid records and **BOF** is **True**, a runtime error results if you then attempt to perform a **MovePrevious**. Likewise, if **EOF** is **True** and you then attempt to perform a **MoveNext**, a runtime error also occurs. Note that if you have an empty record set and add a record, **BOF** and **EOF** behave differently between Jet workspaces and ODBCDirect

workspaces. In a Jet workspace, **BOF** becomes **False** but **EOF** remains **True**, indicating that the current record is still invalid. You should perform a **MoveFirst** or **MoveLast** to make the current record valid. On the other hand, in an ODBCDirect workspace, both **BOF** and **EOF** become **False**. Several of the sample applications on the CD-ROM scroll through records until **EOF** is **True**, indicating that there are no more records. The following scrolls backward through the records until **BOF** is **True**.

```
With rsEmp
  ' Move to last record
  .MoveLast
  ' Scroll through records in descending order
  Do While Not .BOF
    ' Put the record in edit mode
    .Edit
    ' Give them a raise!
    !Emp_Salary = !Emp_Salary + cRaise
    ' This is a local update; the database
    ' is not getting updated yet!
    .Update
    ' Scroll to prior record
    .MovePrevious
   Loop
```

• **Bookmark** is a property whose data type is a **Variant** array of **Byte** data. It uniquely identifies each record in a record set. **Bookmarkable** is a Boolean that determines whether the current record can be bookmarked. You can save your current position in a current record by saving the bookmark to a variable of type **Variant**. This allows you to quickly move back to it at any time. If you create a copy of the record set using the **Clone** method, the bookmarks are identical. The following code segment stores the current bookmark, moves to the end of the record set, and then moves back to the saved bookmark:

```
' rsEmp is a currently open RecordSet
If rsEmp.BookMarkable = False Then
  MsgBox "Cannot Save Position!"
Else
  ' Save the position
  Dim vBookMark As Variant
  vBookMark = rsEmp.BookMark
  ' Scroll to the end
  rsEmp.MoveLast
  ' And back again
  rsEmp.BookMark = vBookMark
End If
```

• **CacheSize** is a **Long** that sets or returns the number of records from an ODBC data source that will be cached locally. I discussed its usage earlier in the chapter when we discussed the **QueryDef** object.

• The **CacheStart** property (Jet workspace only) is used with dynaset-type record sets. It allows you to specify which record will be the first record in a cache. Suppose you have 50 records cached locally and you want the new cache to begin with record 40. You can move to that record and then set the **CacheStart** property equal to the **Bookmark** property of that record. If you then rebuild the cache (see **CacheSize**), the 40th record will then be the first record in the new cache.

• The **Connection** property (ODBCDirect workspace only) is a string containing connection parameters and was discussed earlier in this chapter with the **Databases** topic.

- The **DateCreated** and **LastUpdated** properties (Jet workspace only) return the date and time the base table (not the record set itself) was created and last updated.
- The **EditMode** property is a **Long** that contains a constant indicating the current edit state for the first record. **dbEditNone** indicates that the record is not being edited. **dbEditInProgress** indicates that the current record is currently being edited. **dbEditAdd** indicates that the current record is a new record that hasn't been updated to the database, which occurs when the **AddNew** method has been invoked.
- **Filter** (Jet workspace only) contains a string restricting what records appear in the record set. It is essentially the **WHERE** clause in an SQL query except that the word **WHERE** is omitted. Set the **Filter** property on an existing **RecordSet** object to refine which records will be included in the record set. My recommendation is that you will usually achieve better performance by opening a new record set where the SQL statement includes the **WHERE** clause.
- **Index** (Jet workspace only) is used to alter the order of records. If you set this property to the **Name** property of a valid **Index** object, Jet will use that index on the database to order the records.
- The **LastModified** property is useful to determine which record was the last to be modified in a record set. **LastModified** contains the bookmark of the last record to be altered or added. If no record has been updated or added, this value is not valid.
- **LockEdits** is a Boolean indicating whether pessimistic locking (**True**) or optimistic locking (**False**) is in effect. Under pessimistic locking, the 2K page on which the record is stored is locked as soon as the record is edited. Under optimistic locking, the page is not locked until the record is saved. In general, I recommend optimistic locking, but you need to take extra care for concurrency issues, as I discuss in Chapter 12 and at various points throughout this chapter.
- The **Name** property is a **Variant** of subtype **String** that you use to reference the object within the **RecordSets** collection. Because you can open the same **RecordSet** object more than once, you can have duplicate **Name** properties. To uniquely identify a given **RecordSet** object, assign the **RecordSet** to an object variable.
- **NoMatch** (Jet workspace only) is a Boolean indicating the success or failure of the most recent **Find** or **Seek** operation. If you search for a record and it is not found, this property is set to **True**. You should evaluate this property each time you perform a **Find** or **Seek**. See the **Find** method later in this section.
- **PercentPosition** returns a single between 0 and 100 that gives an approximate percentage of the relative position within the record set. For instance, if there are 1,000 records and you are currently displaying record 125, **PercentPosition** will be equal to 12.5. You can set this property to move within a record set (**rsEmp.Per-centPosition=12.5**), but it is not a precise method for moving around. You are better off moving using the **Bookmark** property.
- **RecordCount** is a **Long** that returns the number of records in the current **RecordSet** object. When you first open the **RecordSet**, this property may not be accurate until you have scrolled through all of the records. In Listing 5.6, I illustrate using the **MoveLast** method to scroll to the last record, which ensures that the **RecordCount** property is accurate. **RecordCount** reflects the actual number of records, whereas **AbsolutePosition** is 0 based. (**RecordCount** does not reflect any records that have been deleted.) The following code updates a Data control's **Caption** property to display the current record number and the number of records. If it is not necessary to know how many records there are, you may want to omit this step because moving to the last record has an adverse impact on performance.

```
Data1.Caption = "Record " & rsEmp.AbsolutePosition + 1  & _
  " of " & str$(rsEmp.RecordCount) & " records"
```

- **RecordStatus** (ODBCDirect workspace only) is a **Long** that reflects the current status of

each record in the **RecordSet**. The constants can be **dbRecordUnModified**, **dbRecordModified**, **dbRecordNew**, **dbRecordDeleted**, or **dbRecordDBDeleted**. This is meaningful only in optimistic batch updating, because other types of record sets cause the record change to be updated to the database immediately. In an optimistic batch update, you can determine from each record's **RecordStatus** what type of change will be made to the database.

- **Restartable** is a Boolean that returns **True** if the **RecordSet** can be requeried (see the **ReQuery** method later in this section). If the property returns **False**, you will have to close the record set and reopen it to refresh the records.
- **Sort** (Jet workspace only) is a string that sets or returns the sort order for a record set. The syntax is the same as the SQL **ORDER BY** clause except that it omits the **ORDER BY** phrase. In general, I recommend using the **ORDER BY** clause in the **SELECT** statement itself. Altering the **Sort** property of an already opened **RecordSet** object does not alter its sequence; you must set the property before opening it.
- **StillExecuting** (ODBCDirect workspace only) is a Boolean that returns **True** if an asynchronous query is still executing. You cannot access the records in a **RecordSet** object until the query has finished processing. You also cannot reference any other property of the **RecordSet** object until this property returns **False**. The following code opens a **RecordSet** object. Then, a **MoveLast** is performed asynchronously. The code then loops until the move is complete. The **DoEvents** statement allows the screen to redraw while the loop is executing:

```
' Open the RecordSet
Screen.MousePointer = vbHourGlass
Set rsEmp = conEmp.OpenRecordSet _
  ("SELECT * FROM EMPLOYEE", dbOpenDynaset, 0, _
  dbOptimisticBatch + dbRunAsync)
' Move to the last record
rsEmp.MoveLast dbRunAsync
' Don't do anything until the move is finished
Do
 If rsEmp.StillExecuting = False Then Exit Do
 DoEvents
Loop
Screen.MousePointer = vbDefault
```

- **Transactions** (Jet workspace only) is a Boolean that returns **True** if the **RecordSet** object supports transaction processing. Using **BeginTrans**, **CommitTrans**, and **Rollback** has no effect if the **RecordSet** does not support transaction processing. However, you should check the **Transactions** property before issuing a **BeginTrans** to ensure that you don't receive unexpected results. Listing 5.3 earlier in this chapter contains a program demonstrating transaction processing.
- **Type** sets or returns the **RecordSet** type. I enumerated the five different types of record sets at the beginning of this section. The constants associated with these types are **dbOpenTable**, **dbOpenDynamic**, **dbOpenDynaSet**, **dbOpenSnapShot**, and **dbOpenForewardOnly**.
- **Updatable** is a Boolean that returns **True** if the **RecordSet** object is updatable. There are any number of reasons why a **RecordSet** object might not be updatable. For instance, the record set type itself may preclude updates (snapshot types, for example). Your query may combine updatable and non-updatable tables or perhaps it does not include the primary key of a record. You should check this property to ensure that your **RecordSet** object can be updated.
- **UpdateOptions** (ODBCDirect workspace only) determines the **WHERE** clause in an

update statement sent to the database. Normally, you only need to specify the primary key in the **WHERE** clause to uniquely identify the row to be updated. However, in a multiuser environment, you might want to include more columns to ensure that no other user has altered the record between the time you retrieved the row and the time you update it. For instance, if you specify your **WHERE** clause, include all columns from the record; then if the **WHERE** clause fails, you know that the record has been altered. Use **dbCriteriaKey** if you only want to use the primary columns in the **WHERE** clause. **dbCriteriaModValues** causes the **WHERE** clause to include the primary columns and any columns that were changed. Use this option if you are concerned about concurrency issues (see Chapter 11 for more details) but only care about those columns that you altered. **dbCriteraAllCols** uses all columns in the **WHERE** clause. Use this option in a multiuser environment where you absolutely do not want to alter a record if another user has altered it. **dbCriteriaTimeStamp** will use the primary columns and the timestamp column if there is one. This is often just as good as **dbCriteriaAllCols** because any change to the record will cause the timestamp to be changed, and it is more efficient (because the **WHERE** clause is simpler). Along with any of the five prior constants, you can also specify either **dbCriteriaUpdate** to specify that the update be done with an **UPDATE** statement or **dbCriteriaDeleteInsert** to specify that the update be done by first deleting the record and then inserting a new one. Normally, you will want to use **dbCriteriaUpdate** (which is the default) because it is inherently more efficient. However, if for some reason you are altering the primary key, you normally need to delete the first record and then insert a new record.

- The **ValidationRule** and **ValidationText** properties (Jet workspace only) control the validation criteria and what message to display if the record fails validation. I generally recommend validating each field in the **RecordSet** independently. Note, however, that the **ValidationRule** for a **Field** object cannot reference other **Field** objects, whereas the **ValidationRule** for a **RecordSet** can reference any field in the record. I discussed these properties earlier in the chapter in the **TableDef** and **Field** topics.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The **RecordSet** object has a rich set of methods that you use to manipulate fields or data as well as properties. Not all methods are available for all record set types, as noted in the discussion of each:

- The **AddNew** method adds a new record to the **RecordSet**. It does not add it to the database. (Use the **Update** method to make the change to the database.) If the fields have default values, they are set to those values; otherwise, they are set to **Null**. If you create a new record and then move to another record, the changes are lost unless you first invoke the **Update** method (unless you are performing batch updates). In a dynaset-type **RecordSet**, new records are always added to the end of the **RecordSet**. In a table-type **RecordSet**, records are added in their proper sort order if one has been specified with the **Index** property. The new record must have a unique key. If there is no unique key, Jet workspaces generate a “Permission Denied” error, whereas ODBCDirect workspaces generate the message “Invalid Argument.”
- The **Cancel** method (ODBCDirect workspace only) cancels execution of a currently running asynchronous query (for the **RecordSet** object, this is the **OpenRecordSet** method) or **MoveLast** operation. If **dbRunAsync** was not specified, using **Cancel** causes a runtime error.
- **CancelUpdate** cancels any changes that are pending since the last **Update** call. If the user adds a record or modifies a record, the changes are discarded. If performing batch updating, you can also specify an argument, as shown in the following example. **dbUpdateRegular** cancels any pending changes that aren’t cached. This is the default. **dbUpdateBatch** cancels any changes pending in the cache.

```
rsEmp.CancelUpdate dbUpdateBatch
```

- **Clone** (Jet workspace only) creates a copy of the current record set. The bookmarks in each copy of the **RecordSet** are shared, but each **RecordSet** can have different current records. You can issue the **Clone** method on the cloned **RecordSet** without affecting the original **RecordSet** and you can issue another **Clone** on the original **RecordSet** without affecting the cloned **RecordSet**. Cloning a **RecordSet** is useful if you need to access the records in different ways at the same time, and it is more efficient than creating a second **RecordSet** using the **CreateRecordSet** method. The following example creates a cloned **RecordSet** and searches it to see if the primary key

of a record that has just been added already exists:

```
' Assumes rsEmp already exists and that the
' user has just entered a new record
Dim rsClone As RecordSet
' Clone the RecordSet
Set rsClone = rsEmp.Clone
' Now, search through it for the primary key
' txtEmpID is a textbox containing Emp ID
rsClone.FindFirst "Emp_ID = '" & Trim(txtEmpID.Text) & _
  "'"
If rsClone.NoMatch = False Then
  ' Duplicate key found
  MsgBox "Invalid Primary Key!"
  ' Move to the offending record
  rsEmp.Bookmark =  rsEmp.LastModified
End If

' Close the cloned RecordSet
rsClone.Close
```

• The **Close** method closes the **RecordSet** object. You should close all open **RecordSet** objects prior to closing the **Database** or **Connection** object to which the **RecordSet** objects belong.

• The **CopyQueryDef** is used to create a new **QueryDef** object that is the same as the **QueryDef** on which the **RecordSet** object is based.

• The **Delete** method is used to delete the current record in the **RecordSet**. The current record remains current but is inaccessible. You should move to a new record after executing the **Delete** method. It is no longer possible to reference the deleted record, and the change cannot be undone unless you perform a **Rollback** after a **BeginTrans** statement.

• **Edit** causes the current record to be placed into edit mode. Changes to the record are placed into the copy buffer and are made permanent after the **Update** method is issued. Note that in a non-batch-based **RecordSet**, moving to another record without using the **Update** method causes the changes to be discarded.

• **FillCache** (Jet workspace only) fills all or part of a local cache. The syntax is shown in the following code segment. **recordset** is a valid object variable of type **RecordSet** (the **RecordSet** must already be opened). **rows** is the number of records to retrieve. If omitted, the **CacheSize** value is used instead. The **startbookmark** argument specifies the **Bookmark** of the record that should be the first record in the cache. If omitted, the value stored in the **CacheStart** property is used. See the discussion of **CacheSize** and **CacheStart** earlier in this section.

```
recordset.FillCache rows, startbookmark
```

• The **FindFirst**, **FindLast**, **FindNext**, and **FindPrevious** methods (Jet workspace only) allow you to search the record set for a specific record. The search criteria is similar to the SQL **WHERE** clause except that the word **WHERE** is omitted. These methods are not available in ODBCDirect workspaces. Instead, you should use your SQL **SELECT** statement to return those rows that you require. The following example searches the **rsEmp RecordSet** object for all occurrences of the last name “Smith”. If the first one is found (using **FindFirst**), then you use **FindNext** to find all subsequent

occurrences. **FindLast** finds the last occurrence of a criteria, whereas **FindPrevious** finds the occurrence of a criteria prior to the current record.

```
rsEmp.FindFirst "EMP_LNAME = 'SMITH'"
MsgBox "Smith Found at Record " & _
  str$(rsEmp.AbsolutePosition + 1)
If rsEmp.NoMatch = False Then
  Do
    rsEmp.FindFirst "EMP_LNAME = 'SMITH'"
    If rsEmp.NoMatch Then Exit Do
      MsgBox "Another Smith Found at Record " & _
        str$(rsEmp.AbsolutePosition + 1) & "!"
    Loop
End If
```

• **GetRows** copies one or more rows from a **RecordSet** into a two-dimensional array of type **Variant**, where the first subscript refers to the column number (zero-based) and the second subscript refers to the row number (also zero-based). The following example moves 300 rows in a **RecordSet** called **rsCusts** into an array. If the **RecordSet** contains fewer than 300 rows, no error results. Use **UBound** to determine how many rows were moved. The code also displays the value of the 3rd field in the 18th record. Note that the current record is the one immediately after the last one loaded. If you load all of the rows, the current record will be past the last record (in other words, **EOF** will be **True**).

```
' Do not need to declare as array
Dim vArray As Variant

' Copy the first 300 rows
Set vArray = rsCusts.GetRows (300)
' Display how many there actually are
MsgBox UBound (vArray, 2) & " records loaded"
' Display a field
MsgBox varray(2, 17)
' Reposition to the first record
MoveFirst
```

• The **Move**, **MoveFirst**, **MoveLast**, **MoveNext**, and **MovePrevious** methods move to an absolute record within the **RecordSet**. The **Move** method accepts two arguments. The first is the number of records to move relative to the current record. A value greater than zero moves forward, whereas a value less than zero moves backward. You may specify an optional argument, **startbookmark**, which causes the move to occur relative to whichever record is referenced by the **Bookmark**. The following example records the current position and then moves to the last record. It then uses the **Move** method to move 10 rows ahead of the record that was bookmarked. If there are no current records, moving causes an error. An error results if **BOF** is **True** and you **MovePrevious** or if **EOF** is **True** and you **MoveNext**. If you otherwise move forward or backward more records than are in the **RecordSet**, **BOF** or **EOF** is set to **True** as appropriate.

```
Dim vBookMark As Variant

' Bookmark the current record
vBookMark = rsEmp.BookMark
' Move to the last record
```

```
rsEmp.MoveLast
' Move to the 10th record beyond the bookmark
Move 10, vBookMark
```

• **NextRecordSet** (ODBCDirect workspace only) gets the next set of records when more than one SQL statement is specified in the **RecordSet** object's **Source** property (or in the **SQL** property of the **QueryDef** object on which the **RecordSet** is based). If you do specify more than one query, each must be separated by semicolons. I do not recommend this practice (placing more than one query in the **Source** property) and prefer to use entirely different **RecordSet** objects for different result sets. If there are no more records to be fetched, this method returns **False**. Otherwise, it returns **True**.

• **OpenRecordSet** (Jet workspace only) creates a new **RecordSet** object. I discussed how to use it in the **Databases** topic earlier in this chapter.

• **ReQuery** re-creates or refreshes the records in a **RecordSet** object by re-executing the query.

• The **Seek** (Jet workspace only) method is roughly equivalent to the **Find** methods, which search for a value in a **RecordSet**. **Seek** is available only in table-type record sets (and thus, only in Jet workspaces). The value you are searching for must be part of the current index (see the **Index** method earlier in this section). For example, if the current index is on the **Emp_ID** field, you can only search for employee IDs. Further, you can only use these comparison operators: =, <, <=, >, or >=. If you specify =, >, or >=, the search starts from the beginning of the record set and returns the first match. If the search uses < or <=, the search starts at the last record in the record set. The following locates employee ID 350: **rsEmp.Seek=350**.

• **Update** causes the contents of the copy buffer to be saved to the **RecordSet** object. It works only on the current record. If you specify pessimistic locking when you place a record in edit mode, then the record (and the entire page the record is on) remains locked until you issue the **Update** method. The syntax is shown in the following code example. **recordset** is a valid object variable of type **RecordSet**. The optional type argument specifies how to update the data. If you specify **dbUpdateRegular** (which is the default), the changes to the record are written to disk without being cached. If you specify **dbUpdateBatch**, any pending batched updates are written to disk. **dbUpdateCurrentRecord** causes any changes to the current record to be updated to disk regardless of whether there are other pending, batched changes. The **force** argument optionally specifies whether to force the database to be updated even if another user has changed the record. **True** forces the changes to be saved. If not specified, **False** is used, indicating that the changes should not be made if another process has made changes to the records since you invoked the **AddNew**, **Delete**, or **Edit** method. No error occurs, but the **BatchCollisions** and **BatchCollisionCount** properties will be set. I discussed these two properties earlier in this section, including an example of using the **Update** method in a multiuser environment.

```
recorset.update (type, force)
```

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The Data Control

The Data control is the granddaddy of all Visual Basic data access controls and is the only data access control intrinsic to VB. To use the Remote Data control or the Active Data control, you first must add them to the toolbox using the Components selection from the Project menu.

The Data control works only with DAO workspaces, although it does support both Jet and ODBCDirect. To force the Data control to create an ODBCDirect workspace, set the **DefaultType** property to 2 (**dbUseODBC**).

The Data control is flexible in that it provides access to most of the DAO hierarchy that I discussed in first part of this chapter with little coding. Further, the control provides buttons for the user to navigate a record set. The buttons equate to the **MoveFirst**, **MovePrevious, MoveNext**, and **MoveFirst** methods. When a form is opened, the Data control automatically creates a **RecordSet** object and fetches the records from the database.

The Data control also offers the convenience of being a target—a data provider, if you will—for data-aware controls to bind to. Thus, any time the Data control scrolls a record, all controls that are bound to it also update their contents. Data-aware controls include the textbox, label, listbox, ComboBox, picturebox, image, checkbox OLE, MaskedEdit, and RichTextBox controls. The latter two controls come with the Professional and Enterprise editions of VB only. Additionally, the DataList, DataCombo, DataGrid, and MSFlexGrid controls can display multiple values at the same time.

The Data control provides the interface necessary to automatically add and update records. It includes events that make it convenient to process records such as the **Validation** event, which occurs any time there is an action that would cause the current record to no longer be the current record.

In addition to the brief examples shown in this section, Listing 5.4 earlier in this

chapter provides an example of using the Data control.

The Life Of The Data Control

When you open a form containing a Data control, it is initialized before the form's **Load** event. If an error occurs during initialization, an untrappable error occurs. The Data control creates a **Database** object and a **RecordSet** object, and then populates the **RecordSet** and positions itself onto the first record. Alternatively, you can create a **RecordSet** in your code and assign it to the Data control. When this happens, the **RecordSet** is automatically populated and, again, the Data control makes the first record the current record. The Microsoft documentation states that with a Data control, you do not need to perform a **MoveLast**, implying that the **RecordCount** property of the **RecordSet** will be accurate. This has not always been my experience; you will have to experiment with your own database implementation.

Any time the user scrolls a record, the control's **Validate** event occurs, giving you the opportunity to edit any changes made by the user as well as the opportunity to prevent the update and even stop the action the user took. Thus, if the user attempts to close the form, the **Validate** event fires and you can stop the form from closing.

Bound Controls

One of the great conveniences of the Data controls is, of course, the ability to bind a control to a field or column in the database. Controls that can be bound to a data source are data aware. You bind a control to a data source by setting the control's **DataSource** property to the name of the object (such as a Data control) and setting the **DataField** property to whichever field is to be displayed. Where appropriate, bound controls have a **DataFormat** property to customize the display of data (this is new to Visual Basic 6). Also, bound controls have a **DataChanged** property, allowing you to quickly determine which fields of a record have been modified. In Chapter 9, I discuss more advanced uses of these properties as well as the new **DataMember** property, which is not meaningful in DAO applications. I also show the **DataChanged** property being used in the **Validate** event example later in this section.

Using The Data Control

You draw the Data control on a form as you would any other VB control. If you do not like its interface, you can make it invisible (by setting its **Visible** property to **False)**. However, you will have to then supply your own navigation capabilities.

I discuss here the key properties of the Data control that I did not discuss earlier in the chapter when reviewing the Data Access Objects:

- The **Align** property allows you to dock the Data control to one edge of the form. When the form is resized, the control is also automatically resized. Although you can dock the control to the left or right edges of the form, it is rather ugly. I recommend docking on the bottom where the control is out of the way of any toolbars. Possible values are 0 (no

alignment), 1 (align top), 2 (align bottom), 3 (align left), or 4 (align right).

- **BOFAction** and **EOFAction** determine what to do if the user attempts to scroll before the first record or after the last record. If the user is on the first record (**BOF** is **True**) and attempts to move to a prior record, **vbBOFActionMoveFirst** causes the first record to remain the current record. In other words, the attempt to move backward is ignored. This is the default action. **vbBOFActionBOF** causes the Data control to scroll before the end of the file. When this happens, the **Validate** event is triggered for the current record and the **Reposition** event is triggered on the now invalid record. Similarly, **vbEOFActionMoveLast** and **vbEOFActionEOF** ignore the move and move to the **EOF** invalid record. I recommend using **vbBOFActionMoveFirst** and **vbEOFActionMoveLast** for the two properties.

- The **Caption** property is a string you can use to display information about the database connection (or whatever purpose suits your needs). The following code snippet placed in the **Reposition** event will cause the caption to display a “Record x of y records” message every time the user moves to another record:

```
Data1.Caption = "Record " & _
  str$(rsEmp.AbsolutePosition + 1) & _
  " of " & str$(rsEmp.RecordCount) & " records"
```

- The **RecordSetType** property determines what type of **RecordSet** object will be displayed. Using the Data control, you can only specify table, dynaset, or snapshot. To use other types of **RecordSet** objects, you need to create and manipulate the **RecordSet** object in code. All the properties and methods of the **RecordSet** object are available to you by referencing the Data control, as shown in the previous example. If not specified, DAO will attempt to create a dynaset. If you change the **RecordSetType** property in code, you must **Refresh** the Data control.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The Data control has these key methods beyond those that I discussed earlier in the chapter:

- The **Refresh** method rebuilds the results of the **RecordSet** associated with the Data control. Most commonly, you will modify the **RecordSet** object's **Source** property, creating a new SQL statement or associating a **QueryDef** object with it. When you then use the **Refresh** method (**Data1.Refresh**), DAO will rebuild the **RecordSet**. The first record in the new **RecordSet** will be the current record, and bound controls will be updated with the new fields. If there is an error, such as passing an invalid **QueryDef** object or an invalid SQL statement, an untrappable error occurs. Often, you will have two Data controls on a form in a master/detail relationship where the contents being retrieved from the detail Data control depend on a value in the current record of the master Data control. For instance, if you have a Data control displaying department numbers and names, you might have a second Data control that fetches all of the employees in that department. When a new record is displayed in the master Data control, the secondary Data control fetches the employees in the currently displayed department. You place code similar to the following in the master Data control's **Reposition** event:

```
datSecondaryRS.RecordSource = "Select * From Employee " & _
  "Where Emp_Dept_No = " & datPrimaryRS!Dept_No
```

- The **UpdateControls** method is used to get the current record in the Data control's **RecordSet** object and display the values in all bound controls. This is normally not necessary except when you want to cancel all changes made by a user on the current record and restore the values in the bound controls.
- The **UpdateRecord** method saves all changes to the current record to the database, but the **Validate** event is not triggered, as when using the **RecordSet** object's **Update** method. This is useful, for instance, from the **Validate** event itself when you don't want to create a second call to the event.

The Data control also has a number of events that are useful in the client/server environment:

- The **Error** event occurs as a result of any error that occurs while no VB code is being executed—in other words, when the error occurs at the database. If you don't code for these errors, VB displays a message and the error is typically fatal. The syntax for the event is shown in the next code example. **data_error** is the error number generated, whereas **response** is one of two values that you set: **vbDataErrContinue** causes the program to continue execution and **vbDataErrDisplay** causes VB to display the error and act as though

the error was unhandled. Beyond that, the **Error** event is roughly the same as an active error handler in your code. Note that an error that occurs prior to the form's **Load** event does not trigger the event. An example is if the Data control attempts to open an invalid **RecordSet**.

```
Data1_Error (data_error, response)
```

• The **Reposition** event occurs after any record becomes the current record, including when the Data control is loaded and the first record is displayed. This is where you will place code to handle situations unique to each record. For instance, you may want to add code to the **Reposition** event to calculate the age of an employee based on his or her date of birth each time a new employee is displayed. Under the **Caption** method, I included a line of code to change the caption each time a new record is displayed.

• The **Validate** event is where you will place code to ensure that data is valid before being saved to the database. The event occurs *before* a new record becomes the current record (such as when the user clicks one of the move buttons), before the **Update** method, and before a **Delete**, **Unload**, or **Close** operation. You have the opportunity to save the changes or to cancel the operation that triggered the **Validate** event. For example, if the user changes a record and then clicks a move button, you can cancel the move to force the user to correct any input errors. The syntax of the event is shown in the next code example. The **action** argument is a constant that you can evaluate to determine what action caused the **Validate** event to occur. Table 5.15 lists the possible constants. You can code a **Select Case Action** statement to determine which action occurred. The **save** argument is a Boolean that you can evaluate to determine whether any bound data was changed. If the value is **False**, there is usually no need to update the database and there is probably no need to validate the data. If the value of **Save** is **True** when the event is exited, the **Edit** and **UpdateRecord** methods are invoked, causing the change to be saved to the database. You can set the value of **Action** to **vbDataActionCancel** to cancel whatever action the user took. Finally, you can change one action to another. For instance, you might change an **AddNew** method to **MoveNext**. You cannot invoke any methods of the **RecordSet** during the **Validate** event.

```
Private Sub Data1_Validate (Action As Integer, _
   Save As Integer)

Select Case Action
  Case vbDataActionUnload
    ' Prevent the form from being unloaded
    Action = vbDataActionCancel
  Case Else
    ' Check the Salary
    If Val(txtSalary.Text) < 0 Then
      MessageBox "Invalid Salary!"
      ' Prevent the action
      Action -= vbDataActionCancel
      ' Put focus on the error
       txtSalary.SetFocus
       txtSalary.SelStart = 0
       txtSalary.SelLength = Len(txtSalary.Text)
      End If
  '    See if ID has changed
      If txtID.DataChanged Then
        MessageBox "Can't Change ID!"
        Save = False
      End If
End Select
```

```
End Sub
```

Table 5.15 Constants for the action argument of the Validate event.

Constant	Meaning
vbDataActionCancel	Cancel the operation when the **Sub** exits.
vbDataActionMoveFirst	**MoveFirst** method.
vbDataActionMovePrevious	**MovePrevious** method.
vbDataActionMoveNext	**MoveNext** method.
vbDataActionMoveLast	**MoveLast** method.
vbDataActionAddNew	**AddNew** method.
vbDataActionUpdate	**Update** operation (but not **UpdateRecord**).
vbDataActionDelete	**Delete** method.
vbDataActionFind	**Find** method.
vbDataActionBookmark	**Bookmark** property set.
vbDataActionClose	**Close** method.
vbDataActionUnload	Form is being unloaded.

Where To Go From Here

In this chapter, we have covered in depth the Data Access Objects, their use both in code and with the Data control, and numerous examples. Read Chapter 4 to see DAO contrasted with other Visual Basic data models. If you have determined that DAO is the correct model for you, I recommend reading Chapter 11 where I discuss advanced database techniques such as transaction and concurrency management.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Previous | Table of Contents | Next

Chapter 6
Remote Data Objects (RDO)

Key Topics:

- The RDO hierarchy
- RDO vs.DAO
- Event-driven data access with RDO
- Managing asynchronous communications
- Transaction and concurrency management

In Chapter 5, I discussed the usage of Data Access Objects (DAO). DAO is most appropriate for desktop database applications, although you can use it, with or without the Jet engine, for true client/server applications. When you create a DAO ODBCDirect workspace, DAO actually communicates with Remote Data Objects (RDO) through a thin layer, as shown in Figure 6.1. If you are using ODBC data sources, it may make more sense to avoid DAO altogether and use RDO instead. Of course, as I discussed at the end of Chapter 4, it makes even more sense for many to move directly to ADO.

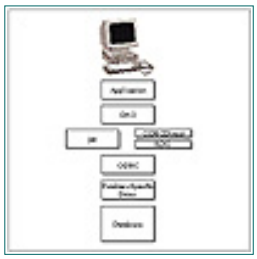

Figure 6.1 How an application communicates with ODBC via the RDO layer.

You can also use RDO to access Jet databases. However, the Jet engine gets

loaded as it does in DAO, and RDO passes its ODBC function requests to the Jet engine, which attempts to map them to Jet functions. This setup is actually more expensive in terms of computing power than simply using DAO. Worse, a given Jet ODBC driver might not support some RDO functions, resulting in potential erratic program behavior.

In this chapter, I review the use of Remote Data Objects, concentrating on in-code techniques with a review of the Remote Data control (RDC).

What Is RDO?

RDO version 1 was introduced with Visual Basic 4 and was tremendously improved with RDO version 2, released with Visual Basic 5. RDO offers a compelling alternative to DAO for those who do not need to use the Jet engine. RDO essentially acts as a thin wrapper around the ODBC API, so it is much more efficient than DAO. DAO developers will recognize familiar features in RDO: Many of the methods and properties are similar and the object hierarchy, although considerably simpler than DAO, is still familiar. Table 6.1 provides a basic cross-reference of RDO objects to their DAO counterparts.

Table 6.1 Cross-reference of RDO objects to DAO objects.

RDO Object	DAO Object
rdoEngine	**DBEngine**
rdoError	**Error**
rdoEnvironment	**Workspace**
rdoConnection	**Database** or **Connection**
rdoQuery	**QueryDef**
rdoColumn	**Field**
rdoParameter	**Parameter**
rdoResultset	**Recordset**
rdoTable	**TableDef**
rdoPreparedStatement	N/A

You will notice two key differences between RDO and DAO: RDO is table and row oriented, whereas DAO is file and record oriented; RDO places more emphasis on procedures and result sets, whereas DAO's primary emphasis is on the retrieval itself. RDO leaves the details of data retrieval to the ODBC driver.

Additionally, RDO provides an event-driven environment in which to manage the database. You can declare most RDO objects with the **WithEvents** clause and then use an event-driven model to handle messages and manage asynchronous communications, for example. This environment not only provides more flexibility than you have with DAO, but it also tends to simplify the program logic.

Overview Of Remote Data Objects

Whereas DAO has up to 17 objects (and 16 collections), RDO has just 10 objects (and 9 collections). If you use the RDC, the Remote Data Objects library is automatically loaded. Otherwise, you must add it as a reference in your VB project. (From the menu, select Project|References. Scroll down the list until you locate Microsoft Remote Data Objects and select it.)

Like DAO, RDO objects are arranged in a hierarchy (see Figure 6.2). The **rdoEngine** is the highest object in the hierarchy. As such, there is only one **rdoEngine** object, and it is created as soon as you reference any RDO object.

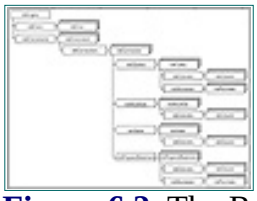

Figure 6.2 The Remote Data Object hierarchy.

Like DAO, RDO maintains information about ODBC errors in the **rdoErrors** collection.

The **rdoEngine** object also has a collection of **rdoEnvironment** objects, which are the equivalent of the DAO **Workspace** objects. The **rdoEnvironment** object contains the **rdoConnection** collection. Each **rdoCollection** object consists of the **rdoQueries**, **rdoResultsets**, **rdoTables**, and **rdoPreparedStatements** collections.

The **rdoQuery** object is similar to DAO's **QueryDef** object and represents a query against the database. Thus, each **rdoQuery** object contains an **rdoColumns** and an **rdoParameters** collection, which are equivalent to DAO's **Fields** and **Parameters** collections. Each **rdoColumn** object represents a column in the query, and each **rdoParameter** object represents a query parameter.

The **rdoResultset** object is similar to the **Recordset** object in DAO. It allows you to manipulate a row from a query and consists of an **rdoColumns** collection.

The **rdoTable** object stores information about an SQL table or view and thus also includes an **rdoColumns** collection. **rdoTable** is included mostly for backward compatibility, and its use is discouraged because its functionality has been incorporated into other objects.

The **rdoPreparedStatement** is also provided for backward compatibility with RDO 1.0. Its functionality—as a database-prepared query or stored procedure—is encompassed by the **rdoQuery** object.

As with DAO, you can create RDO objects by using the Remote Data control or by using methods of the parent object to create them. For example, you can create the **rdoResultset** object by using the **rdoConnection** object's **OpenResultset** method. The **rdoEngine** object is created automatically. The first instance of the **rdoEnvironment** object is created whenever the

rdoEngine object is created. Additional instances of it are created with the **rdoEngine** object's **rdoCreateEnvironment** method. The Remote Data control can create **rdoConnection** and **rdoResultset** objects. The **rdoParameter** object is created automatically for parameter queries. The **rdoTable** object is created automatically whenever a table or view is referenced. The **rdoError** object is created automatically any time an ODBC error occurs.

TIP
The **rdoParameter** object and **rdoParameters** collection are only created and populated if the ODBC driver supports the **SQLNumParams** function. Refer to your ODBC driver documentation if you encounter any problems. If the ODBC driver does not support parameters or is otherwise unable to parse the query, the **rdoParameter** objects and **rdoParameters** collection are not created and a trappable runtime error occurs.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

RDO Collections

As noted, all RDO objects are maintained in collections except the **rdoEngine** object, of which there can only be one. However, Visual Basic 6 allows you to create **rdoConnection** and **rdoQuery** objects that are not part of a collection. Therefore, these objects are not automatically appended to collections when created; you must use the **Append** method to add these objects to the collections and the **Delete** method to remove them. All other objects are automatically appended to their collections, so the collections have only the **Refresh** method and **Count** property.

Exploring Remote Data Objects

In the following sections, I discuss each of the Remote Data Objects and provide examples for their usage. As in Chapter 5, it is not my intention to chop down the rain forest, so I concentrate on the key aspects of the different properties and methods of each object. If you need more information, you might want to refer to the Visual Basic documentation. Where I have included a complete program to illustrate a technique, you can find the source code on the CD-ROM. Note that you might need to alter the path to the database in the code or configure the ODBC data source name prior to running the examples.

The rdoEngine Object

The **rdoEngine** object is the owner of all other RDO objects. It is essentially the RDO library and provides the interface between your VB application and ODBC. ODBC is exposed via its API. RDO places user-friendly wrappers around the API in the form of RDO properties and methods.

The **rdoEngine** is responsible for creating the **rdoEnvironment** and **rdoError** objects. It also provides default values for other objects in the RDO hierarchy.

rdoEngine has one event—**InfoMessage**—which occurs when informational

messages (ODBC **SQL_SUCCESS_WITH_INFO** return codes) are received from the ODBC driver. This event is triggered once for each set of messages received. This event replaces the need for the **ErrorThreshold** and **rdoDefaultErrorThreshold** properties, which are retained for backward compatibility with RDO version 1.

To take advantage of the **InfoMessage** event, you declare the object using the **WithEvents** clause: **Private WithEvents rdoEng As rdoEngine**.

Once you declare a reference to the **rdoEngine** object using the **WithEvents** clause, you can intercept any messages from the database. Listing 6.1, later in this chapter, illustrates the use of this technique.

```
Private Sub rdoEng_InfoMessage ()

Dim sMsg As String
Dim vErr As Variant

' Iterate through all errors
For each vErr in rdoErrors
  sMsg = str$(vErr.Number) & ": " & vErr.Description
  ' Display text of message
  MsgBox "Info - " & sMsg
Next
```

The **rdoEngine** object has a number of properties that you use to set default behaviors of other objects. You can individually override these behaviors when you create the other objects:

- The **DefaultCursorDriver** property specifies which kind of cursor driver to be used by default. Table 6.2 lists the possible values. In general, you will get better performance for small result sets using the ODBC driver cursor library. For result sets of more than 100 rows or so, you should use the server cursor. For optimistic batch processing, use **rdUseClientBatch**. If your result set will have only one row, do not use a cursor. The default value is **rdUseIfNeeded**.

Table 6.2 RDO cursor driver constants.

Constant	Description
rdUseIfNeeded	Use server side if available; if not, use client side. If client side is also not available, use none.
rdUseODBC	Use the ODBC driver cursor library.
rdUseServer	Use server cursor.
rdUseClientBatch	Use optimistic batch cursor library.
rdUseNone	No cursor. Appropriate only with one row at a time.

- The **rdoDefaultPassword** is a string containing the password that will be used when an **rdoEnvironment** object is created, unless another one is specified. If not specified, the default is a zero-length string.
- The **rdoDefaultUser** is a string containing the user name that will be used when an **rdoEnvironment** object is created, unless another one is specified. If

not specified, the default is a zero-length string.

- The **rdoDefaultLogInTimeOut** property is a **Long** that specifies how many seconds to wait when logging on to the server before an error is generated. The default is 15. A value of 0 indicates there is no limit. This value will be used if you do not specify a **LogInTimeout** property for the **rdoEnvironment** object.
- The **rdoLocaleID** property is a **Long** in which you specify the language in which messages should be displayed. If it is not supplied, the Windows locale will be used. Messages are generated from a locale-specific DLL. If you specify a locale but the user does not have the correct DLL, **rdLocaleEnglish** is used because it does not require a DLL. Table 6.3 shows the valid values.

Table 6.3 Valid RDO locale constants.

Constant	Description
rdLocaleChinese	Chinese
rdLocaleEnglish	English
rdLocaleFrench	French
rdLocaleGerman	German
rdLocaleItalian	Italian
rdLocaleJapanese	Japanese
rdLocaleKorean	Korean
rdLocaleSimplifiedChinese	Simplified Chinese
rdLocaleSpanish	Spanish
rdLocaleSystem	The Windows locale

- The **rdoVersion** property returns a string containing the RDO version number.

The **rdoEngine** object also supports these two methods:

- **rdoCreateEnvironment** creates a new **rdoEnvironment** object and appends it to the **rdoEnvironments** collection. The syntax is shown in the following code segment. The **name** argument must be unique within the collection. If the name is not specified, the object will not be appended to the collection. You must specify the **user** and **password** arguments, although you can reference the defaults from the **rdoEngine** object. When you create the **rdoEngine** object, a default **rdoEnvironment** object is automatically created, including the default user and password properties. Listing 6.1, later in this chapter, provides an example of this method.

```
Private WithEvents rdoEng As rdoEngine
rdoEng.rdoCreateEnvironment (name, user, password)
```

- The **rdoRegisterDataSource** essentially duplicates the functionality of the ODBC Administrator in the Windows Control Panel by adding or updating ODBC data source information to the Windows Registry. If the data source name does not exist, it is added; otherwise, it is updated. It is recommended that you use the ODBC Administrator because the requirements for different ODBC drivers vary. For more information, see the VB help file.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The rdoEnvironments Collection And rdoEnvironment Object

The **rdoEnvironment** object represents a single transaction against a database within the context of a single user/password combination. All activity against that database will occur within the context of the **rdoEnvironment** object. A default **rdoEnvironment** object is created when the **rdoEngine** object is instantiated. If you need more objects, you use the **rdoCreateEnvironment** method of the **rdoEngine** object. Once the object is created, you can alter its **User** and **LoginTimeout** properties but not the **Password** property. You can close any **rdoEnvironment** object using the **Close** method except the default object, which cannot be destroyed.

The **Name** property of the default **rdoEnvironment** object is **"Default_Environment"**. The user name and password are both empty strings. If you create another **rdoEnvironment** object and do not specify its **Name** property, it is created as a standalone object (it is not appended to the collection). The **Name** property is then determined by the remote data source and is read-only. This behavior may be an advantage because the object is not exposed to other in-process DLLs. Once the properties of the standalone object are set, you can append it to the collection using the **Add** method. You can use the **Remove** method to remove any **rdoEnvironment** object except the default.

In ODBC, there is a single environment handle—a memory pointer to the ODBC session. All **rdoEnvironment** objects share a single **hEnv** property, which is the ODBC environment handle. Because of this, you can manage a single transaction across multiple **rdoConnection** objects using the **BeginTrans**, **CommitTrans**, and **RollbackTrans** methods. Conversely, you can create multiple **rdoEnvironment** objects if you need to maintain multiple simultaneous transactions. If you issue a **CommitTrans** or a **RollbackTrans**

in an **rdoEnvironment** object, all **rdoConnection** objects are affected.

TIP
Warning!
Do not attempt to use ODBC handles (environment, statement, and connection) unless you fully understand the ODBC API. You must allocate the handles in a specific order, and many ODBC API functions cannot be performed until other functions are executed. Arbitrary use of the handles and API functions can result in program crashes and computer lockups.

As with the **rdoEngine** object, you can program the **rdoEnvironment** object in an event-driven manner. It supports three events: **BeginTrans**, **CommitTrans**, and **RollbackTrans**. Each of these events is fired *after* the corresponding transaction method has completed. To take advantage of these events, you must declare the **rdoEnvironment** object using the **WithEvents** clause as shown:

```
Dim rdoEnv WithEvents As rdoEnvironment
' Set a reference to the default object
Set rdoEnv As rdoEnvironments(0)
```

The **CommitTrans** event, for example, fires after the **CommitTrans** method completes:

```
Private Sub rdoEnv_CommitTrans()
  MsgBox "Transaction Committed"
End Sub
```

Listing 6.1, later in this section, illustrates many of the methods and properties of **rdoEnvironment**.

The **rdoEnvironment** object supports a number of properties:

- The **CursorDriver** property is a **Long** specifying what type of cursor to use within the **rdoEnvironment**. Table 6.2 earlier in the chapter lists possible values. If **CursorDriver** is not specified, the **rdoEngine** object's default cursor driver is used.
- The **hEnv** property is a **Long** containing the ODBC environment handle.
- The **LoginTimeout** property, if specified, determines how many seconds can pass when attempting to log on to the database before an error occurs. If not specified, the default value from **rdoEngine** is used.
- The **Password** and **UserName** properties are both strings and specify the user ID and password for the **rdoEnvironment** object. If not specified, these values default to the values from **rdoEngine**. Once set, the **Password** property is write-only; it cannot be read in your code.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The **rdoEnvironment** object supports several methods:

- The **BeginTrans** method begins a new transaction, as discussed earlier in this section. The scope of the transaction is across all **rdoConnection** objects within the **rdoEnvironment** object. If you need multiple transactions, you must create multiple **rdoEnvironment** objects. Transaction management is discussed in more detail in Chapter 11. Listing 6.1 provides an example of a program that manipulates the salary data of employees in the context of a transaction. The user is allowed to save or discard changes after all of the salaries are displayed and modified. The application, a modification of the transaction example in Chapter 5, is shown in Figure 6.3. Although the application takes advantage of some event-driven concepts, it uses **While Wend** loops to determine when an asynchronously called method has completed. Listing 6.2 later in this chapter provides a completely event-driven model.

Listing 6.1 The modification of employee salaries.

```
Option Explicit
Private WithEvents rEng  As rdoEngine
Private WithEvents renvCoriolis As rdoEnvironment
Private WithEvents rconEmp As rdoConnection
Private WithEvents rrsEmp As rdoResultset
Private rqEmpSal As rdoQuery

Private Sub Command1_Click(Index As Integer)

Dim iRtn As Integer
Dim sRaise As String
Dim sMsg As String
Dim sConn As String
Dim cSalary As Currency
Dim cSaveSal() As Currency

Select Case Index
  Case 0
```

```
' Create environment
Set renvCoriolis = rdoEngine.rdoCreateEnvironment _
  ("Emp", "coriolis", "coriolis")
  ' Set properties
  With renvCoriolis
    .CursorDriver = rdUseOdbc
    .LoginTimeout = 10
    ' Create connection
    Set rconEmp = .OpenConnection(dsName:="", _
      Prompt:=rdDriverNoPrompt, _
      Connect:="DSN=Coriolis VB Example;UID=Coriolis;" & _
        "PWD=Coriolis;", Options:=rdAsyncEnable)
  End With

' Wait for connection to complete
While rconEmp.StillConnecting
   DoEvents
Wend
' Set properties of the connection
With rconEmp
  .QueryTimeout = 10
  ' Create query object
  Set rqEmpSal = .CreateQuery("EmpSelect", _
    "Select * from employee")
    ' Create resultset object
  Set rrsEmp = .OpenResultset("EmpSelect", rdOpenDynamic, _
    rdConcurValues, rdAsyncEnable + rdExecDirect)
      ' Idle until the query is complete
  While rrsEmp.StillExecuting
    DoEvents
  Wend
End With

' Begin transaction
renvCoriolis.BeginTrans
' Scroll through rows
 With rrsEmp
   ' Set up array to save old salaries
   ReDim cSaveSal(rrsEmp.RowCount)

   Do Until .EOF
     txtFields(0) = !emp_no
     txtFields(1) = !emp_fname
     txtFields(2) = !emp_lname
     txtFields(3) = Format$(!emp_Salary, "###,##0.00")
     cSaveSal(rrsEmp.AbsolutePosition) = !emp_Salary
     ' Prompt for changes to salary
     sMsg = "What percent raise for " & !emp_fname & _
        " " & !emp_lname & " making $" & _
     Format$(!emp_Salary, "###,##0.00") & "?"
     sRaise = InputBox$(sMsg, "Raise", "0")
     cSalary = !emp_Salary * (1 + Val(sRaise) / 100)
```

```
          If cSalary <> !emp_Salary Then
            ' If changed, edit the record
            .Edit
            !emp_Salary = cSalary
            ' Save the change
            .Update
          End If
          ' Move to next record
          .MoveNext
        Loop
        ' Commit the changes?
        If MsgBox("Save all changes?", vbYesNo + vbQuestion, _
          "Commit or Rollback") = vbYes Then
          renvCoriolis.CommitTrans
        Else
          renvCoriolis.RollbackTrans
        End If
        ' Display changes (if any)
        .MoveFirst

        Do While Not .EOF
          txtFields(0) = !emp_no
          txtFields(1) = !emp_fname
          txtFields(2) = !emp_lname
          txtFields(3) = cSaveSal(rrsEmp.AbsolutePosition)
          txtFields(4) = !emp_Salary
          If MsgBox("Show next record?", _
            vbOKCancel + vbQuestion, _
            "Display Salary Changes") = vbCancel Then
            Exit Do
          End If
          .MoveNext
        Loop
        .Close
      End With
    ' Close the objects
    rqEmpSal.Close
    rconEmp.Close
    renvCoriolis.Close
  Case 1
    End
End Select

End Sub

Private Sub rconEmp_Connect(ByVal ErrorOccurred As Boolean)
Select Case ErrorOccurred
    Case True
        MsgBox "Connection Failed!", vbOKOnly + vbCritical
    Case False
        MsgBox "Connection Succeeded!"
End Select
```

```
End Sub

Private Sub rEng_InfoMessage()

Dim sMsg As String
Dim vErr As Variant

For Each vErr In rEng.rdoErrors
    sMsg = Str$(vErr.Number) & _
      ": " & vErr.Description
    MsgBox sMsg, vbOKOnly + vbInformation, "ODBC Message"
Next

End Sub

Private Sub renvCoriolis_BeginTrans()
' Put code here for begin trans event

End Sub

Private Sub renvCoriolis_CommitTrans()
' Put code here for commit trans events
MsgBox "Changes Saved"
End Sub

Private Sub renvCoriolis_RollbackTrans()
' Put code here for rollback events
MsgBox "Changes Discarded"
End Sub
```

Figure 6.3 This program, from Listing 6.1, provides examples of using various Remote Data Objects.

- The **CommitTrans** method commits all changes to the database across the entire **rdoEnvironment** object. Likewise, the **Roll-backTrans** method rolls back all changes across the entire **rdoEnvironment** object. If there are several active connections (**rdoConnection** objects), the **CommitTrans** and **RollbackTrans** methods affect all of them.
- The **OpenConnection** method creates a new **rdoConnection** object, which I discuss later in this chapter. The syntax of the method is shown in the next code example. Listing 6.1 has an example of the method being used with a DSN, whereas Listing 6.2 shows a DSN-less connection. In the syntax, **dsName** is the name of the data source or is a zero-length string. If you supply the **DSN** parameter in the connect argument, you do not need to specify the **dsName** argument, but you must still specify an empty string, as shown in Listing 6.1. The **Prompt** argument is a constant, as listed in Table 6.4, and controls whether (and to what extent) ODBC connection dialogs are presented to the user. The **read_only** argument is set to **True** if the connection is to be opened as read-only. If the argument is not specified, the default is **False**, which makes the data read-write. **Connect** is a string specifying connection parameters. If the user ID and

password parameters are not supplied, they are taken from the **rdoEnvironment** object and the **dsName** argument must be supplied. The only setting allowed for the **Options** argument is the constant **rdAsyncEnabled**, which causes the connection to be performed asynchronously. The code example in Listing 6.1 connects in this manner and then executes a **While Wend** loop until the connection is established. Be careful not to attempt to manipulate the **rdoConnection** object until the connection is established, or you will cause an error in your application.

```
rdoEnvironment.OpenConnection (dsName, Prompt, read_only, _
  Connect, Options)
```

Table 6.4 Valid connection prompt constants.

Constant	Description
rdDriverPrompt	If no **dsName** is specified, use default. Otherwise, present ODBC connect dialog.
rdDriverNoPrompt	If not enough connection information, prompt for missing parameters.
rdDriverComplete	If connection includes **DSN** parameter, prompt for any missing information. Otherwise, behaves like **rdDriverPrompt**.
rdDriverCompleteRequired	Behaves like **rdDriverComplete** except doesn't allow the user to change any information already provided.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

DSN-Less Connections

A problem with ODBC is that you can't always rely on the user PC having the correct ODBC settings. When you specify a DSN (data source name), the ODBC manager looks into the Windows Registry for an ODBC configuration matching that name. If it is not found, ODBC may be able to prompt your user for the connection parameters. However, a better answer may be to provide a *DSN-less* connection string. This tells the ODBC manager which database driver to load and then passes on to that driver the information needed to connect to the database without relying on a DSN. The parameters you need to supply vary from database to database. The following code example is a modification of the code shown in Listing 6.1 to connect to the sample Sybase SQL Anywhere database (the entire listing is on the CD-ROM in the project file RDOTransactionsNoDSN.VBP):

```
' Be sure to modify the path!
Set rconEmp = .OpenConnection(dsName:="", _
  Prompt:=rdDriverNoPrompt, Connect:="UID=Coriolis;" & _
  "PWD=Coriolis;Driver=Sybase SQL Anywhere 5.0;" & _
  "DBF=C:\Examples\Coriolis VB Example.db;" & _
  "DBN=Coriolis;DSN='", Options:=rdAsyncEnable)
```

The equivalent connection with Microsoft SQL Server appears in the next code segment. It is simpler than the Sybase SQL Anywhere example because my SQL Anywhere server is a standalone product. (In the preceding example, I need to not only connect to the server, but also I have to start the database engine.) If you connect to an engine that is already running, you generally can omit the **DBF** and **DBN** parameters and substitute an **ENG** parameter that specifies the name of the server.

Listing 6.2, later in this chapter, also performs a DSN-less connection.

```
Set rconEmp = .OpenConnection(dsName:="", _
  Prompt:=rdDriverNoPrompt, _
  Connect:="UID=Coriolis;PWD=Coriolis;Driver=SQL Server;" & _
  "Server=Test;DSN='", Options:=rdAsyncEnable)
```

The rdoConnections Collection And rdoConnection Object

The **rdoConnections** collection contains all **rdoConnection** objects within an **rdoEnvironment** object. You normally create the **rdoConnection** object using the **OpenConnection** method of the **rdoEnvironment** object. When you do so, it is automatically appended to the **rdoConnections** collection. However, you can also create a standalone **rdoConnection** object using the **EstablishConnection** method (discussed later in this section). When you do create a standalone **rdoConnection** object, it is not appended to the collection automatically. You can use the collection's **Add** method to do so.

You can manipulate an **rdoConnection** object synchronously or asynchronously. It also supports event-driven techniques. To take advantage of event-driven processing, you must use the **WithEvents** keyword when declaring the object variable: **Private WithEvents rconEmp As rdoConnection**.

The **rdoConnection** object generates a half-dozen events that can be valuable in an asynchronous environment.

RDO Event-Driven Asynchronous Programming

Listing 6.2 is an application that uses Remote Data Objects in a completely event-driven, asynchronous environment. The application, shown in Figure 6.4, begins by creating an **rdoEnvironment** object. Nine textbox controls display data. Four command buttons connect to the database, run a query, disconnect from the database, and close the application. For the four navigation buttons, I used the Image control. I use the outer buttons as **BOF** and **EOF** buttons to move to the first and last records. I use the inner two buttons to move backward and forward one row at a time.

Listing 6.2 The event-driven, asynchronous RDO application.

```
Option Explicit
Private WithEvents rEng  As rdoEngine
Private WithEvents renv As rdoEnvironment
Private WithEvents rcon As rdoConnection
Private WithEvents rrs As rdoResultset
Private rq As rdoQuery
Private Enum query_direction
    first_time = 0
    forward = 1
    backward = 2
End Enum

Private Sub Command1_Click(Index As Integer)

Dim iRtn As Integer
Dim sRaise As String
Dim sMsg As String
Dim sConn As String
Dim cSalary As Currency
Dim cSaveSal() As Currency

Select Case Index
  Case 0
    ' Connect to the database
```

```
    Caption = "Connecting …"
    Command1(0).Enabled = False
    ' Be sure to change path in your application!
    Set rcon = renv.OpenConnection(dsName:="", _
      Prompt:=rdDriverNoPrompt, _
      Connect:="UID=Coriolis;PWD=Coriolis;" & _
      "Driver=Sybase SQL Anywhere 5.0;" & _
      "DBF=C:\ \Examples\Coriolis VB Example.db;" & _
      "DBN=Coriolis;DSN=''", Options:=rdAsyncEnable)
  Case 1
    ' Run query
    Set rq = rcon.CreateQuery("Test", _
      "Select customer.cust_no, customer.cust_lname, " & _
      "customer.cust_fname," & _
      "item.item_desc, line_item.line_item_no, " & _
      "line_item.line_qty, line_item.line_price, " & _
      "line_item.line_total, orders.ord_no, " & _
      "orders.ord_date " & _
      "From customer, item, line_item, orders " & _
      "Where customer.cust_no = orders.ord_cust_no " & _
      "And line_item.line_ord_no = orders.ord_no " & _
      "And item.item_no = line_item.line_item_no " & _
      "Order By customer.cust_lname, customer.cust_fname, " & _
      "orders.ord_no, line_item.line_no")
    ' Set properties of the connection
    With rcon
      .QueryTimeout = 3
      ' Create resultset object
      Set rrs = .OpenResultset("Test", rdOpenDynamic, _
        rdConcurValues, rdAsyncEnable + rdExecDirect)
    End With
  Case 2
    ' Disconnect
    rcon.Close
    ' Disable movement buttons
    Image1(0).Enabled = False
    Image1(1).Enabled = False
    Image1(2).Enabled = False
    Image1(3).Enabled = False
    ' Enable connect button
    Command1(0).Enabled = True
  Case 3
    End
End Select

End Sub

Private Sub Form_Load()

' Set the query
' Enable/disable command buttons
Command1(1).Enabled = False
Command1(2).Enabled = False
```

```
Command1(0).Enabled = True
Command1(3).Enabled = True
Image1(0).Enabled = False
Image1(1).Enabled = False
Image1(2).Enabled = False
Image1(3).Enabled = False

' Force the form to display
Show

' Create environment
Set renv = rdoEngine.rdoCreateEnvironment _
  ("Test", "coriolis", "coriolis")
' Set properties
With renv
  .CursorDriver = rdUseOdbc
  .LoginTimeout = 10
End With

End Sub

Private Sub Form_QueryUnload(Cancel As Integer, _
  UnloadMode As Integer)

' Ensure all objects are closed
rq.Close
rcon.Close
renv.Close

End Sub

Private Sub Image1_Click(Index As Integer)

Select Case Index
  Case 0
    ShowRec backward
  Case 1
    ShowRec forward
  Case 2
    rrs.MoveFirst
    ShowRec first_time
  Case 3
    rrs.MoveLast rdAsyncEnable
    ShowRec first_time
End Select

End Sub

Private Sub rcon_BeforeConnect(ConnectString As String, _
  Prompt As Variant)

Dim sMsg As String
sMsg = "About to connect to the database using" & vbCrLf & _
```

```
  ConnectString
MsgBox sMsg, vb0konly + vbInformation

End Sub

Private Sub rcon_Connect(ByVal ErrorOccurred As Boolean)

Select Case ErrorOccurred
  Case True
    ' Connect failed!
    MsgBox "Connection Failed!", vbOKOnly + vbCritical
    Dim sMsg As String
    Dim vErr As Variant
    For Each vErr In rEng.rdoErrors
      sMsg = Str$(vErr.Number) & _
        ": " & vErr.Description
      MsgBox sMsg, vbOKOnly + vbInformation
    Next
    ' Reenable connect button
    Command1(0).Enabled = True
    Caption = "Connect Failed!"
  Case False
    MsgBox "Connection Succeeded!"
    ' Enable query button
    Command1(1).Enabled = True
    Caption = "Connected"
End Select

End Sub

Private Sub rcon_Disconnect()

Dim iCtr As Integer

MsgBox "Disconnected"
' Enable connect button
Command1(0).Enabled = True
' Disable move buttons
Image1(0).Enabled = False
Image1(1).Enabled = False
' Clear all text boxes
For iCtr = 0 To 8
  txtFields(iCtr).Text = ""
Next

End Sub

Private Sub rcon_QueryComplete(ByVal Query As RDO.rdoQuery, _
  ByVal ErrorOccurred As Boolean)

Select Case ErrorOccurred
  Case True
    Dim sMsg As String
```

```
    Dim vErr As rdoError
    For Each vErr In rdoErrors
      sMsg = Str$(vErr.Number) & _
        ": " & vErr.Description
      MsgBox sMsg, vbOKOnly + vbInformation, Query.SQL
    Next
  Case False
    MsgBox "Query Successful"
    ' Enable movement buttons
    Image1(0).Enabled = True
    Image1(1).Enabled = True
    Image1(2).Enabled = True
    Image1(3).Enabled = True
    Command1(1).Enabled = False
    Command1(2).Enabled = True
    ' Display first record
    Call ShowRec(first_time)
End Select

End Sub
Private Sub ShowRec(direction As Integer)

Select Case direction
  Case first_time
    ' Do nothing
  Case forward
    If rrs.AbsolutePosition = rrs.RowCount Then Exit Sub
    rrs.MoveNext
  Case backward
    If rrs.AbsolutePosition = 1 Then Exit Sub
    rrs.MovePrevious
End Select
' Display all fields
With rrs
  txtFields(0) = !cust_no
  txtFields(1) = !cust_fname
  txtFields(2) = !cust_lname
  txtFields(3) = !ord_no
  txtFields(4) = !ord_date
  txtFields(5) = !line_item_no
  txtFields(6) = !item_desc
  txtFields(7) = !line_qty
  txtFields(8) = !line_price
End With
End Sub

Private Sub rcon_QueryTimeout(ByVal Query As RDO.rdoQuery, _
  Cancel As Boolean)

' Query timed out
Dim iRtn As IFontDisp
iRtn = MsgBox("The query timed out! Keep working?", _
  vbYesNo + vbQuestion, "Query Error")
```

```
If iRtn = vbNo Then
  Cancel = True
Else
  Cancel = False
End If
End Sub

Private Sub rEng_InfoMessage()

' This code illustrates the interception of any informational
' messages from ODBC
Dim sMsg As String
Dim vErr As Variant
For Each vErr In rEng.rdoErrors
  sMsg = Str$(vErr.Number) & _
    ": " & vErr.Description
  MsgBox sMsg, vbOKOnly + vbInformation, "ODBC Message"
Next
End Sub

Private Sub renv_BeginTrans()
' Put code here for begin trans event
' The application does not actually perform updates
End Sub

Private Sub renv_CommitTrans()
' Put code here for commit trans events
MsgBox "Changes Saved"
End Sub

Private Sub renv_RollbackTrans()
' Put code here for commit trans events
MsgBox "Changes Discarded"
End Sub
```

Figure 6.4 Data is displayed in the event-driven, asynchronous query application.

Of the four command buttons, only Connect and Close are enabled. The Run Query and Disconnect buttons and the navigation buttons are disabled.

When the user clicks the Connect button, the application attempts to perform a DSN-less connect to the database. The **rdoConnection** object's **Connect** event is used to determine whether the connect was successful. If not, all error messages are displayed. If the connection is successful, the Connect button is disabled and the Run Query button is enabled.

When the user clicks Run Query, the application creates an **rdoQuery** object composed of a join of three tables and then attempts to open an **rdoResultset** object using the **rdoQuery** object as the source. The **QueryTimeout** property is deliberately set to a short interval: three seconds. If the query does not complete in three seconds, the **QueryTimeout** event is fired and the user has the choice of allowing the query to continue working. When the query is completed, the **QueryComplete** event of the **rdoConnection** object is triggered. In this event, I evaluate whether the query ran successfully. If not, I iterate through all the **rdoError**

objects. If the query was successful, I call a general procedure to display the first record. The Run Query button is disabled and the Disconnect button is enabled. The navigation buttons are also enabled.

The user can now use the navigation methods to move forward and backward through the data. I did not include the capability for updating the data because this is a query-only application.

If the user clicks the Disconnect button, all objects are closed and the navigation and Disconnect buttons are disabled. The Connect button is re-enabled.

Throughout the listing, various other events monitor database RDO operations.

All database activities, including the **MoveLast** method, occur in an asynchronous manner. In this manner, control returns to the program as soon as database events occur. The use of the event-driven model replaces the awkward coding in Listing 6.1 where I performed a loop after calling an asynchronous method waiting for the method to complete.

- The **BeforeConnect** event is triggered immediately before an attempt is made to connect to the database. It provides an opportunity to, for example, prevent the connection from occurring. For example, your application may automatically connect to the database but need to occasionally work in "offline" mode. The **BeforeConnect** event provides an opportunity for you to ask the user whether he or she wants to proceed with the connect. The two parameters passed to the event are **ConnectString** and **Prompt**. The following example displays the connect string prior to connecting:

```
Private Sub rcon_BeforeConnect(ConnectString As String, _
   Prompt As Variant)
MsgBox ConnectString
End Sub
```

- The **Connect** event is fired immediately after the connect has completed. This can be valuable when your program needs to know when it is okay to manipulate the connection. For instance, you do not want to execute a query against the database until you have successfully connected. The **Connect** event might also provide an opportunity to display and evaluate any informational messages added to the **rdoErrors** collection, as shown in the following example. In Listing 6.2, I use the **Connect** event to ensure a successful connection to the database.
- The **Disconnect** event is triggered when the connection is successfully closed. You might place code in this event to perform housecleaning chores such as closing the **rdoEnvironment** object. No arguments are passed to this event.
- The **QueryComplete** event is triggered whenever *any* query executing within the **rdoConnection** object is completed. The completion of a query does not necessarily indicate it was successful. Therefore, two arguments are sent to the event: **Query** is a reference to the query that just completed; **ErrorOccurred** is a Boolean that returns **True** if an error was encountered. If an error was encountered, you should check messages in the **rdoErrors** collection. In Listing 6.2, the **QueryComplete** event determines whether to display data and enable the navigation buttons in the application.
- The **QueryTimeout** event is triggered when a query, whether synchronous or asynchronous, has been running for the number of seconds specified in the **QueryTimeout** property. Two arguments are passed: a reference to the query that is running and a Boolean **Cancel** that indicates whether to cancel the query. **Cancel** defaults to **True**, so if you don't code this event, the running query will automatically cancel. If you set the value to **False**, the query will continue running for as many seconds as

specified in the **QueryTimeout** property. The event occurs at the **rdoConnection** level for any query within the connection.

- The **WillExecute** event occurs immediately before a query is to run anywhere in the **rdoConnection**. Two arguments are passed: **Query** is a reference to the query that is about to execute, and **Cancel** is a Boolean that allows you to stop the query from running. The default is **False**, meaning that the query will not be prevented from running. **True** has the effect of canceling the query and will also generate a runtime error indicating that the query was canceled. You can use this event to prevent an update of the database or to change the query. An example follows:

```
Private Sub rcon_WillExecute (Query As rdoQuery, _
  Cancel As Boolean)
Dim iRtn As Integer
' Change the query
Query.SQL = "Select * From Employee"
' Give the user a chance to cancel the query
iRtn = MsgBox ("Run the query: " & Query.SQL & _
  "?", vbYesNo + vbQuestion)
If iRtn = vbNo Then
  ' Cancel the query
  Cancel = True
End If
End Sub
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The **rdoConnection** object has a number of properties you use to manipulate the connection:

- The **AsyncCheckInterval** property is a **Long** that specifies how many milliseconds Visual Basic should wait before polling the database to determine whether an asynchronous query has completed. The default is 1,000 (1 second). Because Visual Basic must poll the database to determine whether the query is complete, setting this value too low can adversely impact performance by increasing network and database traffic. On the other hand, if the setting is too large, the user may wait longer for the data. In general, short queries can default to 1 second. For longer queries, consider increasing this value to minimize the number of hits on the database server.
- The **Connect** property specifies the database connection parameters. I discussed the requirements for this property earlier in the chapter in the **OpenConnection** method of the **rdoEnvironment** object.
- The **CursorDriver** property specifies the type of cursor to use. If not specified, the value will default from the **rdoEnvironment** object's **CursorDriver** property.
- The **hDbc** property is a **Long** that contains the ODBC connection handle. Do not use this handle if you are not familiar with the ODBC API.
- The **LastQueryResults** property returns a reference to the **rdoResultset** associated with the last query that successfully completed.
- The **LoginTimeout** property is a **Long** specifying how many seconds to allow for logging on to the database server before generating an error. If not specified, the value will default from the **rdoEnvironment** object.
- The **LogMessages** property is a string used to set ODBC logging. If not set (or set to an empty string), no logging is done. If the value is set to a valid file path and name, ODBC messages are written to that file.

You should let only one application at a time perform logging, and you should minimize the amount of logging you do because ODBC operations are much slower while performing logging. Setting the **LogMessages** property is the same as using the **Log** function in the ODBC Administrator applet in the Control Panel.

- The **QueryTimeout** property specifies how many seconds to wait before an active query within the connection times out. If not specified, the value defaults from **rdoEnvironment**. If the value is larger or smaller than that permitted by the database, the minimum value allowed by the database is used instead.
- The **RowsAffected** property returns the number of rows deleted, inserted, or updated in the most recent query within the **rdoConnection** object.
- The **StillExecuting** property determines whether an asynchronous operation is completed. Listing 6.1 showed the use of this property to determine when a connection was completed, whereas Listing 6.2 used an event-driven method (**Connect**) to signal the completion of the connection. Likewise, Listing 6.1 used this property to determine when a query had finished, whereas Listing 6.2 used the **QueryComplete** event. While this property is **False** (the asynchronous operation is not complete), no other properties of the object are available.
- The **Transactions** property is a Boolean that indicates whether the connected database supports transaction processing. If this property returns **False**, the database does not support transaction processing and the use of the **BeginTrans**, **CommitTrans**, and **RollbackTrans** methods will have no effect.
- The **UpdateOperation** property is used for optimistic batch updates within the connection. It is a **Long** set to either **rdOperationUpdate** (use an **Update** statement for each row being changed) or **rdOperationDelIns** (use a **Delete** statement followed by an **Insert**). Generally, you are better off using **rdOperationUpdate** because it entails less work on the database server and less network traffic. However, if you need to change the primary key of a row, most databases require that you delete the original row and add a new row.
- The **Version** property returns the ODBC version number to which the ODBC driver conforms as a string. Although you may be using ODBC version 3.5, if the underlying ODBC driver conforms to ODBC version 2, this property will reflect version 2.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The **rdoConnection** object supports several methods:

- The **BeginTrans**, **CommitTrans**, and **RollbackTrans** methods perform transaction management chores, as discussed earlier in the chapter when I discussed the **rdoEnvironment** object. Although you can manipulate transactions at the connection level, the scope is environment-wide.
- The **Cancel** method is used to cancel an asynchronous connection. You can only invoke this method while the **StillExecuting** property is **True**.
- The **CreateQuery** method is used to create a new **rdoQuery** object and add it to the **rdoQueries** collection. If you create an **rdoQuery** object using the **DIM** statement (as I do in Listings 6.1 and 6.2), the object is not automatically appended to the collection. The **rdoQuery** object is not associated with any **rdoConnection** object until you set the **ActiveConnection** property of the **rdoQuery**. The syntax for the method is shown in the following code example. The **Name** argument is the **Name** property to be assigned to the object, and the optional **SQL** argument is the **SQL** property to be assigned to the object. If the arguments are not set when the object is created, they must be set prior to using the object (that is, before executing the query).

```
rdoConnection.CreateQuery Name [,SQL]
```

TIP
When To Prepare A Statement
A good rule of thumb when determining whether to prepare a statement is that the amount of benefit derived is inversely proportional to how well the database is maintained. When preparing a query, the database's query optimizer uses various statistics to plan an access path. If these statistics are not up-to-date, the preparation process could actually hurt the query performance. If you don't prepare a statement—in other words, if the statement is executed dynamically—many RDBMSs use an optimization strategy more akin to generalized rules. Check your specific database's documentation for more details.

- The **Close** method closes the **rdoConnection** object.
- The **EstablishConnection** method is used to connect to the server (if you did not do so when you created the **rdoConnection** object). The syntax follows. The arguments are the same as those for the **OpenConnection** method of the **rdoEnvironment** object, discussed earlier in this chapter.

```
rdoConnection.EstablishConnection [prompt, read_only, options]
```

- The **Execute** method runs a specified query, including update queries, against the database. The syntax appears in the next code example. **Source** is a string containing either an SQL

statement or the name of an **rdoQuery** object. For specifying an **rdoQuery** object, the name is case-sensitive. The **Options** argument can contain one or both of the following constants: **rdAsyncEnable** causes the operation to run asynchronously. **rdExecDirect** causes the query to run without first being prepared on the database. In this context, preparation is similar to compilation. If the query will run only once or if it is a short query, it will usually run marginally faster if you specify **rdExecDirect**. If the query is likely to run several times or if it is longer in duration (more than three seconds or so), you might get somewhat faster performance by not specifying **rdExecDirect**. You can execute row-returning queries and stored procedures using **Execute**; however, it is not a good idea because there is no way to get the return values. Instead, you should use an **rdoResultset** for these types of tasks. Listing 6.3, later in this chapter, uses this method to execute an action query.

```
rdoConnection.Execute Source, Options
```

NOTE
The source argument of an **rdoResultset** object can contain more than one SQL statement with some cursor models. Server-side cursors do not allow more than one statement, but depending on the ODBC driver, client-side cursors may allow the use of more than one statement. See your ODBC driver documentation for more details.

• The **OpenResultset** method is similar to DAO's **OpenRecordset**. It creates a new **rdoResultset** object using the syntax shown in the next code example. **source** is either an SQL statement or the **Name** property of an existing **rdoQuery** object. In both Listings 6.1 and 6.2, I illustrate this method using an existing **rdoQuery** object as the data source. **source** is the only mandatory argument. **type** is a **Long** with which you specify the type of result set to open, as listed in Table 6.5. **locktype** is used in multiuser environments for con-currency control. I discuss more advanced aspects of transaction management—including concurrency issues—in Chapter 11. Valid **locktype** constants are shown in Table 6.6. **option** is a **Long** in which you may specify the constants **rdAsyncEnable** or **rdExecDirect**. **rdAsyncEnable** causes the result set to be opened asynchronously. **rdExecDirect** causes the statement to be executed directly instead of first being prepared on the database.

```
Set resultset = rdoConnection.OpenResultset _
(source[, type, locktype, option])
```

Table 6.5 Valid rdoResultset type constants.

Constant	Description
rdOpenForwardOnly	(Default) Open a forward-only-type result set.
rdOpenKeySet	Open a keyset-type result set.
rdOpenDynamic	Open a dynamic-type result set.
rdOpenStatic	Open a static-type result set.

Table 6.6 Valid rdoResultset lockedit constants.

Constant	Description
rdConcurReadOnly	(Default) Use read-only.*
rdConcurLock	Use pessimistic concurrency.
rdConcurRowVer	Use optimistic concurrency based on row IDs.
rdConcurValues	Use optimistic concurrency based on row values.*
rdConcurBatch	Use optimistic batch updating.

**For some databases, these are the only valid lock types for static-type result sets.*

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

More On RDO Result Set Types

As with DAO record sets, you want to choose the RDO result set type that is most efficient while still meeting your requirements.

The forward-only-type result set allows only forward movement through the result set. It is most useful when you need to quickly access certain data with no dynamic forward and backward searching. You cannot update any rows. Membership in the result set is not fixed.

The keyset-type result set is updatable and movement forward and backward is unrestricted. You can use columns from more than one table. Membership in the result set is fixed.

A static-type result set has fixed membership of the rows and columns. Changes made by other users are not detected until the object is closed and reopened or until it is refreshed.

More On Concurrency And Locking

I discussed some of the issues involved in concurrency management as they relate to DAO in Chapter 5. For RDO, the issues are similar, as are your options.

When you open a result set as read-only, there are no concurrency issues, of course, because you will not be updating the data.

The safest locking is pessimistic. When you use it, the entire page of the row being updated is locked as soon as it is placed into edit mode. However, no other user will be able to access any other row on that database page. Further, if too many pages are locked, the database begins to run out of resources, in which case it might escalate the lock to a *table lock*, where the entire table is locked. You can also encounter a situation called a *deadly*

embrace where two processes lock each other out. For instance, if one user has page 1 locked, the database might not release that lock until it can access page 2. Another user might have page 2 locked, and the database cannot release it until it can access page 1. The two processes wait until one lock times out, and the database terminates one of the transactions. In a worst-case scenario, the DBA might have to manually end a transaction.

Note that, unlike with Jet, in a true ODBC environment, the size of the page is a function of both the ODBC driver and the back-end database. With some databases, the page size may be 2K (2,048 bytes), whereas with others, the size might be 4K. For still others, the page size might be an option set when the database was created. Some databases support true *row-level locking*, which means only the row being affected gets locked. Again, the ODBC driver might not fully support this feature. Check your database or ODBC documentation for more details.

With optimistic locking, the page is not locked until the row is about to be updated. Unfortunately, you cannot guarantee that another user hasn't modified the row since you retrieved it. If you invoke a certain type of lock and the ODBC driver does not support it, another level of locking is substituted. ODBC will generate an informational message that is appended to the **rdoErrors** collection.

With **rdConcurRowVer**, ODBC will attempt to use a row identifier, such as a timestamp column (if available), to determine whether the row has changed. If another row identifier is not available, only the row's primary key is used, which does not guarantee that changes made by other users will be detected. My recommendation is to add a timestamp column when practical and use this option where concurrency is an issue.

rdConcurValues causes a column-by-column comparison of all values on the row to be updated. If any values are different than when the row was retrieved, another user has modified the row. A trappable error is then generated. This feature is a safer option than **rdConcurRowVer** when a timestamp column is not available. However, because the database has to do extra work, there is a performance penalty with **rdConcurValues**.

You might want to use **rdConcurBatch** for performance reasons. All the updates are batched together and sent to the database if the database and ODBC driver support it. (I discuss batch updates in more detail later in this chapter.) Even if your database does not support processing of more than one update at a time, you might still want to use **rdConcurBatch**. To do so, set the **BatchSize** property to 1, which causes the updates to be sent one at a time. The advantage is that you can use the **UpdateCriteria** property to fine-tune your **Where** clause. I discuss these properties later in the chapter when I discuss the **rdoResultset** object. See Listing 6.3, later in this chapter, for some examples of batch updating.

If an update fails because another user has altered a record, a trappable error occurs. You can set the current row's **Bookmark** property to itself (**rdoResultset.Bookmark = rdoResultset.Bookmark**), which changes the row to reflect the changes made by the other user. When you do so, you lose

any changes that your program made (though you can save them to some variables first).

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The rdoResultsets Connection And rdoResultset Object

The **rdoResultsets** collection contains all **rdoResultset** objects within an **rdoEnvironment** object. You manipulate **rdoResultset** in much the same manner that you manipulate the **Recordset** object under DAO. It contains the **rdoColumns** collection. Because **rdoColumns** is the default property, you can specify a column name, omitting the reference to the collection and simply providing the **Name** property of the appropriate **rdoColumn** object.

When you create an **rdoResultset** object, it is automatically appended to the **rdoResultsets** collection. It is removed when you close it. You can append the result set to a different collection by changing the **ActiveConnection** property to another valid **rdoConnection** object. If you set this property to **Nothing**, the **rdoResultset** is removed from the collection but does not free its resources.

The **rdoResultset** object is a representation of all rows returned from a query. Even if the query returns no rows, the result set is still created (the **BOF** and **EOF** properties will be **False**).

You can create an **rdoResultset** object using the **OpenResultset** method of the **rdoConnection**, **rdoQuery**, or **rdoTable** objects. I discussed the use of this method as well as implications for certain arguments earlier in the chapter in the **rdoConnection** section.

As with several other RDO objects, you can manipulate the **rdo-Resultset** object in an event-driven manner if you create a reference to it using the **WithEvents** clause. Note that because a number of procedures will be referencing the object, you should declare it at the module level. Although you can declare it using the **Dim** keyword, I prefer the use of **Private**. At the module level, the two keywords are synonymous, but **Private** is more precise: **Private WithEvents rrs As rdoResultset**.

The events that the **rdoResultset** object supports follow:

- The **Associate** event is triggered after you use the **ActiveConnection** property to associate the result set with a new **rdoConnection** object. You might want to place some code in this event to initialize the result set, perhaps with a new query. Conversely, the **Disassociate** event is triggered when an **rdoResultset** object is set to

Nothing and is disassociated from an **rdoConnection** object. The **WillAssociate** event is triggered immediately before a result set is associated with a new connection. The **WillDisassociate** event occurs immediately before a result set is to be disassociated from a connection.

- The **ResultsChanged** event happens after any use of the **MoreResults** method, even if the method returns **False** (that is, there are no new results).
- The **ResultCurrencyChanged** event is essentially the same as DAO's **Reposition** event. It is triggered by any change in the current position with the **rdoResultset**.
- The **RowStatusChanged** event is triggered immediately after any edit, delete, or insert. You can ascertain the actual status of the row using the **Status** property.
- The **WillUpdateRows** event occurs immediately before any update to the database is to occur. You can place your own code in this event to override the default behavior of RDO. It has one argument, **ReturnCode**, as shown in the following sample code. You use it to tell RDO what you have done or simply to modify RDO's behavior. If you set the **ReturnCode** argument to **rdUpdateSuccessful**, RDO will not attempt to perform the updates itself and will set the status of all rows and columns to **rdRowUnModified** and **rdColUnModified**. If you set the **ReturnCode** to **rdUpdateWithCollisions**, RDO will also not attempt to update the database. However, it will not change the status of the rows or columns either; it is your program's responsibility to do so. **rdUpdateWithCollisions** gives you an opportunity to prevent an update without changing row and column statuses, and you normally use it when you are performing optimistic batch updates. (If doing so, it is important to modify row and column statuses in code for those updates that were successful.) **rdUpdateFailed** tells RDO that the update failed. RDO will not attempt to perform the update and will not modify any column or row statuses. However, RDO will generate a runtime error, which you would trap in your **Update** method. The default is **rdUpdateNotHandled**, which tells RDO to go ahead and handle the update using its own rules. This event provides a good place to perform data validation when you are not using the Remote Data control.

```
Private Sub object.WillUpdateRows(ReturnCode as Integer)
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The **rdoResultset** object has a rich set of properties you can use to monitor and manipulate the result set:

- The **AbsolutePosition** property returns the current record number within the result set. You can also use this property to set the current row to a particular row number, as shown in the next code segment. Note that unlike DAO's **AbsolutePosition** property, which is zero-based, RDO's **AbsolutePosition** property is one-based: You cannot use this property to move within a dynamic-type or forward-only-type result set. If there are no rows, **AbsolutePosition** is -1. If no record is current, then **AbsolutePosition** is undefined, and attempting to reference it will result in an error. You can avoid this problem by checking the **Bookmarkable** property. If it is **True**, the **AbsolutePosition** property should be valid.

```
' Move to the third row
rrs.AbsolutePosition = 3
```

- The **ActiveConnection** property returns a reference to the **rdoCon-nection** object to which this **rdoResultset** belongs. Setting this property to **Nothing** disassociates the result set from the connection. See my discussion of the **Associate** event earlier in this section for more information.
- The **BatchCollisionCount** property returns the number of records that did not complete the last batch update. A collision in a batch update occurs when you attempt to update a record that has been altered since you retrieved it. This can only happen in a multiuser environment or when your application opens multiple record sets on the same data. Use this property along with the **BatchCollisionRows** property, which is an array of bookmarks of those records on which collisions occurred. This combination of properties allows your application to move to each row and correct the problem. Once the problem is rectified, you can invoke the **Update** method again. If there are any more collisions, these two properties (**BatchCollisionCount** and **BatchCollisionRows**) are repopulated. Those rows that are successfully

updated have their row and column statuses changed so RDO will not attempt to update them again. Listing 6.3 shows batch updating and the resolution of collisions using the **BatchCollisionCount**, **BatchCollisions**, and related properties and methods. Figure 6.5 shows the application. The application simulates a second user by creating a new query object in another connection and environment. It sends a single action query to the database using the **Execute** method to alter a row, thus forcing a collision to occur. The application refreshes the row in question to display the values as changed by the other user, which effectively loses your changes. Another method to see the new values is to access the **rdoColumn** object's **BatchConflictValue** property.

Listing 6.3 The RDO batch update demonstration program.

```
Option Explicit
Dim WithEvents rEng As rdoEngine
Dim WithEvents rEnv1 As rdoEnvironment
Dim WithEvents rEnv2 As rdoEnvironment
Dim WithEvents rCon1 As rdoConnection
Dim WithEvents rCon2 As rdoConnection
Dim rq1 As rdoQuery
Dim rq2 As rdoQuery
Dim WithEvents rrsEmp1 As rdoResultset

Private Sub cmdUpdate_Click()
Dim sConn As String
Static cRaise As Currency
Dim sMsg As String

If cRaise = 0 Then
    cRaise = 1
Else
    ' So the user can change the data back
    cRaise = cRaise * -1
End If

' In order to do batch updating, set the default cursor
lbStatus = "Creating RDO Engine …"
rdoEngine.rdoDefaultCursorDriver = rdUseClientBatch

' Use default environment for first environment
lbStatus = "Creating 2 Environment Objects …"
Set rEnv1 = rdoEngine.rdoEnvironments(0)
' Create second environment
Set rEnv2 = rdoEngine.rdoCreateEnvironment("Env2", _
  "Coriolis", "Coriolis")
' Now, connect to the database
' Be sure to set your own path!
lbStatus = "Connecting to the database …"
Set rCon1 = rEnv1.OpenConnection(dsName:="", _
 Prompt:=rdDriverNoPrompt, _
```

```
 Connect:="UID=Coriolis;PWD=Coriolis;" & _
 "Driver=Sybase SQL Anywhere 5.0;" & _
 "DBF=C:\ \Examples\Coriolis VB Example.db;" & _
 "DBN=Coriolis;DSN=''", Options:=rdAsyncEnable)

' Wait for connection
Do While rCon1.StillConnecting
  DoEvents
Loop

' Make second connection
Set rCon2 = rEnv2.OpenConnection(dsName:="", _
 Prompt:=rdDriverNoPrompt, _
 Connect:="UID=Coriolis;PWD=Coriolis;" & _
 "Driver=Sybase SQL Anywhere 5.0;" & _
 "DBF=C:\ \Examples\Coriolis VB Example.db;" & _
 "DBN=Coriolis;DSN=''", Options:=rdAsyncEnable)

' Wait for connection
Do While rCon2.StillConnecting
  DoEvents
Loop

' Create the query object
lbStatus = "Opening Result Set …"
Set rq1 = rCon1.CreateQuery("Emp", _
  "Select * From Employee")

' Set properties of the connection
With rCon1
  .QueryTimeout = 3
  ' Create resultset objects
  Set rrsEmp1 = .OpenResultset("Emp", rdOpenDynamic, _
    rdConcurBatch, rdAsyncEnable)
  ' Wait for result set
  Do While rrsEmp1.StillExecuting
    DoEvents
  Loop
End With

With rrsEmp1
  lbStatus = "Editing Records …"
  ' Move through all records
  Do While Not .EOF
    ProgressBar1.Value = .PercentPosition
    Label5.Caption = Str$(.PercentPosition)
    Text1 = !emp_no & " " & LTrim(!emp_fname) & _
      !emp_lname
    Text2 = !emp_salary
    DoEvents
```

```
    ' Put the record in edit mode
    .Edit
    ' Give them a raise!
    !emp_salary = !emp_salary + cRaise
    Text3 = !emp_salary
    ' This is a local update — the database
    ' is not getting updated yet!
    .Update
    ' Scroll to next record
    .MoveNext
  Loop

  ' Simulate a second user changing a record
  lbStatus = "Simulating a second user …"
  Set rq2 = rCon2.CreateQuery("emp2", _
    "update employee set emp_salary = emp_salary + 3" & _
    " where emp_no = 101")

  rq2.Execute rdAsyncEnable
  ' Wait for it to complete
  Do While rq2.StillExecuting
    DoEvents
  Loop
  rCon2.CommitTrans
  rq2.Close
  rCon2.Close

  ' Perform the batch update to the database
  lbStatus = "Batch updating …"
  rrsEmp1.BatchSize = 1
  rrsEmp1.BatchUpdate False, False
  ' Check to see if any collisions
  If .BatchCollisionCount > 0 Then
    ' Move to first collision to demonstrate
    ' use of the batchcollisions array
    .Bookmark = .BatchCollisionRows(0)
    Text1 = !emp_no & " " & LTrim(!emp_fname) & _
      " " & !emp_lname
    Text2 = ""
    Text3 = !emp_salary
    If MsgBox("Collision Detected. See changes made?", _
      vbYesNo + vbQuestion) = vbYes Then
      ' Also see the BatchConflictValue of the rdoColumn
      .Bookmark = .Bookmark
      Text4 = !emp_salary
    Else
      sMsg = Str$(.BatchCollisionCount) & " records " & _
        "have been altered by another user." & vbCrLf & _
        "Force updates anyway?"
      ' Give user opportunity to force updates
```

```
      If MsgBox(sMsg, vbYesNo & vbQuestion) = vbYes Then _
        ' True forces the update
        .BatchUpdate False, True
      End If
    End If
  End If
  ' Close the rdoResultset
  .Close
End With

' Close rdoConnection and rdoEnvironment
rCon1.Close
rEnv1.Close
lbStatus = "RDO objects closed"
End Sub

Private Sub Command1_Click()

End

End Sub
```

Figure 6.5 The RDO batch update sample program detecting a collision with another user.

- The **BatchSize** property controls how many statements will be sent to the server at a time during a batch update. The default is 15. Some databases or ODBC drivers do not support sending more than one statement at a time. I use this property in Listing 6.3.
- The **BOF** and **EOF** properties are similar to the DAO counterparts. If the current record is before the first record, **BOF** will return **True**. Likewise, if the current record is after the last record, **EOF** returns **True**. If both values are **True**, then there are no rows (and the **RowCount** property is equal to 0). If either value is **True**, there is no current record and attempting to access any data results in a trappable error.
- The **Bookmark** property is a reference to the current row. All result sets except forward-only-type support the **Bookmark** property. You can save a reference to the current row by saving the value of this property to a variable of type **Variant**. The **Bookmarkable** property tells you whether you can reference the **Bookmark**. You can also change the current record by setting the **Bookmark** property to a previously saved value. This method is preferred over using the **AbsolutePosition** property. The following code segment saves a **Bookmark**, moves to the first row, and then moves back to the saved position. Listing 6.3 also illustrates some uses of the property.

```
Dim vBookMark As Variant
' Manipulate the result set
With rrs
```

```
    vBookmark = .Bookmark
    .MoveFirst
    .Bookmark = vBookMark
  End With
```

NOTE
If you set the **Bookmark** property to the current **Bookmark**, it has the effect of refreshing the current record. This feature is most useful when another user has changed a row that is currently being edited. The row will be changed to reflect that user's changes, but any changes you have made will be lost.

- The **EditMode** property returns a **Long** indicating the edit mode state of the current record. **rdEditNone** indicates the current row is not being edited. **rdEditInProgress** indicates that the row is being edited and that the current row is in the copy buffer. **rdEditAdd** indicates that a new row has been added. The row in the copy buffer has not been saved to the database.
- The **hStmt** property returns a **Long** containing the ODBC statement handle.
- The **LastModified** property returns the **Bookmark** of the most recently added or changed row. You cannot get a **Bookmark** reference to a deleted row. If no rows have been added or modified, this property contains zero. The following example illustrates the movement to the most recently modified row:

```
rrs.Bookmark = rrs.LastModified
```

- The **LockType** property is used to set or return the concurrency locking type for the result set. See my discussion of the **OpenResultset** method earlier in this chapter where I discuss concurrency settings.
- The **LockEdits** property returns a Boolean indicating whether the current locking is pessimistic (**True**) or optimistic (**False**). Use the **LockType** property to manipulate the locking.
- The **Name** property is a string containing the first 255 characters of the SQL statement. If the row source is a query object, the **Name** property is equal to the **Name** of the **rdoQuery** object.
- The **PercentPosition** property is a **Single** with which you gauge approximately where you are in the result set. For instance, if there are 200 rows and you are displaying the 50th, the property will be equal to 25 (25 percent). You can also use this property to set the current row, although it is not precise. The following example will set the current row to be approximately 75 percent of the way through the result set. For more precise movement, use the **AbsolutePosition** or **Bookmark** properties. On the other hand, if you are updating a long series of records, **PercentPosition** may be useful to maintain a progress bar for the user, as is also shown in the code example.

```
' Move 75% of the way through the result set
rrs.PercentPosition = 75
' pbStatus is a progress bar
With rrs
  Do Until .EOF
    .Edit
    !Emp_Salary = !Emp_Salary * 1.08
```

```
        pbStatus.Value = .PercentPosition
        DoEvents
        .MoveNext
      Loop
    End With
```

NOTE
In Listing 6.3, I attempted to use the **PercentPosition** property to keep a status bar up-to-date as I moved through 35 records. For each record, the value never changed from 50. For larger result sets, it might be more accurate.

- The **Restartable** property returns a Boolean that is **True** if the database supports the re-execution of the same query (via the **Requery** method).
- The **RowCount** property returns a **Long** telling you how many rows are in the current result set. **RowCount** will return -1 to indicate an error. Unlike in DAO, the value of **RowCount** is guaranteed to be accurate. However, when you reference it, the result set invokes a **MoveLast** method if necessary to determine the number of rows. Avoid referencing this property if you do not absolutely need to know how many rows there are. If you do, consider invoking the **MoveLast** method yourself so that you can do it asynchronously. (Note that some ODBC drivers are not capable of returning a row count. An invalid result set might return a **RowCount** of 0 instead of -1.)
- The **Status** property is a **Long** that returns a constant indicating the status of the current row or column (when accessing the **rdoColumn** object). These constants are itemized in Table 6.7. You can also set this property to override the status given by RDO. You might want to do this when managing your own updates, as I discussed in “More On Concurrency And Locking” earlier in this chapter.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Table 6.7 Valid row and column Status constants.

Constant	Description
rdRowUnModified	No updates or adds of this row have been done or are pending.
rdRowModified	The current row or column has been modified.
rdRowNew	The current row or column has been inserted.
rdRowDeleted	The current row or column has been deleted.
rdRowDBDeleted	The row has been deleted in the result set and in the database.

- The **StillExecuting** property returns **True** while an asynchronous operation is executing. You cannot reference any other property of the result set until the asynchronous operation is complete. You also cannot access any method except **Cancel**. This chapter contains several examples of the use of this property, including Listing 6.3. Note, however, that you might want to take an event-driven approach, as I discussed earlier in this section.
- The **Transactions** property returns a Boolean that is **True** if the database and driver support transaction processing (that is, results can be committed or rolled back). If transactions are not supported, the **BeginTrans**, **CommitTrans**, and **RollbackTrans** methods have no effect.
- The **Type** property returns the type of result set, as listed in Table 6.5.
- The **Updatable** property is a Boolean that returns **True** if the current result set can be updated. Just because the result set is updatable does not mean that all columns are updatable. Some columns may not be updatable. Check individual columns within the **rdoColumns** collection. Any number of reasons dictate why a result set is not updatable. If it is opened read-only or is a forward-only type, the rows

cannot be updated. If the columns returned are not sufficient to form a **WHERE** clause, the result set probably will not be updatable. For instance, you generally require a primary key to perform an update. If your result set does not include a primary key, the result set is probably not updatable.

- The **UpdateCriteria** property is used with optimistic batch updates to set how the **WHERE** clause will be constructed in the SQL **UPDATE** statement. Possible values are listed in Table 6.8. In general, you want to use the list of restrictive update criteria that will guarantee the integrity of your data. The more restrictive the criteria (such as **rdCriteriaAllCols**), the more resources are required. Using **rdCriteriaKey** probably does not make sense in a multiuser environment because it does not help at all to ensure that another user has modified a row while you are editing it. **rdCriteriaAllCols** is an absolute guarantee of detecting any changes because you are comparing all columns as you read them to their present values on the database. This comparison is expensive, however. If you have a timestamp column, **rdCriteriaTimeStamp** is a good compromise because any changes by another user will alter the timestamp. If you do not care about the columns you are not updating, then **rdCriteriaUpdCols** is a reasonable choice.

NOTE
My copy of Microsoft's Help file switched the explanations of the **rdCriteriaAllCols** and **rdCriteriaUpdCols**.

Assume you are updating the balance on a bank customer's record. You certainly do not want to update the balance if, in the meantime, another user has updated the balance. Perhaps you don't care that another user has updated information such as an address. In such a case, you would use **rdCriteriaUpdCols**. Assuming the customer's balance is 300 and you want to update it to 500, your **UPDATE** statement sent to the server will look something like the following:

```
UPDATE CUSTOMER
SET BALANCE = 500
WHERE CUST_NO = 10284 AND BALANCE = 300 ;
```

Table 6.8 Valid concurrency UpdateCriteria constants.

Constant	Description
rdCriteriaKey	Use the primary key for the **WHERE** clause.
rdCriteriaAllCols	Use all columns in the **WHERE** clause.
rdCriteriaUpdCols	Use the primary key and those columns being updated for the **WHERE** clause.
rdCriteriaTimeStamp	Use the timestamp column in the **WHERE** clause.

- The **UpdateOperation** property specifies how updates are to occur. The default is **rdOperationUpdate**, which species that updates will be

done with the SQL **UPDATE** statement. **rdOperationDelIns** causes two statements to be generated—a **DELETE** of the original row and an **INSERT** of the new row. Generally, you want to use **UPDATE**, but if for some reason you need to alter the primary key of a record, most databases require that you do so by deleting the original row and then inserting a new row.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The **rdoResultset** has a number of methods that you use to manipulate data:

- The **AddNew** method creates a new row in an updatable result set. If you attempt to add a row to a result set that is not updatable, no error occurs until you attempt to save the row to the database. Note that when you create a new row, the current row remains current. You need to move to the new row using the **LastModified** property. Also note that if you move off the new row without updating the database (with the **Update** or **BatchUpdate** methods), any changes made to the row are lost. You can cancel a pending **AddNew** with the **CancelUpdate** method. The following example creates a new row for the result set **rrs** and then moves to it so it can be edited:

```
With rrs
  .AddNew
  .Bookmark = .LastModified
End With
```

- The **Cancel** method cancels a currently executing asynchronous operation. While an asynchronous operation is running (see **StillExecuting** earlier in this section), the only method that you can use is **Cancel**.
- The **CancelUpdate** method undoes any changes pending from the copy buffer after an **AddNew** or **Edit** operation. You can use the **EditMode** property to determine if there are any pending changes that can be canceled. When there is nothing to cancel, the **CancelUpdate** method is ignored. The **CancelBatch** method cancels all pending changes in a result set opened using the **ClientBatch** cursor library.
- The **Close** method closes the result set and removes it from the **rdoResultsets** collection.
- The **Delete** method deletes the current row. Once it is deleted, you must move to a valid row in the result set. Once a row is deleted, it

cannot be referenced. The only way to undelete the row is in the context of a single transaction using the **RollbackTrans** method. The following example deletes the current row. It then checks the status of the **EOF** property to make sure that the row wasn't the last one in the result set. If it was the last row, a **MoveLast** is performed; otherwise, a **MoveNext** is performed:

```
With rrs
  .Delete
  If .EOF Then
    .MoveLast
  Else
    .MoveNext
  End If
End With
```

- The **Edit** method copies the current row to the copy buffer, enabling it to be edited by the user or by the program. You must use the **Update** or **BatchUpdate** methods to save the changes to the database before moving off the row or the changes will be lost. Note that if the result set uses pessimistic locking, the database page where the rows reside on the database is locked as soon as you invoke the **Edit** method. The lock is not released until you save the change or cancel the edit. Moving to another record without first saving the changes causes the changes to be lost.

- The **GetClipString** method is similar to **GetRows** (which I discuss next) except that it returns the data as a delimited string. The syntax is shown in the next code example. The **Rows** argument specifies how many rows to return. The **Col_delimiter** is an optional **Variant** that specifies what character to use to delimit columns. **vbTab** is the default. **Row_delimiter** is also an optional **Variant** used to delimit rows in the string. The default is **vbCr**. The **Null** argument is an optional **Variant** with a default of an empty string that is used to specify what value to return when a **Null** column is encountered. This function is most useful for populating grid controls that support the **Clip** method. The Ad Hoc Report Writer sample application that I discuss at the end of this chapter uses this method to populate a grid control.

```
ResulsetString = object.GetClipString (Rows, _
  Col_delimiter, Row_delimiter, Null)
```

- The **GetRows** method moves one or more rows from the result set into a two-dimensional array, where the first dimension specifies the column within the result set and the second dimension denotes the row. You must first declare a variable of type **Variant**, as shown in the following example. In invoking the method, specify the number of rows to copy, as also shown in the example. If there are not enough rows to satisfy the request, RDO copies only those rows that are available. Therefore, to determine how many rows were returned, you need to use the **UBound** function on the array.

```
Dim vArray As Variant
' Get 100 rows
vArray = rrs.GetRows (100)
' Display how many were actually copies
MsgBox UBound (vArray, 2) + 1
```

- The **MoreResults** method is used when the data source of the result set contains more than one SQL **SELECT** statement. This method retrieves the rows from the next SQL statement.
- The **Move** method moves a specified number of rows forward or backward in the result set. The syntax is shown in the following code segment. The **rows** argument specifies how many rows to move. If it is negative, RDO will scroll backwards. The **from** argument, which is optional, evaluates to the **Bookmark** of a row from which you want to scroll:

```
rrs.Move rows[, from]
```

- There are four other move methods: **MoveNext** scrolls to the next row; **MovePrevious** scrolls to the prior row; and **MoveFirst** and **MoveLast** move to the first and last rows.
- The **Requery** method is used to re-execute the current SQL **SELECT**, which has the effect of refreshing the result set to show any changes made by other users. When you use this method, all saved **Bookmarks** are invalid. The **rdoResultset** scrolls to the first row, as when you open a new result set. The method has an optional argument, **rdAsyncEnable**, to cause the method to execute asynchronously.
- The **Update** method saves the changes made to the current record to the database unless you use the **ClientBatch** cursor library, in which case the changes are queued until you invoke the **BatchUpdate** method. Listing 6.3 contains examples of using both methods.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Referencing rdoResultset Columns

Every **rdoResultset** has a collection of **rdoColumns**. When manipulating the result set in code, you reference the individual **rdoColumn** objects:

```
txtFields(0).Text = rrsEmp.rdoColumns("Emp_ID").Value
```

The code snippet references the **rdoColumn** object by name, which is actually the column name from the SQL **SELECT**. (You can also reference the object by its ordinal position within the collection.) **rdoColumns** is the default property of the **rdoResultset** object, so the explicit reference is not necessary. Likewise, **Value** is the default property of the **rdoColumn** object and does not have to be explicitly referenced. The following line of code is simpler to read and somewhat faster to resolve and execute:

```
txtFields(0).Text = rrsEmp!Emp_ID
```

Using the exclamation (**!**) operator in this manner is called *reference by implication*. The reference to the column's parent object, **rdoColumns**, is implied.

The rdoColumns Collection And rdoColumn Object

The **rdoColumn** object represents a column within a record set, query, or table. The **rdoColumn** objects within an **rdoColumns** collection are created automatically when you create an **rdoResultset** or **rdoTable**. Although you can use an **rdoTable** object's **rdoColumns** collection to map the underlying database table, you cannot use it (or any RDO methods) to manipulate the table's structures. You need to use **DDL** in an action query to modify a table structure.

Like **rdoResultset**, **rdoEngine**, **rdoEnvironment**, and **rdoConnection**, the **rdoColumn** object supports some event-driven processing. Specifically, it supports two events that you can use before or after a value has changed. To take advantage of this behavior, you need to declare a reference to the object using the **WithEvents** qualifier: **Private WithEvents rcolEmp As rdoColumn**.

- The **WillChangeData** event is triggered immediately before a column value is going to be changed. Two arguments are passed to the event, as shown in the following code example. The **NewValue** argument indicates the new value of the column that will be

modified. You can set the **Cancel** argument to **True**, which has the effect of preventing the change from happening. By default, this value is **False**. If you prevent a change, RDO triggers a trappable error.

```
Private Sub rcolEmp_WillChangeData (NewValue As Variant, _
  Cancel As Boolean)
```

- The **DataChanged** event occurs immediately after the data has been updated to the database.

The **rdoColumn** object has some 16 properties that you use to manipulate data and the object itself. As with the DAO **Field** object, some properties are more pertinent when dealing with tables, whereas others are more pertinent when dealing with queries and result sets. The **Name** property is the name of the underlying database column.

A number of properties tell you about the nature of the column. Listing 6.4 illustrates many of them in use. To try this yourself, create a form with an array of textbox controls named **txtFields**. The first textbox will display and update the value in the first column, the second textbox corresponds to the second column, and so on. As written, this routine is very generic and can be used to perform basic validation on almost any result set. The listing loops through all of the textbox controls and compares their contents to the requirements of the corresponding column of the result set. I have highlighted the lines of interest.

Listing 6.4 A generic routine to edit the contents of textboxes against the data requirements of table columns.

```
' rrs is a result set
' txtFields is an array of textbox controls
Dim iCtr As Integer
Dim vTest As Variant
With rrs
  For iCtr = 0 to 8
    If .rdoColumns(iCtr).KeyColumn = True Then
      MsgBox "Cannot Update Primary Key!"
      txtFields(iCtr).SetFocus
      Exit Sub
    ElseIf .rdoColumns(iCtr).Updatable = False Then
      MsgBox "Cannot Update This Field!"
      txtFields(iCtr).SetFocus
      Exit Sub
    Else
      ' Verify the type of data required
      Select Case .rdoColumns(iCtr).Type
        Case rdTypeChar, rdTypeVarChar, rdTypeLongVarChar
          ' Column is character data
          If .rdoColumns(iCtr).AllowZeroLength Then
            ' Empty String okay
            .rdoColumns(iCtr).Value = txtFields(iCtr)
          Else
            ' Use null if necessary
            If len(txtFields(iCtr)) = 0 Then
              .rdoColumns(iCtr).Value = Null
            Else
              .rdoColumns(iCtr).Value = txtFields(iCtr)
            End If
```

```
            End If
          Case Else
            ' Data is non-character (no dates on this table)
            vTest = txtFields(iCtr).Text
            If IsNumeric(vTest) Then
              ' Contains a number
              .rdoColumns(iCtr).Value = txtFields(iCtr)
            ElseIf Len(txtFields(iCtr)) > 0 Then
              ' Contains non-numeric data
              MsgBox "Invalid Value: " & txtFields(iCtr)
              txtFields(iCtr).SetFocus
              Exit Sub
            Else
              ' Does not contain a number - Can it be null?
              If .rdoColumns(iCtr).Required Then
                 ' Null not allowed
                 MsgBox "Value Required!"
                 txtFields(iCtr).SetFocus
                 Exit Sub
              Else
                 .rdoColumns(iCtr).Value = Null
              End If
            End If
        End Select
      End If
    Next
  End With
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The code first checks the **KeyColumn** property, which returns **True** if the column is part of the primary key. Most databases do not allow you to update the primary key, and I have disallowed any maintenance to it. After that, I check the **Updatable** column. This value is also a Boolean that will be **True** if the column can be updated.

Next, I evaluate the **Type** property in a **Select Case** construct. This property is a **Long** containing one of the constants listed in Table 6.9. These are the valid data types for a column. In the listing, I look for columns that are a character data type and handle those differently than numeric data types. For purposes of brevity, I did not evaluate whether the column's data type was, for instance, date or time. You can add those tests easily enough. If the data type is character, I next check the **AllowZeroLength** property. This is a Boolean that is **True** when character data (**Char** and **VarChar**) can be a zero-length string. If the value is **False**, you must specify **Null** when there is no data. In the listing, if **Null** is not allowed and the text box is empty, I generate an error message.

Table 6.9 Valid rdoColumn Type constants.

Constant	Description
rdTypeCHAR	Fixed-length character (see **Size** property)
rdTypeNUMERIC	Numeric data with precision and scale such as "Numeric (11, 2)"
rdTypeDECIMAL	Numeric data with precision and scale such as "Numeric (11, 2)"
rdTypeINTEGER	Integer with range of approximately +/- 2.1 billion
rdTypeSMALLINT	Integer with range of -32,768 to +32,767
rdTypeFLOAT	Signed 8-byte floating-point number

rdTypeREAL	Signed 4-byte floating-point number
rdTypeDOUBLE	Signed 8-byte floating-point number
rdTypeDATE	Date
rdTypeTIME	Time
rdTypeTIMESTAMP	Timestamp (date and time)
rdTypeVARCHAR	Variable-length character with maximum length of 255
rdTypeLONGVARCHAR -	Variable-length character; maximum length set by database or driver
rdTypeBINARY	Fixed-length binary data with maximum length of 255
rdTypeVARBINARY	Variable-length binary data with maximum length of 255
rdTypeLONGVARBINARY	Variable-length binary data; maximum length set by database or driver
rdTypeBIGINT	Signed, 12-byte integer
rdTypeTINYINT	Signed, 1-byte integer with range of -256 to +255
rdTypeBIT	Single binary digit

If the column's data type is numeric, I then make sure a number is entered into the textbox. If there is non-numeric data, a message is generated. If the text box is empty, I then check the **Required** property, which returns **True** if the column cannot be **Null**.

The **Value** property, of course, returns or sets the value of the column. Although I did not use it in the listing, you may find the **Size** property useful in your edits. It returns the maximum size (in bytes) of the underlying column. This would be a valid edit of character data:

```
If Len(txtFields(iCtr)) > .rdoColumns(iCtr).Size Then
```

The **Attributes** property combines several other properties and may contain one or more constants. **rdFixedColumn** indicates a fixed-length column, whereas **rdVariableColumn** indicates a variable-length column. **rdAutoIncrColumn** indicates that the column is an auto-incrementing column, usually associated with the primary key. **rdUpdatableColumn** indicates that the column can be changed. **rdTimeStampColumn** indicates a timestamp column. To test for any single attribute, you should use the **And** operator: **If rrs!emp_no. Attributes And rdUpdatableColumn Then…**.

The **BatchConflictValue** property is useful when a batch collision occurs (see Listing 6.3 and the discussion of batch updates earlier in this chapter). The **OriginalValue** is similar except that it provides the value of the column before it was altered by the user or application.

The **Status** property indicates whether the column has been modified as a constant, as listed in Table 6.7 earlier in this chapter.

The **SourceColumn** and **SourceTable** properties return string variables containing the names of the column and table from which the data in the column is derived.

The **ChunkRequired** property is a Boolean that returns **True** when you must use the **GetChunk** and **AppendChunk** methods to access binary and long character data from a column. The **BindThreshold** property returns the largest number of bytes of data that can automatically be bound to a column without using the **GetChunk** method. You can also set this property to the largest column size that you want to be automatically bound.

The **rdoColumn** object has only three methods.

The **GetChunk** and **AppendChunk** methods are used with binary data or very long character data (**rdTypeLONGVARCHAR**). **GetChunk** is used to iteratively retrieve a "chunk" of data at a time, perhaps to read a Microsoft Word document that has been stored in a table. Likewise, the **AppendChunk** method appends a chunk at a time to a column. The syntax for each is shown in the next code segment. **bytes** is the number of bytes to read with each iteration. If there are not enough bytes left in the column to fill the buffer, only the remainder of the column is read. The **datasource** argument is the variable from which you are reading to append to the column. You use the **ColumnSize** method to return the actual size of the binary data stored in the column.

```
Dim vChunk As Variant
vChunk = recordset!column.GetChunk (bytes)
recordset!column.AppendChunk datasource
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The rdoParameters Collection And rdoParameter Object

The **rdoParameters** collection contains all parameters of a query. You can use the **rdoParameter** object as a placeholder in stored procedures or in queries. The overall operation is similar to using DAO's **Parameter** objects. Parameters are implicitly created when you place question marks in your query or call to a stored procedure. The parameter can be input, output, or both. An input parameter supplies a value to a query (as in a **WHERE** clause) or stored procedure. An output parameter is used where a stored procedure will return results. RDO creates an **rdoParameter** object for each question mark (**?**) it sees in an **rdoQuery** object's **SQL** property.

With a stored procedure, you must use the **CALL** keyword to execute the procedure on the database. I created the following simple stored procedure to compute the power of a number. Although the following syntax is fairly generic, your database's syntax may differ slightly:

```
CREATE PROCEDURE raise_no
(IN r INTEGER, IN p INTEGER, OUT rp INTEGER)
BEGIN
  SELECT power (r, p) INTO rp  ;
END
```

In this example, the stored procedure expects two input arguments, **r** and **p**. It raises the number **r** to the power of **p** and passes the result to the output argument **rp**. To call this from Visual Basic, you need to create an SQL statement that looks like the following:

```
CALL raise_no (?, ?, ?)
```

In Listing 6.5, I provide a listing for the RDO parameters application that you see running in Figure 6.6. The application allows you to test both parameterized stored procedures and queries. I have highlighted the lines of code that call the **raise_no** stored procedure. Although not terribly complex, the application illustrates both the methodology of calling

stored procedures from Visual Basic as well as how parameters are created and used. (I discuss stored procedures in greater depth in later chapters.)

Listing 6.5 The RDO parameters demonstration program.

```
Option Explicit
Dim WithEvents reng As rdoEngine
Dim WithEvents renv As rdoEnvironment
Dim WithEvents rcon As rdoConnection
Dim WithEvents rrs As rdoResultset
Dim rq As rdoQuery

Private Sub Command1_Click(Index As Integer)

Dim iRoot As Integer
Dim iPower As Integer
Dim iCtr As Integer
Dim sSQL As String

Select Case Index
  Case 0 ' Stored procedure
    iRoot = Val(txtRoot)
    iPower = Val(txtPower)
    sSQL = "Call raise_no (?, ?, ?)"
    Set rq = Nothing
    Set rq = rcon.CreateQuery("SP", sSQL)
    rq.rdoParameters(2).Direction = rdParamOutput
    rq.rdoParameters(1).Direction = rdParamInput
    rq.rdoParameters(0).Direction = rdParamInput
    rq.rdoParameters(0).Type = rdTypeINTEGER
    rq.rdoParameters(1).Type = rdTypeINTEGER
    rq.rdoParameters(2).Type = rdTypeINTEGER
    rq(0) = iRoot
    rq(1) = iPower
    rcon.Execute ("SP")
    While rcon.StillExecuting
      DoEvents
    Wend
    txtResult = rq.rdoParameters(2).Value
  Case 1 ' Parameter query
    Set rq = Nothing
    Set rrs = Nothing
    sSQL = "Select min(emp_salary), max(emp_salary), " & _
      "avg(emp_salary), sum (emp_salary) from " & _
      "employee where emp_gender = ? and emp_dept_no = ?" & _
      "group by emp_gender, emp_dept_no "
    Set rq = rcon.CreateQuery("PARM", sSQL)
    rq(0) = txtParms(0)
    rq(1) = txtParms(1)
    Set rrs = rcon.OpenResultset("PARM")
    While rcon.StillExecuting
```

```
      DoEvents
    Wend
    For iCtr = 0 To 3
      txtSalary(iCtr) = rrs.rdoColumns(iCtr).Value
    Next
  Case 2 ' Close
    End
End Select

End Sub

Private Sub Form_Load()

Dim sCaption As String
Dim sSlash As String * 1

' Connect to database
Show
sCaption = Caption
Caption = Caption & " Connecting …"
Screen.MousePointer = vbHourglass

' Create environment
Set renv = rdoEngine.rdoCreateEnvironment _
  ("VB", "coriolis", "coriolis")
  ' Set properties
  With renv
    .CursorDriver = rdUseOdbc
    .LoginTimeout = 10
    ' Create connection
    Set rcon = .OpenConnection(dsName:="", _
    Prompt:=rdDriverNoPrompt, _
    Connect:="DSN=Coriolis VB Example;UID=Coriolis;" & _
    "PWD=Coriolis;", Options:=rdAsyncEnable)
  End With

' Wait for connection to complete
While rcon.StillConnecting
  DoEvents
  If sSlash = "/" Then
    sSlash = "\"
  Else
    sSlash = "/"
  End If
  Caption = Caption & sSlash
Wend
Caption = sCaption
Screen.MousePointer = vbDefault

End Sub
```

Figure 6.6 The RDO parameters demonstration program.

When the **rdoQuery** object is created using the preceding syntax for the **SQL** property, RDO creates three parameters—one for each question mark placeholder. The **Name** properties of each parameter are simply **Parameter1**, **Parameter2**, and so on. I use the **Direction** property to explicitly tell RDO whether the parameter will be used for input or output. Normally, RDO can ascertain this information by itself. Other possible values are **rdParamInputOutput** and **rdParamReturnValue**. I also specify the data type of each parameter using the **Type** property. Possible values are listed in Table 6.9 earlier in this chapter. Again, this is normally not necessary. In the next two lines of code, I set the **Value** property of the input parameters. Because **rdoParameters** is the default property of the **rdoQuery** object and **Value** is the default property of the **rdoParameter** object, I simplified the syntax as shown, omitting the property names. For illustration purposes, I explicitly reference the properties when I set the return result from the procedure.

Queries work much like their DAO parameter counterparts. Place a question mark in the query wherever you desire a parameter to be filled in at runtime. The form allows the user to supply a gender and department number and use those values as parameter values when creating the result set. The application then determines various summary salary information and displays it.

The rdoTables Collection And rdoTable Object

Microsoft discourages the use of the **rdoTables** collection and **rdoTable** object. They are maintained for backward compatibility, and all the information and methods that they provide are supported by other objects. **rdoTable** provides a definition of a table or view in a query or result set. To reference it, you must first use the **Refresh** method of the collection.

The **Name** property is equivalent to the **SourceTable** property in an **rdoColumn** object. The **Type** and **Updatable** properties are similarly available from **rdoColumn**. The **RowCount** property is available from the result set.

The only method the object supports is the **OpenResultset** method, which is provided by other objects, such as **rdoConnection**.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The rdoPreparedStatements Collection And rdoPreparedStatement Object

Like **rdoTables**, **rdoPreparedStatements** and **rdoPreparedStatement** are provided for backward compatibility only. All their properties and methods are provided by **rdoQuery**. Further, the requirements for ODBC functionality to use **rdoPreparedStatement** are more stringent than they are for **rdoQuery**. If using **rdoPreparedStatement**, you should consider converting. The effort is little more involved than a simple search and replace in your code.

The Remote Data Control

RDO provides a much cleaner, simpler method for developing database applications than does DAO while retaining a high degree of both familiarity and compatibility. The use of the Remote Data control is analogous to the DAO Data control. Its use provides some loss in flexibility but provides an even simpler development scenario.

To use the Remote Data control, you must add it as a component from the Project menu. Once you have done so, it appears on the toolbox and can be used in a manner similar to that for the Data control.

The control generates only four events. The **Error** event works much like the Data control's **Error** summary event except it provides more information, as shown in the following syntax. The **Error** event is triggered when a database error is encountered while no VB code is running. The **Number** and **Description** arguments are the error number and error descriptions from the **Err** object. The **Scode** argument is the ODBC return code. **Source** is the source of the error. **HelpFile** and **HelpContext** refer to the help file and context ID that provide more information about the error. **CancelDisplay** allows you to display the error (**rdDataErrDisplay**) or proceed without displaying the error (**rdDataErrContinue**). The default value is **rdDataErrDisplay**.

```
Private Sub MSRDC1_Error(Number As Long, Description As _
    String, Scode As Long, Source As String, HelpFile As _
    String, HelpContext As Long, CancelDisplay As Boolean)
```

The **QueryCompleted** event occurs after a query has completed executing, whether asynchronously or synchronously. It is most useful, of course, during asynchronous operations. You use this event as you would use the **rdoConnection** except that it does not provide information about the query itself.

The **Reposition** event is triggered *after* a new row becomes the current row, regardless of what caused the row to display (such as a user clicking one of the move buttons). The **RowCurrencyChange** event of the **rdoResultset** is also triggered.

The **Validate** event is perhaps the most useful of the four events generated by the Remote Data control. It occurs *before* any operation that would cause the current row to no longer be the current row. In other words, if the user or application scrolls to a new row, this event is triggered first. Likewise, if the result set is closed, this event is triggered first. The event is also fired prior to the **Update** method. This gives you an opportunity to validate any changes made by the user as well as to cancel the pending operation that triggered the event. The syntax is shown in the next code example. **Action** tells you what operation is pending that caused this event to be triggered. If you change the value to **rdActionCancel**, the operation will be canceled. Possible values are listed in Table 6.10.

Table 6.10 Remote Data control Validation event Action constants.

Constant	Description
rdActionCancel	Cancel the operation when the **Sub** exits.
rdActionMoveFirst	**MoveFirst** method.
rdActionMovePrevious	**MovePrevious** method.
rdActionMoveNext	**MoveNext** method.
rdActionMoveLast	**MoveLast** method.
rdActionAddNew	**AddNew** method.
rdActionUpdate	**Update** operation (not **UpdateRow**).
rdActionDelete	**Delete** method.
rdActionFind	**Find** method (not implemented).
rdActionBookmark	The **Bookmark** property has been set.
rdActionClose	**Close** method.
rdActionUnload	The form is being unloaded.
rdActionUpdateAddNew	A new row was inserted into the result set.
rdActionUpdateModified	The current row changed.
rdActionRefresh	**Refresh** method executed.
rdActionCancelUpdate	**Update** canceled.
rdActionBeginTransact	**BeginTrans** method.
rdActionCommitTransact	**CommitTrans** method.
rdActionRollbackTransact	**RollbackTrans** method

rdActionNewParameters	Change in parameters or order of columns or rows.
rdActionNewSQL	SQL statement changed.

```
Private Sub MSRDC1_Validate (Action As Integer, _
  Reserved As Integer)
```

When the **Validate** event fires, you should check to see whether any bound data has changed. If so, you should perform your business rule edits (I discuss this at length under the **Validate** event in Chapter 5) and cancel the operation that triggered the event if there is a data problem. If the Remote Data control moves to another row, the changes to the current record are made automatically.

The Remote Data control has a number of properties that mostly parallel those of other RDO objects, such as **Connect**, **BatchCollisionRows**, and **Resultset**. The two properties that are not derived from other RDO objects are **BOFAction** and **EOFAction**, which return or set the action to take when **BOF** or **EOF** is **True**. For **BOFAction**, the value **rdMoveFirst**, which is the default, causes the first record to remain the first record when a **MovePrevious** is attempted. Similarly, **rdMoveLast** for **EOFAction** causes the last record to remain the last record when a **MoveNext** is issued. **rdBOF** causes a move before the first record, triggering a **Validate** event on the first row and a **Reposition** event on the now invalid row. **rdEOF** behaves similarly for **EOFAction**. Additionally, you can set **EOFAction** to **rdAddNew**, which causes the **AddNew** method to be invoked.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The Remote Data control has a number of methods derived from other RDO objects: **BeginTrans**, **CommitTrans**, **RollbackTrans**, and **Cancel**. Other methods include the following:

- The **Refresh** method causes the result set to be closed and reopened, similar to the **Requery** method of **rdoResultset**. You will normally use this method either in a multiuser environment or when the parameters being passed to a parameterized result set have changed.
- The **UpdateControls** method causes all bound controls to be repopulated with columns from the result set. This is basically an "undo" method. If the user has changed some value in bound controls before the result set has been updated, this method will cause all of the bound controls to revert to their original values.
- The **UpdateRow** method is essentially the same as the **rdoResultset Update** method except that the **Validate** event is not triggered. If the **ClientBatch** cursor library is being used, the local result set is updated but the database is not updated until you invoke the result set's **BatchUpdate** method.

Remote Data control result sets must be either keyset type or static type.

Bonus: The RDO Ad Hoc Report Writer

Included on the CD-ROM is the RDO Ad Hoc Report Writer application. The source code is not printed here in order to save a few square miles of the rain forest. The running application is shown in Figure 6.7. I urge you to start it and study some of the code. It connects to the specified database and then retrieves a list of tables. You can click on any two tables and list their columns in the listboxes shown in the figure. Use the two combo box controls to specify how the two tables are to be joined. Select which columns you want to display in the report. You can even do a self-join by selecting the same table for both listboxes. When ready, press the Build button to build and display the SQL

SELECT statement. If you want, you can then alter the **SELECT** statement in the textbox. Otherwise, press the Run button to execute the select. The results appear in the grid control at the top of the form. I will expand upon this application in future chapters.

Figure 6.7 The RDO Ad Hoc Report Writer.

Note that you will have to alter the **SELECT** statement that retrieves the table and column names. For instance, to run this application on an Oracle database, change the table select statement to either of the following:

```
SELECT table_name FROM user_tables
' Or you can use the following
SELECT table_name
FROM all_tables
WHERE creator = 'CORIOLIS'
```

An examination of the listing will show that the choice to limit selects to only two tables was arbitrary. The code will work for three, four, or however many tables. Simply change the number of subscripts for the **sActTables** variable and change the loop counters in the **BuildQuery** procedure. You will need to devise a way to show more listboxes or simply display all of the columns in one listbox, similar to the query facility in Visual Basic's Visual Data Manager.

Where To Go From Here

In this chapter, I have provided guidance on the use of RDO, including numerous code examples and as much real-world advice as I could fit into one chapter. If you are doing pure ODBC development and do not want to consider ADO, you might want to move ahead to Chapter 11 where I discuss advanced database concepts. Otherwise, read Chapters 7 and 8 where I introduce ADO and OLE DB and then cover techniques to migrate from RDO to ADO.

You may also want to review Chapter 5, if you have not already done so, for some DAO concepts not covered in this chapter. Many or most of them are equally applicable to RDO. Likewise, take a look at Chapter 4 if you have not done so yet because I spent some additional time discussing the Remote Data control, including building an application with it.

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Previous | Table of Contents | Next

Chapter 7
Introducing ADO And OLE DB

Key Topics:

- Microsoft Data Access Components
- Overview of OLE DB
- Overview of ADO
- Event-driven programming and ADO
- Using the Active Data control
- Using ADO schema objects
- Transaction management with ADO
- Concurrency management with ADO

In this chapter, I will introduce you to the use of the Active Data Objects (ADO) data model, as well as the OLE DB technologies upon which it sits. I will guide you through most of the essential aspects of ADO development. In subsequent chapters, I will guide you through more advanced application of these techniques into a scalable network or Web-based model.

Microsoft Data Access Components

Microsoft Data Access Components (MDAC) is the umbrella over Microsoft's COM-based database development initiatives. MDAC consists of Active Data Objects (ADO), Remote Data Services (RDS), and OLE DB. Strictly speaking, Microsoft also includes ODBC under MDAC. MDAC is common to all Microsoft development tools and is likely to be embraced by other tool vendors.

With VB6 comes MDAC 2.0, which added an event-driven environment much like RDO's event-driven paradigm. However, MDAC is evolving rapidly, and Microsoft is likely to release functional improvements from time to time. A good page to bookmark in your browser is www.microsoft.com/data/ado, where you can keep an eye on the latest trends. You should be able to download and incorporate into VB new releases of MDAC as they occur.

To appreciate why Microsoft introduced MDAC, we need to back up a step and take a look at data access in general.

Universal Data Access

Data warehouses and similar systems seek to make available all corporate data in a central repository where it can be analyzed for the purpose of making more informed business decisions. This often involves copying data from disparate sources such as Access, Oracle, and Excel into a centralized database. The concept is called *Universal Data Storage* (UDS). Finding all business data and then getting it into a central repository is no trivial undertaking. It also does not guarantee that all the information is accessible because often, data is stored in unstructured and inaccessible formats such as email messages and Web pages.

A central underpinning of Microsoft's Universal Data Access (UDA) model is to *not* copy the data into a central location but rather to devise a methodology to access it in its native format. In other words, if you have an **Employee** table stored on a relational database, you should be able to dynamically join it to your email server to find all messages from "John Smith" whose subject is "Widget Sales."

UDA seeks to provide a high-performance interface to relational and nonrelational data sources in a common manner. It is an evolutionary step from ODBC, DAO, and RDO, but UDA will not replace any of those technologies soon.

Recall that in 1990, Bill Gates made a speech at Comdex where he espoused a "document-centric" way of computing. Under this scenario, a user works on a document, not knowing or caring what applications actually build the document. The document might have some text created by Word, computations performed by Excel, and graphics created by PowerPoint. The "shell" that the user is in might well be your Visual Basic application. The first incarnation of this thrust was Dynamic Data Exchange (DDE). DDE evolved into, but was not replaced by, Object Linking and Embedding (OLE), now redubbed ActiveX.

Because all computing and all documents eventually deal with data, UDA is a logical follow-up to Gates's document-centric thrust: The user deals with data, not knowing or caring where *it* is coming from either. UDA seeks to blur or make invisible the distinction between data in a local Microsoft SQL Server database and data on a Web page across the world. To the user, it is all the same. MDAC takes care of the details of massaging and merging that data behind the scenes.

Goals Of MDAC

Microsoft has espoused several goals for its MDAC efforts:

- Provide the programming interface to create data-bound Web pages in Internet Explorer 4.0 and above.
- Provide the programming interface to create dataware middle-tier components in client/server applications, particularly on Internet Information Server (IIS) 4.0 and above.
- Integrate the remoting services previously provided by the Active Data Connector (ADC), which is now a part of ADO.

Active Data Objects

You will sometimes see the “Active” in ADO called “ActiveX.” The two terms are synonymous, and I use the simpler word “Active.” ADO is to OLE DB what RDO is to ODBC. It is essentially a low-overhead wrapper around the OLE DB API. It adds key support for building client/server applications residing on traditional networks or on Web-based networks, including the Internet.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

OLE DB

OLE DB can be considered analogous to ODBC. It provides an interface to applications that make disparate data sources look as though they were the same data source. It logically organizes structured or unstructured data into rows and columns so that it can be accessed in a common way (such as via SQL **SELECT** statements).

An OLE DB data *consumer* is an application that uses OLE DB-provided data. A data *provider* is any software component that exposes its data via an OLE DB interface. A *service provider* is an OLE DB component that does not actually own any data but provides some other service. By definition, an OLE DB service provider is both a data consumer and a data provider. For instance, a query processor that sits between your application and the OLE DB provider reacts to data requests from your program. It sends a query to the database and receives the results. In that aspect, it is a data consumer. Once the data is received, it sends the data to your application and thus also acts as a data provider.

The very nature of OLE DB is to create the data interface in a component-based manner. Providers and consumers are components under COM and DCOM. Their relationship is shown in Figure 7.1. OLE DB defines these components:

- *Enumerator*—Searches for available data sources and other enumerators and is used by consumers that are not customized to a single data source.
- *Data Source object*—A component that contains the mechanism to connect to a database and through which you will manage transactions, manipulate record sets, and so on. The data source is analogous to the RDO environment and the DAO workspace.
- *Session object*—A representation of the current database connection,

roughly analogous to the RDO connection and DAO database or connection objects.

- *Transaction object*—Allows you to work with the database in terms of a transaction or logical unit of work.
- *Command object*—Used to execute SQL commands; roughly analogous to the RDO and DAO query objects. (In OLE DB, an SQL command is termed a *text command*.)
- *Rowset object*—The object with which you manipulate and view data. It is the equivalent of the DAO record set and the RDO row set.
- *Error object*—Contains information about database errors not occurring as a result of code errors and is akin to the DAO and RDO error objects.

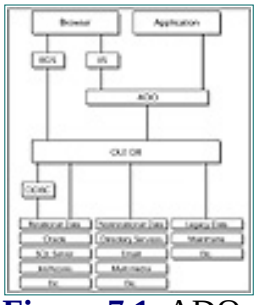

Figure 7.1 ADO as it relates to the application, OLE DB, and the database.

By and large, your application does not need to understand the inner workings of OLE DB any more than it needs to understand the complexities of the ODBC API. These objects can be exposed if you want. However, ADO places a programmatic wrapper around OLE DB, making your task easier.

ADO Overview

ADO allows your application to access any data that is exposed via an OLE DB interface. Further, your application can freely relate any OLE DB data source to any other OLE DB data source. Figure 7.1 shows the relationship of the application (including Internet-based applications) to ADO and OLE DB.

ADO MD

ADO MD (multidimensional) allows your application to access multidimensional data for any OLE DB for which there is an MD interface. Multidimensional data is often used in decision-analysis systems and seeks to "roll up" or summarize data at various meaningful levels. For instance, assume you are analyzing sales and customer data. You might summarize sales at various levels: region, state or province, month, and product. You would want to be able to cut across each of these elements at any level of detail, such as "sales in the northeast for 1999." You might then want to "drill down" by state, product, month, or all three. You can see where this takes on a multidimensional aspect, and you can think of it as roughly akin to a Visual Basic multidimensional array.

ADO And RDS

Remote Data Services (RDS) is a development model where the client application never physically connects to the database. All data access occurs through an intermediary such as Internet Information Server (IIS) or Microsoft Transaction Server (MTS). Instead of the traditional two-tiered application, these applications are three-tiered. Once a separate product, RDS is incorporated into ADO.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

ADO Data Types

ADO specifies a number of data provider-specific data types. Not all data types will apply to all data providers. (In fact, not all data types specified in ADO are exposed to Visual Basic.) For a complete list, see the **DataTypeEnum Enum** structure in the object browser. Table 7.1 lists the most common data types you will encounter.

Table 7.1 Valid ADO data types.

Constant	Description
adBigInt	8-byte signed integer
adBinary	Binary
adBoolean	Boolean
adBSTR	Null-terminated Unicode string
adChar	Fixed-length string
adCurrency	Currency
adDate	Date stored as Double where the whole part is the number of days since December 30, 1899, and the fractional part is the fraction of a day
adDBDate	Date as *yyyymmdd*
adDBTime	Time as *hhmmss*
adDBTimeStamp	Date and time stamp as *yyyymmddhhmmss* plus a fraction in billionths
adDecimal	Number with fixed precision and scale
adDouble	Double
adEmpty	No value
adError	32-bit error code

adGUID	Globally unique identifier (GUID)
adInteger	4-byte signed integer (Long)
adLongVarBinary	Long binary value (**Parameter** object only)
adLongVarChar	Long string value (**Parameter** object only)
adLongVarWChar	Long null-terminated string value (**Parameter** object only)
adNumeric	Number with fixed precision and scale
adSingle	Single
adSmallInt	Integer
adTinyInt	1-byte signed integer
adUnsignedBigInt	8-byte unsigned integer
adUnsignedInt	4-byte unsigned integer
adUnsignedSmallInt	2-byte unsigned integer
adUnsignedTinyInt	1-byte unsigned integer
adUserDefined	User-defined variable
adVarBinary	Binary value (**Parameter** object only)
adVarChar	Variable-length string value (**Parameter** object only)
adVariant	Automation Variant
adVarWChar	Null-terminated Unicode string (**Parameter** object only)
adWChar	Null-terminated Unicode string

Although most of these data types have VB equivalents, some do not. Either way, these ADO data types help you to interpret or set the underlying data.

Using ADO Objects

Figure 7.2 shows the relationship of ADO objects to one another. Notice that the relationships are not hierarchical. In fact, you can program using the **RecordSet** and **Field** objects alone. Also, note that there are only four collections: **Errors**, **Properties**, **Fields**, and **Parameters**. The **Command**, **RecordSet**, and **Connection** objects are all standalone. Finally, note there is no equivalent to the **DBEngine** or **rdoEngine** objects in DAO and RDO. There is no overall “owner” of other ADO objects. Instead, each **Connection** object, for instance, has its own **Errors** collection.

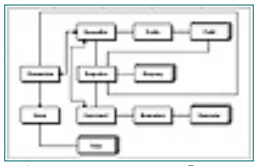

Figure 7.2 The ADO object model. For simplicity, the relationship between **Parameter** and **Properties** is omitted.

The ADO objects shown in Figure 7.2 are briefly explained here:

- The **Connection** object represents a connection to a database. Note

that the database may or may not be a relational source such as Oracle. It can be any OLE DB data source, such as email, an unstructured text file, or legacy data from a mainframe.

- The **Command** object represents an SQL statement such as a **SELECT** or **UPDATE** or some other data provider-specific command.
- The **Parameter** object is much like its DAO and RDO counterparts. You use it to specify variable values in queries and so on.
- The **RecordSet** object is used to view and manipulate data much like DAO's **RecordSet** and RDO's **rdoResultSet**.
- The **Field** object represents a field or a column in a **ResultSet**.
- The **Error** object is used to retrieve and handle error conditions from the database.
- The **Property** object represents a characteristic of various ADO objects. Typically, the **Property** object is used to represent properties of ADO objects that are nonstandard ADO properties. For instance, the **Connection** object will have somewhat different properties based on whether you are connected to SQL Server or Oracle.

To use ADO objects, you must make a reference to the ADO library, unless you are using the ActiveData control. There are actually two libraries: MSADO15.DLL and MSADOR15.DLL. The latter is a scaled-down version of the former, containing only the **RecordSet** and **Field** objects.

In the following pages, I outline the use of ADO, using many of the same application types that I used in Chapters 5 and 6.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The ADO Event Model

Like RDO, ADO supports an event-driven access model. The **Connect** and **RecordSet** objects support events. To gain access to their events, you must declare them using the **WithEvents** clause:

```
Dim WithEvents acon As Connection
Dim WithEvents ars As RecordSet
```

Most events are fired immediately before or after an operation such as a query. Those events triggered immediately before an operation allow you to examine, modify, or halt the operation to occur (such as connecting to a database). They usually begin with **Will** (such as **Will-Connect**). Those events triggered immediately after completion are most valuable in asynchronous operations or in determining if there were any errors. They usually are appended with **Complete** (such as **ConnectComplete**). ADO defines two families of events: **Connection** events occur when transactions begin, are committed, are rolled back, or when a **Command** or **Connection** begins or ends; **RecordSet** events are events that happen when changes are made to a record and when you navigate through records.

ADO provides a good deal more information when events are triggered than do DAO or RDO. Nearly all events pass to the application a status flag, **adStatus**. The possible values of **adStatus** are contained in **EventStatusEnum**. **adStatusOK** indicates that the operation completed without errors (although some events still generate warning messages), and **adStatusErrorsOccurred** signifies that there were errors during the operation. If **adStatus** is set to **adStatusCantDeny**, you are warned that you cannot cancel the pending event. Otherwise, you can set this property to **adStatusCancel** to cancel the operation that is about to occur. This option is applicable only to Will events, of course. Finally, you can set the property to **adStatusUnwantedEvent**, which tells ADO not to notify you of subsequent occurrences of the event.

For events where errors can occur, you will also be passed **pError**, which is a reference

to an **Error** object describing the error. If there was an error, you should iterate through the **Errors** collection, as discussed later in this chapter in the section "The **Errors** Collection And **Error** Object." If there was no error, then this value is **Nothing**. You should also iterate through the **Errors** collection whenever the **Connection** object's **InfoMessage** event is triggered.

Some of the **RecordSet** object's events pass **adReason**, which denotes the reason the event occurred. Typically, you are passed a reason code if, for instance, the user moves to another record. The possible values are enumerated in **EventReasonEnum**. Typical constants are **adRsnMove**, **adRsnAddNew**, and **adRsnClose**. All 15 constants currently defined are listed in the object browser; search on **EventReasonEnum**.

I discuss the specifics of the **RecordSet** events in "The **RecordSet** Object" later in this chapter, and the **Connection** events in the following section.

The Connection Object

The **Connection** object represents an open connection to a data source. If you were to operate on an Oracle database and perform a join to an SQL Server database, you would have two independent **Connection** objects. Each **Connection** object has an **Errors** collection, which is appropriate because two independent database connections can each generate a unique set of errors. Contrast that setup with DAO or RDO, where the **Errors** (or **rdoErrors**) collections are a function of the **DBEngine** or **rdoEngine** objects. If you were to simultaneously execute two different queries on two different databases under RDO, say, things would get interesting if they both generated errors. More to the point, if you were connected to both Oracle and SQL Server in an RDO session, any sequence of errors overlays the most recent sequence of errors. If SQL Server generated an error 1/10 of a second after Oracle, it wins: You would never see the Oracle errors. Under ADO, each connection maintains its own error information, solving this dilemma.

The **Connection** object also has a **Properties** collection, which I discuss later in this chapter.

Closing a connection breaks the open connection to the data source, but you need to set the object to **Nothing** to completely eliminate it from memory (**Set acon = Nothing**).

The **Connection** object is similar to the **rdoEngine** object. This is true for the events generated as well. The following examples show the syntax for each of the events supported by the **Connection** object:

```
Private Sub acon1_BeginTransComplete _
  (ByVal TransactionLevel As Long, _
  ByVal pError As ADODB.Error, _
  adStatus As ADODB.EventStatusEnum, _
  ByVal pConnection As ADODB.Connection)

Private Sub acon1_CommitTransComplete _
  (ByVal pError As ADODB.Error, _
  adStatus As ADODB.EventStatusEnum, _
  ByVal pConnection As ADODB.Connection)

Private Sub acon1_ConnectComplete _
```

```
  (ByVal pError As ADODB.Error, _
  adStatus As ADODB.EventStatusEnum, _
  ByVal pConnection As ADODB.Connection)

Private Sub acon1_Disconnect _
  (adStatus As ADODB.EventStatusEnum, _
  ByVal pConnection As ADODB.Connection)

Private Sub acon1_ExecuteComplete _
  (ByVal RecordsAffected As Long, _
  ByVal pError As ADODB.Error, _
  adStatus As ADODB.EventStatusEnum, _
  ByVal pCommand As ADODB.Command, _
  ByVal pRecordset As ADODB.Recordset, _
  ByVal pConnection As ADODB.Connection)

Private Sub acon1_InfoMessage(ByVal pError As ADODB.Error, _
  adStatus As ADODB.EventStatusEnum, _
  ByVal pConnection As ADODB.Connection)

Private Sub acon1_RollbackTransComplete _
  (ByVal pError As ADODB.Error, _
  adStatus As ADODB.EventStatusEnum, _
  ByVal pConnection As ADODB.Connection)

Private Sub acon1_WillConnect(ConnectionString As String, _
  UserID As String, Password As String, Options As Long, _
  adStatus As ADODB.EventStatusEnum, _
  ByVal pConnection As ADODB.Connection)

Private Sub acon1_WillExecute(Source As String, _
  CursorType As ADODB.CursorTypeEnum, _
  LockType As ADODB.LockTypeEnum, Options As Long, _
  adStatus As ADODB.EventStatusEnum, _
  ByVal pCommand As ADODB.Command, _
  ByVal pRecordset As ADODB.Recordset, _
  ByVal pConnection As ADODB.Connection)
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The **InfoMessage** event is similar to its RDO counterpart. It is triggered whenever an operation has completed successfully but the data provider has returned additional information. In ODBC, the additional information might be an SQL warning.

The **BeginTransComplete**, **CommitTransComplete**, and **Rollback-TransComplete** events are triggered after a transaction has begun, after a transaction has been committed, or after a transaction has been rolled back. The **TransactionLevel** is discussed under **IsolationLevel** later in this section. Use these events for any transaction handling that your application requires and to intercept any errors.

ConnectComplete and **DisconnectComplete** are especially valuable in asynchronous operations, notifying you when the connect and disconnect operations are completed.

The **WillConnect** event occurs immediately before an attempt to connect to the database. **ConnectionString** contains the **ConnectString** property of the **Connect** object. Likewise, **UserID**, **Password**, and **Options** contain other connection parameters. You can modify any of these parameters at this point.

The **ExecuteComplete** event is triggered when a command has completed execution. **RecordsAffected** returns the number of records affected by the command, and **pRecordSet** returns a reference to the record set created by the command, if any. If the command ran successfully but did not return any records, the record set will be empty.

Use the **Connection** object's properties as described here:

- The **Attributes** property specifies whether the connection performs *retaining* commits or aborts. A retaining commit (**adXactCommit-Retaining**) implicitly starts a new transaction whenever a **CommitTrans** is performed. A retaining abort (**adXactAbort-Retaining**) starts a new transaction whenever a **RollbackTrans** is performed. You can set these properties, but not all providers support this. The default is to not perform retaining commits or aborts, which is also my recommendation because it needlessly extends the duration of a transaction.
- The **CommandTimeOut** property defines, in seconds, how long to wait while a command is executing before timing out. If a command times out, ADO cancels the command and generates an error. The default is 30 seconds. A value of 0 means there is no timeout (the command will execute until completed). For most online processing, my recommendation is to set this value much lower than 30. The nature of online processing is

that few records are affected and any lengthy duration is indicative of a larger problem—either with your command or with the database itself. If your command is the cause of a lengthy execution, you need to tune it. If the source of the problem is the database, an excessively lengthy command only compounds the problem. Your DBA needs to investigate a database problem. Balance my advice with some common sense. If you are executing a command over the Internet, the vagaries of traffic will make the performance of your commands less predictable, so you will want to account for that. The **ConnectionTimeOut** property is similar except it sets how long to wait before an error occurs when attempting to connect to a data source. The default is 15 seconds, which is reasonable.

- The **ConnectString** property is much like RDO's **Connect** property. It contains the information necessary to connect to and log on to the database. Visual Basic recognizes only four arguments in the connect string. Any other arguments are passed directly to the data source.

Using The Connect String

The ADO connect string, contained in the **Connect** object's **ConnectString** and the ActiveData control's **ConnectionString** properties, is how you specify various parameters for connecting to the data source. ADO recognizes only four parameters within the connect string. Any others are passed through to the OLE DB provider without ADO attempting to interpret them. The four parameters are:

- **Provider** specifies the name of the OLE DB provider. If supplied, this is the only parameter allowed.
- **FileName** specifies the name of a file containing connection information. This argument is mutually exclusive with any other ADO connect-string parameter.
- **Remote Provider** is used in RDS connections to specify the name of a remote provider. If used, the only other parameter allowed is **Remote Server**.
- **Remote Server** is used to specify the path of the server in an RDS connection. If used, the only other parameter allowed is **Remote Provider**.

With Visual Basic, Microsoft delivers OLE DB providers for Oracle, SQL Server, Jet, Active Directory Service, Microsoft Index Server, and ODBC. Microsoft and other vendors will be developing OLE DB providers for other data sources, and you might want to bookmark the page www.microsoft.com/data to keep track of new developments. In the meantime, ODBC is a near-universal access point for relational databases. The Microsoft OLE DB provider for ODBC allows you to connect to any ODBC data source. To specify the connection to an ODBC data source, set the **Provider** argument equal to **"MSDASQL"**. This is the default provider, so even if it's omitted, ADO will attempt to connect to an ODBC data source. Other parameters in the connect string are database and driver dependent. The following snippet of code creates two **Connection** objects and then connects to the database. Each uses the Microsoft OLE DB provider for ODBC. The first uses a traditional DSN connection, but the second uses a DSN-less connection:

```
Set aCon1 = New Connection
aCon1.Open ("PROVIDER=MSDASQL; dsn=Coriolis VB Example;" & _
"uid=coriolis; pwd=coriolis;")
  Set aCon2 = New Connection
aCon2.Open ("UID=Coriolis; PWD=Coriolis;" & _
  "Driver=Sybase SQL Anywhere 5.0;" & _
  "DBF=C:\ Examples\Coriolis VB Example.db;" & _
  "DBN=Coriolis;")
```

As with RDO, the exact parameters may vary from database driver to database driver.

You can specify a **Database=** argument with either a DSN or DSN-less connection. For the former, the DSN already implies the database (as defined in the ODBC setup). Specifying a different database has the effect of altering the DSN's definition.

Microsoft also delivers an OLE DB driver for Jet, which you would normally use when you need to join an ISAM data source for which no OLE DB provider exists to another OLE DB data source. The provider name is **Microsoft.Jet.OLEDB.3.51**. (Microsoft may have delivered a Jet 4.0 version by the time this book is printed.) For Microsoft SQL Server, the provider name is **SQLOLEDB**. For Oracle, use **MSDAORA**. For the specifics of these and other OLE DB provider connection strings, see the Visual Basic help file (for OLE DB providers provided by Microsoft) or the vendor's documentation.

- The **CursorLocation** property determines where a cursor is to be located (similar to how you set the cursor library under RDO). You can set this value at any time on a **Connection** but it is read-only on an open **RecordSet** object. The value **adUseClient** specifies that a local cursor will be used; in other words, ADO will manage the cursor. This provides some flexibility in that the client-side cursor often has features not supported by a server-side cursor. The value **adUseClientBatch** is synonymous. If you are using RDS, you can only specify **adUseClient**. **adUseServer** specifies that the server will manage the cursor. For larger record sets, this is more efficient and the cursor is more sensitive to changes made by other users. However, you lose the flexibility of the client-side cursor. My recommendation is that if you are generating a large result set, you should use the server-side cursor. Normally, you will do this when generating a report. You should not be creating large record sets in the online transaction environment. Note that **adUseNone** is obsolete and is retained for backward compatibility only.
- The **DefaultDatabase** property is not available with all OLE DB providers and is not available with RDS. It specifies a default database for the **Connection**. This has implications when you join together two different data sources where you would normally have to qualify object names with the database name. Specifying a default database allows you to omit this information.
- The **IsolationLevel** property can be set at any time but does not take effect until the next transaction begins. The property specifies the isolation level of the cursor. Isolation can be considered similar to variable scope in Visual Basic; it specifies how "visible" a cursor is. Table 7.2 summarizes the possible values. Note that RDS permits only **adXactUnspecified**. You will normally want to retain the default values. For instance, **adXactCursorStability** specifies that you will use "cursor stability," meaning that you cannot see changes made in other transactions until they have been made permanent (committed). You normally would not want to see any changes until they are committed to the database. Even more, not specifying cursor stability has a devastating impact on performance at the database level because of all the extra work the database has to do. The opposite of **adXactCursorStability** is **adXactReadUncommitted** (these two constants are mutually exclusive), which does allow you to see changes made in other transactions even if they have not been committed. **adXactIsolated** specifies that the cursor essentially works oblivious to and invisible from all other transactions. Although there are few instances where you would need to use this option, it necessitates the **adXactChaos** constant, which is also on by default. **adXactChaos** specifies that the transaction cannot overwrite pending changes from more highly isolated transactions.

Table 7.2 Valid IsolationLevel constants.

Constant	Description
adXactUnspecified	Provider is using a different isolation level, but the level cannot be determined.

adXactChaos	(Default.) Cannot overwrite pending changes from more highly isolated transactions.
adXactBrowse	Can view changes in uncommitted transactions.
adXactReadUncommitted	Same as **adXactBrowse**.
adXactCursorStability	(Default.) Can see changes in other transactions only after they have been committed.
adXactReadCommitted	Same as **adXactCursorStability**.
adXactRepeatableRead	Cannot see changes made in other transactions, but you can requery to refresh the record set with those changes.
adXactIsolated	Transactions are isolated from other transactions.
adXactSerializable	Same as **adXactIsolated**.

- The **Mode** property specifies the read and write permissions. You can only set it before making the connection to the data source. For RDS, it must be **adModeUnknown**. The other constants are listed in Table 7.3.

Table 7.3 Valid Mode constants.

Constant	Description
adModeUnknown	Permissions have not been or cannot be set.
adModeRead	Read-only.
adModeWrite	Write-only.
adModeReadWrite	Read and write.
adModeShareDenyRead	Others are denied read permission.
adModeShareDenyWrite	Others are denied write permission.
adModeShareExclusive	No one else can access the data source.
adModeShareDenyNone	No one can open the connection with any permissions.

- The **State** property is used to determine the current state of the connection. **adStateClosed** indicates that it is closed, and **ad-StateOpen** indicates that it is open. **adStateExecuting** and **adStateConnecting** indicate that an asynchronous command execution or connection is in process.

The **Connection** object supports eight methods, most of which are similar to their RDO counterparts:

- The **BeginTrans**, **CommitTrans**, and **RollbackTrans** methods are used in transaction management. Unlike with RDO, if the object does not support transactions, calling one of these methods does cause an error. If the provider supports transactions, the **Properties** collection will have an object named **Transaction DDL**. If this property is not present, transactions are not supported. I discuss transaction and concurrency management in Chapter 11. Use **BeginTrans** to begin a transaction. **CommitTrans** makes all changes to the record set during the current transaction permanent on the database and begins a new transaction. **RollBackTrans** undoes all changes to the record set during the current transaction and begins a new transaction.
- **Cancel** cancels the current asynchronous operation (either an **Execute** or a **Connect**).
- The **Execute** method is used to execute a **Command** object. If the command is a row-returning query, the results must be assigned to a **RecordSet** object. In the following code snippet, the first example is used for non-row-returning queries, and the second is for queries that do return rows. The **CommandText** argument is an SQL statement, table name, stored procedure, or some other provider-specific command text. The

RecordsAffected property is a **Long** returning how many records were affected by the command. If there was an error, this value will be -1. The **Options** argument specifies how to interpret **CommandText** and whether to perform the operation asynchronously. Table 7.4 lists possible values.

```
connection.Execute CommandText, RecordsAffected, Options
Set recordset = connection.Execute _
(CommandText, RecordsAffected, Options)
```

Table 7.4 Valid Execute constants.

Constant	Description
adCmdText	Evaluate **CommandText** as a command, such as an SQL statement.
adCmdTable	Generate an SQL query returning all rows from the table named in **CommandText**.
adCmdTableDirect	Return all rows from the table named in **CommandText**.
adCmdTable	**CommandText** is a table name.
adCmdStoredProc	**CommandText** is a stored procedure.
adCmdUnknown	Type of command in **CommandText** is not known.
adExecuteAsync	Execute asynchronously.
adFetchAsync	Remaining rows after the initial quantity specified in the **CacheSize** property should be fetched asynchronously.

• **Open** is used to establish a connection to a data source. Under RDS, the connection is not actually made until a **RecordSet** is opened. The syntax is shown in the following code snippet. **ConnectionString** was described earlier in this section in "Using The Connect String." **UserID** and **Password** specify the user ID and password for the connection. You may set **Option** to **adAsyncConnect** to cause the connection to occur asynchronously. All the arguments are optional.

```
connection.Open ConnectionString, UserID, Password, Option
```

• The **OpenSchema** method has no counterpart in DAO or RDO. It is used to return *schema* information from the database. You can think of a schema as a schematic of the database; it is the layout of tables and columns. The next code example shows the syntax for the method. When executed, it returns a **RecordSet** of type static-cursor, which is read-only. The **query_type** parameter is one of the constants listed in Table 7.5. The **criteria** argument is a further refinement of the **query_type**, as also shown in Table 7.5. (Only the more commonly used query types are shown; refer to the VB help file for a complete listing.) You specify **criteria** in the form of an array, as shown in the code example. Figure 7.3 shows an application, which you can find on the CD-ROM, that opens two schemas using the **adSchemas** query type, as shown in the next example. The code does not specify a **criteria** argument, so all the tables are brought back. Notice that the first two text boxes show the table name and type. The second two, which are from the second command in the code example, show only views. This command is the basis for further refining the Ad Hoc Report Writer project from Chapter 6. You will use the **OpenSchema** method to retrieve information from the database and help automate the process of constructing reports.

```
Set arsSchema(1) = acon.OpenSchema(adSchemaTables)
Set arsSchema(2) = acon.OpenSchema _
```

```
(adSchemaTables, Array(Empty, Empty, Empty, "VIEW"))
```

Table 7.5 OpenSchema query types and optional criteria.

Query Type	Criteria
adSchemaColumns	TABLE_CATALOG
	TABLE_SCHEMA
	TABLE_NAME
	COLUMN_NAME
adSchemaForeignKeys	PK_TABLE_CATALOG
	PK_TABLE_SCHEMA
	PK_TABLE_NAME
	FK_TABLE_CATALOG
	FK_TABLE_SCHEMA
	FK_TABLE_NAME
adSchemaPrimaryKeys	PK_TABLE_CATALOG
	PK_TABLE_SCHEMA
	PK_TABLE_NAME
adSchemaProcedures	PROCEDURE_CATALOG
	PROCEDURE_SCHEMA
	PROCEDURE_NAME
	PARAMETER_TYPE
adSchemaTables	TABLE_CATALOG
	TABLE_SCHEMA
	TABLE_NAME
	TABLE_TYPE
adSchemaViews	TABLE_CATALOG
	TABLE_SCHEMA
	TABLE_NAME

Figure 7.3 The ADO Open Schema demonstration program.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

You can see the beginning of the ad hoc report writer in the application shown in Figure 7.4. After connecting, the user clicks the Tables or the Views button. The left-hand listbox is filled in with the names of all tables on the database. This is done with the following code:

```
Set arsSchema(1) = acon.OpenSchema _
  (adSchemaTables, Array(Empty, Empty, Empty, "TABLE"))
Call Showtables
```

Figure 7.4 This screen allows the display of all the columns in any table.

The code then loops through the **RecordSet** and populates the listbox:

```
Private Sub Showtables()
lbTables.Clear
Do While Not arsSchema(1).EOF
 lbTables.AddItem arsSchema(1)!Table_Name
 arsSchema(1).MoveNext
Loop
End Sub
```

The View button works in a similar manner. Once the listbox is filled, the user can click any table, and the second list box is filled with all the columns on that table using this short piece of code:

```
Private Sub lbTables_Click()
```

```
Dim sTable As String
sTable = lbTables.Text
If sTable = "" Then Exit Sub
' Retrieve the column names
Set arsSchema(2) = acon.OpenSchema _
  (adSchemaColumns, Array(Empty, Empty, sTable, Empty))
lbColumns.Clear
Do Until arsSchema(2).EOF
  lbColumns.AddItem arsSchema(2)!Column_Name
  arsSchema(2).MoveNext
Loop
End Sub
```

The Properties Collection And Property Object

The **Connection**, **Command**, **RecordSet**, and **Field** objects all have a **Properties** collection containing provider-specific **Property** objects. (The **Parameter** object, when present, also has a **Properties** collection.) Figure 7.5 shows an application, contained on the CD-ROM, that iterates through each of the four **Properties** collections and lists their names in listboxes on each tab page. When you click any property, the application returns its value in the textbox. With the **Field** object, each **Field** in the **RecordSet** is listed, along with its **Properties** collection. If you click the **Field** name itself, its value is listed.

Figure 7.5 The ADO Properties demonstration program.

The code to iterate through the properties is not terribly complex. The following example iterates through each of the **Field** objects (**afld**) in a **RecordSet** (**ars**) and, for each, iterates through each **Property** object (**aprop**):

```
For Each afld In ars.Fields
  lbProp(3).AddItem afld.Name
  For Each aprop In afld.Properties
    lbProp(3).AddItem _
      "-" & afld.Name & " : " & aprop.Name
  Next
Next
```

Each **Property** object has four properties itself. The **Name** property identifies the property. In Figure 7.5, I have clicked the **BaseTableName** property; the textbox shows that the **Value** is **employee**. The **Attributes** property is one or more of the constants listed in Table 7.6.

Table 7.6 Valid Attributes constants for Property objects.

Constant	Description
adPropNotSupported	Indicates that the property is not supported by the provider.
adPropRequired	Value of this property must be set before data source is initialized.
adPropOptional	Value of this property does not need to be set before data source is initialized.
adPropRead	Property can be read.
adPropWrite	Property can be written to.

The Errors Collection And Error Object

The **Errors** collection, consisting of **Error** objects, is much like its DAO and RDO counterparts. Any time a provider (data source) error occurs, the collection is cleared and one or more new **Error** objects is placed into the collection. Some provider messages are added to the **Errors** collection but do not halt program execution. Any time you perform an action that might result in an error or warning, you should first invoke the **Clear** method of the collection to clear any entries. Following the database operation, you should check the collection's **Count** property to ensure there were no messages. The following code example illustrates this:

```
Set acon = New ADODB.Connection
' Clear any errors
acon.Errors.Clear
' Make new connection
acon.Open sCon
If acon.Errors.Count > 0 Then
  ' Message from the data provider
  Call ShowErrors(acon.Errors)
End If

Private Sub ShowErrors(errs As Errors)
Dim aErr As Error
Dim sMsg As String
For Each aErr In errs
sMsg = "Message from the data provider:" & vbCr & _
  aErr.Number & "-" & aErr.Description & vbCr & _
  "SQL State:" & aErr.SQLState & vbCr & _
  "Native: " & aErr.NativeError & vbCr & _
  "Source: " & aErr.Source MsgBox sMsg
Next

End Sub
```

The code example generates a message box similar to what is shown in Figure 7.6.

Figure 7.6 Message from the data source provider being displayed to the user.

ADO errors themselves are not entered into the **Errors** collection; they are handled by Visual Basic's normal runtime error-handling system.

The **Error** object has several properties and no methods. The **Number** property is similar to the **Err** object's **Number** property and uniquely identifies the error (or warning) condition. Conversely, the **NativeError** property is the data provider's own error code. **SQLState** corresponds to the ODBC SQL state, as documented in the ANSI SQL standards. It is a five-character string, which is enumerated in your ODBC driver documentation. The **Description** property comes from either ADO or the data source provider itself and is a short description of the error condition. The **Source** property is the name of the object in which the error was generated. Generally, this is the data provider itself. Figure 7.7 shows the text returned from a data provider for an invalid **SELECT**. (The sample **Properties** program on the CD-ROM deliberately invokes an error and then recovers to "fix" the problem.)

Figure 7.7 Error message detailing an invalid **SELECT** statement.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The Command Object

The **Command** object is much like the DAO **QueryDef** and RDO **rdoQuery** objects. It essentially represents a command to be executed by the data provider—typically but not necessarily an SQL statement. If the command is parameterized, the **Command** object has a **Parameters** collection.

The **Command** object has several properties:

- The **ActiveConnection** object is used to associate the **Command** object with a **Connection** object. You can't execute a command outside of the context of a valid connection. Optionally, you can specify a connecting string (see the **ConnectString** property of the **Connect** object earlier in this chapter) in which case ADO will create a **Connect** object but will not assign it to a variable. Setting **ActiveConnection** to **Nothing** disassociates it from a **Connection**. Conversely, if you close the connection, ADO sets the **Command** object's **ActiveConnection** property to **Nothing** automatically. The following example sets the **ActiveConnection** property of **acmd**, an existing **Command**, to the previously opened **Connection acon**:

```
acmd.ActiveConnection = acon
```

- The **CommandText** property is a string containing the command to be executed. The command might be an SQL statement, a stored procedure, a table name, or some other command recognized by the data provider. The following example sets the **CommandText** property to an SQL **SELECT** statement:

```
acmd.CommandText = "Select * from employee"
```

- The **CommandTimeOut** property is a long with a default of 30 that specifies how long the command can execute before generating an error.

A value of 0 specifies that there is no timeout.

- The **CommandType** property is one of the values listed in Table 7.7 and tells ADO how to interpret the **CommandText** property. For SQL statements, you will normally set this property to **adCmdText** or simply leave it at its default of **adCmdUnknown**. **adExecuteNo-Records** indicates that the command is an action query (for example, an **UPDATE** statement) that returns no rows. It must be combined with **adCmdText** or **adCmdStoredProc**. If any rows are returned, they are discarded. With **adCmdTable**, the data provider attempts to generate a query that will return all columns and rows from the table name specified in the **CommandText** property.

Table 7.7 Valid CommandType constants.

Constant	Description
adCmdText	**CommandText** is a command such, as an SQL statement.
adCmdTable	Generate an SQL query returning all rows from the table named in **CommandText**.
adCmdTableDirect	Return all rows from the table named in **CommandText**.
adCmdStoredProc	**CommandText** is a stored procedure.
adCmdUnknown	(Default) The type of command in the **CommandText** property is not known.
adCommandFile	**CommandText** is a saved (persisted) **RecordSet**.
adExecuteNoRecords	**CommandText** is a command or stored procedure that does not return rows.

- The **Prepared** property is a Boolean that, if set to **True**, causes the data provider to prepare (compile) the command before first running it. For short, one-time commands, it generally doesn't make sense to prepare the command because this also takes time. For longer-running commands or if the command will be executed repeatedly, preparing it first probably makes sense. Not all providers support command preparation. The data provider may or may not generate an error if it doesn't support preparation.
- The **State** property indicates the current state of the object. **adStateClosed** indicates that it is closed, whereas **adStateOpen** indicates that it is open. **adStateExecuting** indicates that an asynchronously executing command is still executing.

To execute a **Command**, use its **Execute** method, which I discussed earlier in this chapter in the section "The **Connection** Object." As noted, you can use a record-returning command to create a **RecordSet**. If the command is executed asynchronously, you can use the **Cancel** method while it is executing to halt it.

Use the **CreateParameter** method to create a new **Parameter** object using the syntax shown in the next code example. You must manually append the **Parameter** to the **Parameters** collection, at which time ADO will validate it. The **Type** argument specifies the data type of the parameter, as listed in Table

7.1 earlier in this chapter. The **Direction** argument is used in the same manner as the DAO and RDO **Direction** counterpart; it sets whether a parameter is input, output, or both. See the **Direction** property of the **Parameter** object for more information. The **Size** argument specifies a maximum length or size where the data type is variable length. The **Value** argument sets the actual data value stored in the parameter as an optional **Variant**.

```
Set parameter = command.CreateParameter (Name, Type, _
   Direction, Size, Value)
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The Parameters Collection And Parameter Object

The **Parameters** collection and **Parameter** object work similarly to their DAO and RDO counterparts. **Parameters** represents all the parameters or arguments to a query or stored procedure. See “The **rdoParameters** Collection And **rdoParameter** Object” in Chapter 6 for a generalized discussion of parameterized queries and stored procedures.

Each **Parameter** object has a **Properties** collection, as discussed earlier in this chapter.

Listing 7.1 shows a query being constructed with five parameters. (The running application is shown in Figure 7.8 and is included on the CD-ROM.) The **CommandText** property of the **Command** object has five question marks, each of which represents a placeholder for a parameter. ADO expects, then, that five parameters will be supplied, and it generates an error if there are not enough parameters.

Listing 7.1 Using parameters in ADO.

```
Private Sub cmdRun_Click()
Dim sSex As String * 1
Dim iCtr As Integer

If Option1 Then
  sSex = "M"
Else
  sSex = "F"
End If

' New command object
Set acmd = New Command
acmd.CommandText = _
  "Select emp_no, emp_fname, emp_lname, " & _
  "emp_ssn, emp_dob, emp_dept_no, " & _
```

```
  "emp_salary, emp_gender " & _
  "From employee " & _
  "Where emp_gender = ? " & _
  "and emp_salary between ? and ? " & _
  "and emp_dept_no between ? and ? " & _
  "order by emp_lname, emp_fname"
acmd.ActiveConnection = acon
' Create Parms and Set Values
Set aparm(0) = acmd.CreateParameter("Sex", adChar, _
  adParamInput, 1, sSex)
Set aparm(1) = acmd.CreateParameter("SalFrom", adNumeric, _
  adParamInput, , Val(txtSalFrom))
Set aparm(2) = acmd.CreateParameter("SalTo", adNumeric, _
  adParamInput, , Val(txtSalTo))
Set aparm(3) = acmd.CreateParameter("DeptFrom", adNumeric, _
  adParamInput, , Val(txtDeptFrom))
Set aparm(4) = acmd.CreateParameter("DeptTo", adNumeric, _
  adParamInput, , Val(txtDeptTo))
' Append the parameters
For iCtr = 0 To 4
  acmd.Parameters.Append aparm(iCtr)
Next
' New RecordSet
Set ars = New Recordset
ars.Open acmd, , adOpenDynamic, adLockOptimistic, adCmdText
Call ShowRecs

End Sub
```

Figure 7.8 A parameterized query screen.

Unfortunately, ADO does not automatically create the **Parameter** objects as do DAO and RDO. You use the **CreateParameter** method of the **Command** object to create the five needed parameters (previously declared as an array of type **Parameter**). In each, an appropriate data type is supplied. The **Direction** is set to **adParamInput**, indicating that the parameter is an input parameter. This could have been omitted; it is more appropriate for stored procedures. I also supplied the value of the parameter, but I could have been supplied it after the fact as well (**aparm(0).Value = sSex**). After the parameters are created, I append them to the **Parameters** collection and then execute the query creating an output **RecordSet**.

- The **Attributes** property is used to describe the **Parameter**. It can be one or more of the following constants: **adParamSigned** indicates that the **Parameter** accepts signed numbers; **adParamNullable** indicates that the **Parameter** can accept **Null** values; and **adParamLong** indicates that the **Parameter** can accept very long binary or character data with the **AppendChunk** method.

- The **Direction** property is used to specify the type of **Parameter**, particularly for stored procedures. Normally, the data provider can determine the direction of the **Parameter** but, if it can't, you need to specify the **Direction** yourself. You can do this as part of the **CreateParameter** method (as I did in Listing 7.1) or after the parameter has been created using the syntax **parameter.Direction = constant**, where **constant** is equal to one of the following values: **adParamInput** for input parameters; **adParamOutput** for output parameters; **adParamInputOutput** for parameters that are both input and output; and **adParamReturnValue** for parameters that are used as return values (such as the number of rows returned from a stored procedure). **adParamUnknown** indicates that the **Direction** is unknown.
- The **NumericScale** property sets the scale of numeric values, whereas **Precision** sets the precision of numeric values. Certain SQL numeric data types such as **adNumeric** (see Table 7.1) require that you set the precision and scale of the number. Scale refers to how many digits are in the number, and precision refers to how many of those digits are to the right of the decimal point. A number with a precision of 11 and a scale of 2 is 11 digits wide with 2 of those digits to the right of the decimal.
- The **Size** property sets the maximum width of character data types, such as **adVarChar**. ADO reserves memory based on this property, so it is important that you set this property correctly to prevent program errors.
- The **Type** property specifies the data type of the **Parameter** object, as listed in Table 7.1. Note that the VB documentation specifies some data types that you are not likely to encounter. Use the object browser to see a complete listing.
- The **Value** property sets or returns the actual value of the **Parameter**. Its data type is **Variant** so that you can use any underlying data type.

The **Parameter** object has only one method: **AppendChunk**. The syntax for the method is **parameter.AppendChunk data**, where **data** is the data you are appending to the **Value** property of the **Parameter**. The first time you call this method, any data in the **Parameter** is overwritten. Subsequent calls add data to the end of the existing data. You use this method when you need to move large amounts of character or binary data into a **Parameter**. When you use this method, be sure that the **Attributes** property has **adParamLong** set.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The RecordSet Object

The **RecordSet** object is the ADO object with which you will interact the most. It represents a set of records from the data provider and provides the ability to scroll through the records as well as add, edit, and delete them. Where RDO refers to rows and columns, ADO uses the terms records and fields, much like DAO.

> **NOTE**
> The CD-ROM has a number of applications that illustrate the use of **RecordSet** objects. Figure 7.9 shows an employee maintenance application running. The application is capable of opening multiple maintenance forms within an MDI form. On the CD-ROM, the application name is ADOEmployeeMaint. I expand upon these concepts in the next several chapters, so I leave it until then to dissect and discuss the applications themselves.

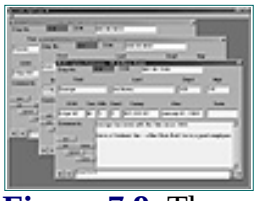

Figure 7.9 The employee maintenance application.

The **RecordSet** object has two collections: **Properties** and **Fields**. The **Field** objects work in essentially the same manner as they do with DAO and RDO (where they are called **rdoColumns**). The default property of the **RecordSet** is the **Fields** collection, and the default property of individual **Field** objects is **Value**. You can use shorthand to reference the value of any field using the familiar exclamation-point convention:

```
ars!emp_no
```

To move through a set of records, you use a cursor. ADO provides four types:

- The dynamic cursor allows unrestricted movement forward and backward through the **RecordSet**. Depending on the data provider, you may or may not be able to use bookmarks to move.
- The keyset cursor is similar to the dynamic cursor except that it prevents access to records that other users have added, and it prevents manipulation of records that other users have deleted. (A record can be deleted from the database after you have retrieved it into your **RecordSet**).
- The static cursor creates a nonupdatable **RecordSet** that allows unrestricted movement but does not see changes made by other users. You would normally use this type of cursor when building reports.
- The forward-only cursor is identical to the dynamic cursor except that you can only move forward through the **RecordSet**.

You want to use the cursor model that meets your needs while using the least amount of resources. For instance, if you only need to loop through the records once, then the forward-only cursor is more efficient than the dynamic cursor. Note that not all cursor types are supported by all data providers.

Set the cursor type before opening the **RecordSet** or when you use the **Open** method.

If you have used DAO **RecordSet** objects or RDO **ResultSet** objects, then you should be pretty comfortable with the ADO **RecordSet**. There are some differences, but they are outweighed by the similarities.

NOTE
In DAO and RDO, moving to a new record or row without first saving your changes causes those changes to be lost. ADO, however, automatically saves your changes when you move to a new record. To *not* save the changes, you must explicity call the **CancelUpdate** method.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The **RecordSet** object supports a number of events that are useful, especially when performing asynchronous operations. I discussed some generalities about ADO events earlier in this chapter in the section "The ADO Event Model." In all **RecordSet** events, **pRecordSet** is passed as a reference to the **RecordSet** for which the event occurred. The syntax of each event follows:

```
Private Sub ars_EndOfRecordset(fMoreData As Boolean, _
  adStatus As ADODB.EventStatusEnum, _
  ByVal pRecordset As ADODB.Recordset)

Private Sub ars_FetchComplete(ByVal pError As ADODB.Error, _
  adStatus As ADODB.EventStatusEnum, _
  ByVal pRecordset As ADODB.Recordset)

Private Sub ars_FetchProgress(ByVal Progress As Long, _
  ByVal MaxProgress As Long, _
  adStatus As ADODB.EventStatusEnum, _
  ByVal pRecordset As ADODB.Recordset)

Private Sub ars_FieldChangeComplete(ByVal cFields As Long, _
  ByVal Fields As Variant, ByVal pError As ADODB.Error, _
  adStatus As ADODB.EventStatusEnum, _
  ByVal pRecordset As ADODB.Recordset)

Private Sub ars_MoveComplete _
  (ByVal adReason As ADODB.EventReasonEnum, _
  ByVal pError As ADODB.Error, _
  adStatus As ADODB.EventStatusEnum, _
  ByVal pRecordset As ADODB.Recordset)

Private Sub ars_RecordChangeComplete _
  (ByVal adReason As ADODB.EventReasonEnum, _
  ByVal cRecords As Long, _
  ByVal pError As ADODB.Error, _
  adStatus As ADODB.EventStatusEnum, _
```

```
  ByVal pRecordset As ADODB.Recordset)

Private Sub ars_RecordsetChangeComplete _
  (ByVal adReason As ADODB.EventReasonEnum, _
  ByVal pError As ADODB.Error, _
  adStatus As ADODB.EventStatusEnum, _
  ByVal pRecordset As ADODB.Recordset)

Private Sub ars_WillChangeField(ByVal cFields As Long, _
  ByVal Fields As Variant, adStatus As ADODB.EventStatusEnum, _
  ByVal pRecordset As ADODB.Recordset)

Private Sub ars_WillChangeRecord _
  (ByVal adReason As ADODB.EventReasonEnum, _
  ByVal cRecords As Long, _
  adStatus As ADODB.EventStatusEnum, _
  ByVal pRecordset As ADODB.Recordset)

Private Sub ars_WillChangeRecordset _
  (ByVal adReason As ADODB.EventReasonEnum, _
  adStatus As ADODB.EventStatusEnum, _
  ByVal pRecordset As ADODB.Recordset)

Private Sub ars_WillMove _
  (ByVal adReason As ADODB.EventReasonEnum, _
  adStatus As ADODB.EventStatusEnum, _
  ByVal pRecordset As ADODB.Recordset)
```

- The **EndOfRecordSet** event is triggered by an attempt to move beyond the end of the **RecordSet**. There is no corresponding **BeginningOfRecordSet** event. It is conceivable that, while in this event procedure, you could add more records to the **RecordSet**. If so, set **fMoreData** to **True**.
- **FetchComplete** is called whenever a fetch operation has completed. **FetchProgress** is called periodically during an asynchronous fetch operation. **Progress** is a count of how many records have been fetched into the **RecordSet** so far, and **MaxProgress** is an estimate of how many records are expected to be fetched.
- The **FieldChangeComplete** event is called after a **Field** object in a **RecordSet** has been changed. The **WillChangeField** event is called before the **Field** is changed. The **Fields** argument is an array containing those **Field** objects that will be or were changed, and **cFields** is a count of the number of objects in the array.
- **MoveComplete** is triggered at the end of any **Move** operation, such as **MoveFirst** or **MoveNext**. This is where you want to place any code that updates the screen based on the now current record. The **WillMove** event is triggered immediately before any of the following methods: **AddNew**, **Bookmark**, **Delete**, **Move**, **MoveFirst**, **MoveLast**, **MoveNext**, **MovePrevious**, **Open**, **Requery**, and **Resync**. Unless **adStatus** is set to **adStatusCantDeny**, you can cancel the operation by setting **adStatus** to **adStatusCancel**. The **WillMove** event is where you normally place any final edits of data changes.
- The **RecordChangeComplete** event is called immediately after one or more records in the **RecordSet** are changed, whereas the **WillChangeRecord** event is called immediately before one or more records are changed. The following methods trigger these events: **AddNew**, **CancelBatch**, **CancelUpdate**, **Delete**, **Update**, and **UpdateBatch**. **cRecords** indicates how many records will be or were changed.
- The **RecordSetChangeComplete** event is called after a **RecordSet** has been changed

due to the **Close**, **Filter**, **Open**, **Requery**, or **Resync** method. The **WillChangeRecordSet** is called immediately before those methods.

The **RecordSet** object has a number of properties that help you monitor or manipulate it:

- The **PageSize** property allows you to break your **RecordSet** set into logical *pages*. For instance, if you had a screen that displayed 20 records at a time, you might set this property to 20. **PageSize** sets or returns the number of records in a logical page. The default is 10. This allows you to take advantage of the **AbsolutePage** property. By setting this property, you cause the **RecordSet** to scroll to the first record of the specified page. **PageCount** returns the total number of pages in the **RecordSet**. Note that the last page might not be full (there could be fewer records on it than specified in **PageSize**). The following example sets a page size of 20 and then moves to page 3. The current record will then be 41, which is the first record on page 3.

```
' Set page size
ars.PageSize = 20
' Move to page 3
ars.AbsolutePage = 3
```

- The **AbsolutePosition** property is a one-based reference to the current record number. It can also be one of the following constants: **adPosBOF** indicates that **BOF** is **True**; **adPosEOF** indicates that **EOF** is **True**; and **adPosUnknown** indicates the **RecordSet** is empty or that the provider does not provide this property. **adPosUnknown** is equal to -1, which makes it compatible with DAO record sets and RDO result sets where -1 indicates no records or rows. Also, see the **RecordCount** property later in this section.

- **BOF** and **EOF** are Booleans that, if **True**, indicate the current record is before the beginning of the **RecordSet** or after the end. If there are no records, both properties will be **True**.

- **Bookmark** is a variant with which you can record the current record and return to it in the same way as in DAO and RDO. If you use the **Clone** method on the **RecordSet**, the new **RecordSet** inherits all the saved bookmarks as well. You can use bookmarks between the two **RecordSet** objects interchangeably. The following code example illustrates saving a bookmark and then scrolling to it:

```
Dim vBookMark As Variant
vBookMark = ars.Bookmark
' Move to the last record
ars.MoveLast
' Move to saved position
ars.Bookmark = vBookMark
```

- The **CacheSize** property sets the number of records that are buffered locally (at the client). The property can be set at any time but only affects the next retrieves. For instance, if you set **CacheSize** to 20 records, when the **RecordSet** is opened, the first 20 records are read and cached locally. When you attempt to move to the 21st record, another set of 20 records is read. You can use the **Resync** method to force records already cached to reflect changes made by other users.

- **CursorLocation** specifies where the cursor is to be located. **adUseServer** specifies that the server's cursor services are to be used. This specification can be more efficient but may not have all the functionality of client-side cursors. **adUseClient** specifies that the cursor will be maintained locally. You may find that with certain data providers, the **AbsolutePosition** and **RecordCount** properties are unavailable if you use **adUseServer**.

- The **CursorType** property sets the **RecordSet** type as discussed at the beginning of this section. Valid values are **adOpenForwardOnly**, **adOpenKeySet**, **adOpenDynamic**, and

adOpenStatic. If you will be performing batch updates, you must use either **adOpenStatic** or **adOpenKeySet**.

- The **EditMode** property is used to return the edit mode of the current record. **adEditNone** indicates that there is no edit in progress. **adEditDelete** indicates that the current record has been deleted, and **adEditAdd** indicates that the current record is new. **adEdit-InProgress** indicates that the current record is being edited, but the changes have not been saved.
- The **Filter** property is a variant that you can use to filter which records in the **RecordSet** can become current. Setting the property to an empty string ("") or to **adEditNone** removes any filtering. When a filter is in place, properties such as **AbsolutePosition** reflect the filter. You can set the filter to a criteria string using comparison operators such as **"EMP_LNAME = 'Smith'"** and you can join those comparisons together using Boolean operators such as **And** and **Or**. There are some restrictions on how you nest these (see the VB help file for more details). You can also set **Filter** to an array of **Bookmarks**. Finally, you can set it to one of the constants in Table 7.8.

Table 7.8 Valid Filter constants.

Constant	Description
adFilterNone	Remove the current filter.
adFilterPendingRecords	For batch mode, view only records changed but not yet saved to the server.
adFilterAffectedRecords	View only records affected by the last **CancelBatch**, **Delete**, **Resync**, or **UpdateBatch** method.
adFilterFetchedRecords	View only the most recently cached records.
adFilterConflictingRecords	View records that failed the last **UpdateBatch** attempt.

- The **LockType** property sets the type of record locking. **adLock-ReadOnly**, the default, specifies that the records cannot be altered. **adLockPessimistic** specifies that a record will be locked as soon as it is edited. **adLockOptimistic** specifies that a record is only locked when it is updated. **adLockBatchOptimistic** specifies that records will not be locked when edited and that the updates will be done in batch mode (and locked at that time). Pessimistic locking, of course, has a tendency to create lock escalations on the server, so it is not generally recommended. Optimistic locking, on the other hand, raises concurrency issues that I address at more length in Chapter 11. Note that the details of how locking is implemented may vary based on the provider. Note also that you cannot use **adLockPessimistic** for local cursors.
- Use the **MaxRecords** property before the **RecordSet** is open to restrict the number of records retrieved.
- The **RecordCount** property returns the number of records in the **RecordSet**. If the **RecordSet** does not support approximate positioning or bookmarks, then referencing this property will cause all records to be retrieved in order to determine an accurate record count. If this property is equal to -1, then ADO cannot determine the number of records. The following code example displays the current record number as well as record count:

```
txtRecCount = ars.AbsolutePosition & " of " &
ars.RecordCount
```

- The **Sort** property is available only when **adUseClient** is set for **CursorLocation**. It allows you to specify a **Field** object **Name** to sort the results by. The **RecordSet** will be accessed in that order. You can optionally specify the direction of the sort by appending **ASCENDING** or **DESCENDING**. The following code example sorts a **RecordSet** of employee records by salary in descending order:

```
If ars.CursorLocation = adUseClient Then
  ars.Sort = "emp_salary DESCENDING"
End if
```

- The **State** property is used to determine the current state of the **RecordSet**. Possible values are **adStateClosed**, **adStateOpen**, **adState-Connecting**, **adStateExecuting**, or **adStateFetching**.
- **Status** returns status information about the current record. It will contain one or more of the constants listed in Table 7.9.

Table 7.9 Valid Filter constants.

Constant	Description
adRecOK	The record was successfully updated.
adRecNew	The record is new.
adRecModified	The record was modified.
adRecDeleted	The record was deleted.
adRecUnmodified	The record was not modified.
adRecInvalid	The record was not saved because its bookmark is invalid.
adRecMultipleChanges	The record was not saved because it would have affected multiple records.
adRecPendingChanges	The record was not saved because it refers to a pending insert.
adRecCanceled	The record was not saved because the operation was canceled.
adRecCantRelease	The new record was not saved because of existing record locks.
adRecConcurrencyViolation	The record was not saved because optimistic concurrency was in use.
adRecIntegrityViolation	The record was not saved because the user violated integrity constraints.
adRecMaxChangesExceeded	The record was not saved because there were too many pending changes.
adRecObjectOpen	The record was not saved because of a conflict with an open storage object.
adRecOutOfMemory	The record was not saved because the computer has run out of memory.
adRecPermissionDenied	The record was not saved because the user has insufficient permissions.
adRecSchemaViolation	The record was not saved because it violates the structure of the underlying database.
adRecDBDeleted	The record has already been deleted from the data source.

- The **Source** property indicates the source of the **RecordSet**. It can be an SQL statement, a table name, a stored procedure, a file name (for a saved **RecordSet**), or a **Command** object. If using the latter, reference the object itself, not the **Name** property of the object, as shown in Listing 7.1.

The **RecordSet** methods are similar to those available in DAO and RDO:

- Use the **Move** methods to move through the **RecordSet**. **MoveFirst**, **MoveLast**, **MoveNext**, and **MovePrevious** scroll to the first, last, next, and prior records. The **Move** method allows you to move an absolute number of records forward or backward. Specify

as an argument the number of records to move. A negative number scrolls backward. You can optionally specify a starting point from which to move (the default is to move from the current record). Specify a valid **Bookmark** as a starting point. Alternatively, you can specify one of the following constants: **adBookmarkFirst** or **adBookMark-Last**, both of which cause moving to start from the first or last record.

- Not well documented is the **Find** method, which is similar to the **Find** method available in the DAO Jet workspace. The syntax is shown in the following code. **criteria** is a string containing the field name to be searched, the comparison operator to use (=, >, <>, and so on), and the value to be searched for (for example, **"emp_salary < 90000"**). You can also use **Like** for pattern searches. **SkipRows** specifies how many rows to bypass from the starting position before searching. If you specify 5, for example, the search will begin on the sixth row after the starting position. The default is 0. Specifying a negative number will cause the search to begin *before* the starting position. **searchDirection** specifies in what direction to search. **adSearchBackward** causes the search to occur backward from the starting position. The default is **adSearchForward**. **start** specifies the position from which to start searching. The default is the current record. However, you can also specify a saved bookmark from which to start searching.

```
' Syntax of the Find method
Find (criteria, SkipRows, searchDirection, start)
' Example of searching for all last names beginning with S
ars.Find ("emp_lname like 'S_*'")
```

- Use the **Requery** method to re-execute the **RecordSet**; it is the equivalent of closing and reopening it. Use the **Resync** method to refresh records in the **RecordSet** to reflect the changes of any other users. You can optionally specify two arguments: **AffectRecords** and **ReSyncValues**. **AffectRecords** defaults to **adAffectAll** to refresh the entire **RecordSet**. You can specify instead **adAffectCurrent**, which refreshes only the current record, or **adAffectGroup**, which refreshes the records that meet the current **Filter** condition. For **ReSyncValues**, specify either **adResyncAllValues**, which causes any pending updates to be canceled, or **adResyncUnderlyingValues**, which specifies that changes are not overwritten and pending updates are not canceled. Note that using the **Resync** method does not retrieve rows added by other users. Also, note that if a row in the current **RecordSet** has been deleted by another user, an error will be generated and added to **Errors**.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The **RecordSet** object's editing facilities, outlined in the following list, are much like those found in DAO and RDO, with some enhancements. If you are in batch mode, records aren't saved to the database until you call the **UpdateBatch** method. While in batch mode, calls to the **Update** method save the changes within the **RecordSet** itself. If you scroll from one record to another, ADO automatically invokes the **Update** method.

- Use **AddNew** to add a new record. You can optionally specify the **Fields** and **Values** arguments. **Fields** allows you to specify which **Field** objects of the **RecordSet** will be made available for editing. You can specify either the **Name** properties or the ordinal numbers of each of the **Field** objects in an array. If you specify **Fields**, you can also specify **Values** as an array of values to place into each of the objects in **Fields**. The items in **Values** must correspond positionally to the **Field** objects and must be equal in number. The following example illustrates:

```
ars.AddNew Array("emp_lname", "emp_fname", "emp_salary"), _
  Array ("Smith", "John", 23000)
```

- **Delete** is used to delete the current record. If you specify the argument **adAffectGroup**, all the records that meet the current **Filter** criteria are deleted.
- **Update** saves any changes made in the current record. You can also set values using the optional **Fields** and **Values** arguments, as in the **AddNew** method. When **Update** is called, the current record remains current.
- **UpdateBatch** is used to update the database with pending changes. The argument **adAffectCurrent** causes only the current record to be updated to the database. **adAffectGroup** causes only changes to records that meet the **Filter** condition to be updated to the database. **adAffectAll** is the default and causes changes to all records to be updated to the database. If there are any batch collisions, a message is added to the **Errors** collection. Use the **Filter** and **Status** properties to find those records with conflicts.
- The **CancelUpdate** and **CancelBatch** methods cancel pending changes. With **CancelBatch**, you can add an optional argument to specify which changes are to be canceled. Use **adAffectCurrent**, **adAffectGroup**, or **adAffectAll**.
- You can save the **RecordSet** as a file, a process known as *persisting*, using the **Save** method. If the **Filter** property is set, only filtered records are saved to the file. The file is not closed until the **RecordSet** is closed. Repeated calls to **Save** cause the records to be appended to the file. The syntax for the method is shown in the next code snippet. The

FileName argument is self-explanatory. The default for **PersistFormat** is **adPersistADTG**, which specifies a **RecordSet** format. It is currently the only value that can be specified, but Microsoft has placed it the **PersistFormatEnum** structure, indicating that other formats will be added in the future. The speculation is that these additions will include formats such as reports or other formatted data. A saved **RecordSet** can be opened later using the **Open** method of the **RecordSet**.

```
RecordSet.Save FileName, PersistFormat
```

- The **Clone** method creates a copy of the **RecordSet**, including all saved bookmarks. You can optionally specify **adLockReadOnly** to make the cloned **RecordSet** read-only. Otherwise, the cloned **RecordSet** has the same locking as the original. When the cloned **RecordSet** is opened, the current record is the first record. The following example creates a cloned **RecordSet**:

```
Dim arsClone As RecordSet
' Clone the original RecordSet
Set arsClone = ars.Clone
```

- The final method of the **RecordSet** object is **Supports**, whose record set features the currently open **RecordSet** supports. You pass it an argument of type **CursorOptionEnum**, and the method returns **True** or **False** to determine if the feature is supported. Note that these features are largely a function of the type of **CursorLocation**, the type of **CursorType**, and the data provider itself. Also, note that the **Supports** method does not guarantee that the feature is available under all circumstances. For example, a **RecordSet** may return **True** for **adUpdate**, indicating that there is nothing in the type and location of the cursor that would preclude updates. However, the nature of the data in the **RecordSet** set, such as a multiple table join, may mean that some fields will not be updatable. The following example illustrates the **Supports** method by testing whether the **MovePrevious** method is supported:

```
If ars.Supports (adMovePrevious) Then
  MsgBox "Can Move Previous"
Else
  MsgBox "Can Not Move Previous"
End If
```

- Constants that you can pass are **adAddNew**, **adApproxPosition**, **adBookMark**, **adDelete**, **adFind**, **adHoldRecords**, **adMovePrevious**, **adNotify**, **adResync**, **adUpdate**, and **adUpdateBatch**. Most of these are self-explanatory. The **adApproxPosition** tests whether the **RecordSet** supports the **AbsolutePosition** and **AbsolutePage** properties. **adHoldRecords** indicates that you can retrieve more records without committing any pending changes.

Previous | Table of Contents | Next

Products | Contact Us | About Us | Privacy | Ad Info | Home

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The Fields Collection And Field Object

The **Fields** collection represents all the fields in a **RecordSet**. The individual **Field** objects represent the individual fields in the **RecordSet**. Each **Field** object contains basic information about the field, such as its value, data type, and size. The **Field** object contains the **Properties** collection, as noted earlier in this chapter in the section "The **Properties** Collection And **Property** Object."

- The **Name** property is the name of the **Field** as stored on the database or as specified in the command that retrieved the data.
- The **Type** property contains a constant indicating the basic data type of the field as listed in Table 7.1. The **DefinedSize** property of the **Field** object contains its maximum width in bytes and is typically used for character fields, such as **adVarChar**. The **ActualSize** property returns the number of bytes actually stored in the **Field**. The **Precision** property returns the number of digits in numeric fields, such as **adNumeric**, whereas the **NumericScale** property returns the number of digits to the right of the decimal point.
- The **Attributes** property is one or more constants that describe characteristics of the **Field**. Some of the more common **Attributes** constants include **adFldUpdatable**, which indicates that the **Field** can be updated; **adFldFixed** indicates that the **Field** is fixed width; **adFldIsNullable** indicates that the **Field** can contain **Null** values; **adFldLong** indicates that the **Field** is a long binary field on which you can use the **AppendChunk** and **GetChunk** methods; and **adFldRowVersion** indicates that the **Field** object contains a time or date stamp, usually for concurrency purposes. Other constants are used less often but are documented in the Visual Basic help file under "Attributes."
- The **Value** property lists the current value of the **Field**, whereas the

OriginalValue property contains the value of the **Field** when first retrieved from the database. The **UnderlyingValue** property retrieves the current value as stored on the database. This is a concern if another user has changed the record after you retrieved the **RecordSet**. I discuss this and other concurrency issues in Chapter 11.

The **Field** object has only two methods, **AppendChunk** and **GetChunk**, which are both used to manipulate long binary and character data fields.

Use **GetChunk** to make iterative calls to retrieve data from a **Field** and store it in a variable. Assume that you have a **Field** object named **"emp_picture"** that contains a bitmap picture of an employee. The following example will retrieve the contents of the **Field** using a buffer size of 2,048 bytes. Once the contents are read, you can display the bitmap.

```
Dim vBuffer As Variant
Dim vPic As Variant
Dim lFldSize As Long

' Size of the Field
lFldSize = ars!emp_picture.ActualSize
Do Until lFldSize < 1
  ' Read a chunk
  vBuffer = ars!emp_picture.GetChunk (2048)
  lFldSize = lFldSize – 2048
  ' Append chunk to variable
  vPic = vPic & vBuffer
Loop
```

The **AppendChunk** method appends data to a **Field** object. See the **AppendChunk** method in the "The **Properties** Collection and **Property** object" earlier in this chapter.

Other ADO Objects

ADO supports three objects used in Remote Data Services: the **RDS.DataControl** object, the **RDS.DataSpace** object, and the **RDSServe.DataFactory** object.

The Active Data Control

Like RDO and DAO, ADO provides a data control, which you can draw on your form to simplify programming. In practice, it is almost identical to the Remote Data control. To use it, you must add it as a component via the Project|Components menu. Select the Microsoft ADO Data Control 6.0.

Figure 7.10 shows an application that was quickly assembled using the Active Data control. It is included on the CD-ROM as ADOData-Control.

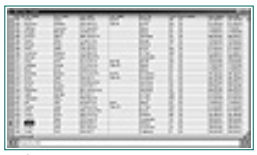

Figure 7.10 An application built using the Active Data control.

The Active Data control supports the same events as the **RecordSet**. It also has an **Error** event triggered when an entry is made to the **Errors** collection. It does not have a **Validation** event as does the intrinsic Data control; instead, use the **WillMove** event.

The Active Data control supports the **Refresh** method, which rebuilds the **RecordSet**. Its properties are basically those of the **Connection** object, and the control can be considered a surrogate for the **Connection** object. Some **RecordSet** properties are also exposed directly in the Active Data control, but most **RecordSet** properties and all **RecordSet** methods are exposed via the **RecordSet** property of the control itself.

The main advantage of the Active Data control is the same as that of the Remote Data control and the intrinsic Data control: simplicity of programming at the expense of some flexibility in application design. The biggest convenience is the fact that you can bind other controls directly to the Active Data control.

Where To Go From Here

This chapter presented a fairly detailed overview of OLE DB and ADO. You should be fairly comfortable with ADO development. If you plan to convert existing DAO or RDO applications to ADO, I encourage you to read Chapter 8. For developing an ADO application from scratch, Chapters 9 and 10 expand on the concepts covered in this chapter. For more advanced database techniques, read Chapter 11. Finally, after covering those materials, I move you into the realm of multitiered applications and Internet-based development in Chapters 12 and beyond.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Chapter 8
Converting To ADO

Key Topics:

- ADO vs. RDO and DAO
- Converting from DAO to ADO
- Converting from RDO to ADO
- Whether or not to convert to ADO

In the last chapter, I covered the key aspects of client/server development with ADO. In prior chapters, I discussed development with DAO and RDO and took the time to compare and contrast the three data models. ADO offers VB developers a solid platform on which to develop client/server applications and, in many cases, may be superior to DAO or RDO. This chapter will help you convert existing applications from DAO or RDO to ADO.

ADO Compared To DAO And RDO

As I discussed earlier in the book, DAO was Visual Basic's first relational data access tool. RDO was an event-driven, object-oriented follow-up to DAO that offered a much more streamlined and powerful approach to database development—but it came at the expense of not being an ideal ISAM development tool. ADO is more akin to RDO than it is to DAO, and RDO developers will have less trouble converting. In particular, DAO developers who have done extensive work with Jet (such as replicas) may have an uphill battle in converting to ADO. Microsoft has attempted to incorporate the functionality of both DAO and RDO into ADO, but you may find it difficult at first to locate the property or method you are looking for. Also, unlike

DAO/RDO, ADO is nonhierarchical, which may be a hurdle for some VB developers. ADO does *not* include all DAO functionality.

My recommendations on whether to convert are based on three questions:

- Does ADO support your current functionality? For DAO Jet developers, the answer may well be no. Carefully review your DAO application. If you are performing tasks such as replication via the Jet engine, it is probably wise to wait until ADO evolves more. For DAO ODBCDirect applications, ADO probably does include all needed functionality. RDO applications should have little or no problem in terms of losing functionality.
- Does ADO give you any technical advantages? Assuming the answer to the first question is yes, you might want to consider any advantages that ADO provides and factor those into your decision. Clearly, ADO is the wave of the future in terms of data access via Microsoft development tools and possibly non-Microsoft tools as well. ADO provides the capability of accessing relational and nonrelational data sources in a relatively transparent manner. ADO also simplifies the data model while providing a more robust asynchronous, event-driven programming environment. This is a big boon to DAO ODBCDirect developers who will benefit immediately by having access to data-source-driven events. There should also be a definite performance boost in ADO applications over DAO ODBCDirect. RDO developers may or may not see an immediate improvement in performance (although that should also come in time) but will benefit from the simpler data model and more elegant event model.
- Does it make economic sense to convert to ADO? This is really a two-part question. I am not a fan of technology for technology's sake. If your present application works and fills your current and future business needs, why change it? On the other hand, if you need to scale your application or migrate it to multitier models, DAO will leave you with no options, and RDO will probably restrict your room for growth. Also, consider the likelihood of having to do more development. Although RDO, say, may suffice to meet your needs, ADO may offer enough advantages in terms of productivity that outweigh the very real economic impact of converting.

ADO References

With DAO and RDO, you need to add a reference to the DAO or RDO library in order to access the data objects in code. With ADO, there are actually two libraries. MSADO15.DLL, the Microsoft Active Data Objects 2.0 library, is the one that you will use most often. You can use MSADOR15.DLL, the Microsoft Active Data Objects 2.0 Recordset library, instead when you only need record set functionality. This "lightweight" library includes the **Recordset** and **Field** objects only.

ADO Objects

Although ADO has a nonhierarchical object structure, many of the objects in

ADO, RDO, and DAO are roughly analogous to each other, as summarized in Table 8.1.

Table 8.1 DAO, RDO, and ADO object cross-reference.

DAO	RDO	ADO
DBEngine	**rdoEngine**	N/A
Error	**rdoError**	**Error**
Property	N/A	**Property**
Workspace	**rdoEnvironment**	N/A
Database	**rdoConnection**	**Connection**
Connection	**rdoConnection**	**Connection**
Container	N/A	N/A
Document	N/A	N/A
QueryDef	**rdoQuery**	**Command**
Field	**rdoColumn**	**Field**
Parameter	**rdoParameter**	**Parameter**
Recordset	**rdoResultSet**	**Recordset**
TableDef	**rdoTable**	N/A
Index	N/A	N/A
User	N/A	N/A
Group	N/A	N/A
N/A	**rdoPreparedStatement**	N/A

ADO has no object analogous to the **DBEngine** or **rdoEngine** objects. **Connection** objects are independent and thus do not need the equivalent of the **Workspace** or **rdoEnvironment** objects. Properties, methods, and events of those objects are largely part of the **Connection** or, in some cases, **Recordset** object.

ADO Events

Like RDO, ADO permits event-driven, asynchronous communications. However, these events are entirely in the context of the **Connection** and **Recordset** objects. ADO's events are a superset of those provided by RDO. Events related to the connection to the data source (including transactions) are largely within the **Connection** object. Events related to the data itself are within the **Recordset** object.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

ADO Properties And Methods

As noted earlier, many DAO methods and properties that are unique to the Jet engine are entirely missing from ADO (although they may be added at some future point). Methods and properties that apply to ODBCDirect and RDO are present in ADO, although you might initially have difficulty finding them. For instance, ADO does not have a **BatchCollisions** property; it relies instead on the **Filter** property, which allows you to filter out all but those records that failed the most recent batch update. As a pleasant surprise, some DAO Jet functionality not present in ODBCDirect or RDO has been added to ADO. Most notable is a **Find** method.

TIP
Use Common Sense
Before you start your conversion, back up all of your files. It may seem obvious, but you should *never* test on production data.

Active Data Control

The Active Data control is largely compatible with the intrinsic Data control and the Remote Data control. You will have to add the control to your project and respecify its properties, but that is a minimal investment. You will also want to seek out Data control events, such as **Validate**, and move that code to their ADO equivalents. Before making that investment in time, however, consider creating a **DataEnvironment** object or even a data-aware class to act as the data source for your bound controls. I discuss these techniques in the next two chapters. Also be aware that some grid controls are not compatible with ADO. Instead, use controls such as the Hierarchical FlexGrid control, which specifically say “OLEDB” in their description.

Converting The Application

If you have read this far in the chapter, you are probably ready to take the leap. To illustrate the process of converting DAO and RDO applications, I am going to use the sample batch update demonstration programs from Chapters 5 (DAO) and 6 (RDO). I will use the Parameter Query project from Chapter 5 as well. You might want to follow along as I outline a straightforward methodology for converting. Of necessity, these applications are not as large as what you may be converting, but there is enough DAO- and RDO-specific functionality in each to provide overall guidance.

Some of the techniques are simple brute-force search and replace. Microsoft or some third party may develop an approach that is more elegant, but I am taking the attitude of just "getting it over and done with." Follow these simple steps:

1. Save your project and all modules to new names or to a new directory.
2. Add a reference to the ADO library.
3. Remove the reference to the DAO or RDO library.
4. Globally replace all DAO/RDO variables with their ADO equivalents using Table 8.1 as a starting point. Modify declarations for the **Connect** and **Recordset** objects to include the **WithEvents** clause.
5. For DAO/RDO objects not represented in ADO:

    a. Globally replace references to the engine objects with an appropriate **Connection** object.

    b. If using only one **Workspace** or **rdoEnvironment** object, globally replace it with a reference to a **Connection** object.

6. Evaluate all RDO events and determine their ADO equivalents. Move code from the RDO events to the new ADO events and delete references to the RDO events.
7. Replace connection strings with their ADO equivalents (see Chapter 7).
8. Scan through the code, looking for any obvious mistakes in methods or property uses. Correct as many as you can.
9. Step through the code, letting the compiler alert you to remaining syntax problems.
10. Exercise all aspects of the program to shake out any remaining bugs or problems.

Once your application is working, let it "settle down" by running it for a few weeks. You might want to eventually modify code to take advantage of ADO functionality. For instance, if you were using the **StillExecuting** property in DAO to test for the completion of asynchronous operations, consider taking advantage of ADO events to accomplish the same purpose in a more streamlined manner.

Converting The DAO Application

For the DAO to ADO conversion test, I use two applications from Chapter 5. I chose the batch transaction program (**DAOBatchUpdate** on the CD-ROM) because

it exploits enough different methods and properties of DAO to be a valid example while being short enough to discuss within the space of this chapter. I also chose the parameter query application (**QDefParmRSDemo** on the CD-ROM) because it illustrates the use of **Parameter** objects and also because it uses the **QueryDef** and **DBEngine** objects not used in the batch update program.

The Batch Update Application

The first step is to remove the reference to DAO and add a reference to the Microsoft Active Data Objects 2.0 library. If you refer to the code, you will see four DAO objects declared in the **cmdUpdate_Click** event:

```
Dim wrkEmp As Workspace
Dim conEmp As Connection
Dim rsEmp As Recordset
Dim rsemp2 As Recordset
```

The two **Recordset** objects are easy. Globally replace **rsemp** with **arsEmp**. Globally replace **conEmp** with **aConEmp**. Delete the **wrkEmp** declaration.

As you will recall, a **Workspace** represents a database session. In ADO, the **Connection** object largely replaces this function. You need to find all places where **wrkEmp** is used and replace it with a reference to the **Connection** object, **conEmp**. The code has four such references.

Next, you want to replace all connection strings. The code has just one:

```
sConn = "ODBC;DSN=Coriolis VB Example;UID=Coriolis;" & _
     "PWD=Coriolis;Database=Coriolis;"
```

The ADO connect string for an ODBC data source, which is what you will handle when converting from RDO or DAO/ODBCDirect, is straightforward. See Chapter 7 for details about the **Connection** object's **Open** method. For this example, you can simply replace the **ODBC;** in the DAO connection string with **PROVIDER=MSDASQL;**, which is a reference to the Microsoft ODBC data provider. You can probably do a simple global replace in larger projects. The revised connect string now looks like this:

```
sConn = "PROVIDER=MSDASQL;DSN=Coriolis VB Example;" & _
  "UID=Coriolis;PWD=Coriolis"
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Note that I also deleted the **DATABASE=** parameter as unnecessary. You can actually simply omit the **PROVIDER** argument because the default is **MSDASQL**. For DSN-less connections, you can generally get away with simply deleting the **ODBC** argument.

At this point, you are probably 60 percent converted. Unfortunately, the remaining 40 percent of the work is a bit tougher. You need to scan through your code for the most flagrant uses of DAO objects, properties, and methods that have no one-to-one equivalency in ADO. The most obvious are uses of the **Workspace** object. In the code sample, the following lines of code immediately jump out:

```
' Create the workspace
  Set aconEmp = CreateWorkspace("ODBCWorkspace", _
    "Coriolis", "Coriolis", dbUseODBC)
' In order to do batch updating, set the default cursor
' driver prior to opening the connection.
  aconEmp.DefaultCursorDriver = dbUseClientBatchCursor
```

Note that the **Workspace** variable has been renamed to a **Connection** variable. The **CreateWorkspace** method is entirely unnecessary; its main purpose here is to establish an ODBC environment, sans Jet, which is the default under ADO anyway. Delete any references to creating a **Workspace** object. The **DefaultCursorDriver** property also has no equivalent in ADO. You will specify these types of properties when you open the **Recordset** objects. Delete that line.

Other **Workspace** properties that you may need to deal with are isolation levels and timeout values. You will set these at the **Connect** object level now. **Workspace** methods that you need to spot are **OpenDatabase** (which you will replace with **Open** at the **Recordset** level), **OpenConnection** (which you will replace with **Open** at the **Connection** level), and **Close** (which you will simply delete).

The next several lines of code look like this:

```
' Now, open the connection to the employee table
sConn = "PROVIDER=MSDASQL;DSN=Coriolis VB Example;" & _
```

```
  "UID=Coriolis;PWD=Coriolis"
Set aconEmp = aconEmp.OpenConnection("CorEmp", _
  dbDriverNoPrompt, False, sConn)
' Make sure the proper locking is set
' and open the record set
Set arsEmp = aconEmp.OpenRecordset _
  ("SELECT * FROM EMPLOYEE", dbOpenDynaset, 0, _
  dbOptimisticBatch)
```

You will alter the code as shown here:

```
' Now, open the connection
sConn = "PROVIDER=MSDASQL;DSN=Coriolis VB Example;" & _
  "UID=Coriolis;PWD=Coriolis"
Set aconEmp = New Connection
aconEmp.Open sConn
' Select from the employee table
Set arsEmp = New Recordset
arsEmp.Open "SELECT * FROM EMPLOYEE", aconEmp, _
  adOpenKeySet, adLockBatchOptimistic, adCmdText
```

In ADO, before you open an object, you will add the new **Set…New** lines of code as you did previously with **aconEmp** and **arsEmp**. Change the **OpenConnection** method to the **Open** method of **aconEmp** as shown. You can add the **adConnectAsync** option later, if you want, to have the connection open asynchronously. For the time being, you are merely replacing the existing code's functionality without enhancing it. Next, you need to replace the **Workspace** object's **OpenRecordSet** method with the **Open** method of **arsEmp**. The **dbOpenDynaset** argument becomes **adOpenKeySet**. Add the **adLockBatchOptimistic** argument to take into account that the DAO **Workspace** had a default of using batch updates. Also, add the **adCmdText** to make explicit that the **Source** of the **Recordset** is an SQL command.

NOTE
I did not use the more intuitive **adOpenDynamic** type of **Recordset** because it does not support batch updating. For batch updating, you must use either **adOpenKeySet** or **adOpenStatic**.

If it is easier for you, you can instead set the **CursorType** and **LockType** properties of the **Recordset** objects to **adOpenKeySet** and **adLockBatchOptimistic** prior to opening the **Recordset** objects.

Later in the code, another **Recordset** was opened. The line of code now looks like this:

```
Set arsEmp2 = aconEmp.OpenRecordset _
     ("SELECT * FROM EMPLOYEE", dbOpenDynaset, 0, _
      dbOptimisticBatch)
```

Replace it with these two lines of code using the same reasoning that applied earlier:

```
Set arsEmp2 = New Recordset
arsEmp2.Open "SELECT * FROM EMPLOYEE", aconemp, adOpenKeySet, _
  adLockBatchOptimistic, adCmdText
```

That pretty much takes care of any references to the **Workspace** object.

Next, scan through the code for uses of the **Recordset** objects. Happily, most of the methods and properties will work as is. A few key methods to spot are the following:

- **Update**—If your code specifies the **dbUpdateBatch** argument, replace the method with **UpdateBatch**. Unfortunately, there is currently no ADO equivalent to the **Force** argument of DAO's **Update** method to force updates to the database in the event of collisions.
- **CancelUpdate**—If doing batch updates, replace this method with the **CancelBatch** method.
- **Edit**—Delete any lines that use this method. The **Recordset** is placed into edit mode automatically any time you make changes to the record.

Also, look for properties that have been replaced. For instance, **Bookmarkable** is replaced by the ADO **Supports** method:

```
arsEmp.Supports (adBookmark)
```

The **BatchSize** property has been omitted from ADO. **BatchCollisionCount** and **BatchCollisions** have been replaced by the **Filter** property. The example has a section of code that looks like this:

```
If .BatchCollisionCount > 0 Then
' Move to first collision to demonstrate
' use of the batchcollisions array
  .Bookmark = .BatchCollisions(0)
  sMsg = Str$(.BatchCollisionCount) & " records " & _
    "have been altered by another user."
  MsgBox sMsg
End If
```

Instead, set the **Filter** property to look for any batch collisions and then use the **RecordCount** property to see how many there are:

```
' Check to see if any collisions
     arsEmp.Filter = adFilterConflictingRecords
     If arsEmp.RecordCount > 0 Then
       arsEmp.MoveFirst
       sMsg = arsEmp.RecordCount & " records " & _
         "have been altered by another user."
       MsgBox sMsg
    End If
```

By and large, that takes care of the changes needed to run the application. If your application uses the **DBEngine**, **QueryDef**, or **Parameter** objects, you need to make additional modifications. I discuss these in the next section.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The Parameter Query Application

The application from Chapter 5 used Microsoft Jet but would have worked just as well in an ODBCDirect workspace. It employs the **Database**, **QueryDef**, **Recordset**, and **Parameter** DAO objects.

Let's use an approach similar to how you converted the batch update project. The first step, of course, is to save all the files under new names so that you don't ruin the existing project. Next, you remove the reference to DAO and add a reference to ADO using the References option on VB's Project menu.

The next step is to change object references to their ADO counterparts or, where there is no counterpart, to the object that provides the closest match in functionality. The object declarations in the existing project follow:

```
Dim dbCoriolis As Database
Dim qdfDeptGenEmpSal As QueryDef
Dim rsReport As Recordset
```

Note that there is no explicit **Parameter** declaration because these objects are created dynamically by DAO when the **QueryDef** object is created.

Globally change all references of **dbCoriolis** to **aconCoriolis**. Change the data type to **Connection**. Globally change **qdfDeptGenEmpSal** to **acmdDeptGenEmpSal** and the data type to **Command**. Change **rsReport** to **arsReport**. Add the **WithEvents** clause to the **Connection** and **Recordset** declarations:

```
Dim WithEvents aconCoriolis As Connection
Dim acmdDeptGenEmpSal As Command
Dim WithEvents arsReport As Recordset
```

Note that in order to use the **WithEvents** clause, you move the declarations to the **General** section of the form module.

You must modify the connection string next. The current string and connection to the database look like the following:

```
sCoriolis = "c:\examples\coriolis.mdb"
Set aconCoriolis = OpenDatabase(sCoriolis)
```

Change these lines of code to this:

```
sCoriolis = "DSN=VB Coriolis Example"
aconCoriolis.Open sCoriolis, "Coriolis", "Coriolis"
```

You use a variation on the **Connect** object's **Open** method this time by specifying the user and password as separate arguments. You also let the data provider default to **MSADSQL** instead of explicitly specifying it.

The **QueryDef** object dynamically created parameters by using the Jet version of a parameterized query:

```
Set acmdDeptGenEmpSal = _
  aconCoriolis.CreateCommand("", _
    "PARAMETERS Dept Integer, Gender Char ; " & _
    "SELECT EMP_DEPT_NO, EMP_GENDER, EMP_NO, EMP_FNAME, " & _
    "EMP_LNAME, EMP_SALARY FROM EMPLOYEE " & _
    "WHERE EMP_DEPT_NO = [Dept] " & _
    "AND EMP_GENDER = [Gender] " & _
    "ORDER BY EMP_DEPT_NO, EMP_LNAME, EMP_FNAME")
```

You need to convert this to the more generic ODBC version (using question marks as placeholders) and create a **Command** object. First, you declare a new object. Next, you set its **ActiveConnection** and **CommandText** properties as shown:

```
Set acmdDeptGenEmpSal = New Command
acmdDeptGenEmpSal.ActiveConnection = aconCoriolis
acmdDeptGenEmpSal.CommandText = adCmdText
acmdDeptGenEmpSal.CommandText = _
  "SELECT EMP_DEPT_NO, EMP_GENDER, EMP_NO, EMP_FNAME, " & _
  "EMP_LNAME, EMP_SALARY FROM EMPLOYEE " & _
  "WHERE EMP_DEPT_NO = ? "AND EMP_GENDER = ? " & _
  "ORDER BY EMP_DEPT_NO, EMP_LNAME, EMP_FNAME
```

You do not have to explicitly create **Parameter** objects. ADO will do that for you automatically. If you do it, the code will execute somewhat faster. On the CD-ROM, I created the application both ways (with and without explicit declarations). If you open the project on the CD-ROM, you will see that the explicit declarations and references are commented out.

However, because you might choose to explicitly create the objects yourself, the following code example shows you how. First, create two objects for the two placeholders in the command text:

```
Dim aparmDept As Parameter
```

```
Dim aparmSex As Parameter
```

Next, code the actual **Create** statements:

```
Set aparmDept = acmdDeptGenEmpSal.CreateParameter
Set aparmSex = acmdDeptGenEmpSal.CreateParameter
```

You next code the data types and directions of the parameters. Strictly speaking, the data provider can normally determine this on its own, but it is good practice to be explicit:

```
aparmDept.Type = adInteger
aparmDept.Direction = adParamInput
aparmSex.Type = adChar
aparmSex.Direction = adParamInput
aparmSex.Size = 1
acmdDeptGenEmpSal.Parameters.Append aparmDept
acmdDeptGenEmpSal.Parameters.Append aparmSex
```

The code uses the **InputBox** function to prompt for what department and sex to search for and then assigns those values as shown:

```
acmdDeptGenEmpSal.Parameters("Dept") = iDept
acmdDeptGenEmpSal.Parameters("Gender") = sGender
```

Parameter objects aren't named in ADO (technically, the names are **Param1**, **Param2**, and so on). Simply change the **Name** property of the **Parameters** collection to the corresponding ordinal position:

```
acmdDeptGenEmpSal.Parameters(0) = iDept
acmdDeptGenEmpSal.Parameters(1) = sGender
```

The only other change of significance is where you open the **Recordset**. Under DAO, it looked like the next example after making the global changes:

```
' Set arsReport = _
' acmdDeptGenEmpSal.OpenRecordset(dbOpenSnapshot)
```

You change it to the ADO format as shown here:

```
Set arsReport = New Recordset
arsReport.CursorLocation = adUseClient
arsReport.Open acmdDeptGenEmpSal, , _
  adOpenForwardOnly, adLockReadOnly
```

The **Open** method of the **Recordset** references the previously created **Command** object. You specify the local cursor library to ensure the record count is accurate. You delete the **MoveLast** method in the DAO code sample, which was merely used to ensure the **RecordCount** property was accurate. You also clean up the code listing where it looped through the resulting **Recordset**. The DAO code sample looped through a **For Next** loop. You change that to a **Do Until** loop, which is somewhat easier to follow:

```
' Display the results
With arsReport
  Do Until .EOF
    txtFields(0).Text = !EMP_DEPT_NO
    txtFields(1).Text = !Emp_Gender
    txtFields(2).Text = !Emp_No
    txtFields(3).Text = !Emp_FName
    txtFields(4).Text = !Emp_LName
    txtFields(5).Text = !Emp_Salary
    ' Display More?
    If MsgBox("Display More?", vbYesNo + vbQuestion) = vbNo
    Then
    Exit Do
    End If
    arsReport.MoveNext
  Loop
End With
```

The CD-ROM includes a second version of this project, which populates a grid control, which is a more natural presentation of the results. The user is prompted for parameter values via textboxes instead of input boxes. The application is shown in Figure 8.1.

Figure 8.1 The parameterized query application converted to ADO.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

DAO To ADO Summary

The process of converting from DAO to ADO is not tough, particularly if you are already using an ODBCDirect data model. Still, although the completed ADO application should function in an equivalent fashion to its DAO predecessor, it does not take advantage of ADO's event-driven model and quite possibly does not take advantage of asynchronous communication. Once your project is up and running, give it some time to settle down—to shake out any bugs you may have introduced. Then, review the code, looking for opportunities to streamline your projects, move code to event procedures, take advantage of asynchronous communications, and so on.

Converting The RDO Application

RDO applications are relatively straightforward to convert to ADO. The basic principles remain the same as in DAO: Back up your work; globally replace RDO objects with their ADO equivalents; eyeball the code for obvious changes; step through the code to catch any other problems; and test thoroughly.

In one sense, the conversion is more complex than DAO in that, with this application at least, there is an event-driven environment from which to convert. On the other hand, the converted application will be a fully asynchronous, event-driven project unlike the converted DAO applications.

For this example, I chose the ad hoc report writer example because it uses just about all of the RDO objects that you probably have used. The project also takes advantage of the asynchronous communication opportunities available in RDO as well as the event-driven model presented in RDO.

You start, as you might guess by my timid nature and fear of unemployment, by backing up all of the work. You then save all the files to new names and begin the surgery in earnest.

First, go to VB's Project menu and change the reference to the Microsoft Remote Data

Objects library to Microsoft Active Data Objects 2.0. Next, you examine variable declarations. The application declares five module-level RDO objects:

```
Private WithEvents reng As rdoEngine
Private WithEvents renv As rdoEnvironment
Private WithEvents rcon As rdoConnection
Private WithEvents rrs As rdoResultset
Private rq As rdoQuery
```

As in DAO, there are no ADO equivalents to the **rdoEngine** or **rdoEnvironment** objects. The properties and methods of these objects have been largely migrated to the **Connection** object. Do a global replace of **reng** and **renv** with **acon**. Do the same for the **rcon** object variable. Delete the **rdoEngine** and **rdoEnvironment** declarations. Change **rdoConnection** to **Connection**, and change **rdoResultSet** to **Recordset**. Globally replace **rq** with **acmd**, and change **rdoQuery** to **Command**. Your module-level declarations should now look like the following example:

```
Private WithEvents acon As Connection
Private WithEvents ars As Recordset
Private acmd As Command
```

Do a global replacement of all occurrences of **RDO.** with **ADODB.**. This is to adjust those events that declare RDO objects as arguments. Finally, do a global replace of **rdoError** with **Error**.

A scan through the code of this application will show that it makes extensive use of RDO collections. Many times, there are no ADO equivalents because **Connection** and **Recordset** objects tend to be standalone. You will adjust for this as you go through the code. In the **cmdChoice_Click** event, you see these lines of code to connect to the database:

```
Set acon = rdoEnvironments(0)
acon.CursorDriver = rdUseClientBatch
Set acon = acon.OpenConnection(dsname:="", _
  Prompt:=rdDriverNoPrompt, _
  Connect:="DSN=Coriolis VB Example;UID=Coriolis;" & _
  "PWD=Coriolis;",Options:=rdAsyncEnable)
acon.QueryTimeout = Val(Text1)
Call WaitConn
```

Begin by changing the **acon** assignment to **New Connection**. The next line of code is intended to use a local cursor library. The property name is now **CursorLocation** and the value is **adUseClient**. You need to modify the connection by invoking the **Open** method of **acon** and using the now-familiar connection string. Alternatively, you could set the **ConnectionString** property of the object or simply pass the DSN, user ID, and password as separate arguments to the **Open** method, as shown in the following example. Notice that you also specify that the open should occur asynchronously. The **QueryTimeout** property is replaced by the **CommandTimeout** property. The revised code now looks like the following:

```
Set acon = New Connection
```

```
acon.CursorLocation = adUseClient
acon.Open "Coriolis VB Example", "Coriolis", "Coriolis", _
  adConnectAsync
acon.CommandTimeout = Val(Text1)
Call WaitConn
```

The call to the **WaitConn** event is meant to flash a message on the form while still connecting. This seems reasonable. The gut of the code in the event is the following line:

```
While acon.StillConnecting
```

The **StillConnecting** property is replaced by the **State** property. Replace the one line of code with the following:

```
While acon.State = adStateConnecting
```

Using the **While** construct, of course, is decidedly non-asynchronous. An asynchronous approach would be to delete the code and move the **BuildTableList** call to the **acon_ConnectComplete** event.

The next line of code is to the **BuildTableList** procedure, which creates a query statement:

```
sQuery = "Select distinct creator || '.' || tname " & _
  from sys.syscolumns "
```

The query had been constructed that way to illustrate the **DISTINCT** keyword. A more direct approach is to query the **SYS.SYSTABLE** table directly:

```
select user_name || ' ' || table_name
from sys.systable, sys.sysuserperms
where user_id = creator
order by user_name, table_name
```

Unfortunately, the specific query—in this case, to retrieve table names and the names of their creators—varies highly from database to database. A better answer is to take advantage of ADO's **OpenSchema** method.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The code passes the query to a procedure that actually creates the **Recordset**. What you do instead is replace the code and create the **Recordset** right in the **BuildTableList** procedure. The entire code block that you seek to replace follows:

```
Dim sQuery As String

If Check1(0).Value = vbChecked Then
  sWhere = "'coriolis'"
End If
If Check1(1).Value = vbChecked Then
  If Len(sWhere) > 0 Then
    sWhere = sWhere + ","
  End If
  sWhere = sWhere & "'DBO'"
End If
If Check1(2).Value = vbChecked Then
  If Len(sWhere) > 0 Then
    sWhere = sWhere + ","
  End If
  sWhere = sWhere & "'SYS'"
End If
If Len(sWhere) > 0 Then
  If Check1(3).Value = vbChecked Then
    sWhere = ""
  Else
    sWhere = " Where creator in (" & sWhere & ") "
  End If
Else
  MsgBox "You must select a table owner!"
  Exit Sub
End If

' Build the query
sQuery = "Select distinct creator || '.' || tname " & _
```

```
  from sys.syscolumns "

If Len(sWhere) > 0 Then
  sQuery = sQuery & sWhere
End If
Call SetResultSet(sQuery)
```

A good chunk of the code concerns building the **WHERE** clause to determine which **CREATOR** to look for. In ADO, the creator is known as the **TABLE_SCHEMA**. You can use those criteria when you use the **OpenSchema** method. You change the **sWhere** variable to an array of four elements (0 to 3) and move the lines of code to clear the listboxes in the application to this procedure. This gives you the ability to specify different schemas and iteratively add the tables to the listboxes. The owner names are still hard-coded at this point. You can refine that in Chapter 9 by using the **adSchemaSchemata** schema, which will return a list of all schemas. The refined code now looks like the following:

```
For iCtr = 0 To 1
  lbTables(iCtr).Clear
Next
sWhere(0) = "Coriolis"
sWhere(1) = "DBO"
sWhere(2) = "SYS"

If Check1(3).Value = vbChecked Then
  Set ars = New Recordset
  Set ars = acon.OpenSchema(adSchemaTables, _
    Array(Empty, Empty, Empty, "TABLE"))
  ' Show results
  Call ShowTables
Else
  For iCtr = 0 To 2
    If Check1(iCtr).Value = vbChecked Then
      Set ars = New Recordset
      Set ars = acon.OpenSchema(adSchemaTables, _
        Array(Empty, sWhere(iCtr), Empty, "TABLE"))
      ' Show results
      Call ShowTables
    End If
  Next
End If
```

Alter the values of the **Where** array to reflect your own database environment or simply pass an argument of **Empty** in the **OpenSchema** method to retrieve all schemas.

The **ShowTables** procedure remains the same except for deleting the listbox **Clear** method mentioned previously and adding a modification to the **AddItem** method. The code currently looks like the following:

```
lbTables(0).AddItem ars.Columns(0).Value
```

Recall that the old query returned a concatenation of the **Creator** column, a period (“.”), and the **Table_Name** column. You will simulate that with the following modified code, replacing the **Columns** reference with **Fields** and using elements 1 and 2 (where the schemas and table names are located):

```
lbTables(0).AddItem ars.Fields(1).Value & "." & _
  ars.Fields(2).Value
```

At this point, the application is capable of listing all the tables on the database. The user sees a screen that looks like Figure 8.2.

Figure 8.2 The partially converted ad hoc report writer.

Next, you follow the code when the user clicks on a table name. The code here that you need to replace is the following:

```
' Build query
sQuery = "Select cname from sys.syscolumns" & _
  " Where creator = '" & sCreator & "' " & _
  " and tname = '" & sTable & "' " & _
  " order by colno"

' Build the result set
Call SetResultSet(sQuery)

' Populate listbox and combo
Do Until ars.EOF
  lbCols(Index).AddItem ars!cname
  cbCols(Index).AddItem ars!cname
  ars.MoveNext
Loop
```

The RDO version of the code generated a query to find all the columns in the table that the user clicked on. Again, you will use the **OpenSchema** method to generate a more generic solution that will work with any database. Using **OpenSchema** turns out to make the code much simpler, as shown in the next example:

```
Set ars = New Recordset
Set ars = acon.OpenSchema(adSchemaColumns, Array(Empty, _
  sCreator, sTable, Empty))

' Populate listbox and combo
Do Until ars.EOF
  lbCols(Index).AddItem ars!column_name
  cbCols(Index).AddItem ars!column_name
  ars.MoveNext
Loop
```

The query was changed to the **OpenSchema** method using **sCreator** as an argument for **TABLE_SCHEMA** and **sTable** for **TABLE_NAME**. In the code where you add to the listboxes, you use the more generic **ars!column_name** instead of referencing the **Fields** collection, as in the **BuildTableList** procedure earlier.

You are getting near the end of what needs to be converted. The **BuildQuery** procedure is fine as is; it merely builds up the SQL **SELECT** statement. When the user clicks the Run Query button, the **SetResultSet** procedure is called. The code follows:

```
If acon.RecordSets.Count > 0 Then
  ars.Close
End If

Set ars = acon.OpenResultset(sQuery, _
  rdOpenDynamic, rdConcurReadOnly, rdAsyncEnable)
Call WaitQuery

You replace the code with these simplified lines:

Set ars = New Recordset
ars.Open sQuery, acon, adOpenKeyset, adLockPessimistic, adCmdText
Call WaitQuery
```

In the body of the **WaitQuery** procedure, you replace the text of the **StillExecuting** property with the **State** property:

```
While ars.State = adStateExecuting
```

After the **Recordset** has been built, the application populates the grid control with the results. You should replace the numerous references to **RowCount** in the procedure with **RecordCount**. The references to **rdoColumns.Count** should be replaced by **Fields.Count**. Alas, you run into a stumbling block when you address the use of the **GetClipString** method of the RDO **ResultSet**. There is no ADO equivalent. The code from the RDO version of the application looked like the following:

```
' Display data
With rs
  sClip = .GetClipString(.RowCount)
  fgResults.Row = 1
  fgResults.Col = 0
  fgResults.RowSel = fgResults.Rows - 1
  fgResults.ColSel = fgResults.Cols - 1
  fgResults.Clip = sClip
  fgResults.Row = 1
  fgResults.Col = 1
End With
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The RDO solution was fairly efficient. A better solution is to bind the data source directly to the grid control, but that is going a little bit out of the way for this example. You can replace the code with a straightforward, if brute-force, solution, as shown next:

```
With ars

  Do Until .EOF
    fgResults.Row = fgResults.Row + 1

    For iCol = 0 To .Fields.Count - 1
      fgResults.Col = iCol
      fgResults.Text = .Fields(iCol).Value
    Next
    .MoveNext
  Loop
End With
```

The CD-ROM contains the code that I used to populate the grid control for the DAO version of the Ad Hoc Report Writer project. It takes a different tack in populating it, which is too lengthy to list here. However, although it is considerably longer, it is actually more efficient.

The application is now fully converted, and the finished product is shown running in Figure 8.3. The query that the application built is a self-join of the **Employee** table listing all employees along with their managers.

Figure 8.3 The RDO Ad Hoc Report Writer converted to work under ADO.

Summary

The RDO example that I used to illustrate the conversion to ADO process was extensive. Between it and the DAO examples, I have covered most of the key methods and properties of ADO objects. Although there was no magic bullet, neither were there deep mysteries. The process is straightforward, if a bit tedious. Still, most developers will find the effort worthwhile.

Where To Go From Here

Assuming you have read Chapter 7, where I introduced the basics of ADO, my recommendation is to take one of your DAO or RDO applications and practice converting it. Run it in debug mode, stepping through the code and using the Immediate window to explore the values of different objects. Take some time also to explore the Microsoft Web site for any developments since this book was written (www.microsoft.com/data/). Then, dive into Chapter 9, where I expand upon the use of ADO, and Chapter 10, where I start introducing advanced topics, such as business objects. If you are not yet comfortable with SQL or your database, review Chapters 2 and 3 and skip ahead to Chapter 11, where I discuss some advanced database issues.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Chapter 9
Advanced ADO Client/Server Techniques

Key Topics:

- Using the **DataEnvironment** object
- Using the **DataReport** object
- Parameterized, hierarchical, grouped, and aggregated **Recordset** objects.

Microsoft has introduced a number of new objects along with ADO, including the **DataEnvironment**, which is introduced briefly in Chapter 4, and the **DataReport**. To support these tools, new designers and viewers have been developed as well.

In this chapter, you'll learn about the new tools that can help you to maximize the productiveness of your client/server development, as well as functionality and performance. The tools that I show you in this chapter can add power to your running application as well as provide an unprecedented leap in development productivity. In addition to reviewing the new tools, you'll also learn about some new, advanced data handling and navigation techniques.

The concepts and tools presented in this chapter are all very new and will evolve over time. I urge you to experiment with the technology presented here and to create bookmarks for the Microsoft Visual Basic home page (**msdn.microsoft.com/vbasic**) and the Microsoft Universal Data Access Web page (**www.microsoft.com/data**). Check the sites often—you'll frequently find new articles and, most likely, new releases of ADO.

Bugs, Bugs, Bugs

I have been programming in the PC world for 20 years now, and I am not ashamed to admit that a lot of my background is on the mainframe. We "old-iron" types have a saying: Before PCs had mice, we had bugs. But alas, mainframes are not unique in that respect.

I encountered a number of bugs as I attempted to push Visual Basic's new tools to various extremes. In this chapter, I share what bugs I found as well as some workarounds. I anticipate that there will be other bugs that I haven't run into yet. Therefore, you should monitor the Visual Basic Web site for any patches, workarounds, and so on for at least the first six to nine months after Visual Basic 6's general release.

The Data View Window

The first item that I want to introduce is the Data View window. To access the Data View window, select it under the View menu (or from the Standard toolbar). All currently defined data links should display as shown in Figure 9.1.

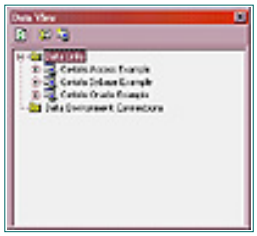

Figure 9.1 The Data View window.

You can right-click the Data Links folder to create a new Data Link or right-click the Data Environment Connections folder to create a new **Connection** object. You can expand any of the existing Data Links to see a list of tables, views, and stored procedures. You can expand each of these and see their properties.

After you have created a new **DataEnvironment** object, you can then drag the object from the Data View window onto the DataEnvironment Designer window to create new **Connection** objects. The **DataEnvironment** object and DataEnvironment Designer are new to Visual Basic 6. Combined, the **DataEnvironment** components provide a new weapon in the arsenal of tools for the developer to attack high-powered client/server applications. In this section I will introduce you to these tools and their usage.

The DataEnvironment Designer

The DataEnvironment Designer is roughly analogous to the User-Connection Designer in Visual Basic 5, which created RDO objects. The DataEnvironment Designer encompasses the functionality of the UserConnection Designer, plus new, ADO-related functionality. For instance, you can create **Connection** objects that access multiple data sources simultaneously. You can drag fields from the designer onto your VB form, and appropriate data bound controls will

be automatically created (!).

To use the DataEnvironment Designer, you must add a reference to it:

1. From the Project|References menu, select Microsoft Data Environment 1.0.
2. Then, from the Project menu, choose Add Data Environment.

Alternatively, you can press the Add Data Environment button on the Data View window's toolbar. The DataEnvironment Designer window will open, and a reference will be added to the Project Explorer window, as shown in Figure 9.2.

Figure 9.2 The DataEnvironment Designer window.

Before you get too far in using the designer, you should have an idea of what it is you need to accomplish.

Using The DataEnvironment Designer

In this section, you'll learn how to use the DataEnvironment Designer by working through example scenarios. I'll walk you through two examples using the sample database. First, you'll look at a report using Microsoft's new **DataReport** object. Then, I will walk you through the creation of a hierarchical **Recordset**. For now, let's take a look at the DataEnvironment Designer. Follow along as I give you the "whirlwind" tour:

1. If you have not already done so, open the DataEnvironment Designer by selecting "More ActiveX Designers" from the Project menu and then selecting the DataEnvironment Designer.
2. Right-click on the **DataEnvironment** object, and select Options. The Options dialog has two tabs—General and Field Mapping.
3. On the General tab, make sure that the Show Properties Immediately After Object Creation and Show System Objects options are both checked.
4. Click on the Field Mapping tab. Here, you can tell Visual Basic how to map data types to bound controls on forms, as shown in Figure 9.3.

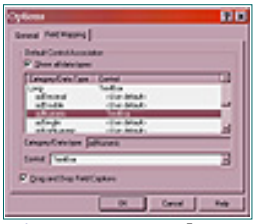

Figure 9.3 The Field Mapping tab of the Options dialog box.

As you can see in Figure 9.3, the dialog box organizes the database data types into categories, such as **Long**. Under each category are specific data types, such as **adDouble** and **adNumeric**. For each category, you

can specify what type of bound control Visual Basic should create when you drag a field onto a form. You can then specify that each data type uses the default type of control, or you can override the default and specify a different type of control. In the Control drop-down list box, Visual Basic lists every ActiveX object registered on your PC, although some are obviously inappropriate choices to use to display data.

5. Close the dialog box after making any changes that seem appropriate. You can always change them again later.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

I am hoping that Microsoft will some day also allow you to specify a default **DataFormat** property for bound controls in the Field Mapping dialog. (The **DataFormat** property, of course, would make it easier to quickly control how data is displayed as opposed to having to change each field individually.)

When you open the DataEnvironment Designer, Visual Basic automatically creates a **DataEnvironment** object and a **Connection** object, I show you how to use these in the next two sections.

The DataEnvironment Object

When you add the DataEnvironment Designer to your project, a **DataEnvironment** object is automatically created along with a **Connection** object named **Connect1**. The **DataEnvironment** object has only two properties: the **Name** property and the **Public** property. The latter is a Boolean where **True** indicates that the object can be shared by other applications.

The Connection Object

The **Connection** object is a “standard” ADO object. I discussed its properties and methods in Chapter 7.

The easiest way to create a new **Connection** object is to drag a table, view, or stored procedure from the Data View window to the DataEnvironment Designer window. Alternatively, you can right-click the **DataEnvironment** object, and select Add Connection. When you do so, the Data Link Properties dialog box displays. There are four tabs: Provider, Connection, Advanced, and All.

- *Provider*—Use to select the OLE DB provider that you will be using.
- *Connection*—Use to select the database, enter the user ID and password, and so on, as appropriate for the OLE DB provider.
- *Advanced*—Use to enter the timeout parameter, locking level, and so

on.

• *All*—Use to view a summary of the **Connection** object's settings, as shown in Figure 9.4.

Figure 9.4 The All tab of the Data Link Properties dialog box.

You can add as many **Connection** objects as needed. To each, you can add multiple **Command** objects. I discuss the **Command** object next.

NOTE
When you create a **Connection** object by dragging from the Data View window, a **Command** object is created automatically.

The Command Object

Like the **Connection** object, the **Command** object is a standard ADO object. I discussed its use in Chapter 7. The following shows you how to add a **Command** object to a **Connection** object and customize it in the DataEnvironment Designer:

1. To add a **Command** object to a **Connection** object, right-click on the **Connection**, and select Add New Command.
Alternatively, you can drag a table, view, or stored procedure from the Data View window onto the DataEnvironment Designer window.

2. Right-click on the new **Command** object to set its properties. Again, you are presented with a tabbed dialog box:

• *General*—On the General tab, you can rename the object or choose a different **Connection** to which this object belongs. Under Source Of Data, you can set the source of the data to be either an existing database object or an SQL Statement.
If you select SQL Statement, you can type in the statement or press the SQL Builder button. Doing the latter allows you to build the query graphically. The SQL Builder is shown in Figure 9.5.

Figure 9.5 The SQL Builder dialog box allows you to drag and drop to build your queries.

Besides an SQL Select, you can also specify a database object as

being the data source of a **Command** object. If you choose to use a database object, you can select Stored Procedure, View, Table, or Synonym. In each case, the Object Name drop-down list box lists the available objects.

- *Parameters*—The Parameters tab of the Connection Properties dialog box shows all the parameters associated with a stored procedure, as well as its data type and direction.

The parameters you see listed on the Parameters tab will generate **Parameter** objects at runtime. ADO normally identifies all the properties of **Parameter** objects automatically, but this dialog box provides a convenient method of verifying or overriding the property values. For more information on **Parameter** objects, see "The **Parameters** Collection And **Parameter** Object" in Chapter 8.

TIP
Adding Multiple Stored Procedures As Commands
You can create multiple **Command** objects from stored procedures by right-clicking on the **Connection** object and choosing Insert Stored Procedures. Visual Basic will then open the Insert Stored Procedures dialog box listing the stored procedures available in the current connection. You can select all or some of the procedures—Visual Basic will then create **Command** objects from them.

- *Relation*—The Relation tab allows you to associate **Command** objects with each other in a hierarchical relationship. I will discuss this later in the chapter in the section entitled "Hierarchical Cursors."
- *Grouping*—The Grouping tab allows you to specify one or more fields on which your output will be grouped, similar to the SQL **GROUP BY** clause.
- *Aggregates*—The Aggregates tab allows you to aggregate your data based on one or more of several function types, such as sum and standard deviation. If the data is grouped, subtotals are taken at the group level.
- *Advanced*—The Advanced tab has different options available depending on the data source. For instance, the Call Syntax tab will show you the syntax for calling stored procedures but is unavailable for SQL **SELECT**s. For SQL **SELECT**, you can set properties such as Cache Size and Lock Type, as shown in Figure 9.6. In the example, I have unchecked the All Records box, and entered a maximum number of records to return.

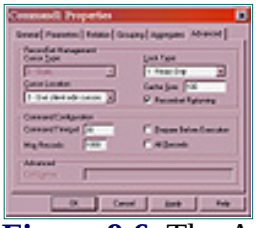

Figure 9.6 The Advanced tab of the Command object Properties dialog box allows you to refine various advanced settings.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Using The SQL Builder

I found the SQL Builder to be a little balky. (I am hoping that by the time VB6 ships, the SQL Builder's procedures will have been refined somewhat.) When you open the SQL Builder dialog box, you face four blank panes:

- The top pane represents a graphical look at the tables.
- The second pane, called the *Grid*, shows columns that you have selected and allows you to specify sort orders, and so on.
- The third pane is the text of the SQL statement.
- The last pane allows you to view the output of your query.

To add a table to SQL Builder, you can drag it from the Data View window. (For whatever reason, I had a hard time figuring that out—someone else had to point it out to me.)

To create a query, click next to each column that you want in your query. The columns appear in the second pane as you click each. To change the order of the columns (not the sort order within a column), drag each column as needed.

To join tables together, click on a column from one table and drag it to the column in the second table. VB draws a line denoting the relationship. By default, VB creates an inner join. Right-click on the join line to expose its properties. You can then change how the tables will be joined.

In the Grid pane, you can assign an alias to columns, if you so desire. If you click in the Sort column next to a database field, you can choose Ascending or Descending. The Sort Order allows you to refine nested **ORDER BY**s. You can type an expression into the Criteria box, such as **> 0**. There are three **Or** columns allowing you to string together several search criterion. There is no overt provision to **And** your search criterion. You can, however, create multiple conditions one below the other, which creates the **And** condition.

To see the results of your query, right-click in the dialog box, and click Run.

Using The DataEnvironment Object

The **DataEnvironment** object can be used much like a Data control by binding controls to it. You can also create it programmatically and manipulate its **Command** objects. In the design environment, the **DataEnvironment** object exposes two collections: **Commands** and **Connections**. At runtime, it also exposes the **Recordsets** collection. Confusingly, these collections are 1 based while other ADO collections are 0 based.

The **DataEnvironment** collections are created at runtime when the **DataEnvironment** object is instantiated. When a **Command** object is executed, it creates a **Recordset** object if it is marked as being record returning.

As I mentioned earlier, you can rapidly build your forms using drag and drop. Here's how:

1. Reposition the DataEnvironment Designer window and your form window so that you can see both.

2. Expand the **Command** object that you want to be the data source for the form, as I have done in Figure 9.7.

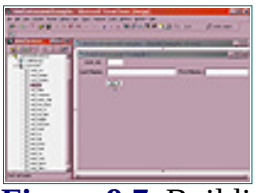

Figure 9.7 Building a form based on a **DataEnvironment** object using drag and drop.

> **NOTE**
> When you create a **Command** object with the DataEnvironment Designer, it is automatically identified as to whether it's record returning. You can override this setting from the Advanced tab of the Command Properties dialog box.

3. Set the Font property of the form to whatever font and size that you want your bound controls to inherit. (The Font property of the form is used to default the Font property of controls subsequently added to the form).

4. Then, simply drag fields from the design window to the form window, as shown in Figure 9.7.

You can drag the **Command** object itself onto the form, in which case Visual Basic will create all the data bound controls at once. You can drag with the right mouse button also. When you do so and drop the **Command** object on the form, you're given the options of creating data bound controls, a data grid, or a hierarchical flex data grid. It doesn't get much easier than this.

The label captions default to the field name; however, you can right-click on any field in the **Command** object within the designer window and override the

default caption. I did this for the last and first name fields in Figure 9.7. You can also override the default control. Unfortunately, Visual Basic does not provide a facility to rapidly change all the fields on one window. Maybe in the next release of VB?

Visual Basic names the controls that it creates for you based on the control and the field. As you can see in Figure 9.8, the name of the textbox for the customer last name is **txtCust_LName** (**Cust_LName** is the field name). I would prefer that VB created an array of textboxes to more easily iterate through them at runtime, but an alternative would be to iterate through the **Controls** collection of the form.

Figure 9.8 Properties of a control bound to the data source.

You can, of course, create the controls manually. You will need to set the **DataSource**, **DataMember**, and **DataField** properties of the control, as seen in Figure 9.9. **DataSource** is set to the name of the **DataEnvironment** object, **DataMember** to the name of the **Command** object within the **DataEnvironment**, and **DataField** to the name of the appropriate **Field** object within the **Command** object.

Figure 9.9 The **DataEnvironment**-driven Item Maintenance application.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Using the objects in code is pretty much the same as I explained in Chapter 7. You will be manipulating **Recordset** objects, **Field** objects, **Connection** objects, **Command** objects, and perhaps **Parameter** objects. The two things to keep in mind are:

- All of the objects will be properties of the **DataEnvironment** object.
- Visual Basic will create the **Recordset** objects for you and name them by adding the letters *rs* in front of the name of the **Command** object from which the **Recordset** objects were created. For example, if your **Command** object is named **Orders**, the resulting **Recordset** will be named **rsOrders**.

TIP
Using Stored Procedures As Data Sources
To bind a control to a stored procedure as a data source, you must make sure that it is marked as Recordset Returning on the Advanced tab of the **Command** object's Properties dialog box. Because the **Command** object will be executed as soon as the form opens, you must also enter a value for each of the input parameters on the Parameters tab.

As we go through the next several sections, I'll provide a number of examples of using the objects in code.

Creating The Basic Data Maintenance Form From The DataEnvironment Object

In this section, we will program a form to maintain the Item table on the database. The first step is to create the appropriate **Connection** and **Command** objects. (If you have not already created a **DataEnvironment** object, you should do that first.)

1. Create a **Connection** object as discussed earlier. I created one to connect to a Sybase database and called it **conExampleSybase**.
2. Next, create a **Command** object whose data source is a table. Select the Item table, and name your **Command** object **Item**.
3. Add a form to your project, right-click on the **Item Command**, and drag it to the form. Select Bound Controls from the context menu.

4. Now, you will need to add navigation buttons.

Your navigation buttons should be pretty much like navigation buttons in any ADO application. I used an array of Image controls, as seen in Figure 9.9. I also added an array of command buttons and a textbox to display the current status. The complete code is presented as Listing 9.1. Although it is bare bones, it performs most functions required of a data form. (You would want to add some code validation and so on.) Note that the only real difference between this application and more traditional ADO projects is that the **Recordset** object is referenced as a property of the **DataEnvironment** object.

Listing 9.1 The code for the Item Maintenance form.

```
Private Sub cmdAction_Click(Index As Integer)
Dim vBMark As Variant
Dim lRec As Long
Dim lCount As Long
vBMark = deItemTest.rsItem.Bookmark
lRec = deItemTest.rsItem.AbsolutePosition
lCount = deItemTest.rsItem.RecordCount
Select Case Index
  Case 0
    deItemTest.rsItem.AddNew
    Text1 = "New Record"
  Case 1
    deItemTest.rsItem.Delete
    If lCount > 1 Then
      Image1_Click (0)
    Else
      cmdAction_Click (0)
    End If
  Case 2
    deItemTest.rsItem.Update
    Image1_Click (3)
  Case 3
    deItemTest.rsItem.CancelUpdate
    deItemTest.rsItem.Bookmark = vBMark
    dispRec
  Case 4
    End
  End Select
End Sub

Private Sub Form_Load()
dispRec
End Sub

Private Sub Image1_Click(Index As Integer)
Select Case Index
  Case 0
    deItemTest.rsItem.MoveFirst
  Case 1
    If deItemTest.rsItem.AbsolutePosition > 1 Then
```

```
      deItemTest.rsItem.MovePrevious
    End If
  Case 2
    If deItemTest.rsItem.AbsolutePosition < _
      deItemTest.rsItem.RecordCount Then
      deItemTest.rsItem.MoveNext
    End If
  Case 3
    deItemTest.rsItem.MoveLast
End Select
dispRec
End Sub
Private Sub dispRec()
Text1 = "Record " & deItemTest.rsItem.AbsolutePosition & _
  " of " & _
  deItemTest.rsItem.RecordCount
End Sub
```

Using Stored Procedures And Parameters With DataEnvironment Objects

Figure 9.10 shows an application included on this book's CD-ROM that uses stored procedures as data sources. As in the stored procedure examples shown in Chapter 7, you have a lot of flexibility in how you code the procedures.

Figure 9.10 The Stored Procedure demo program.

The top half of the form shown in Figure 9.10 is the same Employee Count stored procedure used in Chapter 8. The code to execute the procedure and display the results is shown next:

```
deStoredProcedure.Commands("coriolis_NoEmps").Execute
txtNoEmps = deStoredProcedure.Commands _
  ("coriolis_NoEmps").Parameters(0)
```

Because the procedure has no input parameters, it is not necessary to set any parameter values. The first line of code executes the **Command** object **Coriolis_NoEmps** that has one output parameter. The second line of code then accesses the value of that **Parameter** object and displays its value in a textbox.

The second stored procedure has two input parameters and one output. The code to execute it is as follows:

```
deStoredProcedure.Commands("coriolis_raise_no").Parameters(0) _
  = val(txtRaise(0))
deStoredProcedure.Commands("coriolis_raise_no").Parameters(1) _
  = val(txtRaise(1))
deStoredProcedure.Commands("coriolis_raise_no").Execute
```

```
txtRaise(2) = deStoredProcedure.Commands _
  ("coriolis_raise_no").Parameters(2)
```

In this example, the first two lines of code set the two input parameters equal to the value of whatever is entered into the first two textbox controls. The third line of code executes the **Command** object, and the fourth line displays the result.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The DataEnvironment Event-Driven Model

You do not lose the flexibility of event-driven development or asynchronous techniques when you use the **DataEnvironment** object. The **DataEnvironment** object exposes two events: **Initialize** and **Terminate**. These two events are much the same as the class module's **Initialize** and **Terminate** events and provide you with an opportunity to perform initialization and clean-up chores.

TIP
DataEnvironment Objects And Reusability
You will place processing such as data validation into the **DataEnvironment** module and not into any form (and so on) module that accesses it. If you code the **InfoMessage** event of a **Connect** object, for instance, any form that uses the **DataEnvironment** object as its data source inherits the code for the **InfoMessage** event. This provides considerable freedom to reuse the **DataEnvironment** object from form to form or even from project to project, thereby maximizing code reuse and reliability. Neat!

On this book's CD-ROM, I have included a project called CustOrdDetHierarchy. In it is a **DataEnvironment** object—**deHierarchy**—that I use in examples in most of the remainder of the book.

deHierarchy has one **Connection** object: **cmdCustOrdDet** (so named because the examples access the **Customer**, **Orders**, **Line_Item**, and **Item** tables). The module for the **deHierarchy** object exposes all the ADO **Connect** object events, such as **WillConnect** and **ConnectComplete** (the complete ADO event model is discussed in Chapter 7).

The **DataEnvironment** object takes all the **Command** objects that you create and internally performs a **Set Recordset = command.Execute** method. The names of the **Recordset** objects created are patterned after the **Command**

object's name by prefixing it with an *rs*. A **Command** object named **custsByState** generates a **Recordset** named **rsCustsByState**.

Each of these **Recordset** objects have their event models fully exposed as well. If you double-click **deHierarchy** in the Project Explorer, its code window opens. All **Recordset** objects that will be generated are listed in the object drop-down box without your needing to make a reference to them. You use the events just as you would in a form module that creates an ADO **Recordset**.

To access a **Recordset** from your form, you need to declare a reference to it. If you do so, you then have access to its events at the form level as well (although I am not convinced this is a good idea in most cases). Add a declaration at the module level to create a **Recordset** object:

```
Private With Events rs As Recordset
```

In the form's **Open** event, make an explicit reference to the **Recordset** created by the **DataEnvironment** object:

```
Set rs = deHierarchy.rsCustsByState
```

In the next section, I use the returned reference to navigate a **Recordset**.

Handling Recordset-Returning Stored Procedures

Stored procedures that return **Recordset** objects are shown in the DataEnvironment Designer window with a plus sign next to them. If you expand the stored procedure (by clicking on the plus sign), you can see the fields returned by the stored procedure. The Properties dialog for the **Command** object containing the stored procedure has a checkbox on the Advanced tab indicating that the stored procedure returns a **Recordset**.

Figure 9.11 shows a form that I created to illustrate handling stored procedures that return a **Recordset** object. The form, **frmSPRS**, is part of the same application, **deSPDemo** (included on this book's CD-ROM), in which **frmSPDemo** from Figure 9.11 is included. Both forms will open if you run the application.

Figure 9.11 This form demonstrates how to code applications that use **Recordset**-returning stored procedures.

I used a "canned" stored procedure from the database to create the form. I right-clicked on the stored procedure in the DataEnvironment Designer window and dropped it onto the form. At the prompt, I selected DataGrid. I dragged the stored procedure again and chose Data Bound Controls. I borrowed the navigation buttons from the **frmSPDemo** form.

After declaring a variable, **rs**, to be of type **Recordset** at the module level, I coded the following line into the form's **Load** event:

```
Set rs = deStoredProcedure.rsdbo_sp_tables
```

This line of code does a few things for me:

- By cutting down on the number of "dots" whenever I need to reference the **Recordset**, I actually make the code more efficient. I discuss this a little more in Chapter 11 but, for now, understand that VB only has to resolve **deStoredProcedure.rsdbo_sp_tables** once instead of many times.
- I now gain access to the events of the **rsdbo_sp_tables Recordset** object as discussed in the prior section.
- I save myself a *ton* of typing!

As you can see in the project on the CD-ROM, the code is pretty much the same as for the **frmSPDemo** form example except that references are shortened up, for example,

```
rs.MovePrevious
```

instead of:

```
deStoredProcedure.rsdbo_sp_tables.MovePrevious
```

The data source of both the grid control and the textboxes is the same, so any references to a method or property of **rs** affects both.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Previous | Table of Contents | Next

Using The DataReport Object

The **DataReport** object is a welcome addition to Visual Basic. The Crystal Reports add-on has always been a good tool (and continues to be). The **DataReport** provides some event-driven asynchronous control over your reporting.

The **DataReport** can be bound to any data source, but it works most naturally with the **DataEnvironment** object. You can drag and drop fields from a **Command** onto the **DataReport** or drag the entire **Command** object. In my testing, I found the DataReport Designer to be somewhat finicky and, sometimes, not very intuitive. However, it seems that it will be a great addition to the VB toolkit.

The following is a summary of some of the highlights of the **DataReport** and the DataReport Designer. An in-depth coverage of either is beyond the scope of this book. (Besides, you probably have already used a half-dozen or so report writers in your career.) For more information, read “Writing Reports With The Microsoft Data Report Designer” in the Visual Basic Help file, if you so desire.

To use the **DataReport** object, select Add Data Report from the Project menu. When you do so, Visual Basic adds a library reference. The first time you use it, I recommend pulling up the Object Browser, selecting **MSDataReportLib** in the Library list box, and exploring the objects, methods, properties, and events of the library.

The DataReport Designer is broken into different *sections* (what are often called *bands* in other report writers). By default, Section 1 is the Detail section. It is interesting to note that each section is actually an object (of type **Section**). Each **Section** object is part of the **Sections** collection. The implication, of course, is that the **DataReport** is highly programmable at runtime.

Each **Section** object has several properties. The key properties are **KeepTogether** (**True** means that if a section does not fit on a page, a new page should be started) and **ForcePageBreak**. For the latter property, you can specify that a page break always occurs before the section, after the section, before and after the section, or that no page

break occurs. The **Section** object also has a **Controls** property made up of all of the controls placed on that section.

The **DataReport** has its own toolbox with its own version of design-time controls. RptTextBox, RptShape, RptLine, and RptImage are all self-explanatory. A fifth control is the RptFunction control. You select this and drop it on an appropriate section. Then, you need to set two properties: the **DataField** property and the **FunctionType** property. Among choices for **FunctionType** are **rptFuncAve**, **rptFuncSum**, and so on.

Of more interest are the properties, methods, and events of the **DataReport** object itself as well as how to use the object in your program. By using these properties and methods, you can exercise a lot of flexibility over your report, including the dynamic alteration of its layout.

After you create a report, you can show it in much the same way as you show any form. I created a standard module (included on this book's CD-ROM) that displays a single report modally from the **Main Sub** procedure and then ends the program:

```
Sub main()
drCustomersOrders.Show vbModal
End
End Sub
```

Other methods that you can use programmatically include **Refresh** (to regenerate the report), **PrintReport**, and **ExportReport**. The syntax for the **PrintReport** method is shown as follows:

```
datareport.PrintReport(ShowDialog, Range, PageFrom, PageTo)
```

ShowDialog is a Boolean where **True** causes the Print dialog to be displayed. Setting **Range** to **rdPrintAllPages** causes the entire report to print. **rdRangeFromTo** specifies that only part of the report will be printed. You can then specify the **PageFrom** and **PageTo** arguments. Omitting **PageFrom** or **PageTo** causes the Print dialog to open. The report is printed asynchronously.

The **ExportReport** method allows you to save the report as a file. The syntax is shown in the following code:

```
datareport.ExportReport(index, filename, _
  Overwrite, ShowDialog, Range, PageFrom, PageTo)
```

The **ShowDialog**, **Range**, **PageFrom**, and **PageTo** arguments work the same as for the **PrintReport** method. **Overwrite** is set to **True** to cause files to be overwritten or **False** to prevent files from being overwritten. **filename** is the path and file name to save the report to. If not supplied, the Export dialog is opened. **index** specifies the type of file to generate. You can specify the name of an **ExportFormat** object or an export key (**rptKeyHTML**, **rptKeyText**, **rptKeyUnicodeHTML_UTF8**, or **rptKeyUnicodeTest**).

Visual Basic provides a lot of flexibility in data report exports. The subject is outside the scope of the book, but you can refer to "ExportReport Method" in the VB Help file for more information.

The **DataReport** object supports a number of events. **AsyncProgress** allows you to monitor the progress of the report. The **Processing TimeOut** event is triggered at defined intervals (about one second each) and allows you or the user to cancel a report in progress. The **Error** event is triggered when an error is encountered during an asynchronous event.

The following code example is a bare-bones error-handling routine:

```
Private Sub DataReport_Error _
  (ByVal JobType As MSDataReportLib.AsyncTypeConstants, _
   ByVal Cookie As Long, ByVal ErrObj As _
   MSDataReportLib.RptError, ShowError As Boolean)

Select Case JobType
  Case rptAsyncExport
    MsgBox "Error Exporting Report"
  Case rptAsyncPreview
    If ErrObj = rptErrOutOfMemory Then
      MsgBox "Out of Memory!"
      Unload Me
    End If
  Case rptAsyncPrint
    MsgBox "Error Printing Report"
End Select
ShowError = True
End Sub
```

The next sample code demonstrates the handling of the **Processing TimeOut** event:

```
Private Sub DataReport_ProcessingTimeout _
  (ByVal Seconds As Long, Cancel As Boolean, _
   ByVal JobType As MSDataReportLib.AsyncTypeConstants, _
   ByVal Cookie As Long)

If MsgBox("Report Timed Out. Continue?", _
  vbYesNo + vbQuestion) = vbNo Then
  Cancel = True
Else
  Cancel = False
End If
End Sub
```

You can also programmatically access or set various properties to control the appearance or operation of the **DataReport** object. For instance, the **ExportFormats** property is a reference to the **ExportFormats** collection. **DataMember** and **DataSource** allow you to specify the data source of the report.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Command And Recordset Hierarchies

The power of the relational database has always been to present data from multiple tables and, in the process, relate the data from one table to another. Visual Basic 6 and, in particular, OLE DB, provide an entirely new way of relating rows or records from one table or file to another table or file—the hierarchical **Recordset**. Not only is this method somewhat more intuitive, it is more efficient.

For purposes of this discussion, I will refer to tables. From an OLE DB standpoint, however, the concept of hierarchical **Recordset** applies just as well to any file or data source for which there is an OLE DB data provider.

The hierarchical **Recordset** views tables in parent-child relationships. This is a comfortable enough analogy—we looked at relationships that way when designing the database back in Chapter 2. In the real world, we can't have children without first having parents. In parent-child, hierarchical relationships, records on a child table depend on records on a parent table. Because an order cannot exist without a customer, the **Customer** table is parent to the **Orders** table.

In the **DataEnvironment** object, we can create command hierarchies to represent hierarchical relationships. This is implemented as a **Shape** (which I discuss later in this section). The DataEnvironment Designer also helps you to build grouping and aggregate hierarchies. Visual Basic 6 includes the Hierarchical Flexgrid control to display **Recordset** hierarchies in an intuitive manner.

When you create a hierarchical **Recordset**, child **Recordset** objects are represented as **Field** objects on the parent **Recordset**. In other words, if you have a parent **Recordset** representing the **Customer** table and a child **Recordset** representing the **Orders** table, on each record in the parent

Recordset, there is a **Field** object containing a **Recordset** which, in turn, contains all orders related to that customer.

The easiest way to display and understand a hierarchical **Recordset** is to use it as a data source to a Hierarchical Flexgrid control, as shown in Figure 9.12. The control displays each customer on the **Customer** table. If the customer has any orders, a plus or minus symbol is displayed next to the customer. Click the plus symbol to expand the display to show all orders. Click the minus symbol to not show the order information. For each order, line items are also displayed with the same plus and minus symbols. Again, click the plus sign to display the detail line item information or the minus sign to not display the information.

Figure 9.12 A Hierarchical Flexgrid control displaying a hierarchical **Recordset** hat contains the **Customer**, **Orders**, and **Line_Item** tables.

To create the hierarchical **Recordset**:

1. Add a **Command** object to in the DataEnvironment Designer.
2. After giving it an appropriate name, such as **cmdCustomer**, set its data source to be a table object, and select the **Customer** table.
3. Next, right-click on **cmdCustomer**, and select Add Child Command. Give it a name, such as **cmdOrders**, and specify its data source to be Table. Select Orders for the name of the table.
4. Now, click on the Relation tab, and click the Relate To A Parent Command checkbox.
5. In the Parent Command drop-down list, select **cmdCustomer**.
6. In the Related Definition field, select the two fields on which these commands are related, in this case, **Cust_No** and **Ord_Cust_No**.
7. Click the Add button, and then click OK. The DataEnvironment Designer window will now show the new **Command** (**cmdOrders**) as being a **Field** object underneath **cmdCustomer**.
8. Add a child **Command** to **cmdOrders** using the same technique relating the **Line_Item** table to **Orders** using **Ord_No** and **Line_Ord_No** as the related fields.
   You can relate two **Command** objects after they have already been created, even if you did not specify that one is the child of another.
9. Go to the Properties dialog of the object that is to be the child **Command**, select the Relation tab, and define the relationship as described in the preceding steps.
10. Now, drag the top level **Command** object (**cmdCustomer**) onto a form, and drop it with the right mouse button.
11. Select Hierarchical Flexgrid control. The grid is drawn on the form.
12. Right-click on the grid, and select Properties.
    One of the tabs, as seen in Figure 9.13, is named Bands. Each

Command object represents a band in the control. You can select what fields to display here as well as perform some limited formatting. Unfortunately, I was not able to find a way to format numbers or dates without doing it in code.

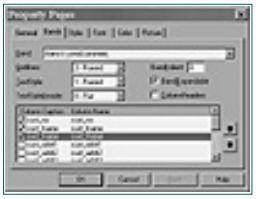

Figure 9.13 The Hierarchical Flexgrid control's Property Pages.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Grouped Command And Recordset Hierarchies

Command objects can be grouped in a manner similar to the way data can be grouped in SQL statements with the **GROUP BY** clause. When you create a **Command** group, a new object is created that is the parent of the object being grouped. In other words, if you group by one or more fields in a **Command**, a new **Command** is created consisting of those summary field(s) and is the parent in the hierarchical relationship. You can group a single **Command** or a **Command** that already has a hierarchical relationship.

Assume that you wanted to group your customers by state and city. You could do so by opening your **Customer** command and, in the Properties dialog box, selecting the Grouping tab. All the fields that comprise the command are listed. Select the one(s) that you want to group by. Note that if you group by state *and* city in the same object, then your groups are composed of each unique occurrence of state and city together. You do not get a state group and, within that, a city group.

When you create a Hierarchical Flexgrid control based on the group hierarchy, you get a band for each **Command** object just as you do with the hierarchical **Command**. Depending on how you build the hierarchy, the Flexgrid control can be fooled. To illustrate, let's use the **Command** hierarchy built earlier with the **Customer**, **Orders**, and **Line_Item** tables. Drag it on the form, create a Hierarchical Flexgrid control as we did earlier, and then test. All works fine. Now, edit the **cmdCustomer** object, and, on the Grouping tab, specify **Cust_St**. Run the form again. This time, only the first grouping is displayed on the Flexgrid control. Using the sample data, only orders placed from California are displayed and there is no way to see other data.

The reason for this is that the definition of the data source object has changed. What the grid control thinks is band 0—the *top-level* band—is derived from the second highest **Command** object. The quickest solution is to simply

right-click on the Flexgrid control and select Retrieve Structure. This will reinitialize the control, but you will have to go into Properties and reselect those fields that you want to display.

Another solution is actually a cute trick: Drag the summary **Command** object onto the form, and drop it anywhere except on top of the Flexgrid. Choose Bound Data control if you dragged with the right mouse button. You will now have a textbox that is displaying the summarized level of the command hierarchy while the Flexgrid control displays the detail. Next, place a command button on the form and attach some code to scroll through the **Recordset**, as shown:

```
deHierarchy.rscmdCustomer_Grouping.MoveNext
```

The form will then *page* through the data one state at a time—the Flexgrid will always display customers and sales for only that state in the textbox. You can quickly turn the form into an inquiry by state form. I further refine this example later in the chapter by moving navigation to the **DataEnvironment** object and adding more functionality, such as a Find dialog.

Aggregating Data

You can create aggregated commands based on hierarchical relationships and/or grouped relationships. The aggregate is a **Field** object, which contains aggregated data based on how you define it. I found using aggregation to be very frustrating and am assuming that the problems are beta-related (or me!). In particular, I had no luck using the grand total aggregate or any meaningful grouping aggregations. Nevertheless, I will walk you through what I learned.

Assuming you have two **Command** objects related in a hierarchical manner, you can edit the properties of the parent object and select the Aggregates tab. In the Name box, enter a meaningful name for the aggregate. Then, you can select the function type (Sum, StdDev, and so on). Below that is a box where you define what you will aggregate on. Depending on what level of the hierarchy you are on, and whether you have grouped the hierarchy, you can aggregate based on Grouping (computations are taken on breaks in whatever field(s) you have grouped by), the child table, or Grand Total. If you choose a child table, then you can select one of its columns in the Field box. For instance, in Figure 9.14, I have a form where I chose to sum the **ord_amount** column. If you chose Grand Total as your grouping level, you can only compute the total based on the Grouping criteria.

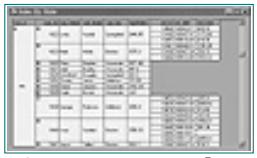

Figure 9.14 This form, grouped by state, shows each customer with total sales and detail sales.

TIP
Provider And The Shape Language
Aggregating, grouping, and establishing hierarchies is all done with the

Shape language, which I discuss later in this chapter. To use the **Shape** language yourself, you must set the **Connection** object's **Provider** property to "**MSDataShape**" and change the **Connection** string from **"Provider=…"** to **"Data Provider=…"**. I did the same for the **DataEnvironment** object.

Alas, I was never able to get any sort of nested computations and spent a considerable amount of time in working the problem out using VB's tools. (When you work with a new package and things don't go as planned, the tendency is to assume that you are making the error). I spent a considerable amount of time debugging the **Shape** language generated by Visual Basic. I discuss the mechanics of what I did later in this chapter.

I wanted to total sales by state. In the Properties window of the **cmd-Customer** object, I added a new aggregate and called it **TotSalesByState**. I selected **Sum** for the function and **Grouping** for the aggregation level. A new **Field** object was added to the parent **Command** object—the same one that contains the **cust_st** summary field.

After I got the **Shape** language squared away, the next trick was to tell VB to use my version and not its own. To do that, I needed to edit the **WillExecute** event of the **DataEnvironment** object. You may recall from Chapter 7 that the syntax for this event includes the **Source** argument. I placed some code into this event to see what command was to be executed and, if appropriate, replaced it with my own source.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Using Hierarchical Commands And Recordsets In DataReports

You can also use your hierarchical **Recordset** as the source for a **DataReport**, though it isn't as straightforward as it might seem. To do this, add a **DataReport** to your project from the Project menu. Then, drag the parent **Command** object on to the **DataReport**. At first blush, it seems as though nothing has happened. But, in fact, the **DataReport** reconfigures itself by adding (or deleting) headers and footers to reflect the number of levels of data in the **Recordset** hierarchy.

Consider the report shown in Figure 9.15; the detail section is composed of individual line items. The **Line_Item** table happens to be the child of all other tables in the command hierarchy. The parent of the **Line_Item** table is the **Orders** table. Visual Basic assigns the first header and footer sections to the **Orders** table. On the report, I use this area to print the order number and date. The footer section can be used to perform calculations (such as summing) on the preceding section (in this case, the detail section).

Figure 9.15 The Customer Sales Report generated by the report designer.

The parent of the **Orders** table is **Customer**, so Visual Basic makes a higher-level header and footer pair of sections for customer-level data. On my report, I use the header area to show the customer number and name.

After this is done, you can drag individual fields onto the report and place them as you wish, within appropriate sections. For instance, I cannot place line item information into the customer header. You can add functions, labels or other controls, and so on.

Encapsulation And The DataEnvironment Object

In most of the examples in this chapter, I have manipulated **Recordset** objects directly from the forms that are displaying them. This approach, while valuable as an illustrative technique, has some problems. My main objection is the redundancy of code—each form that displays a given **Recordset** must re-create the wheel in terms of navigation, data validation, and so forth. Furthermore, in an object-oriented world, we have to keep in mind the issue of ownership and responsibility. The forms that display the **Recordset** objects don't actually *own* the record sets. The **DataEnvironment** object owns them, instead. As Chapter 8 describes, encapsulation demands that manipulation be a function of the owning object.

For this example, I refined the form that I discussed earlier under "Grouped Command And Recordset Hierarchies." On the form itself, I added an array of four command buttons named **cmdMove**. I added two other command buttons named **cmdFind** and **cmdClose**. Finally, I added the new DataCombo ActiveX control. (To use the DataCombo control, add Microsoft DataList Controls 6.0, which adds an OLE DB enhanced list box and combo controls to the toolbox.) Figure 9.16 shows the form.

Figure 9.16 The Customer Detail form enhanced with navigation and find capabilities.

Instead of performing the navigation and find functions within the form, I added them to the **deHierarchy** object. Recall that the object has multiple commands and record sets. I wanted to generalize the routines so that they could be used with any **Recordset** object.

In preparation, I declared a reference to the **rsCusts_by_State Recordset** in the **frmCustOrdDet** form:

```
' In the General Declarations Section
Dim rs As Recordset
Private Sub Form_Load()
Set rs = deHierarchy.rscusts_by_state
End Sub
```

Next, I added an **Enum** for navigational control to the general declarations section of the **deHierarchy** module (you may want to do it in a standard module instead):

```
Public Enum RsMovement
  rsFirst = 0
  rsBack = 1
  rsForward = 2
  rsLast = 3
End Enum
```

Next, I added two **Sub** procedures as shown:

```
Public Sub MoveRS(rs As Recordset, dir As RsMovement)
```

```
Select Case dir
  Case rsFirst
    rs.MoveFirst
  Case rsBack
    If rs.AbsolutePosition > 1 Then
      rs.MovePrevious
    End If
  Case rsForward
    If rs.AbsolutePosition < _
      rs.RecordCount Then
      rs.MoveNext
    End If
  Case rsLast
    rs.MoveLast
End Select
End Sub

Public Sub rsFind(rs As Recordset, sVal As String)
Dim vBMark As Variant
If (Not rs.BOF) And (Not rs.EOF) Then
  vBMark = rs.Bookmark
Else
  Exit Sub
End If
rs.MoveFirst
rs.Find sVal
If rs.EOF Then
  MsgBox "Record Not Found"
  rs.Bookmark = vBMark
End If
End Sub
```

The first **Sub** accepts as arguments a **Recordset** object and a variable of type **rsMovement**, which is the **Enum** created earlier. Back in **frmCustOrdDet**, I coded the procedure to handle navigation:

```
Private Sub cmdMove_Click(Index As Integer)
Dim iDir As RsMovement
Select Case Index
  Case 0: iDir = rsFirst
  Case 1: iDir = rsBack
  Case 2: iDir = rsForward
  Case 3: iDir = rsLast
End Select
deHierarchy.MoveRS rs, iDir
End Sub
```

As you can see, the routine simply checks which button was pressed, sets the direction variable, and makes a call to the **MoveRS** procedure, passing it a reference to the **Recordset** and a direction. Any form or module can share the **MoveRS** procedure, of course.

The Find function allows the user to select a state from the combo box control on the form and click Find. The find procedure in **deHierarchy** is generalized to accept any **Recordset** and any search criteria. It does some basic error checking and executes the search. From **frmCustOrdDet**, the procedure is as follows:

```
Private Sub cmdFind_Click()
deHierarchy.rsFind rs, "cust_st = '" & DataCombo1.Text & "'"
End Sub
```

These principles are not a whole lot different than the techniques we used to abstract classes in Chapter 8 and can be expanded upon to add updating, validation, and so on. Unfortunately, the **DataEnvironment** cannot protect its members. In other words, the **Recordset**s it generates are public by nature and there is nothing to stop any other module from manipulating them directly. You can, of course, use the **DataEnvironment** as a data source to a class module and achieve some degree of encapsulation in that manner.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The Shape Command

When you create a hierarchical **DataObject**, Visual Basic implements it with what is known as a **Shape** command. You are free to create the syntax yourself. For instance, you can create a **Recordset** by using **Shape** language as its record source. The statement to create the hierarchical **Recordset** shown in Figure 9.14 is:

```
recordset.open "Shape {Select * from line_item}", connection
```

In order to use the **Shape** language, you must set the **Connection** object's **Provider** property to **MSDataShape** and alter the **ConnectionString** somewhat:

```
con.Provider = "MSDataShape"
con.ConnectionString = _
  "Data Provider=MSDASQL;dsn=Coriolis VB Example;" & _
  "uid=coriolis;pwd=coriolis;"
con.CommandTimeout = 15
con.CursorLocation = adUseClient
con.Open
```

As mentioned earlier, the **Shape** language generated by the DataEnvironment Designer is flawed. You can view the generated statement by right-clicking on the **Command** object.

I attempted to create the form shown in Figure 9.17 with the designer. As you can see, it subtotals all the sales for each customer, groups the customers by state, and takes a total of sales by state.

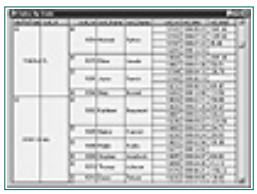

Figure 9.17 This form was generated by overriding Microsoft's **Shape** language.

The syntax that the designer generated is as follows:

```
Shape ( Shape {Select * From customer} As cmdCustomer _
  Append (( Shape {Select * FROM orders} As cmdOrders _
  Append ({Select * From "coriolis"."line_item"} _
  As cmdLineItem Relate ord_no To line_ord_no) _
  As cmdLineItem) As cmdOrders Relate cust_no _
  To ord_cust_no) As cmdOrders, Sum(cmdOrders.ord_total) _
  As sumOrds) Compute cmdCustomer, Sum(cmdOrders.sumOrds) _
  As TotSalesForState By cust_st
```

The syntax is a bit difficult to follow, so I will try to explain it in this section. For now, understand that the correct syntax is more like this:

```
Shape ( Shape {Select * From customer} As cmdCustomer _
  Append (( Shape {Select * From orders} As cmdOrders _
  Append ({Select * From coriolis.line_item} _
  As cmdLineItem Relate ord_no To line_ord_no) _
  As cmdLineItem) As cmdOrders Relate cust_no _
  To ord_cust_no) As cmdOrders, _
  Sum(cmdOrders.ord_total) As sumOrds) Compute cmdCustomer,_
  Sum(cmdCustomer.sumOrds) As Totsalesforstate By cust_st
```

Although it took me quite a while to hack through the language, the only real difference is on the last line. Notice on the second to last line that a **SUM** is done on **cmdOrders.ord_total**. **Ord_total** is a column in the **Orders** table. **cmdOrders** is a child of the **cmdCustomer** object. The **SUM** occurs right before the **COMPUTE cmdCustomer** and creates a **Field** called **sumOrds**, which is a field within **cmdCustomer**. But, on the next line, the generated syntax attempts to reference a field named **cmdOrders.sumOrders**. **cmdOrders** has no such field. I changed that to **cmdCustomer.sumOrds**, and the syntax was correct.

It makes sense that the change was as "simple" as that, because, as I note earlier in the chapter, I could not get any nested functions to work. In experimenting, I found that the designer consistently referenced an object one step lower in the hierarchy than it should have whenever it attempted to manipulate summary fields.

Unfortunately, knowing what is wrong and doing something about it are two different things. You cannot override the language that VB creates in the designer. What I had to do instead was a little trickery. In the **deHierarchy** object, I placed some code in the **WillExecute** event. Recall from Chapter 7 that one of the arguments to this event is the **Source** that will execute. I tested to see what **Source** was about to execute and then overrode it in this way:

```
Private Sub conCustOrdDet_WillExecute (Source As String, _
  CursorType As ADODB.CursorTypeEnum, LockType As _
  ADODB.LockTypeEnum, Options As Long, adStatus As _
  ADODB.EventStatusEnum, ByVal pCommand As ADODB.Command, _
  ByVal pRecordset As ADODB.Recordset, _
```

```
  ByVal pConnection As ADODB.Connection)
If InStr(1, UCase$(Source), "TOTSALESFORSTATE") Then
  Source = "SHAPE ( SHAPE {SELECT * FROM customer} " & _
  "AS cmdCustomer APPEND (( SHAPE {SELECT * FROM " & _
  "{orders} AS cmdOrders APPEND ({SELECT * FROM " & _
  "line_item} AS cmdLineItem RELATE ord_no" & _
  "TO line_ord_no) AS cmdLineItem) AS cmdOrders" & _
  "RELATE cust_no TO ord_cust_no) AS cmdOrders," & _
  "SUM(cmdOrders.ord_total) AS sumOrds) COMPUTE" & _
  "cmdCustomer, SUM(cmdCustomer.sumOrds) " & _
  "AS TotSalesForState BY cust_st"
End If
End Sub
```

This isn't a perfect solution (I rely on finding the value **TotSalesForState**, which I might use in more than one command), but it works. It also provides a valuable opportunity to dissect the **Shape** language.

Before doing so, let's take a step back and ask the $64,000 question: Why bother?

As I noted earlier, hierarchical **Recordset** objects are more efficient than **Recordset** objects created with an SQL join. This is because the parent in a join is repeated for each occurrence of a child, while it (the parent) occurs only once in a hierarchical **Recordset**. To illustrate, assume that you issue a select against a join of fictitious **State** and **City** tables as follows:

```
SELECT state_name, city_name
FROM STATE, CITY
WHERE STATE.st_id = CITY.st_id
ORDER BY state_name, city_name
```

The result set would redundantly include information from the **State** table for each row returned from the **City** table:

```
state_name  city_name
----------  ---------
Massachusetts Attleboro
Massachusetts Auburn
Massachusetts Bedford
Massachusetts Boston
Massachusetts Burlington
```

In a hierarchical **Recordset**, the same query would return only one occurrence of the value **Massachusetts**. In this example, the **State** table is the parent. To create the parent using **Shape**, you would code the following:

```
Shape {Select state_name From State}
```

To create a child, you need to **Append** a new query, and then **Relate** the two objects together:

```
Shape {Select state_name, state_id From State}
  Append {{Select city_name, city_state_id From City}
  Relate state_id To city_state_id}
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Notice that the **Relate** references **state_id To city_state_id**. Unlike SQL, when you join two tables via the **Shape** command, you have to include in the **Select** statements the two columns on which you are joining the tables.

If assigned to a **Recordset**, the columns from the state table cannot be independently accessed. We can fix that problem by providing an *alias*, as shown next:

```
Shape {Select state_name, state_id From State} As cmdStateCity
  Append {{Select city_name, city_state_id From City} As cmdCity
  Relate state_id To city_state_id} As CmdStateCity
```

The key here is the naming of the inner **Select cmdCity**. The resulting **Recordset** created from this would have three **Field** objects: **state_name**, **state_id**, and **cmdCity**. The third **Field** is, of course, another **Recordset**.

The question, then, is how to access the **Fields** within the **cmdState Recordset**. Assume that you had created a **Recordset** named **rsState**. You could access the **cmdCity Recordset** as shown next:

```
Dim vCity As Variant
Dim rsCity As New Recordset
vCity = rsState("cmdCity")
rsCity = vCity.Value
```

Declare a variable, **vCity**, of type **Variant** and a second variable, **rsCity**, of type **Recordset**. Assign to **vCity** the value in the **cmdCity** field. Then, set **rsCity** equal to **vCity.Value**. Visual Basic's example of referencing a child **Recordset** in the Help file was incorrect with my copy of VB6.

A little explanation is in order here. While it is tempting to assume that **vCity**'s underlying data type is **Recordset**, it isn't. It is actually type **Field**. The **Field** object's **Value** property is of type **Recordset**. What's more, you cannot make a direct assignment of **rsCity = vCity**—you will get a mismatch error. Whereas you might assume that **vCity**'s default property is **Value** (the **Field** object's default property is **Value**), **vCity** is still a **Variant**. So, you need to be explicit with your references.

Alternatively, you could reference **vCity** directly:

```
text1.text = vCity.Value("city_name")
```

Although it is strange looking, it works.

When you create parent/child **Recordset** objects, the database only needs to return one copy of each parent record, unlike with the SQL join example earlier in this section. The state of Massachusetts (excuse me, the *Commonwealth* of Massachusetts) has 351 cities and towns. For each of those 351 city records, there is only the one parent record. Assuming the state table has all 50 states and 20,000 or so cities and towns, the parent **Recordset** would include only 50 records. This represents a considerable savings in network traffic and, presumably, database resources.

Figure 9.18 shows another sample application (included on this book's CD-ROM) with which you can experiment with **Shape** language and parent and child **Recordset** objects. There are three command buttons, which display data from one, two, or three tables, respectively. For the application, I used the **Customer**, **Orders**, and **Line_Item** tables. The listing is very lengthy, and so I will only touch upon the highlights.

Figure 9.18 The Shape Testing application.

The goal of the application is, of course, to display the data. This can be a mite tricky as well. If you run the sample application and press the third command button, the first customer record is displayed. That customer's first order is displayed and that order's first line item is displayed. But, if you press one of the top navigation buttons (to scroll to the next customer), the order doesn't change. If you scroll the order, the line item doesn't change. So, what else is going to go wrong today?

You may recall the **StayInSync** property from Chapter 7. If this property is **False** (the default) on a parent **Recordset**, then the parent and child **Recordset**s move independently of one another. To keep the parent and child **Recordset**s in sync, you must set this property to **True** on the parent **Recordset** *before* retrieving the child **Recordset**. On the sample application, checking the Sync Parent checkbox will cause the **Recordset**s to stay in sync.

The application doesn't know what data it will be displaying or how many **Recordset** objects there are. Listing 9.2 shows a mildly dangerous way to generically display data without knowing up front what the child **Recordset** objects are, how many there are, or their field names.

Listing 9.2 A code "trick" to iterate nested Recordsets.

```
Private Sub disprec()
Dim iCtr As Integer
Dim iStart As Integer
Dim iRs As Integer
Dim iFlds As Integer
On Error GoTo BumpIt
ShowRS:
iFlds = rs(iRs).Fields.Count
For iCtr = iStart To iStart + iFlds - 1
  lblFields(iCtr) = rs(iRs).Fields(iCtr - iStart).Name
```

```
  If IsNull(rs(iRs).Fields(iCtr - iStart).Value) Then
    txtFields(iCtr) = "*NULL*"
  Else
    txtFields(iCtr) = _
      rs(iRs).Fields(iCtr - iStart).Value
  End If
Next
Exit Sub
BumpIt:
' This is a dangerous trick!
If Err.Number = 13 Then
  iRs = iRs + 1
  iStart = iCtr
  Resume ShowRS
Else
  MsgBox Err.Description
End If
End Sub
```

First, I should explain that I created an array of **Recordset** objects in the application. If you examine the code listing on this book's CD-ROM, you can see that I did, in fact, make some assumptions about the names of those child **Recordset** objects. A few code changes and you can generalize the routine that creates them as well. Iterate through the **Field** objects testing for a **Null** value. When you encounter a child **Recordset**, a mismatch error will be generated. In the error-handling routine, **ReDim Preserve** your array of **Recordset** objects, and, using the technique I showed earlier, set the new **Recordset** to be the **Field** object that generated the error.

In the preceding example, I use a very similar technique to iterate through the **Recordset** objects displaying their fields. When a child **Recordset** is encountered in the parent, a mismatch error occurs and I increment the **rs** subscript.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

You can consult the VB documentation for other hints on creating **Shape** statements. My recommendation is to generate them in the DataEnvironment Designer and then copy and paste them into your code. Fix any errors using the techniques described at the beginning of this section.

The **Shape** language can get pretty overwhelming for us mere mortals. Even here, though, you can do yourself a favor by starting small, testing, and adding new items. I tell my students the same thing when trying to write complex queries in Oracle or SQL Server. The last hint I would like to leave you with is to treat the **Shape** statement like a nested **If** statement, if necessary. The sample application uses the following **Shape** statement to join three tables. To make it more understandable, I have reformatted it, indenting at each level (parent, child, and sub-child):

```
Shape
  (Shape {Select * From customer} As cmdCustomer
     Append (
      (Shape {Select * From orders} As cmdOrders
         Append (
           {Select * From "coriolis"."line_item"}
              As cmdLineItem
         Relate ord_no To line_ord_no) As cmdLineItem)
           As cmdOrders
     Relate cust_no To ord_cust_no)
As cmdOrders
```

When you look at it this way, it’s not so bad.

Other Data Access Tools In Visual Basic 6

Throughout the first nine chapters of this book, I have attempted to introduce as many of the tools and techniques to access data in a client/server environment as

possible. In fact, the nature of VB's extensibility means that one cannot cover every conceivable tool—most of VB's intrinsic controls are data aware and there is a plethora of third-party components available as well. In Chapter 10, I discuss the class module as a data consumer *and* a data provider in the process of constructing business objects. In Chapters 12 through 15, the discussion moves toward remote database development with an eye toward the World Wide Web and, there, yet more tools are introduced.

Visual Basic 6 continues the trend of an accelerated migration to an object-oriented and database-oriented development tool. VB3 was the first serious attempt at using Visual Basic as a client/server development tool. In my opinion, VB4 was really just a refinement of that attempt. VB5 was as much an improvement over VB4 as VB4 was over the very first release of Visual Basic 1. VB6 is a quantum leap over VB5.

Each time I fire up this beta release of Visual Basic 6 in research for this book, I find something new and wonderful. As I have noted throughout this chapter, some of the edges are a little ragged, but those will be smoothed—hopefully by the time VB is sent into general release.

In this section, I want to briefly overview some of the remaining tools that you should be familiar with.

The Bindings Collection And Binding Object

Visual Basic 6 introduces the **Bindings** collection and the **Binding** object. These objects provide a means to bind any data consumer (such as a textbox or class) to any data provider (such as a data control or **Recordset**).

To use the **Binding** object, you have to declare a reference to the Microsoft Data Binding Collection Library. Then, declare objects as being of type **BindingCollection**, as shown here:

```
Dim bc1 As BindingCollection
```

Next, you bind the **BindingCollection** to a data source. In the following example, I bind it to a **Recordset**:

```
Set bc1.DataSource = rs
```

At this point, **bc1** is a data *consumer*. As such, it doesn't do a whole lot of good, but you can make it act as a data *provider* as well. Next, I am going to bind a couple of textbox controls by adding members to the collection. The arguments that I have provided are:

- The control that will act as the data consumer (in this case, **txtDept** and **txtSal**)
- The property of the control that will display the data (in this case, **Text**)
- The field from the **Recordset** (**emp_dept_no** for **txtDept** and **emp_salary** for **txtSal**)
- A name for the object (**dept** for the first and **salary** for the second)

```
bc1.Add txtDept, "text", "emp_dept_no", , "dept"
bc1.Add txtSal, "text", "emp_salary", , "salary"
```

At this point, the textbox controls are now bound to **bc1**, and **bc1** is bound to the **Recordset**. I will be using this behavior to advantage in the next section.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Visual Basic Format Objects

Format objects allow you to format and unformat data for display. In the following examples, I want to display the employee's gender as red if female or blue if male. To do so, I am going to declare a **Format** object to handle this. To use **Format** objects, you must first declare a reference to the Microsoft Standard Formatting Library. Then, for each **Format** object that you want to create, you must declare a variable of type **StdDataFormat**, as shown in the following code. In my example, I also need to bind the **Format** object to a **BindingCollection** object, so I declare a reference for that as well:

```
Private bc1 As New BindingCollection
Private WithEvents fmtSex As StdDataFormat
```

The sample application on this book's CD-ROM, FormatBindings, opens an Active Data control on the **Employee** table. (The example from the prior section is included in that project as well). In the application's **Open** event, I coded the following:

```
Set bc1.DataSource = Adodc1.Object
Set fmtSex = New StdDataFormat
fmtSex.Type = fmtCustom
bc1.Add txtFields(3), "text", "emp_gender", _
  fmtSex, "Sex"
```

I bound **bc1** to the Active Data control and then created a new instance of **fmtSex**. I set its type to **fmtCustom**. Other options include **fmtBoolean**, **fmtPicture**, **fmtCheckbox**, and so on. See the VB Help file for more information. I then added the object to the **BindingCollection**.

The **Format** object has a number of events. You can use the **Format** event to alter the presentation of the data if necessary. Because the data values can change,

I need to reformat the data each time the value changes:

```
Private Sub fmtSex_Format(ByVal DataValue As _
  StdFormat.StdDataValue)
If DataValue = "F" Then
  txtFields(3).ForeColor = vbRed
Else
  txtFields(3).ForeColor = vbBlue
End If

End Sub
```

As you can see, **DataValue** is a property of the object. There is also an **Unformat** event, which you can use to convert the data back into a form acceptable to the database. Finally, there is a **Changed** event to handle situations where the data has changed.

Figure 9.19 shows the prior two sections in action. The application includes four different format objects and three **BindingCollection**s. It compares each employee's salary to the department average and company average. If greater, the **Format** object sets the **Checkbox** controls. As you can see, the employee shown makes more than the company average but not as much as her department's average salary. Though you can't tell in Figure 9.19, her gender is displayed as red.

Figure 9.19 This application makes use of **Format** and **DataBinding** objects.

Data-Bound User Controls

You can use the **DataBindings** collection to bind controls on a user control to a data source. On this book's CD-ROM, the AXctlLineItem project is a user control compiled to an ActiveX control that is bound to the database.

To create a data-bound user control, create a new project of type ActiveX control. Place fields as needed. For the example application, I used four textboxes that will eventually manipulate data from the **Line_Item** table. Each textbox is exposed through the appropriate **Property Get** and **Property Set** statements. For instance, the following code manipulates a control property, **ItemDesc**, which will eventually be mapped to the **item_description** field on the database:

```
Public Property Get ItemDesc() As String
ItemDesc = txtFields(3).Text
End Property

Public Property Let ItemDesc(sItemDesc As String)
txtFields(3).Text = sItemDesc
End Property
```

To map the properties, select Procedure Attributes from the Tools menu. At the top is a drop-down window that displays the object's properties. Click the Advanced button and you will see where you can set object behaviors. The bottom area is for data binding. Check the Property Is Data Bound box and also the Show Property In Data Bindings Collection At Run Time boxes. Repeat this for each of the four properties in the control and then compile it. You will use the control in conjunction with the DataRepeater control in the next section.

The DataRepeater

The DataRepeater control allows you to display multiple occurrences of a data bound object—such as the ActiveX control created in the prior section—in a format similar to a grid control. To use it, add the Microsoft Data Repeater Control 6.0 to your toolbox, and then draw it on the form. Make it big enough to handle multiple occurrences of the OCX you created in the prior section. (Figure 9.20 shows the running application—the application from this book's CD-ROM is Data-Repeater.) Add an Active Data control whose **Source** is **"Select * From Line_Item"**.

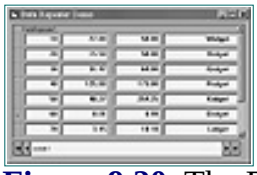

Figure 9.20 The DataRepeater control is used to display multiple occurrences of the contained data bound controls.

Next, find the **RepeatedControlName** property in the Properties dialog for the DataRepeater control. Click it, and all ActiveX objects registered on your PC are displayed. Locate the control you created in the last section and select it. When you do, it will be repeated multiple times inside of your DataRepeater control. For the **DataSource** property, select the Active Data control.

Next, right-click on the control and select Properties. Select the RepeaterBindings tab. There is a drop-down list box listing all the properties of the ActiveX control that you chose to be added to the **DataBindings** collection in the last section. Select each in turn, and select a database field from the DataField box. You can optionally specify a collection key in the next field. The next tab, Format, allows you to format individual fields.

Run the project and you should get a display similar to the one shown in Figure 9.20. To programmatically access the properties and methods of your user control, the **RepeatedControl** property returns a reference to the control. For instance, to set the **ItemDesc** property:

```
DataRepeater1.RepeatedControl.ItemDesc = "Another Item"
```

The current record is indicated by the **CurrentRecord** property.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Where To Go From Here

We have covered a lot of ground in these first nine chapters. In Chapter 10, I'll explore the concept of the business object and build on the technique of using classes as data objects. In Chapter 11, I'll concentrate on the database aspect of your client/server application. After that, my co-author Kurt Cagle takes you into the exciting and scary realm of Web development, DHTML, XML, and so on.

The amazing thing about Visual Basic 6 is that, even in 3,000 pages, not every aspect can be covered in depth. But, it is not my intention to do so. We have built a core of knowledge from Chapter 1 on and added to that very rapidly until getting into the relatively advanced concepts presented in this chapter.

If you have come this far, congratulations! Take some time to explore and expand on some of the sample applications on this book's CD-ROM. In particular, take note of how the latter examples in this chapter take advantage of procedures written earlier. For instance, the final **Shape** application uses the same navigation methodologies encapsulated into the **deHierarchy** that **frmCustOrdDet** used earlier in the chapter. Also, note how we began moving towards encapsulation of methods and properties. The user control in the last section, for instance, makes it impossible to access an item from the database without using its interface. Such techniques promote productivity and reliability of code (and is the focus of the next chapter).

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Chapter 10
Creating Business Objects With Visual Basic 6

Key Topics:

- Object-oriented development
- Creating **Private** and **Public** events, methods, and properties
- In-process and out-of-process ActiveX servers
- Connecting to local and remote business objects

Application development always seems to be 12 to 36 months behind technology, and maybe that is just as well. In the early to mid 1990s, a joke compared client/server development to sex: Everybody is talking about it, but nobody is doing it. Today, the buzz is about distributed objects, COM/DCOM, Web-based applications, n-tier architecture, and so on. Although I certainly don't mean to imply that these technologies aren't viable and even in use in some organizations, they still remain the exception and not the rule.

Throughout this book, I have attempted to keep an eye on where most organizations and developers are technologically, while building a foundation for the architectures of tomorrow. One has only to look at the delays in the release of SQL Server 7—which relies heavily on ADO, DCOM, and distributed objects—to see that some of these new technologies are still evolving. I cannot recommend throwing the tried and true techniques that I have discussed so far in this book into the abyss of remote objects and the like. Instead, I recommend that you proceed slowly, testing the reliability and viability of these techniques as they evolve.

Should you investigate these technologies? Absolutely. Most of the remainder of this book is devoted to these new and exciting development methodologies. My recommendation at this time, if you decide to move forward, is to roll out your deployments with small, non-mission-critical applications as you get your feet wet (in other words, learn how to use the new tools) and as the technology matures and stabilizes.

DCOM, distributed-architecture, n-tier, Web-based applications, and so on are all parts of the same whole: Microsoft's services architecture model, which I introduced in Chapter 1. DCOM is the glue that binds all of these together; its very nature is the distribution of components—ActiveX objects—at appropriate places with the services model (user, business, or data) and imparting them with the knowledge of how to communicate with one another, regardless of the language used to develop them and regardless of their location on the network.

In this chapter, I lead you through the foundations of distributed components with an introduction to creating right-sized and right-located business objects. Note that most of the examples in this chapter are adapted from applications or objects generated by different Visual Basic wizards (such as the Data Object Wizard) to which I then made modifications to illustrate various points.

Introducing The Business Object

A business object encompasses data and rules that perform a business function. More to the point, *any* object in a COM or DCOM environment has three facets:

- *Data*—Business information from the data source, such as a customer record, as well as variables used internally.
- *Behavior*—Methods and events of the object.
- *Interface*—Both the user interface, if any, and a set of "connections" that expose the object to the outside world. In DCOM, these connections are the object's proxy and stub.

COM, DCOM, And ActiveX Summarized

It is worth taking a moment to reflect on what the various pieces of the COM, DCOM, and ActiveX architecture really are and how they fit together. For anyone who has done Java development, ActiveX can be likened to JavaBeans and DCOM can be likened to COBRA. COM (Component Object Model) is as much a development philosophy as it is a development methodology. COM-based development uses previously created components in an effort to promote reusability, reliability, and interoperability. The components are ActiveX servers, as I will be discussing in this chapter. COM in general, and ActiveX controls in particular, focus on the desktop—the presentation layer. In contrast, DCOM (Distributed Component Object Model) focuses on the network and is the technology that enables objects running in separate processes—particularly those running on separate computers—to communicate with one another.

With that quick summary in mind, let's jump into the concept of the business

object in component based development.

This concept of an object is illustrated in Figure 10.1.

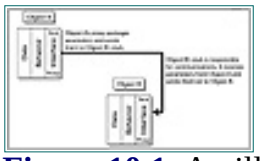

Figure 10.1 An illustration of a client (Object A) calling Object B (running in its own pro-cess)—perhaps to invoke a method of Object B.

Business Object Data

To create a business object, you first need to create its definition or class. In VB, you do this by creating a class module. When the class is instantiated, it becomes an object. Within the class module, you create the data that you need to manipulate, as well as methods and events. Any data that is declared **Private** cannot be seen by other modules, so it is protected from unauthorized use. You may opt to expose **Public** methods, allowing the manipulation of that data, but the manipulation will still be done by the class module itself. This arrangement is called encapsulation; privately declared data, methods, and events are encapsulated by the business object.

The following code declares some variables in a class module:

```
Option Explicit

Private Emp_No As Integer
Private Emp_FName As String
Private Emp_LName As String
Private Emp_DOB As Date
Public bClose As Boolean
```

Because the first four variables are not publicly available to other modules, they are fully protected. The last variable is declared **Public** and can thus be changed by any other object. However, you then need to provide some sort of interface to the outside world to access these variables.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Business Object Behavior

You allow access to objects via properties, methods, and events. Together, those properties, methods, and events that are declared with the **Public** keyword compose the object's interfaces to other objects. You could, for instance, provide an interface that returns to the requesting object the employee's name:

```
Public Property Get EmpName () As String
  EmpName = Emp_FName & " " & Emp_LName
End Property
```

In this example, the property **EmpName** is declared as **Public**, so it is part of the object's interface. If the object's name were **clsEmployee**, you would reference the property using simple dot notation:

```
' Display the customer
txtEmpName.Text = clsEmployee.EmpName
```

EmpName is a read-only property because it was defined with the **Get** modifier. To create a property that can be written to, use the **Let** or **Set** modifiers. Perhaps you want to allow the update of the employee's name but only under strict control. You do this with a **Property Let** statement, as shown:

```
Public Property Let FLName (first As String, last As String)
  Emp_FName = first
  Emp_LName = last
End Property
```

Of course, the preceding code snippet doesn't provide much in the way of data validation. Let's modify the code a little bit:

```
Public Property Let FLName (first As String, last As String)
```

```
  If Len(first) = 0 Or Len(last) = 0 Then
    Err.Raise vbObjectError + 512, _
      "First and last names cannot be blank"
  ElseIf Len(first) > 15 Or Len(last) > 21 Then
    Err.Raise vbObjectError + 513, _
      "Name too long. First <= 15, Last <= 21."
  Else
    Emp_FName = first
    Emp_LName = last
  End If
End Property
```

The preceding code object edits the name lengths for validity and uses the Visual Basic error-handling facilities to flag any data violations. The calling object is not allowed to pass invalid values.

Business Object Methods

Publicly declared methods are implemented as **Sub**s or **Function**s and implement any services that the object needs to provide to the outside world. Of course, the object may also have privately declared methods, which are used internally by the object and are then not part of the object's interface.

There is some overlap between methods and properties. Consider that a **Property Get** exposes a portion of the object—such as a variable. It is similar to a **Function** in that it returns a value to the calling object. Likewise, a **Property Let** or **Property Set** is similar to a **Sub** in that it performs an action without returning a value to the calling object.

An Alternative To The Error Handler In Class Modules

An alternative to using the error-handling facilities for **Property Let** and **Property Get** is to create a property whose sole purpose is error reporting. To do this, create a private module-level variable named **ErrStat** and an array of error messages named **ErrMsg**. Create **Public** properties to expose them:

```
' General Declarations Section
Private ErrStat As Integer
Private ErrMsg (100 To 199) As String

' Class Initialization Procedure
ErrMsg (100) = "First and last names cannot be blank"
ErrMsg (101) = "Name too long. First <= 15, Last <= 21."

Public Property Get Status () As Variant
  Status (0) = ErrStat
  If ErrStat <> 0 Then
    Status (1)= ErrMsg(ErrStatus)
  End If
  ErrStat = 0
End Property
```

Then, in your **Property Let** procedure, update the **ErrStat** variable to reflect success or

failure:

```
Public Property Let FLName (first As String, last As String)
  If Len(first) = 0 Or Len(last) = 0 Then
    ErrStat = 100
  ElseIf Len(first) > 15 Or Len(last) > 21 Then
    ErrStat = 105
  Else
    ErrStat = 0
End Property
```

This might not seem as straightforward as simply relying on VB's error-handling system, and in fact, it isn't. However, it does make the resulting business object more portable and simulates ADO's **Error** object.

TIP

Properties, Methods, And Performance

One other consideration when choosing whether to expose an interface as a property or method is the design of the object. If it is to be an out-of-process server, COM imposes severe performance penalties when communicating across process boundaries. In this case, you are better off using properties as much as possible because properties incur much less of a performance hit than methods. However, see "Relocating The Business Object" later in this chapter, where the rules change.

Your business object could be designed devoid of any publicly declared properties or methods. Consider the **EmpName** property from the last section; it could have been implemented as a **Function**, as shown:

```
Public Function EmpName () As String
  EmpName = Emp_FName & " " & Emp_LName
End Function
```

To an extent, how you build your class and your choices between properties and methods is largely a personal preference. However, my recommendation is that you expose data as properties if for no other reason than it is a more natural way for the calling object to reference the business object. In general, if something that your object exposes is a noun (the employee's name, for instance), expose it as a property. If it is a verb (update employee record, perhaps), it should be implemented as a method. If the method needs to return a value, it should be a **Function**. Otherwise, it should be a **Sub**.

Because the business object "owns" the employee data, it has the responsibility for updating that data. Assuming that a previously opened **Recordset** named **arsEmp** is being used to manipulate the data source, you might create an **UpdateEmp** interface as shown next:

```
Public Sub UpdateEmp ()
  ' Put any data validation here.
  ars.Update
End Sub
```

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Business Object Events

Events provide the foundation of any object-oriented development effort. VB provides the capability for your business object to expose events to which the calling object can react. Assume you want to create an event to let the calling object know that the employee update completed. Perhaps you also want to pass back to the calling object information about the success or failure of the update.

To create the event, you declare it in the General section of the class module:

```
Public Event EmpUpdateComplete (pError As Error, _
  bStatus As Boolean)
```

The **Event** declaration also specifies optional arguments, which the calling object will then have access to. To cause the event to occur, use the **Raise** keyword:

```
Private Sub arsEmp_RecordsetChangeComplete(ByVal adReason As _
  ADODB.EventReasonEnum, _
  ByVal pError As ADODB.Error, adStatus _
  As ADODB.EventStatusEnum, _
  ByVal pRecordset As ADODB.Recordset)
Dim bSuccess As Boolean
If adStatus = adStatusOK Then
  bSuccess = True
End If
RaiseEvent rsUpdateComplete(pError, bSuccess, pRecordset)
End Sub
```

The calling object will then see that declared event just like any other event, as shown in Figure 10.2.

Figure 10.2 Once you declare a reference to a class object, its events are exposed.

Business Objects As Components

In the object-oriented world, an object is an instance of a class. In other words, a class is the definition of an object. A dictionary definition of “person” might generically indicate that a person is a human with two legs, hair, and so on. What you see in the dictionary is not a person; it is the definition of a person. An object becomes an object only after it is *instantiated* (created).

To instantiate a class module, you create a reference to it in whatever module needs to access its properties, methods, and events:

```
Dim empClsDemo as clsEmpDemo
```

In this example, **clsEmpDemo** is the name of a class module that you had previously created. **empClsDemo** is the name of an object that gets created from **clsEmpDemo** when declared.

A component is not an object: Let’s make that clear. A component is prebuilt, binary code that is language independent. When you add the CommonDialog ActiveX control to your project, you are adding a prebuilt component. Because it is binary code, it can be used by any language that understands ActiveX technology. Components typically consist of one or more objects. In Microsoft’s COM and DCOM models, components are ActiveX Servers (either EXEs or DLLs) or ActiveX controls. As with your class module, you must make a reference to them (usually via the Components or References options from VB’s Project menu). When the reference is made, the components’ exposed properties, methods, and events become available to your program.

Although you can certainly create a class module and then reuse it from project to project as needed, its portability is especially enhanced when you compile the class into an ActiveX component. It can then be used by any application, regardless of the development language, potentially at any place on the network (instead of on the client machine). This portability is an especially useful feature because many applications can share the same component.

Components can be *in-process* or *out-of-process* servers. An in-process server runs within the same memory space as the application, whereas an out-of-process server runs in its own memory space. Each approach has advantages and disadvantages. The most significant advantage of an out-of-process server is that it can be shared among other processes. An in-process server belongs to the application. On the other hand, out-of-process servers come at a distinct performance disadvantage: Because calls from one process to another must cross process boundaries, performance can be severely impacted.

I discuss the pros and cons of the different approaches as we go along.

The ActiveX control is essentially an in-process server. Although an ActiveX control is not required to have a visible component, it typically is used that way. The CommonDialog control is one of the more common exceptions to that rule. The biggest drawback of an ActiveX control versus an ActiveX DLL is that the control must be placed in some sort of

container object—such as a form. You can't use an ActiveX control within a standard module, for instance.

Business Objects And Object-Oriented Behavior

To support object-oriented development, objects need to exhibit certain kinds of behaviors. Visual Basic class modules and their resulting objects support these behaviors, which I discuss in this section.

Encapsulation is key. An object must contain all the data that it needs to accomplish its mission (such as maintaining employee data). It must take responsibility for its mission and not allow any other processes to directly affect its data. Instead, it must expose properties, methods, and events to accomplish its goal.

Polymorphism is using the same interface for different objects. For instance, an MS Word document exposes functionality to send a document to the printer using the **Print** interface (or method). Excel exposes functionality using the same interface to send a chart to a plotter: **Chart.Print**. Assume you create two classes—one to manipulate text files and one to draw images. Each might have a **PrintIt** method, as shown next:

```
' In the text class
Public Sub PrintIt (Place As Form)
Dim sText As String
' Code to read the file
Place.Print sText
End Sub

' In the draw class
Public Sub PrintIt (Place As Form)
Place.Line Step(x1, y1) -Step (x2,y2)
End Sub
```

A form module can then use the common interface in a polymorphic manner:

```
Dm myClass As className
myClass.PrintIt Me
```

The previous example illustrates polymorphism using a process known as *late binding*. This means that the compiler cannot know which object is being referenced when the **PrintMe** method is invoked, so it must resolve it at runtime instead of at compilation. Under a different scenario, you could implement polymorphism using *early binding*. The following lines of code implement a generic **Graphics** class:

```
' In the graphics class
Public Sub PrintIt (Target As Form)
End Sub
```

Next, you can borrow the interface provided by the **Graphics** class to create the **Text** and **Draw** classes:

```
' In the text class
Implements Graphics
```

```
Private Sub Graphics_PrintIt(Target As Form)
  Dim sText As String
  ' Code to read the file
  Target.Print sText End Sub

' In the draw class
Implements Graphics
Private Sub Graphics_PrintIt(Target As Form)
  Target.Line Step(x1, y1)-Step(x2,y2)
End Sub
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The code uses the **Implements** keyword, which passes to the **Text** and **Draw** classes all of the interfaces of the **Graphics** class. The advantage comes when you seek to print something on the form, as shown in the next example:

```
Private Sub PrintOnForm (myObject As Graphics)
  myObject.PrintIt Me
End Sub
```

To use this code, you pass it the name of the class whose **PrintIt** method you want to use. Visual Basic is now able to verify the code, perform type checking, and so on at compilation time.

The **Implements** keyword is also used to apply Visual Basic's admittedly weak implementation of inheritance. The **Text** and **Draw** classes inherited all of the interfaces of the **Graphics** class. (The flip side is that they are then required to implement all of those interfaces.) Further, you can customize each class to add its own needed behaviors. You did that previously by changing the definition of the **PrintIt** method. You could have also added other, object-specific interfaces. This process of customization is called *abstraction*. Abstraction allows you to take advantage of common features in objects and ignore the differences. It allows you to borrow from an existing class, as you did with **Graphics**, and customize it as needed. You do that every time you place a control on a form. For instance, if you place three textbox controls on a form, you might make one of them a multiline textbox and ignore that interface (the **Multiline** property) with the other two.

Visual Basic does a fair job of simulating inheritance and multiple inheritance (inheriting from two or more objects to create a new object), using techniques known as *delegation* and *containment*. You are probably already familiar with containment; it states that an object can contain another object. For instance, instead of using the **Implements** keyword, the **Text** class might have declared a reference to the **Graphics** class. Delegation allows you to delegate a function to another object instead of doing it yourself. (Some call this laziness.)

The next example modifies the **PrintIt** method of the **Graphics** class to draw a line on a form. The **Text** and the **Draw** classes each contain the **Graphics** class. The **Draw** class delegates to **Graphics** the responsibility to draw a line. The **Text** class overrides this behavior with its own routine. Any object can still call any of these objects in a polymorphic manner, although the ability to do early binding is now gone.

```
' Graphics class
Public Sub PrintIt (Target As Form)
  Target.Line Step(x1, y1) -Step (x2,y2)
End Sub

' In the text class
Private myGraphics As New Graphics
Public Sub DoPrintIt (Target As Form)
  Dim sText As String
  ' Code to read the file
  Target.Print sText
End Sub

' In the draw class
Private myGraphics As New Graphics
Public Sub DoPrintIt (Target As Form)
  myGraphics.PrintIt Target
End Sub
```

Obviously, any object can contain any number of other objects. This process is called *aggregation*.

The Business Object

In this section, I create an initial model of an object to maintain employee records. Of necessity, the examples provided in the earlier section were oversimplified. In particular, I seek to exploit Visual Basic's newfound ability to create data-aware classes. I first create a simple class module, which will act as the data source in the project shown running in Figure 10.3. Then, I convert the class into an ActiveX server and run it on a remote machine, connecting to it over the network.

Figure 10.3 An MDI application using a data consumer class module as the data source of the employee maintenance forms.

Creating The Business Object

The application that I create is on the enclosed CD-ROM as EmpCLS-Demo.VBP. It includes four forms, a standard module, and a class module. You can generate the application with the Application Wizard, if you want, but I have added more

capabilities here not generated by the Application Wizard. (As shown in Figure 10.3, it is also somewhat less complex.)

The class module is named **clsEmpDemo**. It declares a number of module-level variables as shown:

```
Private WithEvents arsEmp As Recordset
Private WithEvents acon As Connection
Public Event MoveComplete()
Public Event rsUpdateComplete(pError As Error, _
  bSuccess As Boolean, ars As Recordset)
```

The variables are all declared as private, but the two events are declared as public (which is the default).

Connecting To The Data Source

The class module normally has only two events: **Initialize** and **Terminate**. Because I have defined the **DataSourceBehavior** property to be **vbDataSource**, a **GetDataMember** event is also defined.

The **Initialize** event is shown next. It performs a connection to the database and then adds an item to the **DataMembers** collection.

```
' Connect to the database

Set acon = New Connection
acon.CursorLocation = adUseClient
acon.Open "PROVIDER=MSDASQL;dsn=Coriolis VB Example;" & _
  "uid=coriolis;pwd=coriolis;"
Set arsEmp = New Recordset
arsEmp.Open "select * from employee Order by emp_no", _
  acon, adOpenStatic, adLockOptimistic
' This gives a control something to bind to
DataMembers.Add "EmpPrimary"
```

A **DataMember** represents a data source for a data consumer, such as a data-aware control. In essence, it creates a qualifier to the **DataSource** property of a data-aware control. To bind to this class, data-aware controls need their **DataSource** property set to the class (**clsEmpDemo**) and their **DataMember** property to the string added by the class (**EmpPrimary**).

A data provider can provide different data streams, which are identified by the **DataMember** property. When a **DataMember** is added to the **DataMembers** collection, the **GetDataMember** event is triggered. It is here that you associate the actual data source (the **Recordset**) with the **DataMember** as shown:

```
Private Sub Class_GetDataMember(DataMember As String, _
  Data As Object)
     Select Case DataMember
     Case "EmpPrimary"
       Set Data = arsEmp
```

```
    End Select
End Sub
```

I will show you the mechanics of how this event is called when I show you how to "connect" your form to the **clsEmpDemo** object.

Exposing The Business Object

I provided a couple of publicly available properties so that the calling object can be aware of whether the **Recordset** is in edit mode and what the specific edit mode is:

```
Public Property Get EditingRecord() As Boolean
  EditingRecord = (arsEmp.EditMode <> adEditNone)
End Property
Public Property Get EditMode() As Long
  EditMode = arsEmp.EditMode
End Property
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Remote Data Validation

The class contains all the code necessary to manipulate the data. Space limitations preclude listing the entire module. (You can examine the code on the CD-ROM in the file clsEmpDemo.cls.) Listing 10.1 illustrates some basic data validation steps. The first is an evaluation of the reason that the event was triggered. For example, if the user attempts to close the window before saving any changes, the **WillChangeRecord** event is called with an **adReason** of **adRsnClose**. The user is prompted to save his or her changes.

> **NOTE**
> In my beta copy of Visual Basic, the **adRsnClose** code was not being passed. I developed an alternative workaround in the form code itself in the **UnLoad** event, which you can examine on the CD.

Later in Listing 10.1, I evaluate the current **EditMode**. If the mode is delete, for instance, I do not bother to validate the data. If a record is being changed, then I perform some basic validations. (Some validating is done on the database via constraints, as discussed in Chapter 2.) Any data validation errors cause a message to be displayed, listing all the problems, as shown in Figure 10.4. The update is then canceled.

Listing 10.1 Data validation in the **empCLSDemo** object.

```
Private Sub arsEmp_WillChangeRecord(ByVal adReason _
  As ADODB.EventReasonEnum, _
  ByVal cRecords As Long, adStatus As ADODB.EventStatusEnum, _
  ByVal pRecordset As ADODB.Recordset)
  ' This is where you put validation code
Dim bCancel As Boolean
Dim iEmpNo As Integer
Dim iCtr As Integer
Dim sMsg As String
Dim arsEmpNo As Recordset
```

```
sMsg = "The following errors were encountered:" & vbCr

' Why did event trigger?
Select Case adReason
  Case adRsnClose
    ' Prompt to save changes
    Select Case MsgBox("Save Changes before closing?", _
      vbYesNoCancel + vbQuestion)
      Case vbNo
        arsEmp.CancelUpdate
      Case vbCancel
        adStatus = adStatusCancel
        Exit Sub
      Case vbYes
        ' Need to validate data
    End Select
    ' Other reasons
  Case adRsnUpdate
  ' Some reasons omitted for space
  Case adRsnUndoUpdate
End Select

Select Case arsEmp.EditMode
  Case adEditDelete
    ' No need to edit data we are deleting
    Exit Sub
  Case adEditNone
    ' No need to validate
    Exit Sub
End Select

' Validate remainder of data
With arsEmp
  If !emp_DOB > Now Then
    sMsg = sMsg & "Invalid date of birth" & vbCr
    bCancel = True
  End If
  If (!emp_hire_date < !emp_DOB) Or (!emp_hire_date > Now)
Then
    sMsg = sMsg & "Invalid hire date" & vbCr
    bCancel = True
  End If
  If !emp_salary < 0 Then
    sMsg = sMsg & "Salary cannot be less than 0" & vbCr
    bCancel = True
  End If
End With

If bCancel Then
  MsgBox sMsg
  adStatus = adStatusCancel
End If
```

```
End Sub
```

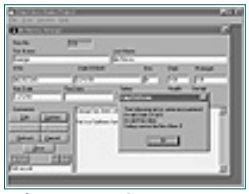

Figure 10.4 Data validation message from the employee class.

Data Handling In The Business Object

The basic data handling is typical of any ADO-based application except that the code lies encapsulated in the class module. To keep any calling objects apprised of what is going on, the module creates publicly viewable events and raises them as necessary. I show next the **arsEmp_MoveComplete** event raised by ADO within the class module. Because it is private, the form that is using the class module as a data source cannot know when a move is complete, so the event procedure raises the publicly viewable **MoveComplete** event.

```
Private Sub arsEmp_MoveComplete(ByVal adReason As _
  ADODB.EventReasonEnum, ByVal pError As ADODB.Error, _
  adStatus As ADODB.EventStatusEnum, ByVal pRecordset _
  As ADODB.Recordset)
RaiseEvent MoveComplete
End Sub
```

Similarly, I added an event to the class module to allow calling objects to be aware of the completion of any update operations. Within the **arsEmp_RecordsetChangeComplete** event, I raise the **rsUpdateComplete** event.

```
Private Sub arsEmp_RecordsetChangeComplete(ByVal adReason As _
       ADODB.EventReasonEnum, ByVal pError As ADODB.Error, _
       adStatus As ADODB.EventStatusEnum, ByVal pRecordset As _
       ADODB.Recordset)
Dim bSuccess As Boolean
If adStatus = adStatusOK Then
  bSuccess = True
End If
 RaiseEvent rsUpdateComplete(pError, bSuccess, pRecordset)
End Sub
```

NOTE
For illustrative purposes, I pass back some of the arguments received by the internal event. Normally, you do not need to nor should you pass this type of information (specifically, the **Error** and **Recordset** objects) back to the calling object. Exposing these objects outside of the class module defeats the purpose of encapsulation and ends up promoting code redundancy; any code that needs to handle these objects should exist within the class module itself and nowhere else.

Accessing The Business Object

To access the interfaces of the class module, you must declare a reference to the module. I used a form, **frmEmpClsDemo**, to display the data. The form's module-level declarations are shown next:

```
Private WithEvents empCLSDemo As clsEmpDemo
Private bChangedByCode As Boolean
Private bBookMark As Variant
Private bEditFlag As Boolean
Private bAddNewFlag As Boolean
Private bDataChanged As Boolean
```

The class module is declared using the **WithEvents** clause, which then enables the form to "see" any publicly declared events.

As shown in Figure 10.2, the events of the class then become available within the code editor. Next, I show you the usage of two of those events:

```
Private Sub empCLSDemo_rsUpdateComplete(pError _
  As ADODB.Error, bSuccess As Boolean, ars As
ADODB.Recordset)
Dim sMsg As String
If bSuccess Then
Else
  Dim vError As Error
  For Each vError In pError
    sMsg = "An error has occurred: " & vbCr & _
      "Err Number: " & pError.Number & vbCr & _
      pError.Description
  Next
End If

End Sub

Private Sub empCLSDemo_MoveComplete()
 lblStatus.Caption = "Record: " & _
 CStr(empCLSDemo.AbsolutePosition)
End Sub
```

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

In the **empCLSDemo_rsUpdateComplete** event, I examine the status (**bSuccess**) returned by the class and, if there was an error, I iterate through the **Errors** collection. As noted earlier, I don't actually encourage you to do this in the calling form—rather, it should be done in the class module itself. I show it here to illustrate the passing of objects from one module to another and, later in this chapter, across process boundaries.

The **empCLSDemo_MoveComplete** event is a little more typical. Here, I simply update the **lblStatus** control's **Caption** property with the current record number. Notice the reference to the **AbsolutePosition** property of the **empCLSDemo** object. To "see" the current record number, I had to expose the **Recordset** object's **AbsolutePosition** property with a **Property Get** statement:

```
Public Property Get AbsolutePosition() As Long
  AbsolutePosition = arsEmp.AbsolutePosition
End Property
```

While writing this book, I hoped that some issues would be resolved by the time the production release of Visual Basic 6 was shipped. An interesting problem that I had was in the **empCLSDemo_MoveComplete** event. I wanted to display the current employee's name on the form's caption:

```
Me.Caption = txtFields(1) & ", " & txtFields(2)
```

Although the textbox controls displayed the correct information, I received the data from the *previous* record if I referenced their **Text** properties in this event. The reason for this discrepancy is interesting if you step through the code. When you click the Next button on the form, the following sequence of code is executed:

1. The **cmdNext_Click** event on the form (which I show next) is

triggered. In that event, I invoke the **MoveNext** method of the **empCLSDemo** object:

```
Private Sub cmdNext_Click()
On Error GoTo GoNextError
empCLSDemo.MoveNext
Exit Sub
GoNextError:
MsgBox Err.Description
End Sub
```

2. The object's **MoveNext** event is called while **cmdNext_Click** is still executing. This event from the **empCLSDemo** object is shown next:

```
Public Sub MoveNext()
If Not arsEmp.EOF Then
       arsEmp.MoveNext
End If
If arsEmp.EOF And arsEmp.RecordCount > 0 Then
    Beep
    ' Moved off the end, so go back
    arsEmp.MoveLast
  End If
End Sub
```

3. As soon as the **MoveNext** method is executed, the **arsEmp_MoveComplete** event is triggered. As shown in Figure 10.5, the **Recordset** has indeed moved to a new record. The **emp_LName Field** object's value is **Wesley**, but the textbox control bound to this **Field** still reflects the value from the prior record: **Benson**. In fact, as shown in Figure 10.6, this is still the case; as the form is notified that the move is complete, the two values are not yet in sync.

Figure 10.5 The bound controls' values are not in sync with the **Recordset** when the **Recordset** object's **Move-Complete** event is triggered.

Figure 10.6 Even when the form is "notified" that the move is completed, the bound controls and the **Recordset** are still out of sync.

4. In the **arsEvent_MoveComplete**, the class object's **MoveComplete** event is raised. Processing control returns to the form momentarily and where the **AbsolutePosition** property is accessed so that the record count can be displayed. Once this is finished, control finally returns to the **MoveNext** event, and as soon as it does, Visual Basic syncs up the values in the bound controls to the values in the **Recordset**. The result is shown in Figure 10.7.

Figure 10.7 The bound controls are finally in sync with the **Recordset** when control is returned to the **MoveNext** event.

What this process suggests, really, is that the notification to the form that the move is complete is a little premature. A better place to put the notification is in the **MoveNext** method. Unfortunately, that choice itself creates some problems because the code is somewhat redundant. (The notification would also have to be placed in the **MoveFirst**, **MoveLast**, and **MovePrevious** methods.) What I ended up doing to solve this problem is create a new publicly viewable property within the class module, as shown next, which returns a formatted string containing the last and first names of the employee in the current record of the **Recordset**.

```
Public Property Get empName()
empName = Trim(arsEmp("emp_lname")) & ", " _
  & arsEmp("emp_fname")
If empName = ", " Then
  empName = "New Record"
End If
End Property
```

The reason for the test to see if **empName** is equal to a comma is to check whether the application is currently processing a new record. In the form module, I altered the event procedure that displays the current record number:

```
Private Sub empCLSDemo_MoveComplete()
lblStatus.Caption = "Record: " & _
  CStr(empCLSDemo.AbsolutePosition)
' Display the customer name on the caption
Me.Caption = empCLSDemo.empName
End Sub
```

The **lblStatus** control and the form's **Caption** property are updated to reflect the current record number and employee name even before the bound control's are updated, as shown in Figure 10.8. I single-stepped through the code in debug mode and took this screen shot immediately after the form's **MoveComplete** event completed. The textboxes have not yet been updated to reflect the current **Recordset** object's information, although the label control and the form's caption have been updated.

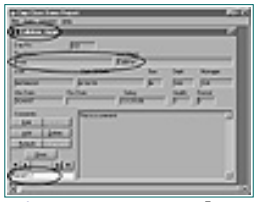

Figure 10.8 The textboxes are not yet updated immediately after the form's **MoveComplete** event fires.

Why go to all this trouble? Figure 10.9 shows the application with some embellishment. I added a Window menu that shows the four currently opened forms, all of which are instances of the employee maintenance form. In an MDI application, the operator might open many instances of different forms, and this menu provides a quick way to locate a specific instance. (Programmatically, you can update the form's **Tag** property with the name of the employee as a way to distinguish different instances of each form.) This is, of course, similar to the way you probably negotiate the many open windows in your own Visual Basic IDE.

Figure 10.9 The application has a Window menu item, allowing the operator to quickly locate a specific form among many.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Binding Controls To The Business Object

To bind the controls to the class module, I set their **DataField** properties to the names of the columns in the database. As shown in Figure 10.10, however, I left the **DataSource** property blank. In development mode, Visual Basic has no way of knowing that the class module will be a data provider as it does when a Data control is placed directly on the form. The association of the controls to the data provider has to occur when the form is instantiated:

```
Private Sub Form_Load()
Set empCLSDemo = New clsEmpDemo
Dim oText As TextBox
' Bind the textboxes to the data provider
For Each oText In Me.txtFields
  oText.DataMember = "EmpPrimary"
  Set oText.DataSource = empCLSDemo
  oText.Visible = True
Next
SetEdit False
End Sub
```

Figure 10.10 I set the control's **DataField** property but left **DataSource** and **DataMember** blank.

The **DataMember** property is set to the value of the object added to the **DataMembers** collection in the **empCLSDemo** object's **Initialize** event. Then, each control's **DataSource** property is set equal to the **empCLSDemo**

object. In a sense, the class module becomes a reusable data control—except that it has no visible components. It provides no buttons to scroll through records and so on. I added the command buttons that you can see in the figures showing the running application. These buttons all invoke methods of the **empCLSDemo** object, such as **MoveNext** and **Update**.

The **SetEdit False** line of code from the previous example calls a procedure that unlocks all but the employee ID textboxes for editing. It also disables some of the command buttons.

Relocating The Business Object

Class modules are powerful weapons in the Visual Basic developer's arsenal. Space limitations prevent me from exploring every facet of the object-oriented potential of the class module. However, a key strength is the ability to compile the class module into an ActiveX server component. When that is done, the component can literally exist almost anywhere on the network and serve as a data provider for multiple clients. The advantages of this approach are numerous. I discussed many of them in Chapter 1 in the discussion of the Microsoft services model. The goal of the services model is partitioning the application into three (and sometimes more) tiers. (The word "layer" comes to mind when discussing tiers; Microsoft actually avoids using the word layer, preferring the term partition or tier.) The client tier is that piece of the application that runs on the client PC—typically the user interface. The business tier contains the business logic and typically handles communications with the database. The data tier is the data source itself.

Before moving further, you need to understand some basic concepts:

- *Persistence* refers to whether an object's state is saved between invocations. Persistent objects "remember" their settings, such as the values of properties, by saving them to disk. This has advantages, particularly with remote objects, by eliminating the need to set many properties and thus reducing the number of calls to the object. Objects that aren't persistent—nonpersistent objects—have the advantage of being simpler to design and implement and are advantageous when there is no need to set properties.
- *Stateless* is a term often used in describing a remote object whose properties are set by calling methods of the object instead of the more traditional *object.property* syntax. As I will discuss in more detail, by calling methods that set multiple properties at once, you can minimize the number of cross-process calls.
- *Process boundary* refers to the way that the operating system segregates separate processes, where a process refers to an application, DLL, and so on. For purposes of this discussion, a process is more akin to a separate program, such as Microsoft Word. The concept of process boundaries has implications in cross-process components.

Modeling Your Application

With the Enterprise edition of Visual Basic, Microsoft bundles a tool called Visual Modeler. The tool is also integrated into Visual Studio. With this tool, you can model an application from scratch or reverse-engineer an existing application. Either way, the result is a graphical model built using a subset of *Unified Modeling Language* (UML), which shows all of the objects in your application and how they connect. The model can be viewed as a logical (business) model or a physical (component) model. Once the application is modeled, you can even generate Visual Basic or Visual C++ source code from the model. Figure 10.11 shows the Employee Class Demo application being modeled by Visual Modeler.

Figure 10.11 The Employee Class Demo project reverse-engineered into Visual Modeler.

The learning curve with Visual Modeler is fairly stiff; entire books can be written about its use. However, the benefit from learning the tool is worth the time invested, particularly for multitiered applications.

- A *cross-process component* is an executable program that makes its services available to other programs. It runs in its own process space. Using DCOM, two separate processes can communicate and share objects. Calls to a cross-process component can be expensive in terms of computer resources. The operating system (Windows) goes to a lot of trouble to keep processes separate—running in their own address spaces. Keeping processes discrete promotes the stability of the operating system because the two programs cannot clobber each other with errant calls. On the other hand, a significant amount of overhead is required to ask the operating system to make calls to other processes.
- An *in-process* component runs in the same address space as the application and is typically a DLL. Running a component as an in-process component saves a lot of overhead but means that the component cannot be shared with other processes.
- A *remote component* runs on another machine on the network. This model exacts a toll in performance because it is not only cross-process, but it also involves network traffic. On the other hand, the component runs on an entirely separate CPU from the application that is using it, which may offset—perhaps by a lot—the overhead incurred in communicating with it.
- With components, the application program or component that is calling or using the properties, methods, and events of another component is called the *client*. The component whose services are being utilized is called the *server* or, more accurately, the *ActiveX server*.

Optimizing Cross-Process Calls

As noted in the text, using cross-process calls to or between objects is inefficient. It is not possible to entirely eliminate cross-process calls, but you can take steps to reduce their impact on performance:

- If you need to set a number of properties at one time, consider creating a method where the properties are arguments to the method.
- Minimize or eliminate late binding. Early binding can reduce the overhead of making cross-procedure calls by 50 percent. To force early binding, use explicit **Set** or **Dim** statements.
- Minimize the use of nested object notation. For instance, assume **object_a** contains a reference to **object_b**. Instead of coding **object_a.object_b.property = value**, use **Set** to create an explicit reference to **object_b**: **Set myVar = object_a.object_b**. Then, you can utilize it directly: **myvar.property = value**. Nested object references are time-consuming for Visual Basic to resolve.
- Use the **With…End With** construct to minimize the number of times Visual Basic has to resolve nested objects.

- *Marshalling* is the method used to invoke the methods and properties of an out-of-process component. With in-process components, you can use the client's stack space to make the calls. With out-of-process components—that is, when crossing process boundaries—the proxy on the client "gathers" together parameters to be passed to the out-of-process component and sends them to a stub on the server component. Counter-intuitively, you are better off passing parameters **ByVal** than you are **ByRef**. With an in-process call, most VB developers use **ByRef** because it involves passing only a pointer to a value or object (as opposed to sending a copy of, say, a 2,000-byte string); however, when making cross-process boundary calls, passing **ByRef** works against you. The other object needs to then make a call back to the client to get the value of the parameter, meaning that parameters sent **ByRef** cause the process boundaries to be crossed twice instead of once.
- The concept of a *thread* is similar to the concept of a process. A thread is a separate line of communication within a process—almost like a process within a process. For instance, Windows 98 runs 32-bit apps as separate threads within the Windows Virtual Machine. (NT runs all applications as separate threads.) Threads are normally protected from each other. You can create an ActiveX server that has separate threads for each client communicating with it.
- *Apartment model threading* is illustrated by a house with separate apartments. Each thread lives in its own apartment, oblivious to what is going on in other apartments. This means that each thread has, for instance, its own copy of global data. This setup is actually the default when creating ActiveX components. Even if the component has a single thread, it resides in its own apartment. You set threading options in the Project Properties dialog box. See "Apartment-Model Threading in Visual Basic" in the Visual Basic Help file for some more details on

options and trade-offs with different approaches.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Creating The In-Process Component

To illustrate the process of separating the user interface services from the business services of the Employee Class Demo project, I separated the **clsEmpDemo.cls** module from the rest of the Visual Basic project. I created a new project, and when asked for project type, I specified ActiveX DLL. The DLL will run in the same space as the client application. I removed the default class module that Visual Basic creates and added the saved **clsEmpDemo** module.

Depending on the type of project you create, the class module will have different properties available. Because this module will be an ActiveX component, the **Instancing** property becomes available. Instancing refers to how and under what conditions objects may be created from the class. The possible values are shown next:

- **Private** means that no object may access the properties and methods of the class—which is useful only for other objects within the same component.
- **PublicNotCreateable** means that any other process can access the class but cannot actually create an instance of the class. This is only appropriate for ActiveX EXEs, and it means that the server application (the ActiveX EXE) must already be running for other processes to use objects from it.
- **MultiUse** means that multiple processes can create objects from one instance of the component. For an ActiveX EXE, objects can be supplied to multiple clients. For an ActiveX DLL, the component can provide multiple objects—but only to the one client.
- **GlobalMultiUse** is the same as **MultiUse** except that you do not have to explicitly declare an instance of the class; it happens automatically as soon as any method or property of the class is referenced.

- **SingleUse** allows any object to create objects from the class, but each object created causes a new instance of the class to be created.
- **GlobalSingleUse** is the same as **SingleUse** except that any reference to a property or method of the class causes the class to be instantiated without first explicitly creating it. Because both **SingleUse** and **GlobalSingleUse** allow multiple instances of the same process on a computer, you cannot use the values for an ActiveX DLL.

I set the **Instancing** property to **MultiUse**.

The **Persistable** property sets whether the component can be persisted as discussed previously. The possible values are **NotPersistable** and **Persistable**.

A new property is the **MTSTransactionMode**, which specifies the level of support the component provides for Microsoft Transaction Server (MTS). Other possible values, if you are placing objects on MTS, are **NoTransactions**, **RequiresTransactions**, **UsesTransactions**, and **RequiresNewTransaction**.

You really don't need to do anything else to the class project except compile it. When you do, the resulting DLL is registered as an ActiveX component, which you can use much like any other ActiveX component. I compiled my project as AXClsEmpDemo.DLL. I typed "Coriolis ActiveX DLL Emp Example" into the Project Description area of the Project Properties dialog box.

Using The In-Process Component

To use the AXClsEmpDemo.DLL component, I had to add a reference to it in my original form-based project. I actually created a new project of type Standard EXE and added the **frmClsEmpDemo** form to it. I then saved that form as **frmClsEmpDemoAX**. Next, I went to the Project References dialog and added the "Coriolis ActiveX DLL Emp Example" component that I created in the last section. Notice in Figure 10.12 that in the References dialog, the project description is used (and, if it's not entered, the project name is used), whereas in the object browser, the project name is used. Notice also that the class looks like any other component in the browser because it *is* like any other component.

Figure 10.12 Referencing the in-process component in the application.

The only other changes to the application were slight modifications in the reference to the object (mainly because I renamed it when I created a component from it):

```
Private WithEvents empCLSDemo As clsEmpDemoAX
```

At this point, you can run the application as before. If you need to debug it, you do not have access to the code internal to the component unless you first

start it in a separate instance of Visual Basic itself.

Figure 10.13 shows the running application.

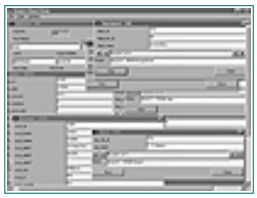

Figure 10.13 This form is using an in-process component (ActiveX server) as its data source.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Creating The Remote Business Object

In Chapters 12 through 15, I guide you through the process of creating remote objects hosted on the Internet (or your corporate intranet). The principles of the remote business object are not much different from those for a business object running locally. Remote business objects offer the advantages of code reuse, encapsulation, and application partitioning. Additionally, they can be shared by multiple clients, take advantage of the server's presumably greater resources, and more effectively manage connections to the database.

For purposes of this chapter, I built a flexible data inquiry application, shown running in Figure 10.14. Running behind the scenes is another application, a remote data server. Let's examine the remote server first.

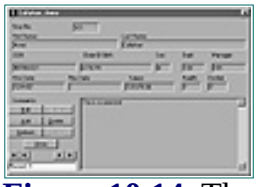

Figure 10.14 The Remote Server demo is running on a Windows 98 client connected to a Windows NT 4.0 server.

The remote server is adapted from the **VBBusObj** sample provided with MDAC and is included on the CD-ROM as VBBusObj.vbp. The guts of the application is a single class module responsible for communicating with the database. In my testing, I located and registered the module on an NT 4.0 server. The **Instancing** property is set to **GlobalMultiUse** so that any object need merely reference a method of the business object to create it. The project's only reference is to the Microsoft Active Data Objects Recordset 2.0 library.

I created a **Public Function**, shown in Listing 10.2. The purpose of the function is to return to the client the names of all of the columns in a table. I had originally sought to use the **OpenSchema** method. However, Visual Basic does not support

this method remotely, and I was forced to instead open a **Recordset**. To minimize database and network traffic, I set the **MaxRecords** property to 1. Notice the use of the **adOpen-Unspecified** and **adCmdUnspecified** arguments on the **Open** method. These arguments refer to the **CursorType** and **Options** properties. On a remote server, these are the only values you are allowed to specify.

Listing 10.2 The **GetSchema** function.

```
Public Function GetSchema(ByVal Connect As String, _
  ByVal sName As String) As ADOR.Recordset
 ' This function returns the requested schema
On Error GoTo ehGetRecordset
' ADO must be registered locally!
Dim objADORs As New ADODB.Recordset
objADORs.MaxRecords = 1
objADORs.Open sName, Connect, adOpenUnspecified, _
  adLockBatchOptimistic, adCmdUnspecified
' Set object pointer for ADOR type recordset
Set GetSchema = objADORs
Exit Function
ehGetRecordset:
Err.Raise Err.Number, Err.Source, Err.Description
End Function
```

The function returns an object of type **ADOR.Recordset**. The **ADOR** comes from the **Recordset** -only library reference. (In Chapter 7, I discussed the two ADO libraries—the Microsoft ActiveX Data Objects Recordset 2.0 library provides only the **Recordset** and **Connection** objects as opposed to the Microsoft ActiveX Data Objects 2.0 library, which provides all ADO objects.) Notice the use of the **Set** command to set the function equal to the **Recordset** returned from the **Open** method.

I will show you how this function is used when I illustrate how to create the client.

The other main function in the remote server application is **Get-Recordset**, as shown in Listing 10.3. This function is similar in functionality to the **GetSchema** function. Notice that in both functions, all arguments are passed **ByVal** instead of **ByRef**. This eliminates a trip (call) to the server from the client, as I discussed earlier in the chapter in “Optimizing Cross-Process Calls.” An examination of the module will show that the object exposes no properties, and the only items passed from process to process are done so via methods.

Listing 10.3 Function used to return a populated **Recordset** to the client.

```
Public Function GetRecordset(ByVal Connect As String, _
  ByVal SQL As String) As ADOR.Recordset
' This function returns an ADODB recordset object
On Error GoTo ehGetRecordset
' ADO must be registered locally
Dim objADORs As New ADODB.Recordset
objADORs.CursorLocation = adUseClientBatch
```

```
objADORs.Open SQL, Connect, adOpenUnspecified, _
  adLockBatchOptimistic, adCmdUnspecified
' Set object pointer for ADOR type recordset
Set GetRecordset = objADORs
Exit Function
ehGetRecordset:
  Err.Raise Err.Number, Err.Source, Err.Description
End Function
```

The remote server provides a **Test** method so those clients can quickly ascertain whether they are establishing contact with the server:

```
Public Function Test() As String
Test = "You are being heard!"
End Function
```

Although not illustrated here, the remote server also has several other methods, including one to return the machine name and one to execute action queries.

The remote server application was built and compiled as an ActiveX EXE. Once compiled, it is registered as an ActiveX component. You need to deploy it on the target server and register it there and then re-register it at the client machine so that the client can find it on the network.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Using The Remote Business Object

The RemoteSrvrClient application, also included on the CD-ROM, consists of four forms:

- **frmSrvLogIn** is used to pass a connection string, a user ID, a password, and a server address to the rest of the application. If you are running the remote server on your local machine, you can leave the server address blank.
- **mdiRemote** is an MDI form that acts as a shell to the rest of the application.
- **frmClsSrvEmp** is a form that displays employee data.
- **frmGeneral** is a form that is built on-the-fly to display whatever data the user chooses.

The project has a single standard module, which contains most of the common code.

The application references four libraries: OLE Automation, vbBusObj (which is the remote server component), Microsoft ActiveX Data Objects Recordset 2.0 library, and the Microsoft Remote Data Services 2.0 library. The last one is the critical piece, enabling communications over the network.

NOTE
The application allows you to connect to any database via any data provider. However, because the menu itself is hard-coded, you would have to dynamically alter the menu **Caption** properties to access other tables.

The forms that display data have these module-level references:

```
Private ds As New RDS.DataSpace
Private bo As Object
Private objadors As Object
```

The **ds** object comes from the Remote Data Services library. **bo** and **objadors** are both

defined as generic objects, meaning that they will be late bound.

When the application is running, the user can select the Data menu. It has two choices: PreBuilt and On The Fly. Both lead to cascading menus. The PreBuilt menu has only one choice, Employee, which creates a form hard-coded to display employee data. The other cascading menu lists all of the tables in the database by name.

If a user clicks, for example, the Customer menu choice, the code shown next is invoked:

```
Private Sub mnuDataRec_Click(index As Integer)
Dim f As New frmGeneral
Screen.MousePointer = vbHourglass
BldForm f, mnuDataRec(index).Caption
f.txtGetRecordset = "SELECT * FROM " & f.Tag
Screen.MousePointer = vbDefault
End Sub
```

A new instance of form **frmGeneral** is created, and a procedure, **BldForm**, is then called, passing the form and the menu item's **Caption** property as arguments. The latter contains the name of the table that will be accessed.

The **BldForm** procedure is in the standard module because the **frmClsSrvEmp** form also uses it. It is shown in Listing 10.4. The procedure creates a basic **SELECT** statement based on the table name passed to it. It then creates an instance of the remote server component using the **CreateObject** function, passing it the name of the component and the name of the server. Notice that **CreateObject** is a method of the **ds** object. Once a reference is obtained by the server and stored in **bo**, a **Recordset** is created using the **GetSchema** method I discussed earlier. Most of the code is spent building the form. The table names are stored in the form's **Tag** property for later use. The **DataField** property of each textbox on the form is set to the name of each **Field** object returned in the **Recordset**. The textbox is then made visible (the form is built with all of the textboxes invisible). Finally, the form is resized to show only the used textbox controls.

Listing 10.4 The procedure to dynamically generate a data form.

```
Public Sub BldForm(f As Form, tname As String)
Dim iIndex As Integer
Dim sCmd As String
sCmd = "Select * from " & tname
Set bo = ds.CreateObject("VbBusObj.VbBusObjCls", sServer)
Set objadors = bo.GetSchema(sConnect, sCmd)
With f
  .Tag = tname
  .Caption = tname
  For iIndex = 0 To objadors.Fields.Count - 1
    .lblField(iIndex).Caption = objadors.Fields(iIndex).Name
    .lblField(iIndex).Visible = True
    .txtFields(iIndex).Text = objadors.Fields(iIndex).Name
    .txtFields(iIndex).Visible = True
    .txtFields(iIndex).DataField =
objadors.Fields(iIndex).Name
  Next
```

```
  .Height = (iIndex * 360) + 1850
  .Show
End With
End Sub
```

Note that a more elegant solution is to create the textboxes as needed. I chose this approach for simplicity, although it limits the number of columns that can be displayed.

When the procedure completes, the form is shown with all of the **Text** properties set equal to the names of the table's columns. A textbox on the form is set equal to a basic SQL statement, such as **SELECT * FROM CUSTOMER**. The user can change the statement if he or she wants. Pressing the Run command button or any of the scroll buttons will cause the form to fill up with data, which is done with the code shown in Listing 10.5. The key line of code is the call to **GetRecordset**, which is in the standard module.

Listing 10.5 This procedure populates the form with data.

```
Private Sub cmdGetRecordset_Click()
Dim oText As TextBox

' Populate form with data from Recordset
On Error GoTo ehcmdGetRecordset_Click
MousePointer = vbHourglass
If Trim(txtGetRecordset) <> "" Then
  Set objadors = GetRecordset(txtGetRecordset.Text, Me)
Else
  Set objadors = GetRecordset(Tag, Me)
End If
For Each oText In Me.txtFields
  oText = ""
Next
If objadors.RecordCount > 0 Then
  bData = True
Else
  bData = False
End If
ShowData Me, objadors
MousePointer = vbNormal
Exit Sub
ehcmdGetRecordset_Click:
  MousePointer = vbNormal
  MsgBox Err.Description
  bData = False
End Sub
```

GetRecordset is shown in Listing 10.6. It is similar to the **GetSchema** method illustrated earlier.

Listing 10.6 The procedure to retrieve the **Recordset** from the remote server.

```
Public Function GetRecordset(tname As String, f As Form) _
```

```
  As Recordset
Dim sCmd As String
On Error GoTo errHandler
If InStr(1, Trim(tname), " ") = 0 Then
  sCmd = "Select * From " & tname
Else
  sCmd = tname
End If
Set bo = ds.CreateObject(sObj, sServer)
Set objadors = bo.GetRecordset(sConnect, sCmd)
Set GetRecordset = objadors
Exit Function
errHandler:
  Screen.MousePointer = vbNormal
  MsgBox Err.Description
  bData = False
End Function
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

When control returns to the form, the **ShowData** method is invoked from the standard module, as shown in Listing 10.7. It essentially moves through each textbox in the **txtFields** array and compares its **DataField** property to get the **Value** of the corresponding **Field** object and displays it. A test is performed to detect and handle the case where the value is **Null**, and the error handler checks for the case where the **DataField** has no corresponding **Field** object. The **lblStatus** control is updated to show the record number, and the form's **Caption** property is updated to show the form's current contents.

Listing 10.7 The **ShowData** procedure is called repeatedly to display data on the form passed.

```
Public Sub ShowData(f As Form, objadors As Recordset)
Dim oText As TextBox
Dim bOkay As Boolean
bOkay = True
If objadors.RecordCount > 0 Then bData = True
If bData = False Then Exit Sub
On Error GoTo errHandler

For Each oText In f.txtFields
  If IsNull(objadors.Fields(oText.DataField)) Then
    oText.Text = ""
  Else
    oText.Text = objadors.Fields(oText.DataField)
  End If
  If bOkay = False Then
    oText.Text = "*Not Found!*"
    bOkay = True
  End If
Next
```

```
f.lblStatus = "Record " & _
  objadors.AbsolutePosition & _
  " of " & _
  objadors.RecordCount
f.Caption = f.Tag & " - " & f.txtFields(0).Text

Exit Sub
errHandler:
If Err.Number = 3265 Then
  ' Field not found
  bOkay = False
  Resume Next
Else
  MsgBox Err.Description
End If

End Sub
```

The final code of interest is the mechanism to scroll through the records. The form has an array of **CmdMove** buttons. When a button is pressed, a call is made to the **ScrollData** procedure in the general module, as shown in Listing 10.8. The form checks whether the **Recordset** object is equal to **Nothing**. If it is, a call is first made to **GetRecordset**. The **ScrollData Sub** performs basic error checking, moves through the referenced **Recordset** object, and then calls the **ShowData** method.

Listing 10.8 This procedure scrolls through the **Recordset**.

```
Public Sub scrolldata(f As Form, index As Integer, _
  objadors As Recordset)
If objadors.RecordCount < 1 Then Exit Sub
Select Case index
  Case 0  ' Move first
    objadors.MoveFirst
  Case 1  ' Move previous
    If objadors.AbsolutePosition > 1 Then
      objadors.MovePrevious
    End If
  Case 2  ' Move next
    If objadors.AbsolutePosition < _
      objadors.RecordCount Then
      objadors.MoveNext
    End If
  Case 3  ' Move last
    objadors.MoveLast
End Select
ShowData f, objadors

End Sub
```

All of the code in both the remote server and the client is built to minimize the amount of network traffic and also to minimize the number of out-of-process calls.

Where To Go From Here

What I have attempted to do in this chapter is introduce the business object, describe how to use the object in Visual Basic applications, and provide some practical hints on improving component performance. I introduced a simple but illustrative remote server and client. Chapters 12 through 15 expand and build on these concepts considerably in Internet- or intranet-based applications. Internet-based components are not the only tools for building a distributed client/server application, of course. You can, as shown in this chapter, deploy the same objects on your NT server just as well as you can deploy them on MTS, IIS, and so on.

Microsoft has included full-blown client and middle-tier (business-tier) applications in the MSADC directory (generally found under Program Files\Common Files\System\MSADC\Samples\Selector), which you can examine in depth. The remote server component in this chapter was adapted to some extent from the middle-tier component in the Selector directory.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Previous | Table of Contents | Next

Chapter 11
Visual Basic 6 Advanced Database Topics

Key Topics:

- Data validation and integrity constraints
- Data dictionary
- Stored procedures and triggers
- Generating primary keys
- Transaction and concurrency management

There are two keys to a successful client/server application: database techniques and component architecture. In terms of component architecture, we seek to maximize performance and reliability. This is done largely through the creation of reusable objects, which I introduced in Chapter 10, and the location of those objects, which will be the subject of the remainder of this book.

In terms of the database, we seek to maximize performance and data integrity. This is accomplished through intelligent database design, which I discussed in Chapters 2 and 3, and sensible database handling techniques, which we will concentrate on in this chapter.

One of the edicts of medicine is to “First, do no harm.” The analogy in client/server is to “First, do not corrupt the data.” Corruption of data involves one or both of two possibilities: inconsistent data, such as an order without a valid customer; or, incorrect values stored on a record. The first situation can happen if we update a database by adding a new order but, for some reason, the customer add fails. The second situation might occur because two users change the same record at the same time.

I will spend a good amount of time in this chapter discussing data integrity constraints and differing approaches for accomplishing it. In doing so, I will introduce the concept of

the data dictionary. I will then lead you through the creation and use of stored procedures and triggers using data validation as a key objective. I will then spend the remainder of this chapter discussing the concepts of transaction management and concurrency as well as a couple other issues that I have found to be troublesome to client/server developers.

Data Validation

Ensuring the integrity of your data is paramount. If your order entry application allows an order to be placed without a valid customer or an address to have an invalid state, you stand to lose a lot of money and business. In this section, I will walk you through some of the issues relating to data validation.

Encoding And Decoding

Certain types of data lend themselves to *encoding*. Encoding is not encrypting as its name implies—rather, it is the abbreviation of common values. Consider the **Emp_Gender** column on the **Employee** table. Employees can be female or male. Storing the fully expanded values wastes space, data entry time, and network traffic utilization. So, it is more typical to encode the values as F or M. The same is true for many other types of data, such as state and provinces. You and I know that F means Female and CA means California. But, the computer doesn't know that. So, we need to provide a facility to *decode* values as well. This process expands the encoded value to its full description. This is done via one or more lookup tables where we store in one column the encoded value and in another column the decoded value, as seen next:

```
St_ID  St_Name
MA     Massachusetts
MD     Maryland
ME     Maine
```

Such lookup tables then become a convenient means to validate data as well, as seen in this example:

```
' Note that this would be a very inefficient method to
' validate. You would normally already have a Recordset open
Set arsState = New Recordset
arsState.Open "Select st_name from state where " & _
  "st_ID = '" & txtState.text & "'", acon
If arsState.RecordCount = 0 then
  MsgBox "State Code Invalid!"
  txtState.SetFocus
End If
```

Referential Integrity

In Chapter 2, I discussed the concept of the referential integrity mechanism and showed you how to create the DDL necessary to create it. In general, this is the preferred method because it involves the least usage of network, database, and client resources. Referential integrity involves the defining of a database integrity constraint, as shown next:

```
Alter Table Customer
Add Constraint fk_valid_state
Foreign Key (cust_st)
References State (st_id)
```

This constraint prevents the entry of a state code that is not already on the **State** table. If such an entry is made, the database generates an error and refuses to insert or update the record. That is all well and good where it is feasible.

What if, though, a country code is recorded on the customer record? While MA might be a valid state code in the US, it is not for Canada. So, you might alter the **State** table to have a compound primary key, as shown next:

```
Create Table State
  (St_Ctry_ID  Char(3) Not Null,
   St_ID Char(2) Not Null,
   St_Name VarChar (21),
   Constraint PK_Cntry_St
   Primary Key (St_Ctry_ID, St_ID) )
```

Then, your constraint on the **Customer** table would look more like this:

```
Alter Table Customer
Add Constraint fk_valid_ctry_state
Foreign Key (cust_ctry, cust_st)
References State (st_ctry_id, st_id)
```

In this way, the combination of country and state (or province, as the case may be) is validated. Of course, that does not mean that the country code entered into the **State** table is valid. You might consider having a separate **Country** table as well:

```
Create Table Country
  (Ctry_ID  Char(3) Not Null,
   Ctry_Name Char(30),
   Constraint PK_Ctry
   Primary Key (Ctry_ID) )
```

And you would add a constraint to the **State** table as such:

```
Alter Table State
Add Contrant fk_valid_ctry
Foreign Key (st_ctry_id)
References Country (ctry_id)
```

You can see where we easily start getting a lot of tables involved, which has the effect of dragging down database performance. I will talk about that in a moment. But, for the time being, we also have the problem of those countries that don't happen to have states. The **Customer** table verifies whether the country/state combination is valid via a foreign key constraint to the **State** table. The **State** table validates the country via a foreign key constraint to the **Country** table. If the customer happens to be from a country with no states (or provinces), then the country will not be listed on the **State** table. There are a

number of solutions.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

You can make the **cust_ctry** and **cust_st** columns on the **Customer** table nullable, which creates a *conditional* constraint. A conditional constraint is one where the data must be valid if present. That solves the problem of a customer legitimately not having a state or province (perhaps the customer is from Aruba or Bermuda). But, it does not help to enforce that the country is, in fact, valid. So, we might create a second foreign key on the **Customer** table like this:

```
Alter Table Customer
Add Constraint FK_valid_cust_ctry
Foreign Key (cust_ctry)
References Country (ctry_ID)
```

Again, this is not enough. Because we have made the **cust_ctry** column nullable, the relationship is not automatically enforced. We cannot make **cust_st** nullable and leave **cust_ctry** as **NOT NULL**. For example, assume the customer is from Bermuda and that Bermuda's **ctry_id** is **BER**. By leaving **cust_ctry** as **NOT NULL**, the foreign key relationship is no longer conditional. The database would attempt to find a row on the **State** table where **st_ctry_id = "BER"** and **st_id** is **Null**. Putting such a row on the **State** table for every country would be awkward. But, we can still trick the database. We can add a constraint on the **cust_ctry** column to enforce that the length of the column must be greater than 0. With a value present, the database then validates the country against the **Country** table while not validating against the **State** table if the state field is not entered. The ending (simplified) **CREATE** statements look something like Listing 11.1.

Listing 11.1 SQL CREATE statements with conditional referential integrity.

```
DROP TABLE COUNTRY ;

CREATE TABLE COUNTRY
 (CTRY_ID  CHAR(3) NOT NULL,
```

```
  CTRY_NAME VARCHAR (35) NOT NULL,
  CONSTRAINT PK_CTRY_ID
  PRIMARY KEY (CTRY_ID) );

INSERT INTO COUNTRY VALUES
 ('USA','United States') ;
INSERT INTO COUNTRY VALUES
 ('CAN','Canada') ;
INSERT INTO COUNTRY VALUES
 ('BER','Bermuda') ;

DROP TABLE STATE ;

CREATE TABLE STATE
 (ST_ID  CHAR(2)  NOT NULL,
  ST_CTRY_ID CHAR (3) NOT NULL,
  ST_NAME  VARCHAR (21) NOT NULL,
  CONSTRAINT PK_ST_ID
  PRIMARY KEY (ST_ID, ST_CTRY_ID) );

ALTER TABLE STATE
ADD CONSTRAINT FK_VALID_CTRY
FOREIGN KEY (ST_CTRY_ID)
REFERENCES COUNTRY (CTRY_ID) ;

INSERT INTO STATE VALUES
 ('MA', 'USA', 'Massachusetts') ;
INSERT INTO STATE VALUES
 ('RI', 'USA', 'Rhode Island') ;
INSERT INTO STATE VALUES
 ('ON', 'CAN', 'Ontario') ;
INSERT INTO STATE VALUES
 ('BC', 'CAN', 'British Columbia') ;

DROP TABLE CUSTOMER  ;

CREATE TABLE CUSTOMER
 (CUST_NO  NUMERIC (9) DEFAULT AUTOINCREMENT,
  CUST_LNAME CHAR (21),
  CUST_FNAME CHAR (15),
  CUST_ST CHAR (2),
  CUST_CTRY CHAR (3) DEFAULT 'USA' ,
  CUST_TIMESTAMP DATE DEFAULT CURRENT TIMESTAMP,
  CONSTRAINT PK_CUST_NO
  PRIMARY KEY (CUST_NO) ) ;

ALTER TABLE CUSTOMER
ADD CONSTRAINT FK_VALID_ST
FOREIGN KEY (CUST_ST, CUST_CTRY)
REFERENCES STATE (ST_ID, ST_CTRY_ID) ;
```

```
ALTER TABLE CUSTOMER
ADD CONSTRAINT FK_VALID_CTRY
FOREIGN KEY (CUST_CTRY)
REFERENCES COUNTRY (CTRY_ID) ;

ALTER TABLE CUSTOMER
MODIFY CUST_CTRY
CHECK (CUST_CTRY  > ' ') ;

INSERT INTO CUSTOMER
(CUST_LNAME, CUST_FNAME, CUST_ST, CUST_CTRY)
VALUES
('Smith', 'John', 'MA', 'USA') ;

INSERT INTO CUSTOMER
(CUST_LNAME, CUST_FNAME, CUST_ST)
VALUES
('Smith', 'Jane', 'MA') ;

INSERT INTO CUSTOMER
(CUST_LNAME, CUST_FNAME, CUST_ST, CUST_CTRY)
VALUES
('Brown', 'Barbara', NULL, 'BER') ;

INSERT INTO CUSTOMER
(CUST_LNAME, CUST_FNAME, CUST_ST, CUST_CTRY)
VALUES
('Brown', 'Bob', 'BC', 'CAN') ;
```

Notice in the SQL **CREATE** statement for the **Customer** table, I have specified an **AutoIncrement** default for the primary key and I have also added a **TimeStamp** column. I will discuss those issues later in the chapter.

The Centralized Data Dictionary Lookup Table

Having too many discrete lookup tables begins to make your design look like an octopus with arms “hanging” off of each master table. If you are displaying data from a single table, you could easily end up joining four, six, or more lookup tables as well. The impact on the database is tremendous.

What I have done to get around this problem is to place all my validation data into a single lookup table with three columns: **dd_type** (the **dd** is for data dictionary) designates the type of lookup, such as state codes or country codes; **dd_val** is the encoded value, such as **RI** for Rhode Island or **BER** for Bermuda; and **dd_desc** is the decoded value (**Rhode Island** or **Bermuda**). The SQL **CREATE** statement looks something like the following:

```
CREATE TABLE Data_Dic
 (dd_type  CHAR (6)  NOT NULL,
  dd_val   CHAR (32) NOT NULL,
```

```
  dd_desc  VARCHAR (64),
  CONSTRAINT PK_LookUp
  Primary Key (dd_type, dd_val) )
```

Some sample data in such a table is shown next:

```
DD_Type    DD_Val    DD_Desc
-------    ------    -----------------
CC         BER       Bermuda
CC         CAN       Canada
CC         USA       United States
SC         AZ        Arizona
SC         MA        Massachusetts
SC         RI        Rhode Island
VRT        CC        Country Code
VRT        SC        State Code
VRT        VRT       Valid Record Type
```

The key to the table's usage is the **VRT** record, which specifies the valid record types and what they are used for.

To use the table in a customer listing, code an SQL **SELECT** statement, something like the following:

```
SELECT cust_no AS "Cust No",
               cust_fname AS "First",
               cust_lname As "Last",
               sc.dd_desc As "State",
               cc.dd_desc AS "Country"
FROM  customer, data_dic cc, data_dic sc
WHERE cust_st  *= sc.dd_val AND sc.dd_type = 'SC'
  AND cust_ctry = cc.dd_val AND cc.dd_type = 'CC'
```

The query retrieves customer information, including the decoded state and country values. This particular example uses Transact-SQL dialect to perform a left outer join. The results are as follows:

```
Cust No  First    Last    State             Country
-------  ------   ------  ----------------  ------------
1        John     Smith   Massachusetts     United States
2        Jane     Smith   Massachusetts     United States
3        Barbara  Brown   (NULL)            Bermuda
4        Bob      Brown   British Columbia  Canada
```

Now, we have placed all of our validation into a single table eliminating the need to join many tables to perform simple reporting. (We are back in the same boat as we were before in that the countries and states are not cross-referenced, but I will come back to that problem in a moment.)

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Now that we have a convenient place from which to decode values (and potentially to encode as well), how do we use the table for validation? It is no longer possible to have referential integrity constraints, because there is no unique key for the database to validate against. The primary key of **Data_Dic** is **dd_type** and **dd_val** combined. dBase developers will remember that you could validate by hard-coding a portion of the other table's key—if SQL allowed this, it would look something like:

```
ALTER TABLE CUSTOMER
ADD CONSTRAINT FK_Val_State
FOREIGN KEY ('SC', cust_st)
REFERENCES DATA_DIC (dd_type, dd_val)
```

The **('SC', cust_st)** would be needed, of course, so that you could validate **cust_st** against **dd_val** where **dd_type** = **'SC'**.

So, what is left to accomplish the validation? The obvious answer is to open record sets from VB to validate our data:

```
Set arLookup A= New Recordset
Set arCust = New Recordset
' acon is an active connection
arLookup.Open "Select * from data_dic Where dd_type = " & _
  "'CC' And dd_val = " & txtCountry.text, acon
If arLookup.BOF = True and arLookup.EOF = True Then
  ' No records
  MsgBox "Invalid Country!"
  txtCountry.SetFocus
  Exit Sub
Else
  ' proceed with update
End If
```

It would appear that even though we have saved the joining of many tables on data retrieval, we have created a situation where each update involves potentially many queries to validate the data. In the preceding example, we would also have to validate state, postal code, and perhaps other fields.

In the next two sections, I will show you two ways to get around that problem.

Other Uses For The Data Dictionary Table

You can use the data dictionary table for many purposes. Besides the obvious uses, such as storing valid state and country codes, I have used it to store tax tables, currency exchange data, general ledger account types, calendars, and so on.

Sometimes, values vary by an independent factor, such as division. For instance, your company may have several divisions (or subsidiaries) that keep their general ledgers independently. In one division, account 1400 might be Fixed Assets while in another division it might be Accumulated Depreciation. In these cases, I have used a four-column table with the first column being **dd_div**. Any data that is common to all divisions has a **dd_div** code of **"ALL"**.

Keeping Common Data In Memory

The typical application often validates against a common set of information. It is a drain on resources to repeatedly request the same information from the database. Depending on the type and amount of data, you might be better off loading the data into memory—perhaps as an array—and accessing it that way.

Consider two common needs in an order-entry application—the state lookup table and the item lookup table. Both of these types of data tend to be accessed a lot, so they represent good areas to explore for minimizing database and network traffic. But, one lends itself to pre-loading into memory, and the other doesn't. Why?

The state lookup table stores the encoded state values (AZ for Arizona, CA for California, and so on), which is data not subject to change. In other words, the data is static. The item lookup table, on the other hand, stores data that is not static—prices can change, items can be added, and so on. As such, the data cannot simply be stored in memory—the application needs to access the database to ensure it has the most current version available.

If a table has many thousands of rows, you probably don't want to load it into memory. First, you will bring client performance to a screeching halt, because all of that data takes up valuable resources (RAM) and will almost certainly be swapped in and out to disk. Second, each time a client logs into an application, a tremendous amount of data is going to be pulled over the network, thus defeating the whole purpose of minimizing network traffic.

The common lookup table that I discussed in the prior section is an example of a table you probably don't want to load into memory, both because it will likely become very large and because it potentially may have some dynamic (non-static) data in it. That doesn't preclude you, however, from loading specific subsets of the data into memory.

Assume you have many different forms that validate state codes. Rather than opening a **Recordset** object in each one, a better solution would be to open the **Recordset** as

Public when the application opens:

```
' Assumes a standard module
Public arState As Recordset
Public arCon As Connection
Sub Main ( )
  ' Display a splash screen to keep them
  ' amused while loading data
  frmSplash.Show
  ' Open the connection object here
  ' Now open the Recordset
  Set arState = New Recordset
  arState.Open "Select * From data_dic Where " & _
    "dd_type = 'CC' Order By dd_val", acon
  UnLoad frmSplash
End Sub
```

A better solution than loading up the **Recordset** and keeping it in memory would be to save its contents to an array and close the **Recordset** to free up database resources. However, you would then have to write your own routines to search the array for the value(s) that you are looking for.

The best solution of all is to not perform the validation at the client—instead, let the database perform the validation using triggers.

The Trigger As A Data Validation Tool

The trigger is a special stored procedure that is automatically invoked by the RDBMS as a result of a predefined action, such as **UPDATE** or **DELETE**. Using triggers, you can have the database enforce your data validation rules. The advantage to this approach is that it is more flexible than referential and check integrity constraints, it is all done on the database, and every program that accesses the database gains the advantage of the data validation rules already being defined. Further, if a change is needed, the change is done in one place—at the database. All programs immediately gain the benefits of the changes.

In the next section, I will discuss stored procedures and triggers, using data validation for many of the examples.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Previous | Table of Contents | Next

Stored Procedures And Triggers

Throughout this book, I repeatedly allude to the most important aspect of client/server development (outside of integrity of the data itself): minimizing network traffic. In Chapter 10, for instance, the design for the remote, out-of-process server was very different than the design for a local, in-process server. In that example (for the remote server), we stacked calls by passing property references as method arguments. With the database, the issue is perhaps even more critical, because the nature of client/server itself is the manipulation of remotely stored data. To the extent that you can minimize the number of times that data has to move between your application and the database server, your application will benefit from increased performance levels.

In the previous sections, we discussed different aspects of data validation. In some of the examples, we sent requests to the server asking if certain data (such as state codes) was valid. The database then had to send the answers back. Only after several roundtrips were we ready to send the update request to the database—hardly an efficient way of doing things.

Stored procedures are small programs that sit on the database performing specific tasks, such as data validation. Whereas stored procedures have to be explicitly called, triggers are essentially stored procedures that occur automatically as the result of a predefined action. You can think of a stored procedure as analogous to a Visual Basic general procedure where a trigger is analogous to an event procedure. The event procedure also occurs as the result of a predefined event, such as a button click.

Stored Procedures On Differing RDBMSs

Unfortunately, stored procedures and triggers are the area where the different dialects of SQL have significant differences. Although they are similar, you

cannot take an Oracle stored procedure to Microsoft SQL Server and expect it to run without some modifications. As discussed in Chapters 2 and 3, Oracle uses a dialect of SQL known as PL/SQL. Sybase uses Transact SQL (T-SQL). Microsoft's SQL Server, traditionally also based on T-SQL, has been moving away to its own dialect of SQL. Still, the differences are outweighed by the similarities. For instance, the following Oracle stored procedure is used to give employees a raise. The code calls the function passing the department number and the percent of the raise—all employees in that department will be affected:

```
CREATE PROCEDURE emp_raise
  (deptno IN NUMBER,
   raise_pct IN NUMBER)
AS
BEGIN
  UPDATE employee
  SET emp_sal = emp_sal * (1 + raise_pct)
  WHERE emp_dept = deptno;
END
```

TIP
Advantages To Stored Procedures
By moving code to the database, we realize three distinct advantages:

- *Code reusability*—All code modules use the same database procedures.
- *Minimization of network traffic*—The server and client only have to communicate in the event of an error, so network traffic is dramatically reduced.
- *Precompilation*—Stored procedures (and triggers) are precompiled on the server and so run more efficiently than does dynamic SQL.

The equivalent Microsoft SQL Server stored procedure takes this syntax:

```
CREATE PROCEDURE emp_raise
  (@deptno int,
   @raise_pct real)
AS
BEGIN
  UPDATE employee
  SET emp_sal = emp_sal * (1 + raise_pct)
  WHERE emp_dept = deptno;
END
```

In this example, the differences between SQL Server and Oracle are small—mainly how parameters are declared. The larger differences show up in the body of the stored procedure—those lines between the **BEGIN** and **END** statements. There can even be differences between different versions of the same RDBMS—vendors tend to enrich the language as the products evolve.

To use any of the examples I present here, you will have to create the stored procedure on your database while making any necessary changes in syntax. See your own database's documentation for additional details. I will provide

some examples in different syntaxes as I go along. This book's CD-ROM includes the text of the stored procedures (and triggers later in this chapter) that I use for examples.

The Lookup Stored Procedure

To illustrate the creation and use of stored procedures, I will start with some modifications to the **Data_Dic** (data dictionary) table created earlier in the chapter. First, let's create a simple stored procedure that we can call whenever we need to retrieve an un-encoded value from the data dictionary using SQL Anywhere syntax:

```
CREATE PROCEDURE sp_lookup
  (IN lu_type CHAR (3), IN lu_val CHAR(32),
   OUT lu_desc CHAR(64))
BEGIN
  SELECT dd_desc INTO lu_desc
  FROM data_dic
  WHERE dd_type = lu_type
  AND   dd_val = lu_val;
END
```

The Oracle equivalent is shown next:

```
DROP PROCEDURE sp_lookup ;
CREATE PROCEDURE sp_lookup
  (lu_type IN data_dic.dd_type%TYPE,
   lu_val IN data_dic.dd_val%TYPE,
   lu_desc OUT data_dic.dd_desc%TYPE ) AS
BEGIN
   SELECT dd_desc
     INTO lu_desc
   FROM data_dic
   WHERE dd_type = lu_type
   AND   dd_val = lu_val ;
END
```

In the Oracle stored procedure, I used a special notation for data types—**%TYPE**—referencing the data type of the underlying column on the **Data_Dic** column. The advantage of this is that if the data type changes—perhaps making a column larger—the stored procedure does not need to be revised.

Using The Stored Procedure In Visual Basic

You can use the stored procedure within your VB project easily. In an ADO environment where only one record is being returned, it is not necessary to open a **Recordset** object. Figure 11.1 shows an application that opens the **Customer** table and displays the full text of the customer's state and country. While the application could have been done using a join between the **Customer** and **Data_Dic** tables (and would probably be a little more

efficient), I chose to use the **sp_lookup** stored procedure to get the un-encoded values from the data dictionary. The ADO objects created in the application are shown next:

```
Private WithEvents acon As Connection
Private acmd As Command
Private acmdDataDic As Command
Private WithEvents ars As Recordset
```

Figure 11.1 This application uses a stored procedure to retrieve values from the data dictionary.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The application issues a connect to the database when the form opens, as shown next. Note that I am using the **stest** user id in these examples for the sake of simplicity. If you choose to work with the **coriolis** user id, the application works identically.

```
Private Sub Form_Load()
Screen.MousePointer = vbHourglass
Set acon = New Connection
acon.CursorLocation = adUseClient
acon.Open ("PROVIDER=MSDASQL;dsn=Coriolis VB Example;" & _
  "uid=stest;pwd=stest;")
End Sub
```

The connection is established asynchronously. When complete, the **ConnectionComplete** event occurs, and the record set is opened. Previous to that, **acmdDataDic** is created as a reference to the **sp_lookup** stored procedure:

```
Private Sub acon_ConnectComplete(ByVal pError As _
  ADODB.Error, adStatus As ADODB.EventStatusEnum, _
  ByVal pConnection As ADODB.Connection)
' Connect complete - open record set
Set acmd = New Command
acmd.CommandText = "Select * from customer"
acmd.ActiveConnection = acon
Set ars = New Recordset
Set acmdDataDic = New Command
Set acmdDataDic.ActiveConnection = acon
acmdDataDic.CommandText = "sp_lookup"
acmdDataDic.CommandType = adCmdStoredProc
acmdDataDic.CommandTimeout = 15
ars.Open acmd, , adOpenDynamic, adLockOptimistic, adCmdText
Screen.MousePointer = vbDefault
```

```
' Command object for the stored procedure.
End Sub
```

TIP
Asynchronous Timing
In the code listing, I deliberately rearranged the order of creating objects in order to highlight the timing issues that can be a real thorn in the side of a developer who is moving from a synchronous data model. ADO will go out and "look" at the stored procedure when the command type property is set. If I had opened the record set first, ADO would almost certainly have waited for the execution to complete before taking a look at the stored procedure. Then, the **MoveComplete** event would trigger, taking precedence over the code remaining in the **ConnectComplete** event. The **MoveComplete** event would then make a call to **ShowRecs**, which would invoke the stored procedure, which would not even as yet exist.

Moral: Create all your objects prior to executing or opening them.

As you will see in a moment, it is not necessary to declare any **Parameter** objects. As soon as the **CommandType** property is set, ADO goes to the data provider and ascertains the stored procedure's parameters. They are set in the same order as defined in the database—the two input parameters are first and the output parameter is last.

Notice also that I create the **Command** object for the stored procedure prior to opening the record set. Working in an asynchronous environment is a bit more complex than working in a synchronous environment. Specifically, without getting the stored procedure created prior to opening the record set, I can run into timing problems if the query completes before **cmdDataDic** is created.

When **ars** is opened, the **MoveComplete** event is triggered as shown:

```
Private Sub ars_MoveComplete(ByVal adReason As _
  ADODB.EventReasonEnum, ByVal pError As ADODB.Error, _
  adStatus As ADODB.EventStatusEnum, ByVal pRecordset As _
  ADODB.Recordset)
txtRecord = "Record " & _
  ars.AbsolutePosition & " of " & _
  ars.RecordCount
ShowRecs
End Sub
```

The RecordCount Property

Note that not all data providers support the **RecordCount** property at all times. If not supported, **RecordCount** is usually equal to -1. It would be a good idea to verify whether the **RecordCount** property is valid before displaying it.

Also, consider that most data providers need to return the entire result set before they can populate the **RecordCount** property. If you are returning more than a couple dozen records, you may want to avoid referencing this property to avoid the performance penalty inherent in unnecessarily returning the entire result set.

As a final note, in my testing, I noted times when ADO became confused as to what the current record number (as returned by the **AbsolutePosition** property) was. This seemed to occur in the development environment while debugging and presented

itself as record numbers below zero (that is, **AbsolutePosition = -14**). If you notice this happening in your application during production, the workaround that I found is to perform a **MoveFirst** followed by a move back to the current record. I was able to first save the current record in a bookmark, which I found curious (because neither VB nor ADO seemed to really know what the current record was). If you save a record location in a bookmark and the bookmark then seems invalid, consider simply using a **Find** on the primary key for the current record, to get yourself back to the current record.

The **ShowRecs** procedure populates the first few textbox controls with the customer number and name and then calls the **GetDataDic** procedure to retrieve the state and country names, as shown next. Note that because the **cust_st** column is allowed to be **Null**, I have to handle that before calling the stored procedure:

```
Private Sub ShowRecs()
Dim iCtr As Integer
Dim sSt As String
For iCtr = 0 To 2
  txtFields(iCtr) = ars.Fields(iCtr).Value
Next
' Get state
If IsNull(ars!cust_st) Then
  sSt = ""
Else
  sSt = RTrim$(ars!cust_st)
End If
txtFields(3) = GetDataDic("CSR", _
  RTrim$(ars!cust_ctry) & sSt)
' Get country value
txtFields(4) = GetDataDic("CC", _
  ars!cust_ctry)
End Sub
```

Finally, the **GetDataDic** procedure is very simple—much more so than calling a parameterized query, as we did in Chapter 7. Because ADO has already looked at the stored procedure, it is not necessary to create **Parameter** objects or to append them to the **acmdDataDic** object's **Parameters** collection. Normally, ADO can determine the direction and type of the parameters also, so those properties (**Direction** and **Type**) do not need to be set. Since this is a non-record-returning query (only the one output value is being returned), it is not necessary to open a **Recordset**. Assign values to the input parameters, execute the command, and then read the output parameter:

```
Private Function GetDataDic(ddType As String, ddval As _
  String) As String
Dim arsDataDic As New Recordset
GetDataDic = ""
acmdDataDic.Parameters(0) = ddType
acmdDataDic.Parameters(1) = ddval
acmdDataDic.Execute
GetDataDic = acmdDataDic.Parameters(2)
```

```
End Function
```

When ADO dynamically creates the **Parameter** objects, their **Name** properties are set to their names in the stored procedures. In this example, the first parameter has a **Name** property of **lu_type** (the name of the corresponding column in the database).

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

About Triggers

As I mentioned earlier, a database trigger is much like a stored procedure except that where a stored procedure must be explicitly called, a trigger is invoked automatically as the result of an action on the database. To illustrate the use of triggers and expand on the concept of stored procedures, let's enhance the data validation aspects of the **Customer** table and data dictionary.

Recall that the data dictionary has a record type of **VRT**, for valid record types. The **VRT** records tell us what each record type is used for and also should restrict the records that are entered into the data dictionary to valid entries. In other words, I should not be able to enter **TX** record types if there is no such thing as a **TX** record type. So, the data dictionary requires a **VRT** record for each record type maintained. This is a sort of implied referential integrity constraint. To enforce this rule, I created a trigger, **tr_dd_validate**, as shown next:

```
CREATE TRIGGER tr_dd_validate
   BEFORE INSERT, UPDATE ON data_dic
REFERENCING NEW AS new_data
FOR EACH ROW
BEGIN
   DECLARE data_type CHAR (3) ;
   DECLARE data_val CHAR (32) ;
   DECLARE data_desc CHAR (64) ;
   DECLARE data_count INTEGER ;
   /* Is record type valid? */
   CALL sp_lookup ('VRT', new_data.dd_type, data_desc) ;
   SELECT COUNT (data_desc) INTO data_count ;
   If data_count = 0 THEN
     RAISERROR 99999
      'Invalid Record Type For Data Dictionary';
```

```
      RETURN ;
    END IF ;
    /* If country state, verify country */
    IF new_data.dd_type = 'CSR' THEN
      SET data_val = LEFT (new_data.dd_val || '   ', 3) ;
      CALL sp_lookup ('CC', data_val, data_desc) ;
      SELECT COUNT (data_desc) INTO data_count ;
      IF data_count = 0 THEN
           RAISERROR 99999 'No Such Country Record Defined' ;
      END IF;
  END IF;
END
```

As you can see, the trigger is defined to fire automatically any time the **data_dic** table is updated or has a row inserted into it. The third line of the trigger uses the **REFERENCING NEW** clause to create a reference to the new data being updated or inserted. You can also use the **REFERENCING OLD** clause to create a reference to the existing data prior to its being updated (which I will do in another trigger momentarily). The trigger declares some work variables and then calls the **sp_lookup** stored procedure that we examined in the last section. It passes a record type of **VRT**, the new record type, and a holder variable in which to return the description of the record if found. Assume you were adding a new country code (**CC**) record. As soon as you (or your application) sends the new record to the database, this trigger automatically checks to see if there is a corresponding **VRT** record, thus preventing any invalid record types from getting into the data dictionary. A few lines later, the **RAISERROR** function is invoked if the record was not found. The 99999 is a user-defined error return code. You can use any return code not already used by the database (check your database documentation for more details). The text of the error along with the error code is passed back to the application. When you display errors, this error is returned like any other database error.

The second part of the trigger handles **CSR** records. What I did here was to create a special kind of record that handles country/state cross-references. I eliminated the **SC** record type, which, you may recall, contained all of the state and province names. The **CSR** record contains values like **CANBC** for British Columbia. The trigger enforces the rule that, before a country/state cross-reference record can be defined, the country must already be defined in a **CC** record. In other words, I can't add "Montana" to the data dictionary until I first add "United States".

To enforce this referential integrity, I created another trigger, **tr_cust_validate**, shown next:

```
DROP TRIGGER tr_cust_validate ;
CREATE TRIGGER tr_cust_validate
   BEFORE INSERT, UPDATE ON customer
REFERENCING NEW AS new_cust
FOR EACH ROW
BEGIN
   DECLARE data_st CHAR (2) ;
   DECLARE data_ctry CHAR (5) ;
   DECLARE data_desc CHAR (64) ;
   DECLARE data_count INTEGER ;
```

```
   /* Is country valid? */
   IF new_cust.cust_ctry IS NULL THEN
     RAISERROR 99999 'Country can not be null!' ;
     RETURN ;
   END IF ;
   CALL sp_lookup ('CC', new_cust.cust_ctry, data_desc) ;
   SELECT COUNT (data_desc) INTO data_count ;
   IF data_count = 0 THEN
     RAISERROR 99999 'Invalid Country:'
       || new_cust.cust_ctry;
     RETURN ;
   END IF ;
   /* Is state valid? */
   IF new_cust.cust_st IS NULL THEN
     SET data_ctry = rtrim(new_cust.cust_ctry)  ;
     CALL sp_lookup ('CSR', data_ctry, data_desc) ;
     SELECT COUNT (data_desc) INTO data_count ;
     If data_count = 0 THEN
       RAISERROR 99999
         'Missing CSR Record On Data Dictionary:'
         || data_ctry;
       RETURN ;
     END IF ;
   END IF ;
   SET data_ctry = rtrim(new_cust.cust_ctry) ||
     new_cust.cust_st ;
   CALL sp_lookup ('CSR', data_ctry, data_desc) ;
   SELECT COUNT (data_desc) INTO data_count ;
   IF data_count = 0 THEN
     RAISERROR 99999 'Invalid State:' ||  new_cust.cust_st ;
   END IF ;
END
```

As you walk through the code, you might notice that the body of the trigger is pretty much the same as the body of a stored procedure. Also, it is not all that much different than a Visual Basic program. The syntax is a little less forgiving, but, once you get used to it, it's not bad.

tr_cust_validate verifies that the country on the customer record is valid. If not, an error is raised and a message is sent back to the application. As soon as SQL encounters the **RAISERROR** command, the update or insert is cancelled. The **RETURN** statement tells the trigger not to continue processing after the error, much like an **Exit Sub** or **Exit Function** command in Visual Basic.

The trigger then validates the country/state combination, or, if there is no state, it validates the proper construction of the data dictionary's **CSR** key.

So, at this point, it is pretty much impossible to enter a customer with an invalid country or state. Better yet, the validation occurs automatically on the server. But, what's to stop someone from deleting a record from the data dictionary that is being referenced? If someone were to delete a **"USA"** country code record while customers were on the database using that record, the database would be corrupt.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

To handle this, I wrote a generic trigger to handle a wide variety of deletions from the data dictionary. What I wanted to do was to avoid having to write a trigger to handle each and every record type in the data dictionary. To support this, I created two new record types: **RTN**, for referenced table name, and **RTC**, for referenced table column. If you want the generic validation routine to be active for a given record type, then you would add one of each of these two records for the given record type. For instance, if I wanted to prevent someone from deleting a country code (**CC**) record that was being used, I would add two records to the data dictionary:

```
dd_type  dd_val  dd_desc
RTC      CC      cust_ctry
RTN      CC      customer
```

These records will tell our new trigger that if someone attempts to delete a **CC** record, it should make sure that that the **cust_ctry** column on the **Customer** table does not reference that **CC** record. Let's look at the trigger:

```
drop trigger tr_dd_validate_delete ;
CREATE TRIGGER tr_dd_validate_delete
  BEFORE DELETE ON data_dic
REFERENCING OLD AS old_data
FOR EACH ROW
BEGIN
  DECLARE data_table CHAR (32) ;
  DECLARE data_column CHAR (32) ;
  DECLARE data_desc CHAR (64) ;
  DECLARE data_count INTEGER ;
  /* Is this a VRT record? */
  IF old_data.dd_type = 'VRT' THEN
    /* Don't let them delete a referenced record! */
   SELECT COUNT(*) INTO data_count
```

```
   FROM data_dic
   WHERE dd_type = old_data.dd_val ;
   IF data_count > 0 THEN
     RAISERROR 99999 'Cannot Delete Referenced VRT Record!' ;
     RETURN ;
   END IF ;
  END IF ;

  /* check for dependencies */
  /* Must maintain RTN and RTC records for this to work! */
  /* Get dependent table name */
  SELECT dd_desc INTO data_table
  FROM data_dic
  WHERE dd_type = 'RTN' AND dd_val = old_data.dd_type ;

  /* Get dependent column name */
  SELECT dd_desc INTO data_column
  FROM data_dic
  WHERE dd_type = 'RTC' AND dd_val = old_data.dd_type ;

  /* Build and run query */
  EXECUTE IMMEDIATE 'SELECT COUNT(*) INTO data_count FROM '
    || data_table ||
  ' WHERE ' || data_column || ' = ' || char(39) ||
     old_data.dd_val || char(39) ;
  /* Run it */

  IF data_count > 0 THEN
    RAISERROR 99999 'Cannot Delete - ' || old_data.dd_val  ||
      ' present on ' || data_table || ' table ' ;
  END IF ;
END
```

This trigger is declared to occur whenever an attempt is made to delete from the table, whereas our earlier examples looked at inserts and updates. Also, this trigger uses the **REFERENCING OLD** clause to obtain a reference to data on the table.

The trigger first makes sure that no one deletes a **VRT** record that is being referenced by another record type. But, the meat of the trigger is the section that follows. If the record type that is being deleted is not **VRT**, then a check is made to see if there are **RTN** and **RTC** records.

Because the trigger cannot know ahead of time what record type might be deleted, it must *dynamically* create an SQL statement. I had some problems troubleshooting this trigger to make this dynamic statement generation portable across SQL platforms (test on your own database, of course). However, as written, the dynamically generated statement should work just fine on most RDBMSs. The statement is run using the **EXECUTE IMMEDIATE** command. If your application were to delete the **USA** country code record, the statement generated would check to see if there were any customers whose country was **“USA”**. If so, the database would issue the error message that you can see in Figure 11.2.

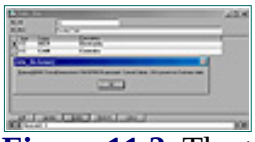

Figure 11.2 The trigger on the database prevents the deletion of a country record if existing customers reference that record.

As a quick note, the statement generated is not particularly efficient. It performs a count of all the customers from that country. If the table was large, the statement would probably yield unacceptable performance. My original intent was to open a cursor and fetch a single row rather than count all the records. While that worked just fine with Oracle, I found that Transact-SQL insisted that all declarations be at the top of the body of the trigger. Unfortunately, to open a cursor, it must first be declared. I found different ways to work around the issue on different platforms, none of them portable enough across all the platforms to present here. Because I had to first determine the existence and values of the **RTN** and **RTC** records, most of the workarounds that I devised involved calling stored procedures, as I did in the first trigger example.

A generalized example of cursor usage is shown next. Consult your database documentation for any specific details:

```
DECLARE data_lname CHAR (21) ;
DECLARE data_fname CHAR (15) ;
DECLARE err_notfound EXCEPTION
    FOR SQLSTATE '02000' ;
DECLARE c1 CURSOR FOR
  'SELECT cust_lname, cust_fname ' ||
  'FROM customer ORDER BY cust_lname, cust_fname' ;
/* Open the cursor */
OPEN c1 ;
/* loop through it */
LOOP
  /* Fetch each record */
  FETCH NEXT c1 INTO data_lname, data_fname ;
  /* No more records? */
  IF err_notfound THEN LEAVE custloop ;
END LOOP custloop ;
/* Close the cursor */
CLOSE c1 ;
```

Maintaining The Data Dictionary

Figure 11.2 showed an application from this book's CD-ROM that maintains the data dictionary. It is a low-tech solution, leaving most of the validation work to the database. Figure 11.3 shows a new country being added. The top two textboxes show the current type of records being displayed in the grid control. If the user adds a new record, the record type is filled into the first column of the new row. Feel free to expand upon it in your own applications.

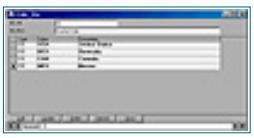

Figure 11.3 The Data Dictionary Maintenance application.

Uses For Triggers

Your job as a client/server developer is to place as much processing on the database server as possible (without overrunning it, of course). I routinely have triggers do such chores as verify ZIP codes whenever customer, employee, or vendor addresses are maintained. Often, applications require an audit trail or transaction table reflecting who did what to whom. Any time a customer record is maintained, for instance, you might want to have a trigger write out to another table the operator, date, and type of change performed, as well as the old data values.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Generating Primary Keys

Every row on the database needs to have a primary key. Although primary keys can be made up of any number of columns and those columns can be any data type, most tables will have a one-column numeric key. Those tables that have more than one column are most often child tables whose key is composed, in part, of their parent table(s) key(s).

Generating a unique, numeric primary key is most often a matter of assigning numbers sequentially. As simple as that sounds, this was a large thorn in my side earlier in my career. In the mid-1980s, I worked on large DB/2 databases with tables whose row counts sometimes numbered into the millions. We didn't have such things as auto-incrementing columns then, so we had to do it the old-fashioned way—painfully. The general gist was to perform a **SELECT MAX** on the primary key column to find the highest used number and add one to it. On the larger tables, this sometimes took several minutes. Users, being unreasonable souls, judged five-minute response times to be unacceptable. So, we developed clever little workarounds, which were never quite perfect but somehow got the job done.

Fortunately, the problem is not so bad today. All database vendors provide some way to generate a sequential number for a primary key (or whatever other purpose you deem). Normally, you specify an option at table-create time. For example, with Sybase and Microsoft SQL Server, you specify the keyword **IDENTITY** when you define the column:

```
CREATE TABLE CUSTOMER
  (Cust_No Int IDENTITY,
   Cust_Lname CHAR (21) , ...
```

SQL Anywhere uses a convention similar to Access with the **AUTOINCREMENT** default (in Access, **AUTOINCREMENT** is an actual

data type—its underlying data type is **LONG**):

```
CREATE TABLE CUSTOMER
  (Cust_No Int DEFAULT AUTOINCREMENT,
   Cust_LName CHAR(21), ...
```

Oracle does not have an equivalent facility (to increment a primary key) specifically bound to a column. Instead, it uses an independent database object, **SEQUENCE**:

```
CREATE SEQUENCE seq_Cust
  CACHE 200
  STARTWITH 100
  INCREMENT BY 10
```

The Oracle **SEQUENCE** maintains a certain number of new numbers, waiting to be used, in memory. The number of items cached is determined by the **CACHE** qualifier (the default is 15). My recommendation is to have enough cache to last about five minutes. For example, if you have 50 users adding an average of 1 customer per minute, a cache size of 250 is reasonable. The Oracle **SEQUENCE** allows you to specify a starting value as well as an increment value. With SQL Server and Sybase, the equivalent would be shown as:

```
(Cust_No Int IDENTITY (100, 10), ...
```

None of the RDBMSs, except for Oracle, cycle back to the beginning when all the numbers are used up. You can tune the cycling behavior in Oracle by specifying **NOCYCLE** (do not start all over again when the numbers are used up) and the **MAXVALUE** or **NOMAXVALUE** clauses.

Except for Oracle, the numbers are created automatically whenever you add a new row to a table. If there is an error, the number does not get reused. For instance, assume that you do an **INSERT** and the next customer number is 220. For whatever reason, the **INSERT** fails. The next insert will be number 221 (depending on whether you alter the default increment of 1). If you issue a **ROLLBACK**, however, the next number will also be restored.

Oracle's **SEQUENCE** has two methods: **CURRENTVAL** and **NEXTVAL**. **NEXTVAL** is used to get the next available number. **CURRENTVAL** is a reference to the number currently being used in the *current database session*. In other words, if you make a reference to **NEXTVAL** and get the number 220, the **CURRENTVAL** in your database session becomes 220. If another users makes a reference to **CURRENTVAL**, the user will not get 220—that person will get whatever number was generated when he or she invoked **NEXTVAL**.

To add a row to the **Customer** table using the **SEQUENCE** object, you have to explicitly reference it. Because the object is not connected to the **Customer** table in any way, the database doesn't know that you are using it to generate customer numbers:

```
INSERT INTO CUSTOMER VALUES
(seq_Cust.NEXTVAL, 'Smith', ...)
```

Although this is a bit more work than you have to do with, say, SQL Server, it also offers somewhat more flexibility. Assume that after you add a new customer, you want to add an order for that customer. For the **ord_cust_no** field, you can simply reference the **CURRVAL** method of the **SEQUENCE**, as shown next:

```
' seq_ord is an existing sequence
INSERT INTO ORDERS VALUES
(seq_ord.NEXTVAL, SYSDATE, seq_cust.CURRVAL ... )
```

The preceding example assumes, of course, that the first three columns on the Orders table are **ord_no**, **ord_date**, and **ord_cust_no**.

I have used the **INT** data type in my examples so far. If you anticipate generating a very large number of keys, or if you anticipate that the value of those keys will be very large, I recommend using a data type of **DECIMAL (38)** (SQL Server), **NUMERIC (38)** (Oracle), or a similar data type. This allows for whole numbers with up to 38 digits.

Result Set Size

Through the last several chapters, I have periodically stressed the importance of reducing to a minimum the amount of data that is retrieved from the database. This helps in the performance of the network, the client, and the database server. Nevertheless, the most common problem I see in beginning and even intermediate client/server development is the temptation to bring an entire table down from the server to the client.

Let’s assume that you have a customer maintenance screen. The lure of presenting all of those customers in a list box (or similar) so that the user can choose which he or she wants to maintain is enticing to even the most seasoned veteran. Don’t do it! Depending on the size of the records, don’t bring down more than a few dozen to a few hundred records. Think of when you go out to Amazon.com to do some book shopping. The server doesn’t send you the entire list of 2 million books—you’d probably sue if they did! Instead, you enter search criteria. Perhaps the name of your favorite author or a subject area. If there are more than 100 books that match your search criteria, Amazon sends them to you 100 at a time. You find the one you want and select it.

Do the same for your users. Find the ways that users locate customers: last name, telephone number, or whatever else is natural to your users and your organization. Place proper indexes on those columns and then bring back a screenfull or a few screenfulls of records for the users to scroll through.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The Nature Of Transactions

The database measures activity (updates, deletes, and inserts) in what are known as *LUWs* (Logical Units of Work). The term LUW is synonymous with a *transaction*—a transaction is a logical unit of work. It is possible to work outside the scope of explicit transactions. When you log onto a database server and start updating records, you implicitly begin a transaction. When you log off, the transaction is ended. Normally, all your work is then *committed* (made permanent).

Managing transactions through your VB program is more work than simply letting the database do it for you implicitly. Imagine that your application is adding a new order. The customer orders 10 screwdrivers. The application generates these changes to the database:

- Inserts a new order
- Inserts a new line item
- Updates the inventory

Now, imagine that the two inserts worked but that the inventory update failed. The customer would get his or her 10 screwdrivers, but your inventory would reflect 10 more in stock than there actually are. The data has been corrupted.

Transactions allow you to handle this situation with a little more grace. This pseudocode illustrates the correct way to handle the order:

```
BeginTrans
Insert new order
If insert worked then
  Insert new line item
    If insert worked then
      Update inventory
      If update worked then
```

```
        CommitTrans
      Else
         RollbackTrans
      End If
    Else
      RollbackTrans
    End If
Else
  RollbackTrans
End If
```

By verifying that each action on the database was successful, we have created an all-or-nothing situation. Recall the term *logical unit of work*. The two inserts and the update together constitute a unit of work that we have logically grouped together. We started by explicitly beginning a transaction. We verified that there were no errors and then committed the work. If there was an error, we then rolled back any changes. Either way, we ended the transaction (a transaction is always ended with a **CommitTrans** or **RollbackTrans**). The beginning and end of a transaction have implications in terms of database performance.

Transactions And Performance

While a transaction is active, you are occupying resources on the server. Inasmuch as resources are finite, it makes sense that you want to begin and end your transaction as quickly as possible. Consider an application with 10,000 users. One after another, each user starts a transaction and then gets up to take a coffee break. Resources on the server will be quickly overrun. Sooner or later, the database will run out of resources and crash, or it will refuse to allow users to log on, or it will simply start terminating transactions and rolling back changes. You, the poor not-so-innocent developer, will be very unpopular. Thus, we want to keep our transactions as short as possible.

Likewise, transactions often need to lock rows of data. When your application locks a row, no other application can access that row. Other applications have to wait for your application to release the lock. Obviously, this can adversely impact performance as well. Because each lock requires a certain amount of memory on the server, too many active locks can adversely impact performance when the server runs short of resources.

Ill-behaved applications that don't quickly release locks on data can lead to a condition known as *lock escalation* and even another condition known as *deadlock*. Assume your application uses pessimistic record locking. Each time a record is edited, it is immediately locked—the lock isn't released until the record is updated. Depending on how the database does locking, other records on the same data page on the server may also be locked. If you have a few hundred users, each maintaining records, the database may be asked to maintain several hundred or more locks simultaneously. This is an expensive operation (in terms of resource usage on the computer) and quickly leads to performance degradation. You may also get into a lock-escalation situation. When the database begins to run short on resources, it will attempt to "consolidate" the locks by escalating them. If it is doing row-level locking, it may escalate them to page-level locking or even table-level locking.

Even worse is the deadlock situation (sometimes called a *deadly embrace*) where two

processes completely block the other with no hope for resolution. Assume application A first updates Smith (placing a lock on him) and then wants to update his account. Application B is updating Smith's account (and places a lock on it) and now wants to update Smith. Application A is locked out until application B releases the lock on the account. But, application B cannot release the lock until application A releases the lock on the customer. The only way to solve this unfortunate situation is for the DBA to terminate one of the transactions. (This can be done manually or through a timeout value.)

A facet of transaction management that is even more crucial than performance is the integrity of the data itself.

Transactions, Locking, And Data Integrity

The nature of the transaction is defined by the concept of data integrity. Consider an application that maintains a general ledger. An entry is placed to debit Cash for $100 and to credit Accounts Receivable $100. There are two updates, and the transaction is not complete until both updates have been made and verified. Using pseudocode, this transaction might look like this:

```
Debit Cash $100
If error then
   Display error
   ' Undo any changes
   Rollback
   Exit Sub
Else
   Credit Accounts Receivable $100
   If error then
      Display error
      ' Undo all changes including debit to cash
      Rollback
      Exit Sub
   Else
      ' Make the changes permanent and end the transaction
      Commit Work
   End If
End If
```

Again, as can be seen in the example, transactions are an all-or-nothing proposition. If the second update fails, we want to undo all changes so that we don't leave the books out of balance.

But there is more to data integrity than making sure that all records are updated. There is the problem of data concurrency.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Concurrency

A few pages back, I discussed the problems of lock escalations with the use of pessimistic locking. The answer, of course, is to use optimistic locking—where the rows aren't locked until they are actually being updated. However, just as premature locking is bad for performance (don't they make a drug for that problem now?), it is great for data integrity. Consider the following scenario:

You go to a bank machine (ATM) and look up your checking balance. The machine says $500. The balance record is loaded into that machine's memory. At that moment, your spouse goes to a second ATM and asks to see the balance. That ATM also reports a $500 balance. The balance record is loaded into the second machine's memory. Since the records aren't locked, neither machine is aware of the other.

You press a lot of keys and swear a little bit. Assuming your bankcard doesn't get eaten, the machine gives you $500. The record in the first machine now reflects the new balance of $0 (about what I am used to), and the database is updated to reflect the fact that you are broke. But, wait! The ATM that your spouse is using still thinks there is $500 in the account. Being somewhat more frugal than you, your spouse withdraws only $300. The ATM dutifully deducts $300 from $500, computes the new balance of $200, and updates the bank record.

So, you started the day with $500, withdrew $800, and still have $200 left. That sounds like a 100 percent rate of return in a single day. That's not too bad for a struggling VB developer, but it's not too good for the bank. Concurrency control seeks to solve this problem by somehow notifying a second process trying to update a record that the record has been changed. In other words, the ATM that your spouse was visiting should have been alerted that the $500 balance was updated by another user. At best, it should then have not given out

any money. At worst, it should at least have computed a balance of negative $300.

This notice to a transaction that the underlying data has changed can also impact—you guessed it—performance. In Chapters 5, 6, and 7, I used some examples of transaction management and concurrency handling in illustrating the use of DAO, RDO, and ADO. I showed you how you can be notified if a record that has been retrieved from the database has since been altered. By definition, to automate the process of determining whether the underlying record has been changed involves frequent communication with the database server, which negatively impacts performance at the client, server, and network levels.

RDBMSs And Row Locking

When it comes to record locking, not all databases are created equally. For some, it might not even matter.

When you lock a row on an Oracle database, only that row is locked. On the other hand, when you lock a row on a Sybase database, every other row on the same *page* of data is also locked. (Databases organize rows into pages. If a record is small, a page might contain many rows.) Oracle, then, performs what is known as *row-level locking* whereas Sybase performs *page-level locking*. Microsoft SQL Server 6.0 performed page-level locking. Version 6.5 performed *pseudo-row-level locking*, whereby the row immediately before and after the row to be updated was also locked. SQL Server 7.0 performs *dynamic row-level locking*, which is to say that it uses an algorithm to determine whether to lock an individual row only or also the rows before and after.

In a very high-volume, transaction-oriented environment, the locking level could be an issue. Obviously, if a database locks 20 rows in addition to the one being updated, the chance of another application being locked out is 20 times greater than if only one row is being locked. On the other hand, if the nature of the updates is that they are random (occurring all over the table instead of being concentrated in just a few pages) or if the transaction volume is not that high, the nature of the locking mechanism might not be that critical. Of more importance might be the duration of the transaction. If a page-level locking database takes 1/100 of a second to update a row and a row-level locking database takes 1/10 of a second to update a row, you might be better off with the former.

As one might speculate, all of these issues—database resources and concurrency handling—are part of the overall issue of transaction management. The overall goals are:

- Ensure that all related updates of the database are handled as an integrated whole.
- Ensure that consistency of data is maintained.

You accomplish these goals by ensuring that your transaction is both complete and of as short a duration as possible, by appropriate record locking and update query tuning.

Optimistic Vs. Pessimistic Locking

Is the glass half-full or half-empty? Pessimistic locking implies preparing for the worst in terms of ensuring that the underlying data does not change unexpectedly. It locks a record as soon as it is edited and does not release the lock until the record has been updated. Optimistic locking takes the view that either no one will change the underlying data, or that if they do, you will be able to safely detect and accommodate the change. The safest—and easiest—tactic is to simply perform pessimistic locking. For an application with a small number of users, this may well suffice. For a larger application, you are pretty much guaranteeing a poor performance model.

Because the database is ultimately used to do locking, pessimistic locking is not available unless you use a server-side cursor model (**CursorLocation = adUseServer**). The locking itself is set with the **LockType** property. Use **adLockPessimistic**, **adLockOptimistic**, or **adLockBatchOptimistic**. The default, **adLockReadOnly**, is not truly a database lock—it merely means that the record set is not updateable, rendering the issue of locking level moot.

Assuming that you are going to use an optimistic locking model, then the issues that face you are whether to perform batch processing, how to determine if the underlying data has changed, and what to do about it if data has changed.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The advantage of batch processing is that it lessens the workload on the server and ultimately reduces network traffic because there are less roundtrip communications between server and client. The disadvantage is that, because you are waiting before sending updates to the server (instead of sending them as soon as changes are made), there is a higher possibility of the underlying data being changed. To update records in batch mode, use the **UpdateBatch** method of the record set. Use the **Filter** and **Status** properties (as I showed you in Chapter 7) to find any records with conflicts—that is, records whose underlying data has changed. Examine the **Errors** collection to determine if there was a problem. You can iterate through the **Fields** collection of a record set to see exactly what values changed.

The following example shows the **UpdateBatch** method being used to send all cached changes to the database. The **MarshallOptions** property is used to send only modified records back (this really only has implications when communicating with a middle tier or perhaps a Web server). The **Filter** property is then set to show an example of how to determine how many changes are pending. Next, the update is performed. The **adAffectAll** argument specifies that all records, even those that do not meet the current **Filter** criteria, should be updated to the database:

```
arsEmp.MarshallOptions = adMarshallModifiedOnly
arsEmp.Filter = adFilterPendingRecords
MsgBox "There are " & arsEmp.Recordcount & _
  "  changes pending."
arsEmp.Filter = adFilterNone
arsEmp.UpdateBatch adAffectAll
```

Next, I will show you how to iterate through all records for which there were conflicts and determine what fields on the underlying records were changed. The **Filter** property is set to only “see” those records for which there was a conflict. I iterate through the **Fields** collection of each one comparing the **UnderlyingValue** property (that is, the value of the field as it currently exists on the database) to the **OriginalValue** field (the value of the field when it was retrieved from the database). I also show the value of the field as it

exists in the database (because your application might have changed it as well):

```
Dim vField As Field
With arsEmp
  .Filter = adFilterConflictingRecords
  .MoveFirst
  Do Until .EOF
    For Each vField in .Fields
      If vField.OriginalValue <> vField.UnderlyingValue Then
        MsgBox "Conflict Detected." & vbcr & _
          "Field: " & vField.Name & vbcr & _
          "Value when retrieved: " & _
          vField.OriginalValue & vbcr & _
          "Current Value: " & vField.Value & vbcr & _
          "Changed Value: " & vField.UnderlyingValue
      End If
    Next
    .MoveNext
  Loop
End With
```

TIP
Dirty Data
You will occasionally run into database theory texts that talk about data being *dirty* or *clean*. All jokes aside (and many come to mind), dirty data is data held in memory that no longer reflects what is actually stored on the database. When a record has **Field** objects with **UnderlyingValue** properties that are different than their **OriginalValue** properties, the data is said to be dirty.

You can quickly set the record set to reflect all changes made using the **Resync** method. This is not the same as re-executing the command that created the record set. The field values will be updated to reflect changes made only to those records already in the record set. Records added since the record set was created will not be reflected. If a record has been deleted, then an error is generated.

Transaction Scope And Batch Updating

There is a basic dichotomy between the goals of batch updating and transaction management. Transactions seek to group mutually depending database updates into a single unit of work so that, if one fails, they all fail. This may be incompatible with some approaches to batch updating, and you will want to think through the consequences of your choices.

Consider that you are, perhaps, updating various employees on the database. You change John’s home address, Cindy’s salary, and Bill’s job title. You then issue an **UpdateBatch** method. If all the updates succeed, fine. But, if they don’t, should you perform a **Rollback**? Is Cindy’s salary in any way connected to John’s home address? Generally, the answer is probably no. So, you probably want to **Commit** the changes to the database even though not all were successful.

If, on the other hand, you changed Cindy’s job title to Manager and changed John’s manager to be Cindy, you might have to adopt a different strategy. Assume that the

update to Cindy failed but the update to John succeeded. You do not want that change to be made permanent, because you now have John assigned to a manager who doesn't exist. Ain't life grand?

The reason I discuss this is to point out how carefully you will want to consider whether or not to use batch updates. As you can see, they tend to cluster together as a transaction when, in fact, they may not really constitute a logical unit of work.

Dirty Data And Batch Updating

Another reason to pause before making a decision to perform batch updating is the likelihood of dirty data. If the underlying database has a lot of activity against it, you increase the likelihood of conflicts when you go to perform your batch update. This can cause aggravation for both you and your users, and, if the application ends up spending time dealing with the record conflicts, you might lose any efficiencies that the batch processing was supposed to provide.

When To Use Batch Updating

My advice is to use batch updating mainly when the underlying database is not likely to cause dirty data problems. If you have a record set that maintains related tables, such as **Orders** and **Line_Item**, then you might want to use batch updating on it as well as a *convenience* to logically group together the updates to **Orders** and **Line_Item**. However, I would also perform an **UpdateBatch** whenever activity to any given order is complete. In other words, don't batch together updates against multiple orders.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

How Does ADO Handle Concurrency? When Is Concurrency Violated?

DAO and RDO allow you to fine-tune your SQL **WHERE** clause on updates so that you have control over conflict detection and, ultimately, concurrency management. In RDO, you use the **UpdateCriteria** property (see Chapter 6).

There are several ways to format the **WHERE** clause in an update. Assume that you have changed employee 100's salary from $40,000 to $50,000:

- *Primary key*—In RDO, this is the **rdKey** option (**UpdateCriteria = rdKey**). This is the loosest control in that it will not cause any conflicts to be detected unless the original record itself was actually deleted.

```
UPDATE employee SET emp_salary = 50000 WHERE emp_no = 100;
```

- *Updated columns*—In RDO, this is the **rdKeyAndModified** option. This method detects any collision between changes you have made and changes that another user has made. In other words, if another user had changed the salary from $40,000 to $55,000, then the update would fail. However, if another user changed the employee's status to "DEAD", the change would still succeed. The nature of your data dictates whether that is okay. (I consulted for a large, national professional organization who shall remain nameless to prevent lawsuits. They had three categories of dead: Possibly Dead; Reported Dead; and Confirmed Dead. While there, a member called complaining that he had been declared "Confirmed Dead" and he vigorously protested the classification. No, I don't make that stuff up.)

```
UPDATE employee SET emp_salary = 50000 WHERE emp_no = 100
  AND emp_salary = 40000
```

- *All columns*—This option is the safest of all, because it guarantees that if *any* change has been made to the underlying database, the update will fail. You can then determine, based on the nature of what was changed, whether to force the update anyway. However, because the generated update statements are so long, the client, network, and server have to work harder.

```
UPDATE employee SET emp_salary = 50000 WHERE
  emp_no = 100 AND emp_lname = 'Brown' AND
  emp_fname = 'Barbara' AND ...
```

- *Primary key and timestamp*—This option, **rdKeyAndTimestamp** in RDO, is just as safe as all columns because the row's timestamp column is modified any time the row is modified, but it is much less resource intensive. It requires that a **TIMESTAMP** column be defined for the table and that the column be included in the result set.

```
UPDATE employee SET emp_salary = 50000
WHERE emp_no = 100 AND emp_timestamp = ...
```

Unfortunately, Microsoft has not yet built into ADO similar **WHERE** clause tuning techniques. To detect concurrency problems, it uses all columns in the **WHERE** clause. Thus, concurrency is said to be violated under ADO when any underlying value has changed—even if, to you, it is not important. Perhaps the capability to tune the **WHERE** clause will be added in a future release of ADO.

TIP
To See The ADO Statements Generated
You can observe the statements that ADO sends to an ODBC data provider by turning on the SQL Trace facilities. Go to the Control Panel, and select the ODBC Administrator applet. Select the Tracing tab, and select the Start Tracing Now button. All SQL activities are written to the log file that you specify, which you can then open in WordPad. Note that this places a tremendous burden on the ODBC manager, so you should use this option only when needed. When done, select Stop Tracing Now.

Where To Go From Here

In this chapter, I have introduced and refined various intermediate to advanced database subjects. I highly recommend learning and using stored procedures and triggers as a matter of course rather than relying on client-side code. You will want to pick up a text that is specific to your RDBMS.

I also attempted to recap discussions of transaction management and concurrency handling procedures discussed in Chapters 5, 6, 7, 9, and 10. For details of specific syntax, refer to Chapter 5 for DAO, Chapter 6 for RDO, and Chapter 7 for ADO.

A couple of texts that I like that you may want to consider are:

- Gray, J. N. and Andreas Reuter: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers 1993. ISBN: 1-55860-190-2
- Hackathorn, Richard D.: *Enterprise Database Connectivity*. Wiley & Sons 1993. ISBN: 0-47157-802-9

While neither is Visual Basic or RDBMS specific, each discusses strategies and concepts in the enterprise client/server model.

For SQL Server development, I also like Bill Vaughn's VB/SQL Server tome (Vaughn, William R.: *Hitchhiker's Guide To Visual Basic & SQL Server*, Microsoft Press, 1997. ISBN: 1-57231-567-9). I have the fifth edition, which is current through Visual Basic 5 and SQL Server 6.5, but an updated edition should be available by the time you read this.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Part III
Visual Basic 6 And The Internet

Chapter 12
The ABCs Of XML

Key Topics:

- Regaining context with XML
- Understanding the XML ActiveX control
- Describing your documents with DTDs
- Building XML applications

The Extensible Markup Language (XML) has gone from being one of thousands of fairly obscure acronyms to one of the hot test uses of the letter X outside of Sculley and Mulder's paranoia. A lot of this has to do with the realization by companies such as Microsoft that one of the biggest problems with information transfer today comes from the difficulty in specifying context.

Regaining Context With XML

To understand why specifying context is a problem, it's important to know that HTML was originally designed as a contextual scheme for classifying documents. **<H1>** was the primary header, **<H2>** the secondary, and **<EM>** the emphasis within text. Blocks of text were contained within paragraph tags, with each paragraph serving a distinct unit of thought. In theory, you could

understand the context of the page by sticking to the notion that a tag's appearance was less important than its meaning, an approach that is consistent with a completely text-based browser such as Lynx.

Of course, this concept became immaterial once people wanted to do more with Web pages than display articles about quantum effects within high-energy particle collisions. Designers became far more interested in the markup aspect of HTML. As the browser wars heated up, Web developers invented all kinds of tags for pulling off specialized functions, such as the egregious **<BLINK>**. A **<BLINK>** tag contains no contextual information (save perhaps that the designer is a twit). One or two such tags were probably fairly harmless, but as the faint call for a more graphically oriented page turned into a roar, other tags such as **<FONT>** appeared. Because any header tag could be replaced with a properly designed **FONT** tag, the underlying structure and meaning of a given page disappeared into a sea of **FONT** tags.

Because a network with billions of documents that contain little-to-no contextual information will eventually become impenetrable, the W3C has worked to create two complementary technologies—Cascading Style Sheets (CSS) for presentation and XML for page context. Style sheets separate the presentation layer from the data in a document, with some interesting consequences. If you can create a structure for representing the data that is separate from how that information is displayed, then the same data can be shown in a wide variety of formats.

Internet Explorer 5 takes this notion literally: Beyond just changing the color of an element, you can use Dynamic HTML (DHTML) behaviors to display data in forms as diverse as a table and a pie chart by simply switching the descriptor for a given element. In and of itself, this is mind-boggling because it makes the presentation of data highly fluid. Anyone who's ever had to display a data form in Visual Basic in more than one mode can appreciate how difficult that is to do with traditional development tools.

By adopting XML, which is more a *coding convention* than a *language* per se, as the data layer, Web developers can provide a context upon which to hang the presentation layer while simultaneously creating a data format that is completely portable because it carries its context within itself. This portability is one of the reasons that an XML-based "world" makes so much sense: As long as your XML document is properly formed, any XML parser can read it. To be sure, you might need to wrap the XML parser's API set in containing classes for a more complex application (hey, Web developers aren't going to be out of a job any time soon), but the data format can be read universally. This feature becomes crucial for applications that may not necessarily be connected to a server at all times and can even be facilitated by utilities that convert SQL record sets into XML code for offline storage on a client's machine.

A second advantage in using XML comes from the rich structure that such a document can contain. An XML document is hierarchical, a structure more reminiscent of programming classes than SQL output. Indeed, although it is certainly possible to use traditional database methods to output structured data, such calls are fairly expensive to make frequently, and they are even more

difficult to update. With XML, the process is much easier—so easy in fact that it is often more convenient to keep data in an XML object than attempt to map it to external classes for processing. This is the approach that I take in this chapter.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Structuring XML

If you are contemplating moving to XML from HTML, and you are dreading the need to learn dozens if not hundreds of new tags, terms, and attributes, put your mind at ease. The only tags that you need to learn are those you create yourself. The reason is simple: For the most part, XML is just a convention for writing out data, and it structurally resembles HTML quite closely. Like its antecedent SGML, XML can get fairly complex, but for most of the uses where Visual Basic programmers or Web developers need XML, the syntax is straightforward. The example in Listing 12.1 provides most of the features of a “simple” XML file.

Listing 12.1 An XML weather report.

```
<?xml version="1.0"?>
<!-- A basic weather report for select West Coast locations -->
<WEATHERREPORT>
    <STATE NAME="California">
        <CITY ID ="Los Angeles">
            <SKIES VALUE="PARTLYSUNNY"/>
            <HI C="31" F="87"/>
            <LOW C="18" F="65"/>
            Partly cloudy.
        </CITY>
        <CITY ID ="Sacramento">
            <SKIES VALUE="SUNNY"/>
            <HI C="36" F="97"/>
            <LOW C="13" F="64"/>
            Sunny and hot.
        </CITY>
    </STATE>
    <STATE ID ="Washington">
        <CITY ID ="Seattle">
            <SKIES VALUE="RAIN"/>
```

```
            <HI C="12" F="54"/>
            <LOW C="9" F="49"/>
            Raining on and off throughout the day.
        </CITY>
        <CITY ID ="Olympia">
            <SKIES VALUE="CLOUDY"/>
            <HI C="11" F="49"/>
            <LOW C="8" F="47"/>
            30% chance of rain toward evening.
        </CITY>
        <CITY ID="Redmond">
            <SKIES VALUE="RAIN"/>
            <HI C="10" F="50"/>
            <LOW C="5" F="41"/>
            Rain, mixed with showers.
        </CITY>
    </STATE>
</WEATHERREPORT>
```

In Listing 12.1, the first line,

```
<?xml version="1.0"?>
```

identifies the document as an XML document through the preprocessor tag **<?...?>**. In more sophisticated XML documents, this tag may also provide a link to a *Document Type Definition*, an external file that makes the structure in an XML document more explicit. DTDs will be covered in much greater depth in the section “Describing Your Data: The DTD” later in this chapter.

NOTE
The preprocessor tags give information to the parser concerning the document’s validity—whether the XML structure contained therein obeys a predefined format. If this format isn’t explicitly defined, the document can be well formed (that is, follow the rules of XML) without being valid. Much of the use of XML as a data-storage mechanism, especially with respect to Microsoft’s parsers, actually works on the assumption that there is no explicit definition of the data; as long as the document follows the rules of XML, it can be used to store data.

As with HTML, comments should be an integral part of any page, describing structures, specialized variables, and anything else that will help other people understand what you intended to do with your XML document. As in HTML, the syntax for XML comments is straightforward:

```
<!-- This is a comment -->
```

Comments can extend across multiple lines because XML structures ignore white space in much the same way that HTML structures do. However, it is not possible to nest comments in XML; the closing comment tag of any interior comment will terminate the comment body in its entirety. In other words, the statement

```
<!-- This is an <!-- embedded --> comment -->
```

will fail to parse properly.

In a typical HTML document, the entire page is contained within the **<HTML> </HTML>** tags. These tags acts as a container for everything else within the document, and there is never more than one such element per document. An XML document likewise has a single containing tag, although there are no restrictions about what you call it. For example, in Listing 12.1, the document container is the **<WEATHERREPORT>** tag. The outermost element in an XML document is usually referred to as the *root* element or *node*, in much the same way that the foundation of a tree (or a family tree) is its root.

This family tree metaphor in Figure 12.1 can actually be useful in getting a better handle on XML data structures. In all but the simplest XML documents (that is, one that has only a root), the root node has a collection of child nodes. In a traditional text document, the child nodes correspond to major headers, but in the data structure displayed in the figure, each child of Weather Report is a State node, a branch off the main trunk. Each State in turn contains one or more City nodes. (For example, the Washington State node contains the City nodes corresponding to Seattle, Olympia, and Redmond.) These nodes may contain other branches or, as in this case, leaf nodes. The leaf nodes terminate the tree at that point; a given node cannot simultaneously be both branch and leaf.

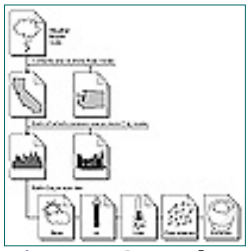

Figure 12.1 The weather report XML data structure resembles a family tree.

One useful way of thinking about this structure is to associate each leaf with a property and each branch with either a class or a collection of classes. For example, the Olympia, Washington, node has a **SKIES** property, a **HI** and **LOW** property, and some associated text. The Olympia node corresponds to the **CITY** class, the Washington node corresponds to the **STATE** class, and the whole document is in turn a collection of **STATE** classes. This relationship between classes and XML nodes is hardly accidental, and indeed, it is one of the things that makes XML such a promising technology.

Within each node, in addition to the tag name for that node (such as **STATE**, **CITY**, **LOW**, and so on), there may be one or more attributes. An attribute helps define a node more clearly in exactly the same way that attributes in HTML clarify the role of the tag. An HTML example illustrates this. The image tag (**<IMG>**) typically comes with several attributes that define the name or identifier for the image (**ID="myPic"**), the source URL of the image (**SRC="http://www.mysite.com/images/mypic.jpg"**), the dimensions of the image (**HEIGHT="120" WEIGHT="90"**), and even what events the element supports (**ONMOUSEOVER="HiliteMe();"**).

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

An XML tag is not that different, except that you can define what attributes are associated with a given tag. Thus, the **HI** and **LOW** tags have two attributes each—the **C** for Celsius and **F** for Fahrenheit. An element doesn't need any given attribute, and although some attributes are more convenient (for example, an **ID** tag is extremely useful in an XML document), none are strictly required. Additionally, a tag doesn't need any specific attributes defined. Here again, it's worth looking at the corresponding **<IMG>** tag in HTML; although **<IMG>** usually contains a **SRC** attribute, such attributes as **WIDTH** and **HEIGHT** may or may not be provided, and attributes such as **BORDER** may appear comparatively rarely. An XML tag likewise could have one attribute, dozens, or none at all, although if a formal DTD is provided with a given XML document, it is possible to have default values for given attributes.

In the simplest form of XML documents, all attributes' values are given as strings. (It is possible to specify different formats with the appropriate parser, but this falls beyond the scope of this book.) As a general rule of thumb, an attribute serves to define a characteristic of a tag or provides a link to additional data about the tag. The **SRC** attribute of an **<IMG>** tag, for example, points to the location of a graphical file to be used for the image.

> **NOTE**
> If you have an opening tag with no closing tag within your XML document, or if you attempt to put a closing tag after a self-terminating tag, such as the one in the following code snippet, the parser will fail.

The one primary difference in structure between XML and HTML concerns strictness. XML treats every element as a container. In other words, any given tag must have a closing tag, and any other tag defined between an opening and closing tag must also be closed between the two. XML closes its tags exactly the same way that HTML does: **<TAG>** is terminated with **</TAG>**.

Although most contemporary HTML authoring programs are moving toward

XML-like code, HTML does include certain elements that are invalid in XML. For example, the **<IMG>** tag in HTML is typically treated as a single element with no closing tag. XML writes the same tag as **<IMG SRC="mydata.jpg"></IMG>**. However, because many nodes in an XML structure don't have text associated with them, XML also includes a shorthand notation, **<IMG/>**, with the terminating slash moved to the end, so you don't have to include the end tag. You can see an example of this in Listing 12.1:

```
<SKIES VALUE="RAIN"/>
```

Text is treated in a slightly different way by XML than it is by HTML. XML supporters tend to fall into two camps; one stresses XML as a document markup language, and the other sees XML as a convenient database format. Because you can use XML to describe documents, it is perfectly possible to have a block of text with embedded XML elements. In this case, the text in each block is actually treated by the parser as belonging to distinct nodes. For example, the expression

```
<WARNING>This is a <HILITE>test</HILITE>, this is only
a test.</WARNING>
```

is actually broken up internally as

```
<WARNING>
    <INTERNAL_TEXT>This is a </INTERNAL_TEXT>
    <HILITE>test</HILITE>
    <INTERNAL_TEXT>, this is only a test.</INTERNAL_TEXT>
</WARNING>
```

where **<INTERNAL_TEXT>** is a node that contains default text for the document. This in turn means that any XML element has a text property that consists of all the text of its subnodes. The text for **<WARNING>**, for example, is "This is a test, this is only a test." whereas the text for **<HILITE>** is just "test". An advantage of this is that a browser that doesn't support an XML parser can still output the text of the document, although it loses any relevant formatting.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Parsing XML

I've bandied about the term "parsing" throughout the last section without really defining it. Like many computer expressions, "to parse" has a wide variety of meanings depending on the context. For XML parsers, more specifically, to parse means "to import the document into a dynamic data structure that can be traversed recursively."

I should explain that mouthful in simpler English. An XML parser is usually built as a recursive descent parser. In short, this means that the parser program performs the following steps:

1. Read the prolog, or preprocessor instructions, to get information about the document not contained within the XML tree itself.
2. Read the first tagged element of the XML structure.
3. Record the name of the element as its tag name.
4. Read through the rest of the element to determine which properties the given node supports and then assign these into attribute-value pairs.
5. If the next tag found is not a closing tag, read the next tag and define it as a child of the current node. For this tag, jump to Step 3.
   If the next tag found is a closing tag, the node is defined. Move to the following node and determine whether it is a closing node (in which case, the parent node is also closed) or a new node (in which case, another child is defined for the parent node).
6. Let this process continue until all nodes have been added into this structure.

In essence, the parser walks a tree, going up a branch until it hits a leaf, adds all the leaves for a given branch, backtracks down a branch, and repeats the process until all leaves are hit (see Figure 12.2). Programming this process is perhaps not as immediately obvious as iterating through a linear list, but

programming recursive walks is not all that much harder either, as I discuss in detail in the next section.

Figure 12.2 The parser starts at the root node, walks up each branch until it reaches a leaf, moves to the next leaf, and then backtracks to the last branch and takes the other fork.

Writing a simple XML parser is not a terribly complicated undertaking, although supporting all the features of XML 1.0 adds considerably to the amount of work involved. Fortunately, Microsoft has been a prime mover to get XML into the marketplace, and it has built a number of reasonably powerful XML parsers that automate the process of reading and traversing XML documents. This book concentrates on the XML parser that is part of Internet Explorer 5.0 because it has applications both in DHTML and Visual Basic. Internet Explorer 4.0 includes a more primitive C++-based XML parser that must be included as an ActiveX control, and Microsoft also supports a Java-based parser, which can be useful in Web pages outside of Microsoft's Internet Explorer, but which has only limited utility in Visual Basic.

Internet Explorer 5.0 makes extensive use of XML. Indeed, you can make a compelling argument that IE5 isn't an HTML browser at all, but rather an XML browser that can coincidentally parse HTML as well. XML appears all over the place in IE5:

- XML can extend the presentation markup in the HTML document stream. This subject is covered in some detail in Chapter 14.
- XML forms the foundation of dynamic properties and behaviors.
- XML can be embedded within HTML documents as data stores or referenced via external files.
- The Client Capabilities, IE5's successor to cookies, saves and retrieves session, form, and property information in XML format.

Because of the pervasiveness of XML in Internet Explorer 5, Visual Basic programmers can access and manipulate XML data either by using the IE5 Document Object Model (frequently referenced as the DOM) or by using the Microsoft XML library. Each offers benefits and drawbacks, although in general, if you are working with an HTML document with embedded XML, you should go the former route, whereas if you want to deal purely with XML files, you should use the XML library.

The MS XML library is fairly extensive, quite powerful, and more than a little counterintuitive. Because it's worth understanding how the library objects work on their own before incorporating XML into DHTML applications, the next several sections discuss the Microsoft XML parser 2.0 API in depth as they are referenced from Visual Basic.

NOTE:
To directly manipulate XML documents, you need to include the XML

library in your Visual Basic project. To do this, select References from the VB project menu. Find and check Microsoft XML, version 2.0 (MSXML.DLL) to add it to the associated references and then press OK. Note that this is version 2.0 of the Microsoft XML parser, not a 2.0 version of XML itself.

Leveraging The MS XML API

Any time you deal with database structures, you're going to find the API to be complex, sometimes cumbersome, and yet never quite as complete as you'd like. The XML API is much like that; the improvements from earlier versions are noticeable, but you still need to put some effort into getting anything useful out of them. Still, there are only a handful of objects to worry about in the library—the DOMDocument, IDOMNodes, IDOMNodeLists, and IDOMError classes.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The Documents Of DOM

DOM, of course, refers to the Document Object Model, the term that Microsoft has applied to the architecture of Internet Explorer 5 pages. Note here that there's no specific mention of XML in any of the class names—yet another indication that Microsoft sees XML as the foundation of HTML development in the future.

Of all the classes, the DOMDocument acts as the primary interface to the XML library. From this class, you can load and save XML documents, navigate to specific nodes in the XML tree, add, remove, or relocate nodes within the tree, and even determine when things have gone wrong. The methods and properties of the DOMDocument are summarized in Tables 12.1 and 12.2, with detailed descriptions following.

Table 12.1 Methods of the DOMDocument object.

Method Name	Description
Abort	Aborts an asynchronous download
CreateNode	Creates a new node
GetAttribute	Retrieves a root node's attribute by name
GetTypedAttribute	Retrieves a root node's attribute as a typed value
InsertNode	Positions a node into the tree
Load	Loads an XML document from a file
LoadXML	Loads an XML document from a string
NodeFromID	Retrieves a node with the given ID
RemoveAttribute	Removes an attribute from the node
RemoveNode	Removes the node from the XML tree

ReplaceNode	Replaces one node with another
SaveNode	Returns an XML string of the current document
SetAttribute	Sets the value of a named attribute to a string
SetTypedAttribute	Sets the value of a named attribute to a specific type

Table 12.2 Properties of the DOMDocument.

Property Name	Type	Description
Async	Boolean	If XML file can load asynchronously, then Async is true
Attributes	IDOMNodeList	A list of attributes within a given node
ChildNodes	IDOMNodeList	A collection of the children of the current node
DataType	Variant	The data type of the node
Definition	IDOMNode	The definition of the node within the DTD or schema
DocumentNameSpaces	IDOMNodeList	A collection of namespaces for the document
DocumentNode	IDOMNode	The document's root node
DocumentType	IDOMNode	Information from the prolog's !DOCTYPE node
LastError	IDOMError	The last error that occurred
NameSpace	IDOMNode	The definition for the namespace on the node
NodeName	String	The name of the node's tag
NodeType	XMLNodeType	The type of node (text, node, prolog, processing instruction (PI), comment, and so on)
NodeTypedValue	Variant	The value of a node, as a typed constant
NodeValue	String	The value or text of a node
ParentNode	IDOMNode	The node to which the current node belongs
QualifiedNodeName	String	Returns the name of the node without its namespace qualifier
ReadyState	Long	Gets the load state of the XML document
Specified	Boolean	Indicates whether a definition exists for the node in the DTD
URL	String	The read-only URL of the XML document

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Loading A Document

Before you can work with an XML document, you need to load it into the data structure. The DOMDocument provides three ways to do this: loading the document from a file, loading the document asynchronously from a URL, or converting a string into a document. Of the three, loading a document from a file is far and away the easiest. Listing 12.2 prompts the user for an XML file to load and then loads it, generating an error if the document couldn't load properly.

Listing 12.2 The **LoadXMLFile** function returns an instance of the DOMDocument.

```
' Declare a public variable in form or module
' to hold XML structure
Public DOMDoc as DOMDocument

Public Function LoadXMLFile(Optional Filename As String) _
  As DOMDocument
    Dim CurReason As String
    Set LoadXMLFile = Nothing
    ' If no filename is supplied, prompt for one
    If Filename = "" Then
        With Screen.ActiveForm.CommonDialog1
            .Filter = "XML Files (*.xml)|*.xml"
            .Filename = "*.xml"
            .CancelError = True
            .DialogTitle = "Select an XML File to Load"
            On Error GoTo CancelLoad
            .ShowOpen
            On Error GoTo 0
            Filename = .Filename
        End With
```

```
    End If
    ' Initialize the document object
    Set XMLDoc = New DOMDocument
    ' Load the document.
    ' Load Syntax: Sub load(bstrUrl As String)
    XMLDoc.Load Filename
    ' Check to see if error occurred ("" indicates no error)
    If XMLDoc.lastError.reason <> "" Then
        MsgBox "XML Document unable to load. Reason:" _
        + XMLDoc.lastError.reason
    Else
        Set LoadXMLFile = XMLDoc
    End If
    Exit Function
CancelLoad:
    On Error GoTo 0
End Function
```

The **LoadXMLFile** function calls upon the Common Dialog box to show an open box. (Indeed, most of the code in the function is involved in setting up the dialog.) When an XML file is located, it is loaded into the parser. If the XML file is well formed—that is, it conforms to XML logic—then an instance of the new DOMDocument is returned, which contains the full XML structure.

Especially if you are creating XML structures by hand, it can be notoriously difficult to get an XML document to parse. The **LastError** property will retrieve a description of the last error that occurred, including the exact location of the error in the XML file that the parser aborted. **LastError** is an IDOMError class that is covered later in this chapter.

In today's wired world, it is just as likely that the source of your file will come from the Internet as it will from a local directory. Although XML files are usually fairly compact, an XML document with thousands (or even tens of thousands) of nodes is not altogether rare. As a consequence, sometimes you will want to download the file asynchronously. Although the DOMDocument object does support this, it doesn't raise any events, making it necessary to set up a timer or make a **SetTimer** API call to determine whether the document is done loading. On the other hand, Visual Basic 6 also includes a new capability that lets you add a control to a form at design time—without putting a base control on the form first. The advantage is that you can make your code much more portable; as long as a reference to the active form can be obtained, you will be able to create the required control.

The code in Listing 12.3 demonstrates one way of making an asynchronous load. You should place this code within a form because it relies on the form as a container for the timer.

Listing 12.3 Loading an XML file asynchronously.

```
Option Explicit

Public DOMDoc As DOMDocument
Public Node As IDOMNode
Public NodeList As IDOMNodeList
```

```
Public WithEvents DownloadTimer As Timer

Private Sub DownloadTimer_Timer()
    Static dtCount As Integer
    Dim Response As Integer
    dtCount = dtCount + 1
    If DOMDoc.readyState = 4 Then
         ' A readystate of 4 indicates that the download
         ' is done or it failed.
        DownloadTimer.Enabled = False
        Me.Controls.Remove DownloadTimer.Name
        dtCount = 0
        If DOMDoc.lastError.reason <> "" Then
            MsgBox (DOMDoc.lastError.reason)
            Exit Sub
        End If
    Else
        If dtCount Mod 20 = 0 Then
            Response = MsgBox("Load is taking longer" & _
                       "than expected. Do you wish to" & _
                       "continue?",
                       vbYesNo)
            If Response = vbNo Then
                DownloadTimer.Enabled = False
                Me.Controls.Remove DownloadTimer.Name
                dtCount = 0
            End If
        End If
    End If
End Sub

Public Sub AsynchXMLLoad(URL As String)
    Static Ct As Integer
    Set DownloadTimer = Me.Controls.Add("VB.Timer", _
    "CTimer" + CStr(Ct))
    Ct = Ct + 1
    DownloadTimer.Interval = 1000
    DownloadTimer.Enabled = True
    DOMDoc.async = True
    DOMDoc.Load (URL)
End Sub

Private Sub Form_Load()
    Set DOMDoc = New DOMDocument
    AsynchXMLLoad ("c:\bin\EmbeddedXML.xml")
End Sub
```

The code involving the DOMDocument object is not all that different from the initial call; to perform an asynchronous load, you need to set the **Async** property of the object to **true**, which indicates that the DOMDoc can continue loading in the background while

other activities take place. Setting **Async** to **false** forces the program to halt until the file is completely downloaded or an error occurs.

To ascertain whether either of these two states has occurred, you need to check the **readyState** property of the DOMDocument. This is exactly the same property as the **readyState** associated with the Internet Explorer browser upon downloads. When this property is set to 4, the download has either completed or has raised an error that can be checked.

Until the download is complete, the program checks a timer that has been added at runtime. This is a new feature of Visual Basic 6; in earlier versions of VB, you would specifically have to create the timer at runtime. Because the only reason to have the timer in place is to help handle downloads, it makes more sense to create it when needed and then destroy it when the task is handled. To do this, the **Controls** collection now supports two new methods: **add** and **remove**:

```
Function add(ControlIdentifier as String, _
             ControlName as String, _
             Optional Container as Object) as Object

Sub remove(ControlName as string)
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

ControlIdentifier is the pair **ProgID.ClassID**, such as “Excel.Work-Book” or “VB.Timer”, **ControlName** is the **Name** property of the new control, and **Container**, which is optional, is the form or other container that owns the control. In Listing 12.2, the timer control was added, supplied a unique name, and then set to fire once a second. When you declare a variable called **DownloadTimer** as a **Timer** variable with events, the program intercepts the event when the timer does fire and calls the **DownloadTimer_Timer()** event handler. This in turn checks to see if the file has finished downloading, displaying an error message if the file wasn’t found or failed to parse. If the file is still downloading, it will continue until either the file does finish or 20 seconds have passed. The user is then prompted about whether he or she wants to extend the download period or terminate the file retrieval.

Other than the extended code involved in querying the download, there is no real difference between **LoadXMLFile** and **AsyncXMLLoad**. In both cases, the XML parser handles the hard part of converting the file into an internal document. Loading from a string is just about as straightforward, although it is covered in Chapter 14.

Troubleshooting Your XML Document

The IE5 XML parser is unforgiving of errors in the documents it parses. For people accustomed to the fairly generous ways that most browsers accept bad HTML, this can come as a bit of a shock, but it actually comes from necessity. In HTML, the parser can usually make a pretty good guess about the intent of the code because an already established structure is in place. If you inadvertently switch the order of closing format tags, such as **<B><I>This is a test</B></I>**, the browser’s parser knows enough to switch them back internally.

However, with XML, you are the one defining that structure. The parser knows only the rules that XML provides and can’t second-guess the creator of

the XML document. You might accidentally create bad code in several places:

- In the prolog, the tag begins with **<?xml**—no spaces and all lowercase. The version number appears next, again in lowercase: **version="1.0"**. Finally, the closing bracket should include a question mark as well: **?>**.
- In the word processor used to create the XML, disable curly quotes. In general, you should probably write an XML file in an editor specifically designed for creating HTML, but avoid WYSISIG editors; they have a tendency to alter erroneous tags, and certain formats for XML tags look wrong to these editors. I'd recommend an application such as Visual InterDev, Allaire's Home Page (my personal favorite), or HoTMetaL.
- Some XML parsers are case sensitive, but others are not; assume the worst, and establish a convention for your tags and attributes as all uppercase or lowercase. The IE5 parser in particular is case-sensitive, so **<MYTAG></mytag>** will generate an error.
- Attribute values must be surrounded by quotes. This differs from most modern strains of HTML, where such expressions as **<IMG SRC=myImage.gif>** are valid. This will generate an error in XML.
- Any standalone tag must be terminated with a slash. For example **<IMAGE REF="myImage.gif"/>** is valid, but **<IMAGE REF= "myImage.gif">** is not unless a corresponding **</IMAGE>** tag exists.
- Similarly, a standalone tag cannot enclose any text, nor have a closing tag. In other words, **<IMAGE REF="myImage.gif"/>This is my picture</IMAGE>** will generate an error because of the terminating back slash in the **IMAGE** tag.
- You can embed HTML text into XML documents, but to do this, you need to set the data type of the node through a DTD structure. Without this, the XML parser will attempt to interpret HTML as if it were XML with all kinds of headaches as a consequence. Embedding HTML in an XML document is discussed later in this chapter in the section "Describing Your Data: The DTD."
- Finally, outside of the prolog, there can be only one root node in an XML document; it encompasses all other nodes. Although this root node doesn't need to have the name **<DOCUMENT>**, it usually serves as a convenient identifier for the root.

In addition to this, if you do provide a DTD—in essence, the rule book of the document—then you need to follow all the naming and data conventions of that document as well. You make the rules; you have to live by them.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Navigating An XML Document

Once you load an XML document, the information is stored in a fairly complex data tree in memory. The key to accessing this tree lies in getting a handle on the root node of the tree. Fortunately, the DOMDocument returns a reference to that very root, although it is not necessarily the root of the data tree. If the document has no prolog or XML header (it doesn't necessarily need one, by the way), then the DOMDocument object contains the root of the data set. However, if the prolog is included or there are any comments outside of the data set, then the start of the data section could be one of several nodes.

The **documentNode** property of the DOMDocument always contains the start of the data set—that part of the XML document that actually holds the data. For example, Listing 12.4 shows the weather report document with special nodes indicated.

Listing 12.4 An annotated XML weather report.

```
<!-- Root Node for the Document, has no name or tag -->
<!-- Root ChildNode 0, nodeName is XML -->
<?xml version="1.0"?>
<!-- Root ChildNode 1, nodeName is ! -->
<!-- A basic weather report for select West Coast locations -->
<!-- Root ChildNode 2, nodeName is WEATHERREPORT -->
<!-- node is documentNode -->
<WEATHERREPORT>
    <STATE NAME="California">
        <CITY ID ="Los Angeles">
            <SKIES VALUE="PARTLYSUNNY"/>
            <HI C="31" F="87"/>
            <LOW C="18" F="65"/>
            Partly cloudy.
        </CITY>
        <CITY ID ="Sacramento">
            <SKIES VALUE="SUNNY"/>
```

```
            <HI C="36" F="97"/>
            <LOW C="13" F="64"/>
            Sunny and hot.
        </CITY>
    </STATE>
    <STATE ID ="Washington">
        <CITY ID ="Seattle">
            <SKIES VALUE="RAIN"/>
            <HI C="12" F="54"/>
            <LOW C="9" F="49"/>
            Raining on and off throughout the day.
        </CITY>
        <CITY ID ="Olympia">
            <SKIES VALUE="CLOUDY"/>
            <HI C="11" F="49"/>
            <LOW C="8" F="47"/>
            30% chance of rain toward evening.
        </CITY>
        <CITY ID="Redmond">
            <SKIES VALUE="RAIN"/>
            <HI C="10" F="50"/>
            <LOW C="5" F="41"/>
            Rain, mixed with showers.
        </CITY>
    </STATE>
</WEATHERREPORT>
```

In this case, the root node has three children—the prolog tag, a comment, and the document node. The **length** property can confirm this. For either the DOMDocument or most DOMNodes, **length** will return the number of children that the node supports. (This also gives a hint of how Java is influencing even Internet Explorer; **length** refers to the number of items in a given collection in Java or JavaScript.)

To actually retrieve the node, you need to use the **childNodes** collection. Each node has a **childNodes** property, an instance of an IDOMNodeList. The IDOMNodeList provides much of the navigation capability within the XML structure and can be used to retrieve specific nodes. However, it should be pointed out that the IDOMNodeList is not a collection in the traditional Visual Basic sense. It doesn't support **For Each** style enumeration, and it doesn't have a default index (or a key, for that matter) as a collection does. Instead, it relies on the **item(n)** function, where **n** is the zero-based index of the members of the collection. Thus, the first node is **item(0)**, the second node **item(1)**, and so forth. For the weather report, then, the **documentNode** is the same as the third child (index of 2) of the root node:

```
DOMDoc.documentNode=DOMDoc.childNodes.item(2)
```

The **item()** function returns an IDOMNode object, which has a subset of the same methods and properties as the DOMDocument object. The IDOMNodeList methods are outlined in Table 12.3, and the methods and properties for the IDOMNode object are listed in Tables 12.4 and 12.5. Neither IDOMNodes nor IDOMNodeLists are directly createable in Visual Basic; you can't use the **New** operator to create a standalone IDOMNode. Similarly, IDOMNodes don't raise any events, although with some clever programming, it's possible to create an IDOMNode surrogate that will (as shown later in this chapter).

Table 12.3 Methods and properties of the IDOMNodeList.

Method Name	Description
CurrentNode	Returns a pointer to the current node in the list
Item	Returns a pointer to the indexed node in the list
Length (property)	The number of nodes in the list
MoveTo	Sets the current node to the indexed item
MoveToNode	Sets the current node to the passed node
NextNode	Sets the current node to the next node in the list
PreviousNode	Sets the current node to the previous node in the list

Table 12.4 Methods of the IDOMNode object.

Method Name	Description
GetAttribute	Retrieves a root node's attribute by name
InsertNode	Positions a node in the tree
RemoveAttribute	Removes an attribute from the node
RemoveNode	Removes the node from the XML tree
ReplaceNode	Replaces one node with another
SetAttribute	Sets the value of a named attribute to a string

Table 12.5 Properties of the IDOMNode.

Property Name	Type	Description
Attributes	IDOMNodeList	A list of attributes within a given node
ChildNodes	IDOMNodeList	A collection of the children of the current node
NodeName	String	The name of the node's tag
NodeType	XMLNodeType	The type of node (text, node, prolog, comment, and so on)
NodeValue	String	The value or text of a node
ParentNode	IDOMNode	The node to which the current node belongs
Specified	Boolean	Indicates whether a definition exists for the node in the DTD

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Navigating through the XML tree is unfortunately not as intuitive as it could be, but it's still relatively powerful. The key is to realize that navigation happens primarily through the **ChildNodes** collection of each node. One of the properties of this collection is the **currentNode** object, which acts as a pointer into the list of nodes in each child node collection. You can then move this pointer by invoking one of the member functions of the child nodes object.

This can best be seen in the weather report, a fragment of which is repeated in Listing 12.6 for reference. The child nodes are evident, by the way, not just by being enclosed in wrapper tags but also by being indented. Indenting is an invaluable visual aid for deciphering the structure of an XML document.

Listing 12.6 Navigating through the weather report.

```
<?xml version="1.0"?>
<!-- A basic weather report for select West Coast locations -->

<WEATHERREPORT> -- DocumentNode
    <STATE NAME="California">
        <CITY ID ="Los Angeles">
            <SKIES VALUE="PARTLYSUNNY"/>
            <HI C="31" F="87"/>
            <LOW C="18" F="65"/>
            Partly cloudy.
        </CITY>
        <CITY ID ="Sacramento">
            <SKIES VALUE="SUNNY"/>
            <HI C="36" F="97"/>
            <LOW C="13" F="64"/>
            Sunny and hot.
        </CITY>
    </STATE>
    <STATE ID ="Washington"> -- MoveTo(1) (i.e., 2nd item)
```

```
        <CITY ID ="Seattle"> -- MoveTo(0) (i.e., 1st item)
            <SKIES VALUE="RAIN"/>
            <HI C="12" F="54"/>
            <LOW C="9" F="49"/> -- MoveTo(2), then _
                 getAttribute("F")
            Raining on and off throughout the day.
        </CITY>
        <CITY ID ="Olympia">
            <SKIES VALUE="CLOUDY"/>
            <HI C="11" F="49"/>
            <LOW C="8" F="47"/>
            30% chance of rain toward evening.
        </CITY>
        <CITY ID="Redmond">
            <SKIES VALUE="RAIN"/>
            <HI C="10" F="50"/>
            <LOW C="5" F="41"/>
            Rain, mixed with showers.
        </CITY>
    </STATE>
</WEATHERREPORT>
```

For example, to retrieve the low temperature node from the weather report in Seattle, Washington, you could use the **MoveTo** method in conjunction with the **CurrentNode** method:

```
DOMDoc.documentNode.MoveTo 1     ' This moves the pointer to
                                 ' the Washington State node
DOMDoc.documentNode.CurrentNode.ChildNodes.MoveTo 0
                                 ' This moves the pointer to
                                 ' the Seattle City node
DOMDoc.documentNode.CurrentNode.ChildNodes.CurrentNode. _
       ChildNodes.MoveTo 2       ' This moves the pointer to
                                 ' the Low node.
```

Of course, because the **MoveTo** method returns the **CurrentNode**, this can be shortened to the following:

```
DOMDoc.documentNode.MoveTo(1).MoveTo(0).MoveTo 2
```

Similarly, you use the **PreviousNode** and **NextNode** methods to move the **CurrentNode** to the previous or next node in the list. One caveat in using any of these navigational moves: You can move outside the bounds of the list by calling **PreviousNode** at the first node, **NextNode** at the last node, or passing an index beyond the bounds of the **ChildNodes** list.

The **MoveToNode** method is one of those functions that makes you wonder why it was added. **MoveToNode** takes a node as a parameter and sets the **CurrentNode** to that node—if the node being passed is in the **ChildNodes** list that calls the function. Otherwise, it sets the current node to **Nothing**. I'm sure that somewhere someone has a need for that function, but I'm puzzled about what the need would be.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Node Names, Node Values, And Attributes

The **nodeName** property retrieves the name of the tag associated with a given node. It's worthwhile here to distinguish between a *node* and a *tag*. A start tag acts as the opening delimiter for a node, just as the end tag acts as a closing delimiter. Everything within those two tags forms a part of the node; a node's children are, technically speaking, a part of the node as a consequence, as is its text.

The **documentNode** property of the DOMDocument in the weather example has a **nodeName** of "WEATHERREPORT", whereas the second child of that node has the **nodeName** of "STATE". Because the node is a container, you'll never end up retrieving a "/STATE" or "/WEATHERREPORT" **nodeName**; that serves simply as the terminator for the node.

The **nodeValue** property, on the other hand, can be a little confusing. For example, the **nodeValue** of **WEATHERREPORT** is "Partly cloudy. Sunny and hot. Raining on and off throughout the day. 30% chance of rain toward evening. Rain mixed with showers."

This confusing (and contradictory) weather report was brought to you by the twisted notion of **nodeValue**; everything that isn't a tag is returned as a string into **nodeValue**. This makes sense if you look at XML's other role as a document markup language. The formatting of a page should be independent of its content, such that you could retrieve the text of the entire document if you need it.

The weather report resolves itself only when the document's scope narrows to the **CITY** level. For example, to retrieve the forecast for Olympia, Washington, you'd navigate as follows:

```
Dim Node as IDOMNode
Set Node=DOMDoc.DocumentNode
Set Node=Node.ChildNodes.MoveTo(1) ' Move to Washington
Set Node=Node.ChildNodes.MoveTo(1) ' Move to Olympia
MsgBox "Today's forecast for Olympia,Washington is" +Node.NodeValue
```

NOTE
You can set the **nodeValue** of a given node, but you should do so only with a node that contains only text. If it contains any children nodes, those nodes will be deleted and replaced with the new text.

This will cause a dialog box to pop up with the message "Today's forecast for Olympia, Washington is 30% chance of rain toward evening."

Of course, it would be useful to get the rest of the information about that weather report. To do that, you need to read the attributes of a given node. The implementation of attributes in the IE5 parser uses a clever trick; it treats the collection of attributes in a given node as if it was a second DOMNodeList belonging to that node. This IDOMNodeList is contained in the **Attributes** property. The attribute name is then given for each attribute by the **nodeName**, and the value of the attribute is given by the **nodeValue**.

For example, to retrieve the low temperature in degrees Fahrenheit in Olympia, you navigate to the **LOW** node for that city and then retrieve the attribute from its index:

```
Dim Node as IDOMNode
Set Node=DOMDoc.DocumentNode
Set Node=Node.ChildNodes.MoveTo(1) ' Move to Washington
Set Node=Node.ChildNodes.MoveTo(1) ' Move to Olympia
Set Node=Node.ChildNodes.MoveTo(1) ' Move to the LOW
                                   ' temperature node
Set Node=Node.Attributes.MoveTo(1) ' Retrieve the attributes
                                   ' property
                                   ' and get the second node
                                   ' ("F")
MsgBox "The low for Olympia was "+Node.NodeValue+ _
  "degrees "+Node.NodeName
' Displays a dialog box saying
' "The low for Olympia was 47 degrees F"
```

Obviously, it is more preferable to just reference the attribute by name. The **GetAttribute** and **SetAttribute** functions do just that:

```
Dim Node as IDOMNode
Set Node=DOMDoc.DocumentNode
' Move to Washington
Set Node=Node.ChildNodes.MoveTo(1)
' Move to Olympia
Set Node=Node.ChildNodes.MoveTo(1)
' Move to the LOW temperature node
Set Node=Node.ChildNodes.MoveTo(1)
MsgBox "The low for Olympia was "+Node.GetAttribute("F")+ _
  " degrees F"
' Displays a dialog box saying
' "The low for Olympia was 47 degrees F"
Node.SetAttribute "F","50"
MsgBox "The low for Olympia was "+ _
  Node.GetAttribute("F")+ _
  " degrees F"' Displays a dialog box saying
  ' "The low for Olympia was 50 degrees F"
```

The **SetAttribute** method takes both the attribute name and the new value, creating the attribute if it doesn't already exist. The **GetAttribute** function takes only the attribute name, and if that attribute doesn't exist, the function returns a null value. This can in fact be a useful test to determine whether a given attribute exists because even if an attribute has been set to a blank string, a blank string is still not the same as a null and can be tested with the **IsNull()** function. The **IsValidAttribute()** function illustrates this:

```
Function IsValidAttribute(Node as IDOMNode, _
  Attribute as String) as Boolean
```

```
        ' Returns true if the given node has the attribute ' listed
        IsValidAttribute=Not IsNull(Node.GetAttribute(Attribute))
End Function
```

These functions provide the building blocks for navigation, but obviously, you will probably not want to access elements in an XML structure by specifying the third child of the first child of the second child of the root node. To get more out of XML, you need to be able to walk the tree.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Walking The Tree

In Internet Explorer 4 and 5, you can use the **all()** method of the document to retrieve an object that has a specific ID value. It's useful to provide a similar function for your XML document; reference a collection of all records that have an ID property (or any property, for that matter), and return the object that corresponds to that property. You can break this down into two distinct steps—building the list and then retrieving the appropriate keyed item.

As mentioned earlier, XML makes use of a recursive descent parser to interpret the document. You can take advantage of this same structure to get information out of the XML tree by using recursion. In other words, the way to retrieve something from the document is to process a node and then process the node's children, then their children, until eventually, the whole tree is handled. Because you use the same operation throughout the tree, this process essentially involves calling the processing function within itself.

The **GetAttributeNodes** function uses recursion to populate a dictionary object with pointers to each of the nodes that have the requested attribute. Because these are object references, they can be used to retrieve the actual node within the XML structure. The code for **GetAttributeNodes** is shown in Listing 12.7.

> **NOTE**
> The dictionary object is available as part of the Scripting ActiveX component that comes with IE5 and provides much of the functionality of a collection without the overhead or error handling. To add it to your Visual Basic projects, select References from the Project menu and check Microsoft Scripting Runtime (SCRRUN. DLL). The beta version of the Visual Basic documentation indicates that it may actually be included in the VB language itself as a default type, but the current beta doesn't include it as a native function.

Listing 12.7 Walking an XML tree to retrieve a list of nodes.

```
Public Function GetAttributeNodes(Node As IDOMNode, _
  Attr As String, Optional ByRef AttrList As Dictionary) _
  As Dictionary
    Dim isRoot As Boolean
    Dim index As Integer
    Dim ChildNode As IDOMNode
    If AttrList Is Nothing Then
        Set AttrList = New Dictionary
        isRoot = True
    End If
    If Not IsNull(Node.getAttribute(Attr)) Then
        AttrList.Add Node.getAttribute(Attr), Node
    End If
    For index = 0 To Node.childNodes.Length - 1
        Set ChildNode = Node.childNodes.Item(index)
        If ChildNode.nodeType = NODE_ELEMENT Then
            GetAttributeNodes ChildNode, Attr, AttrList
        End If
    Next
    Set GetAttributeNodes = AttrList
End Function
```

Not all nodes are created equal. Nodes that handle text don't have children, but nodes that handle elements do. Attribute nodes can't contain data structures but do have **nodeNames**. Comments are handled differently as well.

The action that a node can take is known as its **nodeType**. The **node-Type** of an XML node determines whether a node is an element, text, a comment, or some other XML structure. Only element node types have children and attributes; any other type of node causes the tree walker to fail. Table 12.6 contains a list of all the node types.

Table 12.6 Node type values.

Node Type	Description
NODE_ATTRIBUTE	Specifies an attribute node
NODE_CDATA	A text node that contains protected characters
NODE_COMMENT	A comment within the XML body
NODE_DOCTYPE	The initial declaration of the document as an XML document
NODE_DOCUMENT	The documentNode object
NODE_ELEMENT	A structural node
NODE_ENTITYREF	A node in the DTD that defines a replaceable entity
NODE_NAMESPACE	A node in the DTD that contains a namespace
NODE_PCDATA	A text node (Parsed Character Data)
NODE_PI	A node in the DTD containing a processing instruction
NODE_WHITESPACE	Indicates that node preserves white space
NODE_XMLDECL	An element declaration node within a DTD

Recursion is a powerful programming technique that works especially well with XML. The **GetAttributeNodes** function is initially called with only two arguments—the document node of the XML structure to parse and the attribute to search for. The third argument, an optional dictionary list, will be created if a dictionary object isn't passed as a parameter. The routine then adds the node to the list if it has the requisite attribute explicitly defined. Finally, it iterates through all the children of the list and calls the same function for each child.

You can use the **GetAttributeNodes** function to implement an ID lookup function, as mentioned previously. By passing the attribute **"ID"** to the function and then looking up the requested ID value in the returned dictionary, you can retrieve a pointer to the node that has that particular ID. The **GetIDNode** function in Listing 12.8 does just that.

Listing 12.8 The **GetIDNode** function returns a pointer to the requested ID node.

```
Public Function GetIDNode(xmlNode As Object, RequestedID _
  As String, optional IDTag as string) As IDOMNode
       Dim TargetNode As IDOMNode
       Dim IDList As Dictionary
       Dim startNode as IDOMNode

       ' If no IDTag is supplied, use "ID"
       if IDTag="" then IDTag="ID"
       ' If the root node is passed, get the document node
       If typename(xmlNode)="IXMLDOMDocument" then
             Set startNode=xmlNode.documentNode
       Else
       ' Otherwise use the current node
              Set startNode=xmlNode
       End If
       Set TargetNode = Nothing Set IDList = _
            GetAttributeNodes(startNode, IDTag)
       If Not TypeName(IDList(RequestedID)) = "Empty" Then
             Set TargetNode = IDList(RequestedID)
       End If
       Set GetIDNode = TargetNode
End Function
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

This is a convenient way to get at “named” entries in your XML document. For example, to retrieve the node associated with Olympia, Washington, you simply need to say:

```
Set node=GetIDNode(DOMDoc,"Olympia")
```

You can use three other routines in conjunction with the **GetIDNode** function to simplify navigation further. **GetNamedNode**, **GetNamed-NodeCount**, and **GetNodeIndex** query the children of the node passed to it. **GetNamedNode** (see Listing 12.9) returns the first node it finds with the requested name, although you can pass in an index that will cause it to get the second, third, and so on node instead. **GetNamed-NodeCount** (see Listing 12.10) indicates how many children nodes of the passed node have the requested node name. Finally, **GetNodeIndex** (see Listing 12.11) returns the position of the current node within the set of **childNodes**.

Listing 12.9 Code for the **GetNamedNode** function.

```
' Function GetNamedNode()
' Takes as an argument the parent node of the nodes to be
' searched, the nodeName to be searched for, and a 0-based
' index for retrieving subsequent nodes if more than one node
' in the children collection has the requested name.
' Defaults to 0, the first node found.
' If no nodes match the name, the function retrieves
' the value Nothing.
' Usage:
'    dim node as IDOMNode
'    dim CityNode as IDOMNode
'    ' Set CityNode to the CITY node associated with
'    ' Olympia, Washington
'    ' Retrieve the first FORECAST node
'    set node=GetNamedNode(CityNode,"FORECAST")
'    ' or
'    set node=GetNamedNode(CityNode,"FORECAST",0)
'    ' Retrieve the second FORECAST node
'    set node=GetNamedNode(CityNode,"FORECAST",1)
Public Function GetNamedNode(node As IDOMNode, _
 tagName As String, Optional position As Integer) As IDOMNode
    Dim NamedNodeList As Dictionary
    Dim TargetNode As IDOMNode
    Dim NumNodesFound As Integer
```

```
    Dim Index As Integer
    Dim tempNode As IDOMNode
    NumNodesFound = 0
    Set TargetNode = Nothing
    Set NamedNodeList = New Dictionary
    If node.nodeType = NODE_ELEMENT Then
        For Index = 0 To node.childNodes.Length - 1
            Set tempNode = node.childNodes.Item(Index)
            If tempNode.nodeType = NODE_ELEMENT Then
                If tempNode.nodeName = tagName Then
                    If NumNodesFound = position Then
                        Set TargetNode = tempNode
                        Exit For
                    Else
                        NumNodesFound = NumNodesFound + 1
                    End If
                End If
            End If
        Next
    End If
    Set GetNamedNode = TargetNode
End Function
```

Listing 12.10 Code for the **GetNamedNodeCount** function.

```
' Function GetNamedNodeCount()
' Takes a parent node and the nodeName to be searched
' and returns the number of nodes in the parent's
' children that have that nodeName.

' Usage:
'    dim node as IDOMNode
'    dim CityNode as IDOMNode
'    dim ForecastCount as Long
'    ' Set CityNode to the CITY node associated with
'    ' Olympia, Washington
'    ' Retrieve the number of FORECAST nodes
'    ForecastCount=GetNamedNodeCount(CityNode,"FORECAST")

Public Function GetNamedNodeCount(node As IDOMNode, _
  tagName As String) As Long
    Dim nodeCount As Long
    Dim childNode As IDOMNode
    Dim Index As Long
    nodeCount = 0
    For Index = 0 To node.childNodes.Length - 1
        Set childNode = node.childNodes.Item(Index)
        If childNode.nodeType = NODE_ELEMENT Then
            If childNode.nodeName = tagName Then
                nodeCount = nodeCount + 1
            End If
        End If
    Next
    GetNamedNodeCount = nodeCount
End Function
```

Listing 12.11 Code for the **GetNodeIndex** function.

```
' Function GetNodeIndex()
' Takes a parent node and a possible child node,
' and returns the position of the child within the parent's
' children. If not found, the function returns -1.
' Usage:
'    dim node as IDOMNode
'    dim CityNode as IDOMNode
'    dim ChildNodePosition as Long
'   ' Set CityNode to the CITY node associated with
'   ' Olympia, Washington
'   ' Retrieve the first FORECAST node
'   set node=GetNamedNode(CityNode,"FORECAST",0)
'   ' Determine its position in the list of CityNode's
'   ' Children
'    ChildNodePosition=GetNodeIndex(CityNode,node)

Public Function GetNodeIndex(parentNode As IDOMNode, _
  childNode As IDOMNode) As Long
    Dim Index As Long
    GetNodeIndex = -1
    For Index = 0 To parentNode.childNodes.Length - 1
        If parentNode.childNodes.Item(Index) Is childNode Then
            GetNodeIndex = Index
            Exit For
        End If
    Next
End Function
```

These functions can work together as an alternative navigational system. For example, to find the HI temperature in degrees Fahrenheit for Olympia, you could use the following:

```
Dim OlywaNode as IDOMNode
Dim temperature as string
Set OlywaNode=GetIDNode(DOMDoc,"Olympia")
Temperature= OlywaNode.GetNamedNode("HI").getAttribute("F")
Msgbox "The HI temperature for Olympia was "+Temperature +" F"
```

Similarly, to print the forecasts for all the cities in the list, you could use the code in Listing 12.12.

Listing 12.12 The subroutine **PrintForecast** outputs the first forecast of each city and state.

```
Public Sub PrintForecast()
       Dim StateIndex as Integer
       Dim CityIndex as Integer
       Dim StateNode as IDOMNode
       Dim CityNode as IDOMNode
       Dim Forecast as string
       Dim Buffer as string
       For StateIndex=0 to GetNamedNodeCount(DOMDoc,"STATE")
          Set StateNode=GetNamedNode(DOMDoc,"STATE", _ StateIndex)
          For CityIndex=0 to GetNamedNodeCount(StateNode, _ "CITY")
                Set CityNode=GetNamedNode(StateNode,"CITY", _ CityIndex)
                Forecast=GetNamedNode(CityNode, _ "FORECAST").NodeValue
                Buffer= CityNode.GetAttribute("ID")+","
                Buffer=Buffer+StateNode.GetAttribute("ID")+":"
                Buffer=Buffer+Forecast
                Print Buffer
          Next CityIndex
       Next StateIndex
```

```
End Sub
```

You could create a simple XML weather-report client with what I have already covered in this chapter. However, there is much more to XML than a simple tree. XML documents can hold HTML and more structured data types than text. To unlock these capabilities, it’s necessary to dig into the Document Type Description.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Describing Your Data: The DTD

You've just developed a weather browser that will be used by thousands of people across the world. The product takes off, but there's a problem. Your competitor, Wacky Weather Tools, has come out with a product that also holds a significant share of the weather browser market. While you and your competitor stare across the street at one another, your customer base begins to complain about the fact that Jane in Hoboken (a loyal customer of your service) can't share her data with Fred in Peoria (a dirty, treacherous customer of that other company) (see Figure 12.3). Indeed, it may even cause the Sunny Days Browser Company, a small upstart, to get a foothold in the industry.

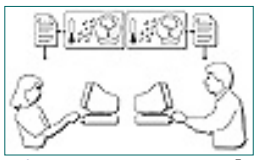

Figure 12.3 With propietary formats, duplication of data and incompatibility between machines becomes common.

This incompatibility is a problem perfect for XML. Although your marketing department and their marketing department are fencing with press releases, you call Jan over at Wacky Weather Tools to discuss the idea of developing a weather standard, an XML data format that both companies can use in addition to your own proprietary formats. A couple calls to Sunny Days brings them on board as well, and the three of you hammer out a Document Type Description, or DTD, that provides the necessary information for all three of your browsers to get a minimal amount of functionality. This DTD is then provided to all interested parties as a set of guidelines for creating documents (see Figure 12.4).

Figure 12.4 A DTD makes it much easier to consolidate data, as well as share it.

DTDs gain a considerable amount of their power from the fact that they are parsed in with your XML documents. If your XML is improper, the parser won't compile the document, although better parsers will indicate where you've gone wrong. This prevents data standards from deviating accidentally and ensures that everyone who needs to use your DTD can in fact read it, either manually or via specialized viewers.

This scenario is precisely what has happened for a number of industries and scientific disciplines, although in many cases, the documents were based upon the older (but lexically similar) SGML. For example, the mathematics community has set a standard convention called the Mathematical Markup Language for use in transcribing complex mathematical notation. The materials and chemicals discipline has the Chemistry Markup Language, a way of describing complex molecules in a consistent fashion. Recently, the Open Financial Exchange (OFX) was ratified by a board of banks and financial institutions, providing a standard for transmission of financial data into the 21st century. All of these standards are instance languages of SGML.

XML is likewise beginning to spawn children of its own. The Channel Definition Format used by Microsoft to specify channel information on the Internet was instrumental in bringing XML to the attention of Web developers. More recently, Macromedia, Microsoft, Hewlett-Packard, AutoDesk, and Visio submitted the Vector Markup Language to the W3C, the committee that arbitrates decisions concerning Internet standards. This language is used for transmitting two-dimensional vector-based graphics over the Internet, and it is based on the XML format. You can view its proposal on the Web at www.w3.org/TR/NOTE-VML.

Similarly, SMIL (Synchronized Multimedia Integration Language) is an XML-based proposal before the W3C that would provide synchronization information for integrated media, essentially an HTML for digital video and audio (the proposal for SMIL is located at www.w3.org/TR/REC-smil/). This technology is still in its infancy but could revolutionize our ability to access information from any media, not just text.

NOTE
A competing specification backed by Adobe, Netscape, IBM, and Sun is the Precision Graphics Markup Language (PGML), which is based upon Adobe's PDF and PostScript formats. The specification for PGML can be found at www.w3.org/TR/1998/NOTE-PGML. This should be taken as an indication that not everyone in the industry is behind XML as a data standard, as well as give you an idea about the amount of back-room politicking involved in getting a standard that everyone agrees upon.

In all of these cases, what has been agreed upon is a common description (the

DTD) that serves as the guide and enforcer of standards. In most cases, you probably will not need DTDs developed at the same level as those that govern the behavior of major software companies, banks, or colleges, but the same reasons that make XML DTDs useful to these companies apply to your department, company, or studio:

- A DTD ensures that the XML data files that you create will work for others who rely on the same format.
- An application that uses a DTD may be able to correct errors in the document before they get promulgated.
- A DTD can be set up to provide default values, simplifying the amount of coding needed for a given file.
- A DTD can define *entities*, macro expressions that can be replaced by other expressions or even entire documents.
- A DTD makes linking between documents possible.
- A DTD can define certain attributes and node values as having data types other than strings (for example, dates).

Building a DTD is not a trivial undertaking, which is part of the reason it is not always required. A simple application (such as the weather report example) doesn't require a lot of definition, but a true interchange format (such as a real weather browser application) might have a DTD that runs thousands of lines long. Moreover, because one of the advantages of XML is its ability to incorporate relational information, the final creation of such a DTD can be serious work. Is it worth it? Usually, yes.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Specifying The Elements

One of the primary tasks of the DTD is to establish a framework in which the document lives—its rule book, so to speak. This rule book can determine the relationship between different tags, such as how a <**LOW**> tag relates to a < **CITY**> tag. An internal DTD (one contained within the same document as the XML data) may look similar to Listing 12.13.

Listing 12.13 The weather report XML structure with an internal DTD.

```
<?xml version="1.0"?>
<!DOCTYPE WEATHERREPORT [
<!ELEMENT WEATHERREPORT (STATE)*>
<!ELEMENT STATE (CITY)*>
<!ELEMENT CITY (SKIES,HI,LOW,PRECIPITATION?,FORECAST)>
<!ELEMENT SKIES (#PCDATA)>
<!ELEMENT HI (#PCDATA)>
<!ELEMENT LOW (#PCDATA)>
<!ELEMENT FORECAST (#PCDATA)>
<!ELEMENT PRECIPITATION (#PCDATA)>
<!ATTLIST STATE
       ID ID #REQUIRED
       TITLE CDATA #IMPLIED>
<!ATTLIST CITY
       ID ID #REQUIRED
       TITLE CDATA #IMPLIED>
<!ATTLIST SKIES
VALUE (SUNNY|PARTLYSUNNY|PARTLYCLOUDY|CLOUDY|SHOWERS|RAIN)
"SUNNY">
<!ATTLIST HI
       C CDATA #REQUIRED
       F CDATA #IMPLIED>
```

```
<!ATTLIST LOW
       C CDATA #REQUIRED
       F CDATA #IMPLIED>
<!ATTLIST PRECIPITATION
       CM CDATA #IMPLIED>
]>
<WEATHERREPORT>
    <STATE ID="California">
        <CITY ID="Los Angeles">
            <SKIES VALUE="PARTLYSUNNY"/>
            <HI C="31" F="87"/>
            <LOW C="18" F="65"/>
              <FORECAST><![CDATA[<B>Partly</B>cloudy]]>
               </FORECAST>
        </CITY>
        <CITY ID="Sacramento">
             <SKIES VALUE="SUNNY"/>
             <HI C="36" F="97"/>
             <LOW C="13" F="64"/>
             <FORECAST>Sunny and hot.</FORECAST>
        </CITY>
        <CITY ID="San Francisco">
            <SKIES VALUE="PARTLYCLOUDY"/>
            <HI C="26" F="79"/>
            <LOW C="14" F="58"/>
            <FORECAST>Partly cloudy and humid.</FORECAST>
        </CITY>
    </STATE>
    <STATE ID="Washington">
        <CITY ID="Seattle">
            <SKIES VALUE="RAIN"/>
            <HI C="20" F="68"/>
            <LOW C="15" F="59"/>
            <PRECIPITATION CM="2.4"/>
            <FORECAST>Raining on and off throughout the day.
            </FORECAST>
        </CITY>
        <CITY ID="Olympia">
            <SKIES VALUE="PARTLYSUNNY"/>
            <HI C="23" F="73"/>
            <LOW C="14" F="57"/>
            <FORECAST>Partly sunny after morning clouds.
            </FORECAST>
        </CITY>
        <CITY ID="Redmond">
            <SKIES VALUE="RAIN"/>
            <HI C="18" F="65"/>
            <LOW C="12" F="54"/>
            <PRECIPITATION CM="4.2"/>
            <FORECAST>Snowstorms in the afternoon.</FORECAST>
        </CITY>
```

```
    </STATE>
</WEATHERREPORT>
```

The DTD is contained as part of the prolog and should appear immediately after the **<?xml** declaration. Such a DTD starts out with a **<!DOCTYPE** node, which essentially points to the root tag of the XML data. The term after the **DOCTYPE** will usually have the same name as the root node of the structure (in this case, WEATHER-REPORT). An internal DTD, such as shown here, usually doesn't require much elaboration beyond this point, although the rules differ somewhat for an external DTD, covered later in this chapter.

The internal DTD, bounded by square brackets, contains a number of different data types The **<!ELEMENT>** node lists a given element and describes what children elements it contains:

```
<!ELEMENT WEATHERREPORT (STATE)*>
<!ELEMENT STATE (CITY)*>
<!ELEMENT CITY (SKIES,HI,LOW,PRECIPITATION?,FORECAST)>
<!ELEMENT SKIES (#PCDATA)>
```

NOTE
If an element can hold all the elements defined within the document, you can also use the simpler **ALL** argument: **<!ELEMENT WEATHERREPORT ALL>**. This is especially useful when dealing with external entities (discussed later in this chapter).

This example declares that the **WEATHERREPORT** element contains zero or more **STATE** elements; the asterisk (*****) tells the parser that if it had been left off, the weather report could have had one and only one state within it. Likewise, the **STATE** element has zero or more **CITY** elements. Just on the basis of these declarations, the XML tags **<WEATHERREPORT/>** and **<WEATHERREPORT> <STATE ID="California"/></WEATHERREPORT>** would both be considered perfectly legitimate.

The **CITY** element is more complex. It shows that the **CITY** must contain **SKIES**, **HI**, **LOW**, and **FORECAST**, and may include the **PRECIPITATION** node (the question mark at the end makes the element optional). Moreover, because the list is separated by commas, these elements must appear in the order given; you can't have the **LOW** tag appearing before the **HI** tag, for example. If the order of the data is unimportant or if it may be coded in more than one order, you can use the pipe character (|) to separate the elements instead.

Note here that the **CITY** element can have only one instance of the listed children; there is no asterisk terminating the list. This makes sense here because more than one **LOW** or **FORECAST** for a given city makes no sense.

The **SKIES** node in turn indicates that the data it contains is parsed character data (**PCDATA**). To signal to the parser that **PCDATA** is not a tag name itself, you need to precede it with a pound sign (**#PCDATA**). Parsed character data includes any and all forms of text, including no text at all—which is the reason that a **SKIES** node takes the **#PCDATA** notation, even though it doesn't actually contain any text.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

An element can contain a mixture of both other elements and text. In this case, it is necessary to use the pipe character to indicate this fact. For example, in the first XML listing in this chapter, a city node's forecast wasn't encased in **<FORECAST>** tags. To create the DTD to allow this, you'd use the following expression:

```
<!ELEMENT CITY (#PCDATA|SKIES|HI|LOW|PRECIPITATION|FORECAST)*>
```

This indicates that there may be zero or more of any of the defined elements, including generic text. Note that you don't need to indicate that a given entry is optional; with the pipe notation, all elements are optional.

Adding Attributes

A data file has a number of different ways of presenting information. Although it is tempting to have a unique element—a separate node—for each piece of information that you need, this approach often works indifferently at best in an OOP environment. On the other hand, it usually doesn't make a lot of sense to put large quantities of text within an attribute string (especially when that text could potentially have embedded quotes that play havoc with the parser). Finding a good balance between using too many and too few attributes takes practice and experience and realistically is a critical part of creating the DTD in the first place.

The **<!ATTLIST>** directive tells the parser to associate a given parser name (and potential values) to a node. As long as the node **<!ELEMENT>** has been defined previously, the < **ATTLIST>** for that element can appear anywhere, although the normal convention is to place it after the elements have been defined. A simple < **ATTLIST>** for the **HI** element shows the basic structure:

```
<!ATTLIST HI
        C CDATA #REQUIRED
        F CDATA #IMPLIED>
```

The term immediately following the **<!ATTLIST** directive is the element that the attribute

list belongs to. After that comes the name of the attribute (here “C”) and an indication of what type of data the attribute is. In this case (as will typically be the case), the attribute is a **CDATA**, or character data, type, which simply means that it contains text characters.

The final term can take one of these three values: **#REQUIRED**, #IMPLIED, or **#FIXED**. A required attribute must be defined within the tag. Omitting it causes the parser to raise an exception. An implied attribute, on the other hand, doesn’t have to be given. For example, with the value given previously, because the Fahrenheit temperature can be ascertained from the Celsius temperature, it is not, strictly speaking, required. An example where it makes more sense is a tag that points to an image file. You can create an **<!ATTLIST>** where the **WIDTH** and **HEIGHT** are implied. If they are left out, then the browser displaying the image can retrieve this information directly from the image file, whereas if they are included, these values overwrite the document defaults and the image gets scaled.

In some cases, it is worthwhile to have an attribute that provides a default value. For example, suppose that the **CITY** attribute had another field called **SRC**, which indicated who provided the data. Typically, the data is collected by the National Weather Service, which makes it a logical default value. You can indicate this within an **<!ATTRLIST>** tag simply by providing the default:

```
<!ATTLIST CITY
       SRC CDATA "National Weather Service">
```

If this is the only source, you might want to set this value as a fixed value. In this case, you apply the **#FIXED** attribute to the end of the expression:

```
<!ATTLIST CITY
       SRC CDATA "National Weather Service" #FIXED>
```

Now, if you attempt to load or change the value of the attribute, the parser will raise an exception. Fixed attributes are useful mechanisms to enforce version-specific information in an XML document.

In some cases, you might want to limit what value a given attribute can take by providing enumerated values. You can provide a list of possible values, and the XML document will only parse the document if the attribute value is in that list. An example of this appears in the **SKIES** declaration:

```
<!ATTLIST SKIES
VALUE (SUNNY|PARTLYSUNNY|PARTLYCLOUDY|CLOUDY|SHOWERS|RAIN)
"SUNNY">
```

The **CDATA** expression has been replaced with a list of possible values, from **SUNNY** to **RAIN**, with a default value of **SUNNY**. If you attempt to set the **SKIES** attribute to **SNOW**, the parser will complain, but it will be perfectly happy with **PARTLYCLOUDY**. This technique is especially useful for situations where an attribute can take a Boolean value: **<!ATTLIST ISRAINING VALUE(TRUE,FALSE) “TRUE”>**, for example, will let you set the **ISRAINING** attribute to either **TRUE** or **FALSE**, but nothing else.

NOTE
At the time of writing, the IE5 parser will properly disallow changes to fixed or enumerated attributes but doesn’t seem to support defaults. Check with the Internet Client SDK at www.microsoft.com/sitebuilder/workshop for updated information on this and related topics.

In addition to enumerated lists, the **<!ATTLIST>** also supports IDs. If you want to use XML as a data structure, IDs are pretty much essential. In the preceding section of this chapter, the **GetIDNode** function iterated through the document to find all ID references. However, if you have a DTD, you can get this capability automatically by setting the type of the attribute to **ID** instead of **CDATA**. For example, the **STATE** node in Listing 12.12 uses this capability:

```
<!ATTLIST STATE
        ID ID #REQUIRED
        TITLE CDATA #IMPLIED>
```

Here, the first ID is the property name, and the second ID indicates that the attribute should be stored in the internal ID list. One problem with native ID support is that each ID in the document must be unique and must follow variable-naming conventions (such as an ID cannot start with a number, and only alphanumeric characters and the underscore character are allowed). For example, “Los Angeles” is not a valid ID name, although “LosAngeles” or “los_angeles” is. Because such IDs often can be rather cryptic to read, a good idea is to include a **TITLE** attribute as well. This attribute contains the “friendly name” of the node.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

To access an ID node, you can use the DOMDocument's **nodeFromID** method. It takes an ID as an argument and returns the corresponding node and is similar in function to the **GetIDNode** defined in this chapter. To get the Los Angeles node, for example, you could call:

```
Set node=DOMDoc.nodeFromID("los_angeles")
Msgbox node.getAttribute("TITLE")
```

This will raise a dialog box with the expression "Los Angeles" in it. The **nodeFromID** method is faster than the **GetIDNode** function but won't work unless a DTD is present that defines ID nodes.

Entertaining Entities And Embedded HTML

Here's a quandary. Your weather browser is, well, dull. It contains just text, raw text, with no pictures to relieve it. Your marketing people warn you about the fact that your competitor's Wacky Weather Browser has pictures and fancy formatting of the forecasts, and as a consequence, its browser is outselling yours two to one. How do you add a little bit of levity to the weather?

This particular problem actually has several different solutions (why do you think I chose it, after all?), depending upon the level of sophistication you want to integrate into your XML application. At the very lowest level, embedding HTML into text, you can gain fine-grain control of your data, although usually at the cost of the most work.

Why embedding HTML into XML causes a problem should be obvious with a little bit of examination. For example, if you want an image to appear within your XML document, the most logical course is to do something like this:

```
<CITY>

        ...
```

```
<FORECAST>
<IMG SRC="sunny.jpg" align="left">Skies will be clear
and warm today, with high clouds forming late.
</FORECAST>
```

There's only one minor problem with this approach. The parser will see the **<IMG>** tag as an XML tag. If the document has a DTD, the document won't parse at all because the **<IMG>** tag isn't defined. Even if the document doesn't have a DTD, the tag will still cause the parser to fail because it lacks a closing tag. Either way, you won't be able to get the document to load, let alone give you anything meaningful.

As it turns out, one solution to this dilemma involves invoking the same mechanism used to display HTML markup code in a browser window. There, obviously, HTML passed to the browser is interpreted as HTML. One way around having tags be interpreted as HTML is to replace the tag symbols "<" and ">" with substitute expressions: **<** and **>** (for less than and greater than). Not surprisingly, because this syntax originated within SGML, you can do exactly the same thing here:

```
<FORECAST>
&lt;IMG SRC="sunny.jpg" align="left"&gt;Skies will be
clear and warm today, with high clouds forming late.
</FORECAST>
```

The characters **&** and **;** act as escape characters, and within XML any expression defined to be so escaped is called an *entity*. Entities act as macros, replacing simple expressions with more complex ones. For example, the **<** entity replaces the string **lt** with the character "<" when the document parses, ensuring that the bracket won't be taken as the start of a new XML element tag. XML automatically predefines only five entities, given in Table 12.7.

Table 12.7 Predefined entities in XML.

Entity Symbol	Is Replaced By
<	< (Less-than sign)
>	> (Greater-than sign)
&	& (Ampersand)
'	‘ (Apostrophe)
"	“ (Double quote mark)

However, entities are so useful in XML that they extend far beyond these predefined objects. Essentially, an entity in XML can replace an expression with any other expression; through the use of entities, you can essentially define a whole separate macro language.

To build an entity, you need to declare it within the DTD of your document. For example, you might want your output to replace the character F with the expression "degrees Fahrenheit". To do this, you set up the entity tag somewhere within the DTD as follows:

```
<!ENTITY F " degrees Fahrenheit">
```

Then, within the body of the XML document, you use the **&F;** notation to signal that you

want the entity to be replaced:

```
<FORECAST>The temperature will be 25 &F;</FORECAST>
```

When the document parses, this will get converted to:

```
<FORECAST>The temperature will be 25 degrees Fahrenheit
</FORECAST>
```

Obviously, it would be better if, instead of the word *degrees*, the ° symbol could appear. You can accomplish this by using character equivalents. They may be familiar to you if you've ever needed to embed a nonstandard character into an HTML page. In this case, the character is represented by a numeric value as defined by the 10,646 character-encoding scheme. This number, called a *character reference*, can be used to represent any character in any alphabet. In the Windows world, these values are the Unicode values for displaying characters.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

For example, the character reference value for the degree symbol is 136, and the character is referenced as **ˆ** (the pound sign signals to the parser that the following number should be treated as a character reference rather than a straight entity). The entity declaration for this might then be:

```
<!ENTITY deg "&#136;">
```

It should be noted that you don't need to restrict an entity to text fields in the XML data structure. You can just as readily put entities in attribute tags or even in XML tags. For example, the **LOW** and **HI** temperature references might look like this

```
<LOW C="10 &deg;C" F="40 &deg;F">
```

which would be parsed into:

```
<LOW C="10 °C" F="40 °F">
```

This is fine for output to straight text, but the degree character will probably not show up properly if the text is used for HTML output. To do that, you can cause the parser to defer expanding escaped characters by replacing the ampersand, its own expansion character, in the entity definition

```
<!ENTITY DEG "&amp;#136;">
...
<FORECAST>The temperature is 25 &DEG;C</FORECAST>
```

which would be parsed into:

```
<FORECAST>The temperature is 25 &#136;C</FORECAST>
```

NOTE
The particularly observant reader may have noted that I used an entity expression

within the definition of an entity. One of the things that make entities so powerful is that you can use one entity to create another entity—anywhere within a DTD. This feature can also lead to a certain loss of control within the DTD, so it should be used sparingly.

When an HTML browser then displays that expression, it converts the **ˆ** into a degree symbol inside the Web page.

Back to the initial problem of displaying the graphic image, it is worth noting that you could also actually embed HTML code into an entity:

```
<!ENTITY imgSunny "&lt;IMG SRC='images/sunny.jpg'
ALIGN=left&gt;">
...
<FORECAST>&imgSunny; It will be clear and sunny
day today.</FORECAST>
```

It's still necessary to turn the tag brackets into entity declarations because the expression gets interpreted during the parse. Still, this is a superb way of setting up a table of predefined graphics (such as **imgPartlyCloudy**, **imgRain**, and so forth) that could then be referenced as needed. The **nodeValue** for the forecast node then appears as:

```
<IMG SRC='images/sunny.jpg' ALIGN=left> It will be a clear
and sunny day today.
```

Still, it would be nice not to have to deal with escaped characters at all, because:

- They require a certain amount of editing of straight HTML files in order to be useful.
- They make the markup less legible than brackets and other characters do.
- They can be especially inconvenient when dealing with dynamic HTML.

So what other alternatives are available?

Because this particularly vexing problem was understood when the XML spec was initially created, there is a mechanism in place for retaining both formatting and "dangerous characters." Within a given text node in the document's main body, you can indicate to the parser that the enclosed text is "dangerous" by encasing it within a **<![CDATA[>** tag. This tag declares that the enclosed text is character data that should not be interpreted by the parser. For example, the forecast given previously can also be displayed as:

```
<FORECAST>
<![CDATA[
<IMG SRC='images/sunny.jpg' ALIGN=left> It will be a clear
and sunny day today.
]]>
</FORECAST>
```

In this case, the entire expression is protected and will output exactly as displayed. An interesting consequence of this is that you can use the **CDATA** tag to display code,

which relies heavily on line breaks to indicate functionality, as is the case with the forecast in Listing 12.14.

Listing 12.14 Using **CDATA** to store code in an XML document.

```
<FORECAST>
<![CDATA[
<SCRIPT LANGUAGE="JAVASCRIPT">
function rollover(me){
       me.src='images/sunnyHi.jpg";
       }
function rollout(me){
       me.src='images/sunny.jpg";
       }
</SCRIPT>
<IMG SRC='images/sunny.jpg'
       ONMOUSEOVER='rollover(this);"
       ONMOUSEOUT="rollout(this)
       ALIGN=left>
       It will be a clear and sunny day today.
]]>
</FORECAST>
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Examining External Entities And DTDs

By now, wheels may be turning in your head about the potential usefulness of entities in XML. I've saved the best for last. Entities aren't specifically limited to objects defined within the XML document. An entity can also retrieve text or binary information from an external file, something called an *external entity*. External entities come in two flavors—**PUBLIC**, which references a file or data stream via a URL, and **SYSTEM**, which pulls the file up through the native file system. The declaration for either flavor is identical:

```
<!ENTITY EntityName PUBLIC "myURL">

<!ENTITY EntityName SYSTEM "myFilePath">
```

An external entity could be an XML document; indeed, this is typically the case for documents that serve as collections of other documents. Consider the weather report again in this light. Each state node (and subnodes) could actually make up a distinct document—Alabama.XML, Alaska.XML, and so forth. A complete weather report for the nation could then consist of the reports for each state, as shown in Listing 12.15.

Listing 12.15 An XML document made up primarily of external entities.

```
<?xml version="1.0">
<!DOCTYPE NATION [
<!ELEMENT NATION ALL>
<!ENTITY ALABAMA SYSTEM "c:\weather\Alabama.XML">
<!ENTITY ALASKA SYSTEM "c:\weather\Alaska.XML">
<!ENTITY ARKANSAS SYSTEM "c:\weather\Arkansas.XML">
...
<!ENTITY WYOMING SYSTEM "c:\weather\Wyoming.XML">
]>
```

```
<NATION>
       &ALABAMA;
       &ALASKA;
       &ARIZONA;
       ...
       &WYOMING;
</NATION>
```

This example begins to illustrate some of the power of external entities in an XML document. If the files in question were generated on a recurring basis, an XML structure such as Listing 12.15 could serve as a collector of this data for later processing by a JavaScript applet or a Visual Basic DHTML application. Moreover, a program could actually generate an XML file that passed parameters to a server-side application, which in turn returns additional XML files, as shown in Listing 12.16.

Listing 12.16 An example of how an XML file could be generated from server-side Active Server Pages (ASP) calls.

```
<?xml version="1.0">
<!DOCTYPE NATION [
<!ELEMENT NATION ALL>
<!ENTITY ALABAMA PUBLIC "http://www.WeBWeather.com/cgi-bin/
GetWeather.asp?state=Alabama&time=0125PM">
<!ENTITY ALASKA PUBLIC "http://www.WeBWeather.com/cgi-bin/
GetWeather.asp?state=Alaska&time=0125PM">
<!ENTITY ARKANSAS PUBLIC "http://www.WeBWeather.com/cgi-bin/
GetWeather.asp?state=Arkansas&time=0125PM">
...
<!ENTITY WYOMING PUBLIC  "http://www.WeBWeather.com/cgi-bin/
GetWeather.asp?state=Wyoming&time=0125PM">
]>
<NATION>
       &ALABAMA;
       &ALASKA;
       &ARIZONA;
       ...
       &WYOMING;
</NATION>
```

Because these calls are handled by the parser, the data reaches the client transparently, structured in a navigable form and easily integrated into an HTML client through the use of Internet Explorer 5's support of XML tags.

NOTE
You can import HTML documents into an XML structure in a similar manner, but you'll need to make a few minor changes to the document first. If you define an HTML page as an entity, the parser will attempt to parse the text as it loads, which, given the slight incompatibility between HTML and XML, almost guarantees that the load will fail. If you place <**![CDATA[** and **]]**> tags before and after the document, the whole document will be treated as one long **CDATA** block by the parser, effectively transparent to the program.

The use of **PUBLIC** and **SYSTEM** isn't limited to entities. In most cases, especially when working with a preexisting data structure format, you can link the DTD itself to an external file. The syntax for this is straightforward; in the prolog, the declaration becomes:

```
<!DOCTYPE MyRootnode SYSTEM "c:\myPath\myDTDFile.dtd" [
<!ELEMENT ...>
<!ATTRIBUTE ...>
<!ENTITY ...>
]>
```

It can also become:

```
<!DOCTYPE MyRootnode SYSTEM
"http://www.myServer.com/myDTDFile.dtd" [
<!ELEMENT ...>
<!ATTRIBUTE ...>
<!ENTITY ...>
]>
```

The DTD file itself then consists of everything that would have been inside the brackets of an inner DTD:

```
<!ELEMENT WEATHERREPORT (STATE)*>
<!ELEMENT STATE (CITY)*>
...
<!ATTLIST LOW
       C CDATA #REQUIRED
       F CDATA #IMPLIED>
<!ATTLIST PRECIPITATION
       CM CDATA #IMPLIED>
```

It is perfectly acceptable to have both an external DTD file and an internal one; anything declared within the internal file will automatically override element declarations or entity definitions in the external DTD. Also, because entity definitions typically concern individual documents rather than the data structure as a whole, entities will typically be defined within the internal DTD.

The Document Type Definition serves as a blueprint for the XML document, one that you can update dynamically and use to incorporate other documents and provide validation and defaults for the data contained therein. It can also be frustratingly complicated; the last couple of sections only begin to hint at the degree of complexity and sophistication that XML ("a simplified version of SGML for the Web"—hah!) brings to both Web developers and Visual Basic programmers.

Still, although XML makes an effective data transfer mechanism, it also forms much of the underpinnings of the new Web, and Internet Explorer 5 and Visual Basic 6 are the perfect vehicles for realizing this. It's worth examining XML in a different light—as the successor to HTML.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Building An XML Application

It’s sometimes hard to remember that HTML is not yet even 10 years old. Certainly in its brief life, that child of SGML has become one of the most pervasive media on the planet. However, like any technology that explodes into prominence, HTML may make it difficult for new and possibly better formats to succeed. Already, Web developers face the daunting (and all too common) problem of creating Web pages that work for even a handful of the browsers currently on the market. Adaptable plug-ins such as Macromedia Shockwave or Adobe Acrobat can provide a certain degree of uniformity, but at the cost of requiring specialized production software. The two versions of DHTML that Microsoft or Netscape offer are so different from one another that using one for specialized page display virtually guarantees that your Web page will fail (sometimes spectacularly) on the other. Java applets, even when they do live up to the promise of develop-once-play-anywhere, often perform sluggishly and unpredictably; again, to do any significant development with them, you need a real programming environment to support it with associated costs.

What about XML? Is it the panacea for the common site? In a word, no. XML is cool technology, and people are only just beginning to understand where it can be used. In time, it may replace HTML in many applications, although certainly not all of them. For starters, with XML, not only do you have to define the data, but you also become responsible for formatting output and displaying that data, something that HTML has already become fairly adept at assuming. Additionally, there is no consistency yet with XML parsers; the API set that one supports may be completely different from that which any other parser supports, making it exceedingly difficult to use on all platforms.

You can still build HTML applications targeted toward a specific browser or browser set. If you are using Internet Explorer (any version, not just IE5), you

can also leverage the powers of DHTML applications that I'll be creating in Chapter 14, simply by incorporating the IE5 MSXML.DLL component into your VB application. IE5 gives you advanced support of XML and is the target of choice if you're using XML for formatting as well as data display. However, through the use of DHTML applications, you can also target Internet Explorer 3.0 and 4.0, and using Visual Basic IIS applications let you extend that to any browser, although at a loss of interactive control.

With a standard in place, it's now time to get the next browser out the door, one that takes advantage of all the capabilities hammered out in the WIML format (that is, the Weather Information Markup Language, based on XML). Determining which pieces you need to develop illustrates both the power and dilemma of the new Internet programming paradigm. Where does your application reside? Consider the following constraints in building such a "browser" (see Figure 12.5):

Figure 12.5 XML data can come from a database query as readily as from a static local file.

- The primary browser will use XML principally as a data source running within an ActiveX-compatible browser. Although you can use XML purely as a markup language, this solution makes sense from the standpoint of supporting the largest number of ActiveX-compatible browsers.
- The XML data source is a WIML-compatible file residing at a URL. The XML document itself will almost certainly have external entity references, which are in turn generated from an SQL Server database that is updated every 10 minutes with information from all over the country. From the standpoint of the client, however, the XML document exists as a complete entity. The browser doesn't need to know anything about how the document is made. The server in turn only needs to know how to generate and transmit the XML file; it is ignorant of what the client will do with the data.
- The logic of the client page is generated by a DHTML application component written in VB6. The component may be sensitive to some of the advanced features of IE5, but it degrades easily to IE4.
- For non-DHTML browsers (or ones that don't support ActiveX), a page server written as an IIS control in VB6 resides within a separate ASP page, providing HTML 3.2 compliant code.

This spanning between client and server provides a good example of where application development is moving. The client and server are both intelligent, with both containing not only the logic of their respective environments but also any specialized modules such as the weather client and weather server programs. Even "shrink-wrapped" products are moving to this paradigm: The client resides within an application rather than a Web page, but with Visual

Basic 6, the logic that handles the mediation between client and server could very well be the exact same module.

Where To Go From Here

This chapter examined XML in some depth, especially as it applies to data usage in Visual Basic. Microsoft has made it quite clear that its future product strategies will be very XML dependent, and while some of the material covered here is still in a preliminary state, XML is likely going to end up as one of the primary formats for communicating disparate types of data between applications, computers, and networks.

Over the next few chapters, the WeBWeather browser and server will be built, piece by piece, incorporating DHTML, ADO, ASP, and the whole host of other acronyms that make up Microsoft's take on the Internet. I still have yet to cover a few facets of the XML parser (such as creating and saving XML on the fly), but they will be discussed in more detail in the next couple of chapters.

Now it's time to beat Wacky Weather in the weather browser war…. (May the best cold weather front win.)

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Chapter 13
Serving Up The Web

Key Topics:

- The history of serving up the Web
- The problems with scripting
- Web Classes—VB's Web solution
- Beyond the canon: real-world Web
- Linking events
- Getting browser capabilities

The balance between client and server was once fairly well defined; the client was a dumb terminal with just enough intelligence to display pixels on the screen and maybe to sound a bell. The server was *the* computer; it did all the processing, database management, and computation. As the client has gained in strength and speed, the server's role has shifted as well because it increasingly acts as a source of data and little more. The Internet also changes this equation—to the extent that it is sometimes difficult to tell where the server ends and the client begins.

Visual Basic 6 blurs this distinction to a point where the server and client roles switch from one machine to the other several million times a second. Microsoft has introduced several significant additions to Visual Basic's lexicon that make it a viable Web tool on both the client and the server.

Serving With Distinction: A History Of CGI

The earliest Web developers (the Neolithic protohumans of the late 1970s,

Homo unixis) wrote server programs as a way of sending documents from one computer to another. The Internet's architecture brings with it a situation that is atypical of most dedicated servers: The system was essentially designed as a way of maintaining computer-to-computer communications in the event of a major disaster. Because of that, such networks are fundamentally stateless: A computer is connected to another computer only at the time of the transaction. Before and after that time, the computer knows absolutely nothing about the other computer. This communication protocol is remarkably robust because it places the emphasis on the information being communicated rather than the source, but it makes the other type of server transaction—database processing—considerably more complicated.

People deal with the world symbolically, and their relationship with computers seems especially filled with the baggage of mankind's metaphors. One notable example of this is the notion of a "file." In the material world, a file is a container, a holder of data about a thing or process. All the information in a file is related in some way, and the whole is kept together under a convenient name (usually written with a messy ink on a peeling label that is guaranteed to fall off at the most inconvenient time).

From the standpoint of a computer, however, a file is simply a string of bytes transferred from one location to another. A file server reads the data from the file into a stream of bits being sent out through a modem, where it is probably turned into acoustical waves and then into modulated pulses of electromagnetism. A Transport Communication Protocol/Internet Protocol interpreter, better known as a TCP/IP stack, takes the file before it is sent and breaks it into discrete chunks that have absolutely no respect for where the cuts occur, collects the packet on the other end, and then reassembles the chunks into something resembling the initial file. Finally, a program on the client side does something with that file, either storing it on a hard drive or blasting it into memory to be interpreted.

The earliest Internet protocols dealt strictly with files as discrete units, although not everything that is transported is a file. The File Transport Protocol (FTP) illustrates this point well; most FTP commands obviously handle the transmission or receipt of files from one computer on the Internet to another. However, most FTP calls request additional information from the computer that is generated by the server without files to reference. For example, **MKDIR** sends a command to the host computer to create a new directory, usually with some additional status information indicating that the command succeeded or failed.

The Hypertext Transport Protocol (HTTP) takes the model one step further. The backbone protocol for the Web, HTTP is a file transfer mechanism coupled with a linking mechanism. Although part of the purpose of a browser is to display the information contained within an HTML file, another part is devoted to handling the links in a document. The links, when clicked, send a request to the server to retrieve another document (hence the *hypertext* in HTTP).

The ability to retrieve documents is certainly useful, but shortly after the paradigm was introduced, people wanted HTTP to do more. A file is a unit of

information, usually a structured set of data; however, it is also a stream of bytes sent by the server—and if the server can send that set of bytes, why not send bytes that are the result of a query or equation? In this way, the notion of Computer Gateway Interface (CGI) was born. This program reads the URLs that are sent to it and checks whether the requested URL corresponds to an executable program. If the file can run as a program, CGI then passes any additional data within the URL as name-value pairs of parameters to the equation, which in turn sends a string back to the client machine for it to interpret.

Because the first Internet servers were Unix-based machines, the initial CGI programmers wrote using either C or Unix shell commands. However, C is not and never has been a terribly good language for manipulating text; most C string functions exist as libraries distinct from the kernel of the language, and C in its usual form does not have a native string class or data type. A number of new languages arose, principally based upon modules of Unix-based word processors. The names of these languages hint at the acronym mania that seems endemic in the computer industry: SED (String Editor), AWK, GREP, and so forth.

In the early 1990s, Larry Walls and Randal Schwartz wrote a highly flexible new language, called Perl, with a distinctive mantra: “With Perl, there’s always more than one way to do anything.” Combining string processing with operating system access, Perl rapidly became the lingua franca of the Internet, and even today, a healthy portion of all CGI scripting is performed in Perl, especially on Unix servers.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The Rise Of ASP

Microsoft's entry into the Web server market came extremely late in the game. On the client side, Netscape's Navigator browser quickly captured the hearts and minds of Internet users throughout the mid-1990s. Internal evidence seems to indicate that Microsoft had become fixated on the sudden rise of America Online in the commercial information services market, a field pioneered by CompuServe in the early 1980s. In comparison to the surge of AOL, the Internet seemed small potatoes.

Microsoft rolled out its Microsoft Network (MSN) in early 1995 with great fanfare. (I was actually present at the preliminary developer's conference for MSN when it rolled out, after writing a baseball simulation piece for the network.) The response to MSN was underwhelming at best; technical problems, insufficient servers, and a poorly designed user interface plagued the service just as the Internet was entering the radar of the general public. Three years later, MSN was quietly shut down and its developers and support staff reassigned to new ventures.

However, a curious artifact of MSN still survives. Developers working with MSN made use of a tool, code-named *BlackBird*, created by Microsoft. This tool was intended to be released to the general developer audience, but MSN's poor reception and problems with BlackBird kept it off the shelves. When Microsoft decided to capture the minds and hearts (and wallets) of the Internet-using public in December 1995, Blackbird became an essential weapon in that effort.

In early 1996, Microsoft released two new products: Internet Information Server (IIS) and Visual InterDev. These programs, of course, were essentially the server and development environment from Blackbird, retooled to take advantage of the protocols of the now red-hot Internet. IIS became a major selling point for Windows NT systems, which had actually been languishing

somewhat in sales.

IIS introduced Active Server Pages (ASP), a concept that, although hardly new, dramatically simplified the process of creating CGI programs. An ASP document is a template that mirrors what the final Web page will look like. Unlike templates in Perl, however, the template document contains scripting code. When an ASP is requested from a client, IIS runs the scripts within the template document. The results are sent as a stream to the client, indistinguishable from any other HTML document.

The power of ASP comes from its ability to communicate with databases before the data is sent down the pipe. Perl had rudimentary data access capabilities at the time (although it has improved dramatically in that area), but it's worth remembering that Microsoft established ODBC as an industry-wide database exchange format years before Perl was even conceived. By serving up the data along with the Web page structure, Microsoft made database-driven Internet sites far easier to create and more appealing to use. This appeal, combined with NT's aggressive pricing, has made a significant dent in what had once been non-Microsoft turf—the server market.

Until recently, Visual Basic has had a fairly peripheral role in server-side programming. The primary scripting language in ASP is a stripped-down version of VB called VBScript, which borrows most of the syntax but none of the typechecking of its older brethren. Although powerful in its effect, VBScript does have a number of significant limitations (discussed in more detail in the next section) that have frustrated developers for some time. With Visual Basic 5, Microsoft did introduce ActiveX DLLs, components written in VB or other COM generators that had no visual interface but otherwise exposed a set of properties and methods that could be used to modify ASP output. These controls typically added adjunct capabilities to ASP—performing calculations, searching databases, retrieving local or remote data files, or checking the state of software or hardware devices. Much more rarely did the DLL actually serve up the whole page, primarily because Visual Basic 5 doesn't really have an environment for creating Web pages of any sophistication.

Visual Basic 6 changes that limitation in such a way as to possibly establish a new paradigm of Web development. Visual Basic 6 introduces two new document types: Internet Information Server applications (IIS apps) and Dynamic HTML applications (DHTML apps) for the server and client, respectively. Central to both of these is the notion that "Web pages" within the browser can offer the same functionality as traditional forms, plus many additional benefits that forms can't supply:

- Because of their comparatively large size, traditional windowed applications don't translate well across the Web.
- Web pages can be designed to be cross-browser and cross-machine compatible, whereas a Visual Basic 6 windowed app will only run in 32-bit Windows.
- Forms are fixed. Once a form is compiled, it's much more difficult to modify its interface. A Web page, on the other hand, is considerably more malleable, incorporates such features as resizing, and can

incorporate new ActiveX and Java components without needing to be recompiled.

- Incorporating sophisticated graphics in a form is difficult at best; such features as irregularly shaped buttons and rich formatted text (things that multimedia products such as Macromedia Director have supported for years) can make even the best VB programmer tear out his hair in frustration. The multimedia engines within the latest crop of browsers support these two features, and Internet Explorer 4 and 5 provide an embarrassing wealth of riches for multimedia programmers, including 3D manipulation, transparency, layered graphics support, animation paths, and sequence controllers.

Why haven't Web applications become the primary means of developing client/server systems? Part of the answer comes from the wild roller-coaster ride of technology releases. Web browsers are released with radically new features every 9 to 12 months, and two major interests (the Microsoft/Intel COM-based consortium versus the IBM/Sun/Netscape Java contingency) strive to create the most marketable products, so it's difficult to determine which standard to follow. From the standpoint of the actual developer, another problem creeps in—the difficulty of creating and maintaining scripted code.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The Problem With Scripting

Scripting an ASP or a Web page in theory should be simple: The language is a type-insensitive version of Visual Basic or Java, and the syntax is straightforward. However, what is simple in theory can often get remarkably ugly in practice. For example, an ASP that reads from a database to populate a table might look something like Listing 13.1. Don't worry if the code doesn't make sense yet; ASP is covered in much greater detail throughout this chapter.

Listing 13.1 A typical example of an ASP script for building a table.

```
<%
dim conn      ' This declares the connection
dim rs        ' This declares the record set
dim fIndex    ' This defines an index for iterating
              ' through fields

set conn=Server.Create("ADODB.Connection")
conn.Open="WeatherReports"
set rs=conn.Execute("SELECT * FROM ReportData _
  WHERE State='Washington';")
%>
<HTML>
<HEAD>
</HEAD>
<BODY>
<TABLE>
<THEAD>
<%
for fIndex=0 to rs.fields.length-1
%>
```

```
<TH><%=rs.fields(fIndex).name%></TH>
<% next %>
</THEAD><TBODY>
<%
rs.MoveFirst
while not rs.EOF
%>
<TR>
<%for fIndex=0 to rs.fields.length-1%>
<TD><%=rs(fIndex)%></TD>
<%
next
%>
</TR>
rs.MoveNext
wend
%>
</TBODY></TABLE>
rs.Close
%>
</BODY></HTML>
```

If you had trouble following exactly what was happening, don't feel bad. ASP introduces what is in essence a macro language into the body of an HTML document; but with even fairly simple examples, such as Listing 13.1, the HTML can get overwhelmed by all of the script. This problem gets worse when JavaScript for the client is added to the mix, especially when the JavaScript relies upon information output from the ASP script. Add the mangling from most HTML editors (including, unfortunately, Microsoft's Visual InterDev and FrontPage), and you begin to appreciate the problems inherent in using script to automate your page output.

Another related problem to this creeps into adding script to Web pages. Once the page is marked up, a WYSIWYG-based HTML editor may no longer even understand the code, let alone be able to reconstruct enough of the page to edit it. In essence, once the Web designer hands off the page to the programmer, making changes to the HTML is nearly impossible for the designer, often forcing the programmer to work as a Web designer as well. Some programmers are adept at graphic design, but the whole left-brain/right-brain duality seems even more apt in page design than in most fields: The best programmers usually make lousy graphic designers and vice versa.

Performance is an issue with scripting. The binding of objects takes a certain amount of time, and an object defined in a script is implicitly late bound. This means that it can take a thousand times as long to initialize an object in a script as it does to implement the same thing in a compiled language. Because this initialization is occurring in the latency-filled environment of the Web, an additional three to five seconds per binding adds up fast.

A final limitation of any sort of scripting languages, whether client or server, is that such code is essentially completely exposed. This is of less concern on the server side (although even there such code isn't completely safe), but on the

client side scripting is fully visible to anyone. Even if a Web page includes a script as an external file, an astute person can download a copy of the same script simply by typing the URL of that script into the browser's window. From the standpoint of a developer, this means that hours or even days of work on scripting code can be stolen in seconds. More seriously, for a database administrator, open JavaScript code essentially provides complete access to a database, especially with such technologies as Remote Data Services (RDS) making it possible to query a database from the client side. Script is fundamentally unsecured.

My point is not that scripting doesn't have its place. Script is lightweight in more than functionality. It has minimal need for a programming environment, it occupies far less bandwidth than an ActiveX control or Java component, it's easily modified, and it can be self-generating—all of which ensure that for many smaller tasks, scripting will not disappear anytime soon.

It would be nice for both designer and programmer if the same table from the earlier example could be expressed like Listing 13.2 instead.

Listing 13.2 An idealized template for ASP output.

```
<HTML>
<HEAD></HEAD>
<BODY>
<WS:MYTABLE>
<TABLE>
<THEAD>
<TH>Placeholder Heading 1</TH>
<TH>Placeholder Heading 2</TH>
</THEAD>
<TBODY>
<TR>
<TD>Placeholder Data 1</TD>
<TD>Placeholder Data 2</TD>
</TR>
</TBODY>
</TABLE>
</WS:MYTABLE>
</BODY>
</HTML>
```

In this case, the table displayed is strictly a placeholder for design purposes and will be replaced later by the real table. The code is completely human and machine readable, which means that the designer can create this code and even let a programmer work with the implementation while she continues to redefine the appearance of the interface. Furthermore, there's nothing in the HTML that gives away details about how the table gets generated or exposes sensitive password or database information.

NOTE
Although there are some similarities between server-based IIS apps and client-based DHTML apps, their differences are enough to treat them as

distinct subjects. In Chapter 14, I cover DHTML applications in much greater detail.

What you see in Listing 13.2 is a simple template file for an IIS application. The template is contained within a specialized ActiveX DLL, located on the server, which can be requested as if it were a standard ASP. From the client's point of view, *there's no difference between this application and any other HTML page*. This point is important: Just as with an ASP, an IIS application can generate output for *any* browser, as long as Internet Information Server 4.0 is used on the server side.

The IIS application, on the other hand, is written in Visual Basic—not VBScript, but the full-blown Visual Basic 6.0 for Windows, Professional or Enterprise Edition. This means that components and variables can be early bound for improved performance, that you can use classes with all the benefits they bring to VB, and that you have the full weight and power of the Visual Basic developing and debugging environment to rely on. This is one of the reasons why programmers are getting so excited about the latest release of Visual Basic.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

MIISSIISSAPPI? Spelling Out IIS Applications

For many VB programmers who have been painfully slugging out code in Visual InterDev 1.0, one of the secretly whispered hopes was that Microsoft would write an add-in to Visual Basic that would let developers write ASP code from within VB.

Visual Basic and VBScript are syntactically quite close, but one is a language with data-typing (even if the typing is weaker than for C++), whereas the other is essentially type-agnostic. This means that even if you wrote your ASP code in Visual Basic, you still need to strip out the type declarations at the beginning, change all **New** calls to **CreateObject** calls, and eschew classes unless you want to include them as ActiveX servers. Using Visual Basic, you also lose the benefit of embedding HTML code in your scripts, although you can always use the **Response.Write** syntax and grit your teeth.

Microsoft could very well have offered this option of writing ASP code in VB. Admirably, it did not. Instead, the product developers thought through the reasons for using ASP in the first place and worked toward a solution that solved the underlying problems with ASP scripting. What they came up with was the notion of a *Web Class*. It should be pointed out that this is not a class in the traditional sense that Visual Basic uses it. A Web Class has several unique attributes:

- Instead of forms, Web Classes use HTML-based templates that are output to a browser (which doesn't have to be Internet Explorer 4 or 5).
- The Web Classes can modify the templates before serving them by replacing custom tags in the template with content. A class can even replace the content of a tag with other custom tags, which can in turn be replaced with additional data or tags, making it possible to create complex recursive structures.
- Web Classes act as servers, and form and query information can be

sent to them in exactly the same way that such information is sent to an ASP. This makes possible exceedingly complex Web applications that handle data validating, formatting, and error processing.

- You can set up Web Class events in the final HTML document that query back into the server. This feature can provide a certain level of dynamic functionality, even for documents that don't support DHTML.
- Web Classes can also maintain state across a session through a number of different mechanisms, from session-persistent variables to cookies on the client side.
- Web Classes automatically expose the Active Server Pages 2.0 Object Model, making it far easier to integrate database and other ActiveX data providers, as well as data structures such as the File System Object and Dictionaries.

IIS applications simply provide an alternative to scripted Active Server Pages with all the benefits and drawbacks that ASP has. If ASP is working with a non-DHTML browser, the pages still need to be refreshed through server calls. On the other hand, an IIS app designed jointly with a DHTML application on the client side completely rewrites all the rules for client/server development. The IIS app customizes the data in response to the DHTML app, which in turn customizes the presentation in response to the data it receives. Integrated VB 6-based client/server applications are covered in Chapter 15.

Creating The Canonical "Hello World"

You knew it was coming. *The Computer Book Writer's Handbook*, page 127, paragraph 3, outlines a requirement to include at least one "Hello, World" example in any book. The Hello World Foundation receives 10 cents for every appearance of a Hello World program in any computer book in the world (including derivative rights for "Guten Tag, Welt," "Hola, Mundo," and "G'day, Mates"). They use the money to underwrite the pizza and highly caffeinated soda industries, staples of programmers everywhere.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Creating a “Hello World” program with templates is straightforward and I cover it shortly, but it’s entirely possible to do it without templates.

1. Open Visual Basic 6 and start a new project by selecting File|New Project. Select the IIS Application project type (see Figure 13.1).

Figure 13.1 To create a new IIS application, double-click the IIS Application project type icon in the New Project dialog box.

2. Open the Project Explorer window, and expand the view to show the Web Class icon (see Figure 13.2).

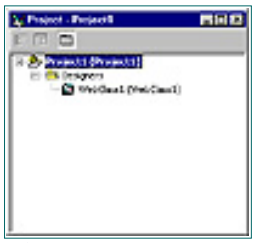

Figure 13.2 From the Project window, you can add new Web Classes.

3. Double-clicking the Web Class icon will launch the Web Class designer, a specialized window that lets you edit templates, add events, and write event handlers (see Figure 13.3).

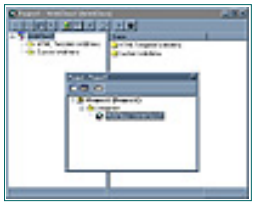

Figure 13.3 The Web Classes are examples of Microsoft’s Designers, specialized

code templates that encapsulate complex code into usable design-time packages.

4. For now, ignore the HTML Template WebItems and the Custom WebItems categories. Double-click the WebClass icon to launch a standard procedure window for **WebClass_Start**. This procedure should already have data in it, as shown in Listing 13.3.

Listing 13.3 The default output for a Web Class.

```
Private Sub WebClass_Start()
    ' Write a reply to the user
    With Response
        .Write "<html>"
        .Write "<body>"
        .Write "<h1><font face=""Arial"">WebClass1's _
          Starting Page</font></h1>"
        .Write "<p>This response was created in the Start _
          event of WebClass1.</p>"
        .Write "</body>"
        .Write "</html>"
    End With
End Sub
```

The code in Listing 13.3 provides the standard output for a Web Class if you leave it in its default condition. You can modify this document yourself to create the canonical “Hello, World” Web page, as shown in Listing 13.4.

Listing 13.4 Modified code output for the “Hello, World!” page.

```
Private Sub WebClass_Start()
    ' Write a reply to the user
    With Response
        .Write "<html>"
        .Write "<body>"
        .Write "<h3>Well, here goes nothing:</h3>"
        .Write "<h1>Hello, World!</h1>"
        .Write "<p>There, feel better?</p>"
        .Write "</body>"
        .Write "</html>"
    End With
End Sub
```

Once you’ve finished, you can run the Web page just as you would any standard VB program (press the Play button or F5). One of two possible things will happen. If you are currently running Windows 95 or 98, Peer Web Services will start if it is not currently active. Formerly known as the Personal Web Server, Peer Web Services provides a local server for testing Internet applications on a LAN. If you are running Windows NT (which will almost certainly have IIS active), you won’t see any notification of Web services being initiated.

Eventually, the server will display the Web page shown in Figure 13.4. It is worth noting that somewhere along the line, a new ASP file called WebClass1.ASP was apparently created; this was actually generated by the Web Class itself. However, there’s more

happening than just the creation of a Web page file. With a bit of searching (primarily in the Temporary Internet Files folder in your Windows directory), you can find the contents of this particular page. The results (shown in Listing 13.5) may surprise you.

Figure 13.4 The "Hello, World!" output produced by your modified page.

Listing 13.5 The rather unexpected content of WebClass1.asp.

```
<%
Server.ScriptTimeout=600
Response.Buffer=True
Response.Expires=0

If (VarType(Application("<WC<WebClassManager")) = 0) Then
   Application.Lock
   If (VarType(Application("<WC<WebClassManager")) = 0) _
     Then
       Set Application("<WC<WebClassManager") = _
       Server.CreateObject("WebClassRuntime.WebClassManager")
   End If
   Application.UnLock
End If

Application("<WC<WebClassManager").ProcessNoStateWebClass _
   "Project1.WebClass1", _
             Server, _
             Application, _
             Session, _
             Request, _
             Response
%>
```

Rather than contain HTML code, the ASP code actually creates a distinct process in memory to run Project1.WebClass1, which is the project you just created. All the data is actually contained within the Web Class, and the purpose of the ASP file is to create the project's DLL in memory and retain state information across pages. Clearly, there's more going on than meets the eye.

Beyond The Canon

You can set the name of the ASP document (and the server where it is hosted by default) from within Visual Basic. Open the Properties window and select the WebClass1 object. This is an instance of the Web Class. Within this window, you can set both the name of the object and **NameInURL** for the class. This latter value is important because it is effectively the URL of the Web Class. For example, if the **NameInURL** was changed from WebClass1 to WeatherReport, the URL becomes

http://www.myServer.com/Project1/WeatherReport.ASP. You can also assign the folder project location by changing the project name of the IIS app: Select Project Properties from the Project menu, and then change the Project Name to the name of the folder where you want to host the application.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

A Pavlovian Response

If you have ever worked with ASP code, you might have noticed the line **With Response** and wondered whether the "Response" object called here has anything to do with the ASP **Response**. As a matter of fact, it *is* the ASP **Response**, with the primary purpose of outputting text (usually HTML text, but this isn't a requirement) to the browser. This gives you a hint about what's happening under the hood, something worth stating explicitly:

A Web Class accesses the same objects that a traditional ASP does, including the **Response**, **Request**, **Application**, and **Session** objects. If you can program ASP, you can program Web Classes.

The **WebClass_Start** method is invoked whenever the Web Class is first loaded, and as a consequence, it serves as the starting point for your "Web site." In the simple example featured previously, the output was generated completely through the use of **Write** statements, and as written, it isn't any different from serving an HTML page with the same script. However, because this is a Visual Basic procedure, you can do a great deal more with the same script. For example, the same code that was displayed earlier in this chapter could be integrated into a VB procedure with few modifications. Listing 13.6 reads the content of a table from a database DSN and outputs the result into a table.

Listing 13.6 Subroutine that outputs the contents of Washington's weather data.

```
Private Sub WebClass_Start()

        dim conn as Connection     ' This declares the
                                   ' connection
        dim rs as RecordSet        ' This declares the record
                                   ' set
        dim fIndex as integer      ' This defines an index for
                                   ' iterating through fields

        set conn=Server.Create("ADODB.Connection")
        conn.Open="WeatherReports"
```

```
        set rs=conn.Execute("SELECT * FROM ReportData _
          WHERE State='Washington';")
        Response.Write "<HTML><HEAD></HEAD><BODY>"
        Response.Write "<TABLE>"
        Response.Write "<THEAD>"
        for fIndex=0 to rs.fields.length-1
             ResponseWrite "<TH>"+rs.fields(fIndex).name+"</TH>"
        next
        Response.Write "</THEAD><TBODY>"
        rs.MoveFirst
        while not rs.EOF
             Response.Write "<TR>"
             for fIndex=0 to rs.fields.length-1
                  ResponseWrite "<TD>"+rs.fields(fIndex).name _
                     +"</TD>"
             next
             Response.Write "</TR>"
             rs.MoveNext
        wend
        Response.Write "</TBODY></TABLE>"
        Response.Write "</BODY></HTML>"
        rs.Close
End Sub
```

Assuming it is connected to an ODBC-compliant database named Weather, Listing 13.6 will iterate through all the fields of the weather database to produce headers in a table and then output each record as a row in the table. This is almost an exact duplication of what the ASP script did in Listing 13.1, but it has the advantage of providing an early-bound (and hence faster and more optimized) set of connections to both ASP and database objects. This subroutine also works within the IDE to enable such useful features as Intellisense and parameter hinting and provides a much higher level of security than naked ASP files could provide.

If you are unfamiliar with ASP, the action of the **Response** object may be a bit of a mystery. **Response** writes information to the client and can operate in one of two modes. In the default mode, when **Response.Write** is called, that information is automatically sent down to the Web browser. This is the preferred mode for working with browsers that display pages as they receive them, but in certain circumstances (such as with tables), it can cause significant delays while the table itself is built. (Internet Explorer 5 supports incremental table build-ing, although IE4 does not.) In such situations, it’s preferable to buffer the output. You can accomplish this buffering by setting the **Response.Buffer** property to true and then calling either **Response.Flush** to immediately write the current contents of the **Response** buffer (but keep it active) or **Response.End** to flush the contents and terminate the **Response** object.

TIP
Why Should You Use Buffering?
In addition to handling table output, the buffering capabilities of the **Response** object are meant to accelerate the perceived download speed of a Web page. For example, a call to a loaded database takes time, especially if that database is located on a different server. By buffering the initial setup of the page (loading background and support graphics, any preliminary explanatory text, or specialized formatting), the browser appears to load the page quickly. This both gives the user something to engage his attention and gives the asynchronous call to the database time to load. Buffering can also be useful in dealing with errors, as illustrated later in this section.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The **Response** is the primary (although not the only) output device in ASP, and it has a full collection of properties and methods to support its mission (see Table 13.1). Visual Basic 6 supports Internet Information Server 4.0, which implements the expanded ASP 2.0 library.

Table 13.1 Properties and methods for the **Response** ASP object.

Property or Method	Description
AddHeader	Adds an HTTP header to the document before it is transmitted to the client.
AppendToLog	Writes a message to the server log.
BinaryWrite	Writes data directly to the client without converting it into Unicode. This is especially useful for writing out image and sound data directly to the client.
Buffer	A Boolean property that indicates whether the **Response** object outputs text to an intermediate buffer (**Buffer=true**) or not (**Buffer=false**) before writing it out to the client.
CacheControl	A Boolean property that indicates whether the output can be cached to a firewall for retrieval.
CharSet	Returns the character set of the document as indicated by the HTTP header.
Clear	Empties the ASP buffer. This is especially useful for handling error conditions.
ContentType	The content type is a string variable that indicates what format the document is in. It defaults to **text/html** but can be anything from a program reference (**application/msword**) to an image (**image/jpeg**).

Cookies	A collection of cookies that are sent to the client.
End	Flushes the output buffer and closes the **Response** object.
Expires	Sends a message to the client's cache indicating how long the document will remain current in minutes. For example, **Response.Expires=240** will keep the document current in the cache for 240 minutes, or 4 hours.
ExpiresAbsolute	Gives an absolute date indicating to the cache when a page is no longer current. For example, **Response.ExpiresAbsolute = #7/21/2002 00:00:00#** will expire on July 21, 2002, at midnight.
Flush	Transmits the contents of the buffer to the Web page and clears the buffer. The **Response** object remains in memory after a flush.
IsClientConnected	A Boolean flag that checks to see whether the client is still connected to the server. This property is especially useful for error-handling situations, where the client may have disconnected from the server.
Pics	PICS, an acronym for Platform of Internet Content Selection, is used by browsers to determine what content a given page has. It is typically used as a way of filtering out material that is inappropriate for minors. You can set this specialized HTTP header as a string. More information about PICS is available at www.rsac.org.
Redirect	Used for diverting a browser to a new URL. The **Redirect** method must be called before the **<HTML>** tag is sent to the browser.
Status	Sends a status code to the browser. Perhaps the most well known of these is the status code '404' (Document not found). In general, it is usually sent automatically by the server, but you might need to send specialized information that the server doesn't handle. Check out www.w3.org/pub/WWW/protocols/rfc2068/rfc2068.txt for more information about server-side status codes.
Write	Outputs text information to the client, automatically converting it to the specified client set (ISO-LATIN-7, if not otherwise specified).

Many of the **Response** object properties and methods deal in some way with header information, which is information sent to the browser prior to the document itself that instructs the browser about specialized information about the document. Frequently, header information is contained within **<META>** tags, such as the PICS specification for setting ratings information. Headers can get quite complex and, for the most part, fall outside the scope of this

book.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Adding Templates To Web Classes

Although the **Response** capabilities are powerful, the code given previously doesn't offer all that much of an advantage compared to ASP script. The **Response.Write** commands, which write information to an internal buffer that is sent when the subroutine ends, are at best clunky and awkward; at worst, they are interwoven into the code, thus making the HTML generation and maintenance a real nightmare. Ideally, VB should be able to specify some form of template file into which the table can be added at runtime. This is precisely what Visual Basic 6 can do through the use of *templates*.

A template is a modified HTML file that is created ahead of time and imported into Visual Basic. Templates can be either standard code that is included within the DLL for convenience or, more likely, HTML code with special tags that will be replaced by new content. By working with custom tags, the Web Class offers several advantages over traditional ASP: The designer can create the Web page with his own HTML editing tools, adding the specialized tags through the agency of "escape hatches" that most such programs have for working with nonstandard code. Both standard browsers ignore undefined tags, which means that you can use the tags to encompass placeholder text that in turn will get overwritten in the final output with the live data.

The weather browser discussed in the preceding chapter offers a good example for creating templates. For instance, suppose that you need a simple page that displays the weather information for a whole state. Because the weather database has several different states' data within it, the template should be generic enough to handle all possible states and to produce a tabular output as needed. (Getting the requisite state for display will be discussed later in this chapter.) Listing 13.7 shows an example of the template structure.

Listing 13.7 Structure of an HTML template for displaying a weather table.

```
<HTML>
<HEAD></HEAD>
<BODY bgcolor="#C0FFC0" text="black">
<h2>Weather for</h2>
```

```
<h1><WS:STATE>WhichState</WS:STATE></h1>
<WS:WEATHERTABLE>
This will be a table.
</WS:WEATHERTABLE>
</BODY>
</HTML>
```

TIP
Weather Report On CD-ROM
The examples in this chapter assume that you have installed a weather database onto your server and given it a DSN of "Weather". A sample weather database called Weather.mdb is located on the CD-ROM for your use.

To use this template (or any template for that matter) within your Web Class, you take the following steps:

1. Create the template file in any HTML editor and save it to a working directory.

2. Open your IIS application if it is not already open, open the Web Class Editor, and right-click the HTML Template WebItems folder (see Figure 13.5). Select Add HTML Template from the pop-up menu that appears.

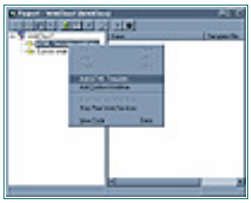

Figure 13.5 Open the Web Class Editor and right-click the HTML Template WebItems folder to add a new template.

3. Browse to the location of the HTML file you created and then open the document. The Web Class automatically creates an internal copy of this document, so you don't need to worry about destroying your original.

4. A new template called Template1 is created within the HTML Template WebItems folder. You can rename the template by pressing F4 to display properties and then changing the Name property. In the example given here, the name was changed to Weather-Summaries.

5. While you have the properties page open, change the **TagPrefix** property to "WS:" from the default value "WC@". The *tag prefix*, which is used by the Web Class to determine what tags it can change, is an example of a *namespace*. Namespaces play an important part in XML documents as well.

6. Within the **General_Declarations** handler, define a variable to hold which state the weather applies to:

```
Public State as String
```

7. Replace the code in the **WebClass1_Start** method with the following:

```
Private Sub WebClass_Start ()
        ' For purposes of illustration, predefine a state.
        ' This will change.
        State="Washington"
        WeatherSummary.WriteTemplate
End Sub
```

8. Select the **WeatherSummary** object and **ProcessTag** method and insert the code in Listing 13.8.

Listing 13.8 ProcessTag event handler for the WeatherReport template.

```
Sub WeatherSummary_ProcessTag(ByVal TagName As String, _
  TagContents As String, SendTags As Boolean)
       dim conn as Connection   ' This declares the connection
       dim rs as RecordSet      ' This declares the record set
       dim fIndex as integer    ' Declaration for a field
                                ' index
       dim buffer as string
Select Case TagName
       Case "WS:STATE"
             TagContents=State
       Case "WS:WEATHERTABLE"
             set conn=Server.Create("ADODB.Connection")
             conn.Open="Weather"
             set rs=conn.Execute("SELECT * FROM Weather
             WHERE State='"+State+"';")
             buffer=""
             buffer=buffer+ "<TABLE>"
             buffer=buffer+ "<THEAD>"
             for fIndex=0 to rs.fields.length-1
                buffer=buffer+"<TH>"+rs.fields(fIndex).name+ _
                 "</TH>"
             next
             buffer=buffer+ "</THEAD><TBODY>"
             rs.MoveFirst
             while not rs.EOF
                    buffer=buffer+ "<TR>"
                    for fIndex=0 to rs.fields.length-1
                          buffer=buffer+ "<TD>"+
                          rs.fields(fIndex).name+"</TD>"
                    next
                    buffer=buffer+ "</TR>"
                    rs.MoveNext
             wend
             buffer=buffer+ "</TBODY></TABLE>"
             rs.Close
             TagContents=buffer
End Select
```

9. Run the Web Class just as you would a regular Visual Basic program, and a weather report for Washington should appear within your browser (see Figure 13.6).

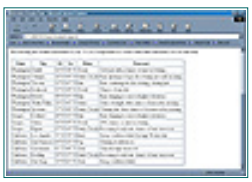

Figure 13.6 The WeatherSummary Web Class produces a table showing the weather for each city in the state.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

If everything goes according to plan, you should see the salient weather information for each city in Washington state (where the dominant weather condition is usually *rainy*).

Within the **Start** event handler, you set the **state** variable to a specific value. You set it here because the **Start** handler is called only once, when a request is made to the server. Once **Start** defines the variable, it forces the template to start processing with the **WeatherTem-plate.WriteTemplate** method, priming it to generate the requested output. Note that in a real application, you would probably set the **state** variable using form data from an HTML form. (I cover this in greater detail when I detail the **Request** object.)

What's My Tagline

The bulk of the work in the Weather Report Web Class is handled within the **ProcessTag** event handler. Each template has its own **ProcessTag** handler, making it possible to customize the HTML output to fit the situation. This routine relies heavily upon the tag prefix. This string of characters, starting a tag, signals to the Web Class that the contents of the tag should be processed. The default tag prefix is "WC@" (WC presumably for Web Class); the @ symbol simply serves as a distinctive visual separator. You can replace this tag with any of your own (such as the "WS:" tag covered in Step 5 earlier), either by setting this property at design time (using the **TagPrefix** property for the template in the property list) or by setting it at runtime using **WeatherReport.TagPrefix="WS:"**.

Why would you want to set it at runtime? One possible use is setting up two sets of tags, one for Microsoft output ("MS:") and the other for Netscape output ("NS:"). With the tag prefix set to "MS:", only Microsoft code gets output to the page; the Netscape tags are untouched. Setting the tag prefix to "NS:" will output only Netscape Navigator code.

The **ProcessTag** handler is called by the Web Class every time its parser finds a tag

that starts with the prefix. In the weather example, for instance, the Web Class will file the template's **Process** tag twice, when it encounters <**WS:STATE**> and <**WS:WEATHERTABLE**> (see Listing 13.9). When it does so, the handler gets passed the tag's name (**WS:STATE**), its contents (**"WhichState"**), and a Boolean value called **SendTags**, discussed shortly.

Listing 13.9 Weather Report template with substitution tags highlighted.

```
<HTML>
<HEAD></HEAD>
<BODY bgcolor="#C0FFC0" text="black">
<h2>Weather for</h2>
<h1>
<WS:STATE>WhichState</WS:STATE>
</h1>
<WS:WEATHERTABLE>
This will be a table.
</WS:WEATHERTABLE>
</BODY>
</HTML>
```

The declaration for the **ProcessTag** handler definitely bears examination:

```
Sub WeatherSummary_ProcessTag(ByVal TagName As String, _
  TagContents As String, SendTags As Boolean)
```

Notice that only the **TagName** is passed by value. **TagContents** and **SendTags** are implicitly passed by reference. **TagContents** holds the text of everything within the specified tag. If its value is changed, then when the routine ends, the new value is used to substitute the old value when the page is sent to the browser. In the example in the preceding section, the **TagContents** value (which was initially "This will be a table.") gets replaced with the actual table generated from the Weather Report database.

The **SendTags** parameter indicates whether the enclosing custom tags should be included in the output. It usually defaults to false; for a custom tag, you probably don't want the actual tags appearing in the final Web page. However, in certain circumstances, it can prove useful to include the tags. Consider that the prefix doesn't necessarily have to be a specialized set of characters but could actually correspond to legitimate HTML tags (such as XMP, which formats output with line spaces and specialized code intact—usually used to illustrate sample code fragments, hence the name). If you set the tag prefix to "XMP", the parser will process every <**XMP**> tag it finds; if you set **SendTags** to true within the handler, the <**XMP**> tags get preserved as well, which means that you wouldn't need to explicitly wrap them around the outputted text.

The text that replaces the placeholder data can be anything—straight unformatted text, HTML code, or even additional custom tags. This last point offers some interesting possibilities. For example, suppose a weather report accessed reports for each city in the state. However, some states may also have customized information peculiar to the area. (Tidal charts make sense for Seattle but make no sense for Spokane, a city located more or less in desert.) Rather than create a generic handler

for all these possible additional pieces, you might use a **<WS:SPECIALCONDITIONS>** tag to pull information from the database that includes the tags **<WS:TIDEHI>** and **<WS:TIDELO>**.

Normally, the Web Classes handle only one level of parsing. Even if the **<WS:TIDEHI>** and **<WS:TIDELO>** tags were included, they wouldn't be processed because the tag containing them would have already been handled and the parser head would move to the next text it found in the original document. You can, however, tell the Web Class to rescan the document once it has been parsed originally by using a Boolean property of the template called **ReScanReplacements**. When this property is set to true, the **ProcessTag** function calls itself recursively, replaces any new valid tags with their appropriate values, and then repeats the process until no new tags can be found. You need to set this property prior to processing the templates.

TIP
A Word Of Warning
Unless you specifically plan to embed custom tags in your replacement tag, leave **ReScan-Replacements** set to false. Setting it to true unnecessarily can have a significant adverse effect upon performance. The default value is false.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

A Humble Request

So far, the Weather Report Web application produces generalized weather output for one state—Washington. For those people who live in the other 49 states, such an application is pretty useless and likely to doom marketing efforts if you need a separate browser app for each state. Obviously, the key to making this more than a strange curiosity is to provide some way for the user to select which state he or she wants to view. In other words, for Web applications to be useful, you need some way of getting information *to* them as well as *from* them.

Fortunately, conveying such information has not been a problem for Web applications for some time. The HTTP protocol defines two different mechanisms for sending information from the client to the server: the **GET** protocol and the **POST** protocol. **GET**, the older of the two, works essentially by sending a string of name-value pairs to the server attached to the URL of the processing CGI program. For example, to send a request to display the statewide weather information for California, you could place the following **GET** request into an anchor link tag:

```
<A HREF="http://www.WeBWeather.com/WeatherReport.asp?state=
  California">California Weather</A>
```

The expression after the question mark constitutes the *query* and is known as a *query string*. Each tag pair within the query is separated by an ampersand (&), and spaces are replaced with plus signs (+). Thus, to retrieve the weather for Walla Walla, Washington, you'd use the following:

```
<A HREF="http://www.WeBWeather.com/WeatherReport.asp?state=
Washington&city=Walla+Walla">Walla Walla Washington Weather
</A>
```

Although **GET** requests are relatively useful for passing small amounts of data, they have some notable problems:

- Query strings have a limit of 1,024 characters, making them problematic for passing large amounts of text.
- Query strings need to be encoded so that the information they pass doesn't contain problematic characters (such as spaces, tabs, quotes, pluses, ampersands, or similar characters).
- Query strings appear as part of the URL when sent, making them less than desirable for sending potentially sensitive information (such as passwords or credit card numbers).

In response to this, the W3C established a new protocol, **POST**, with HTTP 1.0 in 1994. **POST** works by defining within the Web page a **Form** object that contains recognized **<INPUT>** elements. Each **<INPUT>** element, including listboxes, textboxes and textarea boxes, buttons, and checkboxes, has a unique identifier (a name or ID property) and a value property, which can be used to transmit information.

The **POST** protocol solves many of the problems inherent in the **GET** protocol. You can use it to pass large blocks of text, it performs encoding in the background, and it doesn't appear on the command line when sent. For most serious applications, **POST** is more widely used than **GET**, although older servers don't always uniformly support it. IIS applications support both **POST** and **GET**, although Microsoft's documentation stresses that **POST** is the preferred method for sending information.

Of course, sending the information to the server is relatively easy. It's getting the information on the server side that's the tricky part. Fortunately, both ASP and IIS support a number of objects that are useful on the receiving end of things; the most important of these is the **Request** object. **Request** is to server-side input what **Response** is to server-side output. It accesses the data sent to it from the client and puts it into a number of different formats, depending upon the means of transmission and the information involved. The properties and methods of the **Request** object are summarized in Table 13.2.

Table 13.2 Properties and methods of the **Request** object.

Property or Method	Description
BinaryRead	Accesses binary information sent from the client, such as a picture.
ClientCertificate	A collection of client certificate fields, as specified in the X.509 standard.
Cookies	A collection of cookies sent as part of the request.
Form	A collection of form variables sent via **POST**.
Item	The default procedure for the **Request** object, **Item** returns a keyed or indexed item that was sent to it.
QueryString	A collection of query string objects sent as part of a **GET** protocol link.
ServerVariables	A collection of server variables, accessible by name or index.
TotalBytes	The number of bytes sent to the server, usually referenced in order to retrieve data with the **BinaryRead** method.

In most cases, the objects returned by the **Request** object are not traditional Visual Basic collections; rather, they implement **IRequestDictionary**, a specialized form of one of the most useful structures in ASP programming—the Dictionary.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

I Request A Dictionary

Dictionaries are a staple of the ASP developer, and their recent inclusion within Visual Basic 6 attests to their popularity. Dictionaries act much like lightweight collections; passing a key to a **Dictionary** object returns a value. However, that value can be anything: a number, a string, an object, or even another dictionary. This makes them ideal for creating quick, complex data structures as varied as linked lists, binary trees, or arrays. Moreover, they are considerably less sensitive to invalid keys than collections are, which is essential to using them in languages such as JavaScript, which have no error-handling capabilities (see Table 13.3).

Table 13.3 Dictionary properties and methods.

Property or Method	Description
Add	Adds a new item to the Dictionary and associates it with a key.
CompareMode	Sets or retrieves the comparison mode for key references. Can be either **BinaryCompare** (the default), **TextCompare**, or **DatabaseCompare**.
Count	Returns the number of items in a Dictionary. Property is read-only.
Exists	Used to determine whether a given key exists in the Dictionary.
Item	Default property of the Dictionary, **Item** takes a key and returns the associated item.
Items	Used for iterating through a Dictionary, **Items** takes an integer and returns the item corresponding with that integer.
Key	Actually a method of the **Item** property, **Key** returns the key for a given item.
Keys	The **Keys** collection is used for iterating through the keys of a Dictionary. It takes an index (1 based) and returns the key corresponding to that index.
Remove	Removes the item for the key passed to it.

RemoveAll	Removes all the items in a Dictionary.

The base Dictionary class built into ASP (and hence available to IIS applications in Visual Basic 6) works by creating an internal hash table. Hashing is a fairly common technique in programming. In essence, when a key is hashed, a subroutine converts the key into a numeric value; a simple example of this might be a routine that adds all the ASCII values of each character in the key to create an identifier. Good hash routines usually are able to make the identifier unique, although the **Dictionary** object has ways of resolving hash collisions (where two different keys hash to the same value). The advantage of hashing is that it is fast. For small arrays, for example, the time for finding an item in an array is comparable to that for finding it in a hash table, but for anything beyond a couple of dozen items, a hash table can often be hundreds of times faster.

Accessing a Dictionary is quite straightforward. The **Add** method (obviously) is used to add an item to the Dictionary. You can also add a key to the Dictionary for accessing that item, although it's not strictly necessary. The key, like the item, can be any data type, although typically, both are strings. For example, the following code illustrates creating a **SeattleReport** Dictionary:

```
Dim SeattleReport as Dictionary
Dim WashingtonReport as Dictionary
Set SeattleReport = new Dictionary
SeattleReport.Add "skies","cloudy"
SeattleReport.Add "hiF",65
SeattleReport.Add "loF",41
SeattleReport.Add "forecast","Cloudy with rain changing _
  to showers."
WashingtonReport.Add "Seattle",SeattleReport
```

Accessing this information is just as straightforward. To get the forecast for Seattle from the Washington Report, you'd write the following:

```
Dim Forecast as string
Dim SeattleReport as Dictionary
Set SeattleReport=WashingtonReport.item("Seattle")
Forecast=SeattleReport.item("Seattle")
' **** Or
Set SeattleReport=WashingtonReport("Seattle")
Forecast=SeattleReport("forecast")
' **** Or even
Set SeattleReport=WashingtonReport("Seattle")
Forecast=SeattleReport.Items(4)
' **** Because the forecast was the fourth item to be added
'      to the report.
```

You can iterate through the elements of a Dictionary by using the **Count**, **Items**, and **Keys** collections. One useful technique where Dictionaries really shine is as a way to store fairly complex data, such as records in a database where the initial structure of the records is unknown. They also come in handy in output, as the code **GetStringFromDictionary** illustrates. The code in Listing 13.10 will convert a dictionary into a user-delimited string, contained by a second delimiting set of characters.

Listing 13.10 GetStringFromDictionary, a function that takes a Dictionary object and

returns it as a formatted string.

```
Function GetStringFromDictionary(Dict as Dictionary,itemDelim _
  as String=",",keyDelim as String=":",startDelim as _
  string="[",endDelim as String="]") as String
       Dim Key as String
       Dim Value as String
       Dim Buffer as String
       Buffer="["
       For index=0 to dict.Count-1
             Value=Dict.Items[index]
             Key=Dict.Keys[index]
             Buffer=Buffer+Key+KeyDelim
             If IsObject(Value) Then
                   If TypeName(Value)="Dictionary" Then
                         Buffer=Buffer+
                         GetStringFromDictionary(Value)
                   Else
                         Buffer=Buffer+TypeName(Value)
                   End If
             Else
                   Buffer=buffer+cstr(Value)
             End If
             If index<Dict.Count-1 then
                   Buffer=Buffer+itemDelim
             End If
       Next
       GetStringFromDictionary=Buffer
End Function
```

NOTE
The first element of a Dictionary is item 0, not item 1 as it is with some VB collections.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

To illustrate the **GetStringFromDictionary** function, consider a routine that reads the contents of a Dictionary and formats the result into a URL:

```
<A HREF=http://www.WeBWeather.com/WeatherReport.asp?city=
Olympia&hiF=64&loF=47&skies=Cloudy&precipitation=2.25>
```

The **GetStringFromDictionary** function could be called to format the query reference, prior to the HTML page being output:

```
Dim QueryOutput as String
Dim WeatherData as Dictionary
Dim URL as String
Set WeatherData=new Dictionary
WeatherData.add "city","Olympia"
WeatherData.add "hiF",64
WeatherData.add "loF",47
WeatherData.add "skies","Cloudy"
WeatherData.add "precipitation",2.25
QueryOutput=GetStringFromDictionary(WeatherData,",","&","=","","")
URL="http://www.WeBWeather.com/
WeatherReport.asp?"+QueryOutput
```

This is an admittedly simplistic example of how you can use dictionaries, but it does serve to show that dictionaries can appear in nearly all aspects of ASP development. Other places where dictionaries are used include creating quick, easily resizable arrays, linked lists, and binary trees, storing information transparently and otherwise providing a semblance of structure to a collection of items.

The ASP/IISAPI objects that Web Classes expose include a **Request** object in addition to several others. The **Request** object in turn holds a number of specialized Dictionary objects that implement the **IRequestDictionary** interface. **IRequestDictionary** implements most of the same behavior as the default scripting Dictionary class but populates the dictionaries from the client. For this reason, an **IRequestDictionary** doesn't explicitly include either the **Add** or **Replace** methods (or their variants).

The Daily Post

As mentioned previously, one of the biggest problems with the **GET** protocol is that it doesn't work terribly well in those places where client-to-server communication is needed the most:

- Transmitting large blocks of text (as in a multiline text field)
- Sending state information (such as the state of a checkbox)
- Getting highly formatted code upline (a block of HTML text)

As a consequence, around 1994 the **GET** protocol was gradually replaced by the **POST** protocol, a part of the HTTP 1.0 specification. This particular protocol made use of the **<FORM>** tag, which acted as a generalized container for any type of generalized input, such as a textbox, a selection of radio buttons, a combo box or listbox, and so forth. The **POST** protocol, which offered protection from prying eyes, was actually the first step necessary in creating secure channels across the Internet.

The VB6 Web Classes specifically handle all transactions through the **POST** protocol, even when the **GET** protocol is explicitly set up in the final Web page output. Because this means that data sent via the query string method appended to a URL actually must be converted into HTTP 1.0 format, the Visual Basic documentation discourages the use of the query string and **GET** protocol whenever possible.

The **<FORM>** tag can actually use either protocol, by the way, although the default behavior for a form is to use **POST**. Although this discussion is a bit of a diversion, it's worth looking more closely at the attributes in a **<FORM>** tag because certain default parameters aren't always covered within books on standard HTML:

```
<FORM
ACTION="http://www.WeBWeather.com/WeatherReport.asp"
METHOD="POST"
TARGET="self"
ENCTYPE="multipart/form-data">
<!-- Input controls-->
</FORM>
```

The **ACTION** attribute contains the URL to send the data; typically, it is either a CGI script or an ASP. This example uses an ASP that contains the WeatherReport Web Class. The **METHOD** can be either **POST** or **GET**, although it defaults to **POST**. The **TARGET** is the frame that receives the results from the CGI script (and usually defaults to **"self"**, the frame in which the call was made). Finally, the **ENCTYPE** attribute determines how data is encoded for transmission and can take on either the value **"multipart/form-data"** (the default) or the rather unwieldy **"application/x-www-form-urlencoded"**. The latter is used to send binary data, such as a picture, to the server from the client.

A form is simply a container, much like a **<DIV>** or even a **<P>** tag. You can (in theory) place any HTML within a form that you can place outside of it, which means that you can format form input controls within tables or **<DIV>** tags. The important elements from a form's standpoint, however, are the **<INPUT>** elements—textboxes, listboxes, control boxes, buttons, and related "controls." An **<INPUT>** tag has at least three required attributes (**NAME**, **TYPE**, and **VALUE**), although if you are working with Internet Explorer 4 or later, you may also want to include an **ID** attribute:

```
<INPUT TYPE="Text" NAME="City" ID="City" VALUE="Clear">
```

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The **TYPE** attribute indicates what type of **<INPUT>** container is being used. You would use both a **NAME** and **ID** attribute because Internet Explorer and Netscape, as is typical, use different standards for data input; IE refers to everything by its **ID**, whereas Netscape uses the **NAME** attribute to describe its named elements. This is one of the biggest stumbling blocks that most people have when encoding forms for multiple platforms. In general, it is wise to use both and provide the same name for **NAME** and **ID**. The **VALUE** attribute has more meaning with some types of **<INPUT>** than others, but in most cases, **VALUE** contains the text string value that the control will provide when the form is transmitted.

NOTE
Note that an **<INPUT>** tag does not have to be inside a **<FORM>** tag. Especially when used in conjunction with DHTML, **<INPUT>** tags provide ways for client-side JavaScript to retrieve or display values. However, an **<INPUT>** tag outside a form is never transmitted to the server; in that respect, the **<FORM>** object acts as a container.

The HTML form in Listing 13.11 demonstrates how you can put these elements together to transmit information about current local conditions, as you might find in a Web-based weather site.

Listing 13.11 HTML code for retrieving current weather conditions into a form.

```
<HTML>
<HEAD>
<META NAME="GENERATOR" Content="Microsoft Visual Studio 6.0">
<TITLE></TITLE>
</HEAD>
<BODY>

<H1>WeBWeather</H1>
<H2>Current Weather Conditions For Washington</H2>
```

```
<FORM ID=weatherSubmitForm NAME=weatherSubmitForm
METHOD="POST"
 ACTION="WeatherReport.asp">

<INPUT TYPE=HIDDEN ID=state NAME=state VALUE="Washington">
Choose a city:
<SELECT ID=city NAME=city>
       <OPTION SELECTED VALUE=seattle>Seattle</OPTION>
       <OPTION VALUE=olympia>Olympia</OPTION>
       <OPTION VALUE=redmond>Redmond</OPTION>
       <OPTION VALUE=walla_walla>Walla Walla</OPTION>
       <OPTION VALUE=spokane>Spokane</OPTION>
</SELECT><BR>
High Temperature
(&deg;F):<INPUT ID=HiF NAME=HiF VALUE="" ><BR>
Low Temperature (&deg;F):<INPUT id=LoF name=LoF value="" >
<BR>
Current Conditions:<SELECT ID=skies NAME=skies>
       <OPTION SELECTED VALUE=clear>Clear</OPTION>
       <OPTION VALUE=partlySunny>Partly Sunny</OPTION>
       <OPTION VALUE=partlyCloudy>Partly Cloudy</OPTION>
       <OPTION VALUE=cloudy>Cloudy</OPTION>
       <OPTION VALUE=rainy>Rainy</OPTION>
       <OPTION VALUE=thunderstorm>Thunderstorm</OPTION>
       <OPTION VALUE=snow>Snow</OPTION>
       <OPTION VALUE=sleet>Sleet</OPTION>
</SELECT><BR>
24-hour Forecast:<BR>
<TEXTAREA ID=Forecast NAME=Forecast STYLE="HEIGHT: 88px;
  WIDTH: 276px"></TEXTAREA><BR>
<INPUT type=submit><INPUT TYPE=reset>
</FORM>
</BODY>
</HTML>
```

NOTE
Note the use of the **HIDDEN** element in the form to record the information about which state the city belongs to. A hidden element is a convenient way to add information to the page that's necessary to trans-mit but that shouldn't be seen by the person viewing your Web page.

The HTML code here obviously only works for the state of Washington, although later in this chapter, I introduce support tags to turn it into a more generic template. No real effort has been made to "prettify" this input page; its primary purpose is to demonstrate how you set up a form and how the Web Class accesses that form.

The Submit button at the bottom of the form sends a message to the browser to read all the values in the button's form, encode them into a safe format (which is where the **"multipart/form-data"** value for the encode tag comes in), and then send them to the URL specified in the **ACTION** element.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Enabling Events In Web Classes

Sandy, a meteorologist in Washington, selects a city, adds the requisite data, and then presses the Submit button. What happens next? At the moment, nothing. The application needs to process the information, and to do that, it must have some event signal the Web Class that data has been submitted. This is the role of Web Class events.

To work with the data submission page, you need to add the template in Listing 13.11 to your Web Class and create event hooks for handling the form, as follows:

1. Open the Web Class Editor, and right-click HTML Template WebItems. Choose Add HTML Template from the menu, and select the template file created in Listing 13.11 (which is called Weather1Template1.htm on the CD-ROM).

2. Change the name of the template to WeatherSubmit by clicking once on the Template1 label until it becomes editable and then changing the name (see Figure 13.7).

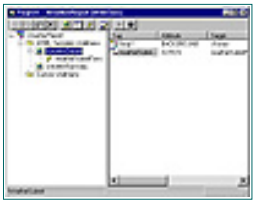

Figure 13.7 The Web Class Editor, disp-laying the new template and its associated form object.

3. In the DHTML outline (the right pane of the Web Class Editor), you can see not just the body object, but also a form object called **weatherSubmitForm**. The Web Class Editor automatically assigns the ID of the form as its name.

4. Double-click the **weatherSubmitForm** object in the right pane. This launches the Code Editor for the **WeatherSubmit_ weatherSubmitForm** event. This event gets called whenever someone presses the Submit button within the form. For now, you can get an echo of the data that was sent to the Web Class by putting the code in Listing 13.12 into the event handler.

Listing 13.12 The event-handling code invoked whenever the form is submitted.

```
Private Sub WeatherSubmit_weatherSubmitForm()
    Dim index As Integer
    For index = 1 To Request.Form.Count
        Response.Write Request.Form.Key(index) + ":" +
        Request.Form.Item(index) + "<br>"
    Next
End Sub
```

5. When you close the code window, you should note that several changes have taken place, including the new **weatherSubmitForm** event handler associated with the WeatherSubmit template and the replacement of the "<none>" target for the **weatherSubmit-Form** with "weatherSubmitForm".

6. Open the **WeatherReport_Start** event handler, and replace the code listed here:

```
Private Sub WeatherReport_Start()
      WeatherSubmit.WriteTemplate
End Sub
```

7. Run the Web Class. Set a high and low value to the temperatures, choose a city and a current weather condition, and provide a forecast (such as "Cloudy giving way to meteor impacts in the afternoon, resulting in snow and nuclear winter conditions"). Then, press Submit. You should get a listing of all the attributes that you set, looking something like Figure 13.8.

Figure 13.8 Raw output of the Web Class, showing the attributes passed from the client.

The process of creating a handler for the Submit button touches on one of the more sublime yet powerful aspects of Web Classes: the Web event. In essence, you can use the Web event for everything from refreshing the page to validating input data and updating databases. It extends the notion of a Visual Basic event to encompass the browser and server together as a single application. Web events are covered in more detail later in this chapter in the section "Linking Events."

Form-Fitting Code

When Sandy presses the Submit button in the WeatherReport, Visual Basic fires the event associated with the **weatherSubmitForm**. This event, shown earlier in Listing 13.12, then iterates through the list of all items sent from the form and outputs their keys and values back to the Web page.

The **Form** collection holds the contents of all the data that was sent via the **POST** statement. In this simple example, nothing actually happens with the data; it is basically output in a simple HTML page. (The <**HTML**>, <**HEAD**>, and <**BODY**> tags are added by default.)

In most cases, you'll want to do something more useful with the data. One of the primary purposes of the event handler is to provide a business rule to validate the data and a mechanism to handle invalid data. With the Weather form, the data could be invalid in several ways:

- Sandy could have failed to put values into the two temperature boxes.
- Sandy could have inadvertently switched the high and low temperatures.
- Non-numeric information could have been added to the temperature boxes.
- The values might fall outside the range of valid data. (A slip might have turned 87 degrees into 877 degrees, a sweltering day indeed.)
- The forecast could have been forgotten.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Even if the data is valid, Sandy should have the chance to do an eyeball check to verify that the data is correct before committing the weather report to a database. Should the data be invalid, the program shouldn't give her the option to submit the data at all, only to correct it.

The code in Listing 13.13 shows some (although hardly all) of the possible validations, as well as an example of how to use the **Form** collection. I've also used a simple (albeit devious) trick to keep the amount of retyping to a minimum: If Sandy wants to correct her data, a small JavaScript handler tells the browser to go to the previously viewed page. In both Netscape and Internet Explorer, the browser retains form information in the history. This means that you don't need to actually maintain any state information on the server.

Listing 13.13 Validation event handler for the WeatherReport submission.

```
Private Sub WeatherSubmit_weatherSubmitForm()
       Dim HiF as integer
       Dim LoF as integer
       Dim Index As Integer
       Dim EnableCommit as boolean ' If true, lets the user
                                   ' commit the data to the
                                   ' database; if false, the
                                   ' user can't commit data.
       Dim HasWarnings as boolean

       EnableCommit=True;
       HasWarnings=False;
       With Response
             .Write "<HTML><HEAD>"
             .Write "<STYLE>"
             .Write ".ErrorStyle {color:red}"
             .Write "</STYLE>"
             .Write "<BODY>"
             .Write "<h2>Your weather report information:</h2>
            For index = 1 To Request.Form.Count
                   .Write Request.Form.Key(index) + ":" +
                   Request.Form.Item(index) + "<br> <br>"
```

```
	Next
	If isNumeric(Request.Form("HiF")) then
		HiF=CInt(Request.Form("HiF"))
		If HiF>130 or HiF<-40 then
			.Write "<div class=ErrorStyle>
			Warning: The high temperature (" +
			cstr(HiF)+" &deg;F)may be in
			error.</div>"
			HasWarnings=True
		end if
	else
		if Request.Form("HiF")="" then
			.Write "<div class=ErrorStyle>  Error:
			No high temperature was given.
		       </div>"
		else
			.Write "<div class=ErrorStyle> Error:
			The high temperature is not a valid
			number.</div>"
		end if
		EnableCommit=False;
	End if
	If isNumeric(Request.Form("LoF")) then
		LoF=CInt(Request.Form("LoF"))
		If LoF>130 or LoF<-40 then
			.Write "<div class=ErrorStyle>
			Warning: The low temperature (" +
			cstr(LoF)+" &deg;F) may be in
			error.</div>"
			HasWarnings=True
		end if
	else
		if Request.Form("LiF")="" then
			.Write "<div class=ErrorStyle> Error:
			No low temperature was given.
		  </div>"
		else
			.Write "<div class=ErrorStyle> Error:
			The low temperature is not a valid
			number.</div>"
		end if
		EnableCommit=False;
	End if
End if
If enablecommit then
	If LoF>HiF then
		.Write "<div class=ErrorStyle>Error:
		The high temperature (" + cint(HiF)+"
		&deg;F) is less than the low temperature
		(" + cint(LoF)+" &deg;F).</div>"
		EnableCommit=False;
	End if
End if
```

```
        If Request.Form("Forecast")="" then
              .Write "<div class=ErrorStyle>Warning: No
              forecast was given.</div>"
              HasWarnings=True
        end if
        if EnableCommit then
              if HasWarnings then
                     .Write "<br> <br"
                     .Write "<div class=ErrorStyle>The form
                     contains one or more warnings, indicating
                     that data may possibly be erroneous. You
                     can commit this report, although you should
                     check the information carefully before
                     doing so.</div>"
              end if
              .Write "<div>"
              .Write "<a href='javascript:history.back()'
              >Edit Data </a>"
              Set Session("ReportData")=Request.Form
              .Write "<a href=WeatherReport.ASP?WCI=
              ReportCommit&amp;WCU>Commit Record</a>"
              .Write "</div>
              .Write "</body></html>"
        else
              .Write "<div>"
              .Write "<a href='javascript:history.back()'>
              Edit Data </a>"
              .Write "</div>
        end if
        end with
End Sub
```

Most of the code in Listing 13.13 is straightforward. You can use the **Request.Form** object much like the ADO **RecordSet** object; passing the name of the property (such as “HiF”) to the form object will give you the value defined in the form. The handler outputs the data from the form and then filters it through a series of validation tests. If a questionable value comes up that may still be legitimate (such as -43 in Barrow, Alaska), the handler issues a warning but doesn’t block the ability to commit the data. On the other hand, if the data is obviously bad (such as 32F) or missing, then only the edit option is available; Sandy cannot commit the report to the server.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

If the data is acceptable, then a quandary pops up. The logical next step is to load a template and submit the data to the database. However, form data retains its value only from the form page to the subsequent page. There has to be a way to save that information to the actual data-storing page. Fortunately, such a mechanism exists: the **Session** object.

Session information causes data to persist from one page to the next and more or less corresponds to a public type declaration in Visual Basic. The information contained within session variables remains available as long as the user is dealing with the same Web Class. If the user browses to a page on a different Web site and then returns to the first page, session information will be lost.

In the preceding example, the report form's data was saved to a session variable called **ReportData**:

```
Set Session("ReportData")=Request.Form
```

Because **Request.Form** is an object, you should use the **Set** keyword to assign it. (Because the **Session** object is itself a Dictionary, this is not strictly necessary, but it is good form.) You can then retrieve the object in the data commit page by requesting the **ReportData** object:

```
Dim formData as Dictionary
Set formData=Session("ReportData")
```

To actually commit the data, you need to define a Web item that acts as a confirmation page—in this case, a template called **ReportCommit**. The code that calls the **ReportCommit** Web item

```
.Write "<a href=WeatherReport.ASP?WCI=
ReportCommit&amp;WCU>Commit Record</a>"
```

shows the actual Web event connection format; if this page had been a template, you could have assigned the connection by right-clicking the appropriate link tag and selecting the **ReportCommit** template.

Commiting The Data

The real power of ASP, and hence Web Classes, lies in the ability to communicate with databases through the use of ADO. Programming script-based ASP through a program such as Visual InterDev can give beginning programmers the impression that the Active Server Pages technology actually includes ADO as part of its core set and that ADO and ASP are inextricably intertwined. With Visual Basic, the distinction becomes obvious…and meaningful. You can create libraries of data access routines to hide the intricacies of the actual database connections, making your code easier to read and maintain.

NOTE
The database model chosen here, by the way, is extremely simple—one table with no foreign keys and no normalization—primarily because it makes it easier to focus on the Web technologies. A real-world application would of course normalize most of the data and would be considerably more extensive in terms of what specifically is saved.

The **ReportCommit** part of the Web Class illustrates how you can simplify your code. Here, the Web Class updates the city's record with the newest weather and then returns a confirmation page that lets users modify different city records or change to a different state.

The **ReportCommit** template is likewise very simple and uses the ReportCommitTemplate.htm file as its source. Note the use of placeholders in the file, which is shown in Listing 13.14.

Listing 13.14 ReportCommitTemplate.htm source file.

```
<HTML><HEAD><TITLE>Weather Report Updated</TITLE></HEAD>
<BODY>
<H2>Weather Report Data has been updated.</H2>
Do you wish to:
<DIV><A HREF="WeatherSubmitPlaceholder">Update Another
  City</A></DIV>
<DIV><A HREF="ChooseStatePlaceholder">Select a different
  State</A></DIV>
<DIV><A HREF="MainPagePlaceholder">Return to the Main
  Page</A></DIV>
</BODY></HTML>
```

Creating the new template item is nearly as simple as creating the original WeatherSubmit template item:

1. Open the Web Class Editor, and right-click HTML Template Web Items. Choose Add HTML Template WebItem from the menu.

2. Change the name of the Web item to **ReportCommit** and then double-click the node to open the **ReportCommit_Respond** event handler. This gets called whenever the **ReportCommit** item is requested from the client.

3. Type the code from Listing 13.15 into the **Respond** handler. By encapsulating data calls, the code can be made much cleaner and easier to maintain.

Listing 13.15 Response code for the **ReportCommit** Web item.

```
Private Sub ReportCommit_Respond()
       Dim ReportData as Dictionary
       Set ReportData=Session("ReportData")
       WeatherUpdate ReportData
       ReportCommit.WriteTemplate
End Sub
```

4. The **WeatherUpdate** subroutine is used to actually update the weather database. To define it, open up the General Declarations code section, and add the code from Listing 13.16. Because of the nature of the application, you should close the connection and recordsets when they are not needed.

Listing 13.16 The **WeatherUpdate** subroutine updates the weather database.

```
Private Sub WeatherUpdate(WeatherRecord As Dictionary)
    Dim conn As Connection  ' This declares the connection
    Dim rs As Recordset     ' This declares the record set
    Set conn = Server.Create("ADODB.Connection")
    conn.Open = "WeatherReports"
    Set rs = conn.Execute("SELECT * FROM ReportData WHERE
    State='" + WeatherRecord("State") + "' and City='" +
    WeatherRecord("City") + "';")
    rs!HiF = WeatherRecord("HiF")
    rs!LOF = WeatherRecord("LoF")
    rs!Skies = WeatherRecord("Skies")
    rs!Forecast = WeatherRecord("Forecast")
    rs.Update
    rs.Close
    conn.Close
End Sub
```

5. Open the Web Class Editor and click the ReportCommit template. In the right pane, hyperlink references appear for each **<A>** tag within the template. Because none of the links within the template actually have an **ID**, Visual Basic assigns them the names Hyperlink1, Hyperlink2, and so forth, in the order that they appear in the Web page. Right-click Hyperlink1 and select Connect To Web Item from the pop-up menu.

6. In the dialog box that appears, choose the WeatherSubmit template and click OK. This sends a message to the server to execute the template's **Respond** event (as well as the **ProcessTag**, even if you've included custom tags in your HTML template).

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The **Respond** handler of any Web item gets called automatically when the Web item is requested from the client. This is probably the best place to process most of the code in your Web application, using such events as the **Start** method of the Web Class object only for initialization and beginning traffic control (that is, where the user will start; see the next section for more details on this):

```
Private Sub ReportCommit_Respond()
        Dim ReportData as Dictionary
        Set ReportData=Session("ReportData")
        WeatherUpdate ReportData
        ReportCommit.WriteTemplate
End Sub
```

NOTE
Note that both the record set and connection would be closed anyway because the **Connection** object is declared within the scope of the handler; however, it's good practice to make these default actions explicit, for ease of code maintenance if nothing else.

Notice that it's the **ReportCommit_Respond** handler is pretty threadbare. The first step that the handler needs to do is pass the form's data on to the processing routine (**WeatherUpdate**). It does this by retrieving the session variable "ReportData" and coercing it into a Dictionary form. There's a trick here worth noting: **IRequestDictionary** implements most of the same interface (and all of the same data interface) as a standard Dictionary. Because it's not possible to explicitly declare an object as **IRequestDictionary**, you need to coerce it into the other form to do anything with it.

The **WeatherUpdate** routine is an extremely simple ADO script: An ADODB connection object gets created and connected to the WeatherReport's DSN. Once a handle exists, the connection executes an SQL script to retrieve the record associated with the current state and city. The individual fields that need to change

in the record are modified and then the full record is updated to commit these changes to the database. Because there isn't a significant need to keep the connection open, the recordset is destroyed and the connection released prior to leaving the handler.

Once the data is updated, the ReportCommit template is sent back to the client, reporting that the database received and processed the information. The **WriteTemplate** method in turn calls the template's **Respond** event, as well as **ProcessTags** events if the template contains custom tags.

Implementing Push With WriteTemplate

Although push technology as a marketing tool has lost a certain amount of its initial hype, there is something to be said about periodically updating content in a browser. This is especially important when supporting older browsers because the mechanism that they have in place for refreshing pages frequently consists of unreliable meta tags.

The method **WriteTemplate** forces a refresh of the HTML template's contents to the browser. You can use this method in conjunction with a timer on the server to regularly update information on a Web page. For example, a server that tracks such weather variables as barometric pressure (or stock quotes or machine system states) would be able to send this data to older browsers that don't have the support that IE4+ offers for data access.

Linking Events

Although the ReportCommit template is almost embarrassingly simple, the final step in connecting the links to other Web items in the Web Class highlights another useful object in the programmer's toolkit: the custom event.

When you right-click a link within the template document (such as a hyperlink or a form), you're given the option of connecting to a Web item or a custom event. A Web item is a template with its associated code (such as the **Respond** or **ProcessTags** event handlers). A custom event, on the other hand, is simply an event handler that gets called when the client makes a request to the server through that link. In essence, you take over the task of writing the HTML code that gets output, rather than rely upon a pre-existing template. You use the **Response** object to output information, just as you did in the **Start** event for the Web Class earlier in this chapter; indeed, unless your Web Class output is extremely simple (one or two template pages), you are much more likely to use either Web items or custom events to handle your output for most pages.

When you connect either an event or a Web item to a link in the template, Visual Basic automatically replaces the contents of the link with its own specific notation. For example, in the simple weather server being developed here, you will obviously need to change which state or region is under consideration (unless, of course, you happen to live in Washington). Because the application itself needs to be as dynamic as possible, the states should be pulled from the database rather than hard-coded into the page; this guarantees that adding a new state into the database won't require reworking the ASP code.

The ReportCommit template included three lines for navigating from the confirmation page to other places within the Web site:

```
<DIV><A HREF="WeatherSubmitPlaceholder">Update Another
   City</A></DIV>
<DIV><A HREF="ChooseStatePlaceholder">Select a different
   State</A></DIV>
<DIV><A HREF="MainPagePlaceholder">Return to the Main
   Page</A></DIV>
```

NOTE
Sometimes, the interests of pedagogy can put a writer into an awkward situation. The database referenced throughout this chapter is essentially a one-table flat file. In a real working database, the states would likely be referenced in one table, the sky conditions would be in a second, and the remainder of the data would end up in a third, with appropriate cross keys. It would have made retrieving the states list almost trivially easy. However, this more complex situation actually provides a good example to illustrate the use of custom events as well as serves as a warning to think about your database design before you start building the code to reference it.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

In the last section, the first of these hyperlinks was connected to the WeatherSubmit template. The second link won't actually go to a template object. Instead, the link will invoke a new custom event, **onRequestStates**, which will in turn query the database and return a sorted list of all the states that have weather information:

1. Open the WebClass Editor and click the ReportCommit template.
2. In the right pane of the editor, select Hyperlink2 and right-click on it to invoke its menu. Choose Connect To Custom Event to do just that.
3. Note that a new node called Hyperlink2 has appeared as a child to the ReportCommit Web template. This is a custom event associated with the link. You can either choose to retain the name or modify it into something more descriptive. For this example, change the name to **onRequestStates**.
4. Double-click the **onRequestStates** event handler to open its code window, and type the code in Listing 13.17. This will force a list of states to be output to the client.

Listing 13.17 The **OnRequestStates** event handler.

```
Private Sub ReportCommit_OnRequestStates()
       Dim conn As Connection  ' This declares the connection.
       Dim rs As Recordset     ' This declares the recordset.
       Dim States as Dictionary
       Dim State as string

       ' Create the initial header HTML and output it
       ' to the client
       Response.Write "<HTML><HEAD><TITLE>Get New State
       </TITLE></HEAD>"
       Response.Write "<BODY>"
       Response.Write "<H1>States</H1><P>Click on a state to _
                       display all of its cities for _
                       editing.</P>"
       ' Flush the output to ensure that something's seen
       ' on the client
```

```
        Response.Flush
        ' Open the connection
        Set conn = Server.Create("ADODB.Connection")
        conn.Open = "WeatherReports"
        ' Create a new dictionary to hold the list of states
        set States=new Dictionary
        ' Retrieve a list of all states ordered alphabetically
        Set rs = conn.Execute("SELECT State FROM ReportData
        ORDER BY State ASC;")
        Iterate through the list and remove duplicates.
        rs.MoveFirst
        while not rs.EOF
              ' Get the state from the current record
              state=rs!State
              ' If the state is not within the dictionary,
              ' then add it; otherwise, ignore it
              if not States.Exist(State) then
                    States.Add State,State
              End if
              ' Move to the next record
              rs.MoveNext
        wend
        ' Connection to database no longer needed.
        rs.Close
        conn.Close
        ' Output a form with a combo box showing all the states
        ' as well as a submit button.
        Response.Write "<FORM id=stateForm name=stateForm _
                        action='" + URLFor(WeatherSubmit)+"' _
                        method='POST'>"
        Response.Write "Please select a State:"
        Response.Write "<select name=""State"" id=""State"">"
        For each State in States
               ' Retrieve the session level State variable and
               ' see if it belongs to the listed state. Make
               ' that option the selected one if it does.
               If Session("State")=State then
                     Response.Write "<OPTION value='"+State+"' _
                     selected>"+State
               Else
                     Response.Write "<OPTION value='"+ _
                     State+"'>+State
               End if
        Next
        Response.Write "</SELECT>
        Response.Write "<INPUT TYPE='SUBMIT'>"
        Response.Write "</FORM>
        Response.Write "</BODY></HTML>"
End Sub
```

Almost all the contents of Listing 13.17 should be familiar to you if you've worked with ADO, although the operation itself is complex. A request is made to the database to retrieve

the list of all states, ordered by state. If the state is not contained within a storage Dictionary (called, not surprisingly, **States**), then the state is added. Note that the state is added as both a value in the Dictionary and as a key to reference the element. Dictionaries require key-value pairs even if the key and value are the same thing. By adding the state only once into the Dictionary, you end up with a list of all the states in the database, sorted alphabetically.

If it can be helped, most developers probably don't want to have to figure out what the precise URL is for each call to the IIS applications. This is where the **URLFor** function comes in:

```
Response.Write "<FORM id=stateForm name=stateForm _
                action='" + URLFor(WeatherSubmit)+"' _
                method='POST'>"
```

The **URLFor** function translates the Web items or templates into a URL on the client side, making it much easier to add these events to the template code. The function can take one of two forms: **URLFor(*TemplateVariable*)**, which provides the URL address for the referenced template (such as **WeatherSubmit** in the example here), or **URLFor(*WebItem, CustomEvent*)**, which will call the requested custom event belonging to that Web item or template.

URLFor is especially handy because it provides a way to incorporate calls to Web items from JavaScript or VBScript events. For example, you can set up an image to act as a link to a Web Class template without using anchors by tapping into the **onClick** event for the picture:

```
Response.Write "<IMG SRC='http://www.myimage.com/image.jpg'
onclick='document.location="+URLFor
(WeatherSubmit,onRequestStates)+"'>"
```

The **onClick** event gets invoked whenever the image is clicked and in turn requests the **onRequestStates** HTML code. You could similarly incorporate **URLFor** routines within JavaScript or VBScript code, with the caveat that directly modifying client-side code with server-side routines can make the process of debugging something of a nightmare.

The only other programming of note within the **onRequestStates** command comes with the **State** session variable:

```
If Session("State")=State then
```

This particular session variable is not defined within the page; its primary purpose is to save and retrieve the last state so selected, which means that the ASP code to actually set the session variable is included in a revised WeatherSummary page. As little is gained by reprinting it here, the primary WeatherSummary page is provided on the CD-ROM.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Getting Browser Capabilities

As a Web developer, I voice the plaint that is spoken almost ritualistically by those who make a living on the Web:

“Oh, why can’t Microsoft and Netscape agree on standards?”

The ugly truth about Web development is that only the simplest Web pages can safely make the transition from Microsoft’s Internet Explorer 4.0 to Netscape Navigator 4.0. The likelihood that even a straight HTML Web page will work on every browser has become almost vanishingly small.

One of the things that makes VB6 so compelling is that you can use it to customize the output of a Web page so that it will work effectively regardless of which browser the page is delivered to. Templates written specifically for Netscape 4 or Internet Explorer 5 or even Opera 2.0 or Lynx can be created in the appropriate tool and then attached to the Web Class. Web Classes can likewise replace customized tags with browser-specific variants, making it easier to build a cohesive design without necessarily compromising your programming resources.

However, this customization capability is fairly meaningless if the Web Class can’t detect what type of browser it is sending these pages to. Fortunately, when the client establishes a collection to the server, the client sends a fairly broad set of information about itself to the server, where it is in turn stored in a special Dictionary for server objects. This object, **ServerVariables**, has access to information about both the server and the client, although only a few properties have any real utility in most applications. The routine in Listing 13.18, **DisplayServerVariables**, will send a table of the current variables in play to your browser (note that this code isn’t a part of the weather project).

Listing 13.18 DisplayServerVariables subroutine and one way to call it.

```
Public Sub DisplayServerVariables(optional whichVariable _
 as Variant)
    Dim index As Integer
    Response.Write "<TABLE>"
    With Request
       If IsMissing(whichVariable) then
         For index = 1 To .ServerVariables.Count
            Response.Write "<TR><TD>"
            Response.Write Request.ServerVariables.Key(index)
```

```
            Response.Write "</TD><TD>"
            Response.Write Request.ServerVariables.Item(index)
            Response.Write "</TD></TR>"
          Next
        Else
            Response.Write "<TR><TD>"
            Response.Write whichVariable
            Response.Write "</TD><TD>"
            Response.Write Request.ServerVariables(which _ Variable)
            Response.Write "</TD></TR>"
            Response.Write "</TABLE>"
        End if
        Response.Write "</TABLE>"
    End With
End Sub

Public Sub WebClass1_Start()
    DisplayServerVariables
End Sub
```

Of all the server variables, the one that is most immediately useful is **HTTP_USER_AGENT**. You can use this variable, which contains identification information about the browser, to determine what type of HTML code is sent to the client. The results, however, can be a little cryptic, as Figure 13.9 illustrates quite clearly. Internet Explorer 5 is described as *Mozilla/4.0 (compatible; MSIE 5.0b1; Windows 95)*, whereas Netscape Navigator 4 has the similar designation *Mozilla/4.05 [en] (Win95; I)*.

Figure 13.9 The contents of **HTTP_ USER_AGENT** for Internet Explorer 5 and Netscape Navigator 4 show that things are not as clear-cut as one could hope.

Although it is possible to parse this information into something meaningful, the real power of **USER_AGENT** comes in conjunction with the **Server** object that's part of ASP. The **Browser Client Capabilities** object is a DLL belonging to IIS4 that reads a file, BrowsCap.ini, and extracts the data associated with the **USER_AGENT**. For example, the latest BrowsCap.ini file (June 1998) had the references corresponding to the Internet Explorer 5.0 beta shown in Listing 13.19.

Listing 13.19 A fragment of the BrowsCap.ini file.

```
;; MSIE 5.0
[IE 5.0]
browser=IE
Version=5.0
majorver=#5
minorver=#0
frames=TRUE
tables=TRUE
cookies=TRUE
backgroundsounds=TRUE
vbscript=TRUE
javascript=TRUE
javaapplets=TRUE
```

```
ActiveXControls=TRUE
Win16=False
beta=False
AK=False
SK=False
AOL=False
crawler=False
CDF=True

; MSIE 5.0 beta 1 browsers
[Mozilla/4.0 (compatible; MSIE 5.0b1; Windows 95)*]
parent=IE 5.0
platform=Win95
beta=True

[Mozilla/4.0 (compatible; MSIE 5.0b1; MSIECrawler; Windows 95)*]
parent=IE 5.0
platform=Win95
beta=True
crawler=True

[Mozilla/4.0 (compatible; MSIE 5.0b1; Windows 98)*]
parent=IE 5.0
platform=Win98
beta=True

[Mozilla/4.0 (compatible; MSIE 5.0b1; Windows NT 5.0)*]
parent=IE 5.0
platform=WinNT
beta=True
```

The **USER_AGENT** key will index a major entry (such as **[IE5]**) or will indicate what browser the key belongs to via the **parent** and **platform** properties. The **beta** property in turn indicates whether the release is a beta release or final (at the time of this writing, Internet Explorer 5 was still in beta). All the other attributes, such as the version number or whether it supports tables, are listed in the parent—here, **[IE5]**.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

From Visual Basic, getting the capabilities information is not quite as intuitive as it should be. First, you need to use the **Server** object (one of the objects supported by Active Server Pages) to create an ActiveX control, **MSWC.BrowserType**. You can then read the various properties of the appropriate browser by setting them as properties to the newly created object. For example, the functions **GetBrowserID** and **GetBrowserVersion** in Listing 13.20 simply encapsulate these calls:

Listing 13.20 GetBrowserID and **GetBrowserVersion** functions both rely on the **Server** object and the **MSWC.BrowserType.**

```
Public Function GetBrowserID() as string
    ' This will return the abbreviated form of the browser name
    ' such as IE for Internet Explorer or NS for
    ' Netscape Navigator
    Dim objBrowser as Object
    Set objBrowser=Server.CreateObject(MSWC.BrowserType)
    GetBrowserID=cstr(objBrowser.Browser)
End Function

Public Function GetBrowserVersion() as Single
    ' This will return the version number of the browser as type
    ' Single.
    Dim objBrowser as Object
    Set objBrowser=Server.CreateObject(MSWC.BrowserType)
    GetBrowserVersion=CSng(objBrowser.Version)
End Function
```

If the requested property doesn't exist for the given browser, then **MSWC.BrowserType** will return **FALSE** for that particular property. This capability is invaluable for determining which scripting language is available (if any); whether the browser supports frames, ActiveX, or Java applets; what platform the browser is running on; and a host of other attributes. If the specific **USER_AGENT** is not found in the list, the BrowseCap.dll will attempt to parse the reference to the closest version that does match. Failing that, the DLL will return a reference to the default browser, which has few capabilities.

You should use the **BrowseCap** object to target your code to fit as many of your browser objectives as possible. By modifying the file *before* it gets to the client, you cut down on the amount of unsafe and redundant code that is executed on the client side. This makes for cleaner Web documents with tighter security and better focus.

NOTE
IIS 4.0 comes with a version of BrowsCap.ini, but because browser capabilities change swiftly, you should maintain the latest version of this file so that you have the most accurate version. CyScape Inc. maintains the BrowsCap.ini list and will usually have the latest version (even Microsoft's file at www.backoffice.microsoft.com/downtrial/moreinfo/bcf.asp was several months older than CyScape's, which can be found at www.cyscape.com/asp/browscap).

Where To Go From Here

It is well nigh impossible to provide all the information about Web Classes and server-side development in one chapter. (It's tough even in a whole book.) Web Classes marry the versatility, data-type richness, and ease of use of Visual Basic with the powerful mechanisms that make up Active Server Pages, which in turn combine server-side programming with the robust database capabilities of ADO.

In Chapter 14, the focus shifts to the other innovation of Visual Basic 6: the Dynamic HTML application. Although IIS applications are built around ASP and are intended as a way of dynamically generating HTML code (and other code) to a wide range of browsers, DHTML applications work only with Internet Explorer 4.0 and later but take advantage of the incredible robustness of the DHTML Document Object Model as the canvas. Chapter 15 looks at integrating client- and server-side code into a single cohesive whole, while also looking at some of the practicalities of creating and deploying Web-based applications.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Chapter 14
The Dynamic Client

Key Topics:

- The role of the client
- DHTML applications
- Internet Explorer Object Model
- Tables
- User input

All too often in the past, client/server programming actually meant server programming, with a side of client to go. The client in database programs usually received short shrift, being primarily considered a window into the more complex world of server manipulation, rules validation, and network-to-network communication. With DHTML applications in Visual Basic, this is about to change.

The Role Of The Client

The irony of most client/server programming is that the visual interface—the client part of the equation—is usually the part of the equation that is given the least amount of thought. At the same time, the client is the part of the program that people deal with the most. A well thought-out client application should be mostly transparent, interfering as little as possible with data interaction.

That is not to say that the program should be dull and uninteresting, made up of nothing but text boxes, combo boxes, and the occasional button. Actually, this is exactly what transparency in the client does *not* mean. All too often, an

interface looks like the cockpit dashboard in a commercial jet, with dozens of combo boxes, scroll bars, buttons, and switches to accomplish specialized purposes. While this can provide a highly detailed view of the data, such interfaces, for the most part, suffer from providing too much information.

At this point, you might be questioning such a statement—you can never provide too much information to the person using your application, right? Wrong. Most client/server programs provide too much information. The user has to sort out what is relevant, place the relevant information into context, and then figure out how to discard what is not needed. The more information that has to be reviewed and discarded as unimportant, the longer it takes the user to get the important information, and the more unpleasant it is to work with the program.

Thus, in a way, the process of building a good interface is to create a screening mechanism to present the important information in an obvious manner while keeping secondary data in the background. Primary information should require the least amount of work to get, while secondary information should be accessible in a logical and consistent fashion.

Unfortunately, most programming tools actually used to discourage this way of thinking. Before the advent of visual programming, putting together an interface often required getting out graph paper and plotting each corner of each control object's window. Visual Basic changed much of that, because with VB you can drag the rectangle of an object's boundary on a form, move the object around, resize it, and delete it with a key press.

Although VB made it easier to create good interfaces, it also made it easier to create bad interfaces. The pain involved in making interfaces in C++ served as a serious incentive to plan the work ahead of time. On the other hand, with Visual Basic, you can create an interface in very little time that has dozens of controls. This has the dual effect of cluttering up the useful information with secondary or even irrelevant data and making it more difficult for the application to modify itself if the focus of the data changes. Think about the code necessary to resize a form with 45 controls, and you begin to get an idea of why such interfaces break down.

Related to this is the amount of work necessary to provide different views of the data. Contemporary database design has long known the importance of *normalization*. Normalization means to pull redundant database elements into their own tables, thereby storing data in the most efficient manner possible. In client interface design, there is a certain amount of normalization that occurs as well. The presentation of the data should change depending on what information needs to be gleaned. (For example, a weather map will use different symbols in the display of temperature gradients than it will when showing rainfall.) In Visual Basic, each of these views frequently requires its own separate form, which in turn requires separate programming layers and introduces coding problems. Although a certain amount of generalized programming can cut this down somewhat (the rainfall and temperature distribution could actually use the same map with some work), this added complexity makes it more difficult to maintain the programs and doesn't work in all situations. A five-day forecast, for example, would not fit well in the

same format as a rainfall map. A Web page, on the other hand, offers an interesting alternative to the standard form.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Previous | Table of Contents | Next

The Web Page As Client

An HTML page can be a remarkably flexible vehicle for presenting information. Indeed, the advantages that it offers as a client interface can appear compelling at first glance:

- Most browsers automatically cause the page to flow so that it fits within the window, regardless of the window's size.
- Text can have a rich format. Different text styles, underlying and emphasis elements, and even colored text can be used at a fairly low cost. You can sort of do this with an RTF control, but the format of RTF is considerably more complex and doesn't support the ability to layer over background graphics.
- Images can be embedded in a document with very little effort, in such a way that text flows around them to accommodate their presence.
- Tables can be set up in a wide number of formats and can handle columns or rows that span across the table. In both Netscape and Microsoft version 4 browsers, the background elements can be modified as well.
- Linking between Web pages is trivial, and several pages can operate in sync with the use of frames.
- HTML pages are computationally inexpensive, intrinsically network aware, and remarkably extensible with ActiveX components, Java applets, and scripting support.

The downside is that the level of support for specific features varies widely from browser to browser and from platform to platform. For the most part, Internet Explorer 4 and Netscape Navigator 4 both support the HTML 3.2 specifications, but the support for HTML 4.0, the most recently adopted HTML standard, is noticeably weak with Navigator. Even with Internet Explorer 5 and Netscape Navigator 5 looming on the horizon, many people are

still using Internet Explorer 3.x and Netscape Navigator 3.x, which don't even uniformly support HTML 3.2.

In the meantime, more corporations are investing in the creation of *intranets*, internal networks that can more precisely define which browser (and version) will access the network data. By standardizing network data for use on a particular browser, companies can take full advantage of the browser's features and keep redundant code to a minimum. Moreover, if the browser could be embedded within an external application, then the application could take advantage of the rich formatting inherent in HTML while still providing the powerhouse of a compiled programming language and IDE.

The notion of a browser working in conjunction with Visual Basic is not new. As early as 1995, the idea that the browser would disappear as a separate application was practically a mantra to both Microsoft and Netscape. This is only logical. When you get right down to it, the browser isn't really an application in the true sense of the word, because you can't actually do much with a displayed Web page. When Internet Explorer 3 was released, Microsoft took the wise step of actually making the "browser" a bundle of components, consisting primarily of the rendering engine for the page (served by SHDOCVW.DLL) and a shell that could call into the browser. This browser component was also just as easily controlled from any other ActiveX container, including Visual Basic.

Internet Explorer 4 raised the stakes by making everything within a Web page an element that could be controlled through a Document Object Model. While IE3 had an object model, it was fairly limited in what it could do. On the other hand, IE4's rich and feature-filled object model has become a very attractive way to display even non-Internet information. Visual Basic 5 could control the Internet Explorer 4 Web browser component in a fairly comprehensive manner, although, for the most part, this information was documented in a haphazard fashion.

Moreover, this solution worked reasonably well when the browser was hosted within Visual Basic (a technique that is discussed in greater detail in this and the next chapter). However, if you wanted to use a Visual Basic as a control to manipulate elements within the Internet Explorer application, you could do so only with a great deal of difficulty. The Visual Basic application either had to be a sizeable ActiveX component (expensive in terms of download time) or a standalone executable running in its own memory space, with the inherent performance drawbacks that such a solution imposes. VB5 offered ActiveX documents, which essentially ran an instance of a Visual Basic application within the shell of Internet Explorer. But, for the most part, the development community has offered a tepid reception to such documents.

Visual Basic 6 offers another solution—DHTML classes—which presents fewer drawbacks than either trying to control an out-of-process server or running a Visual Basic program in the shell's process. DHTML classes are the client-side analog to IIS classes—they reside within a Web page and have complete access to the Internet Explorer document object model for that page. In other words, DHTML classes can control any aspect of a Web page all the way from reacting to **onclick** events to rewriting major portions of the Web

page in response to them. They provide an interesting solution to the dilemma of working with static forms, because a Web page can be resized with no reprogramming, can alter itself to fit data at very little server-side expense, and can load or purge controls as needed.

Although this won't make the standalone Visual Basic application disappear any time soon, client-side Web classes may eventually outnumber their EXE-based brethren, especially in networked environments that deal extensively with databases.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The Dynamic HTML Application

Dynamic HTML applications use Dynamic HTML, cascading style sheets, and a thin ActiveX component to build and control Web pages on the client side. DHTML applications are strictly client-based programs—they can be deployed from any Web server, but they can only be run within Internet Explorer 4 (or greater). As such, they are not very effective within the context of the Internet (in which IE4 makes up only about 30 percent of the browsers) but make perfect sense within the more restricted environment of corporate intranets.

If the core of Internet Information Server applications lies with the Active Server Pages object model, then the basis for Dynamic HTML applications resides in the Internet Explorer object model. Another way of thinking about this is that a client-side Web class can intercept any event that IE4 (or IE5) fires, access or modify any property of the browser, and call any method. In essence, the HTML Web page replaces the traditional Visual Basic form.

Setting Up A DHTML Application

Creating a DHTML Web application is as easy as creating an IIS application (and for that matter, as hard). It's instructive to re-create the archetypal "Hello World" application that we created in Chapter 13 with the DHTML Application Template.

Creating A Client Side "Hello, World!"

This sample will create a very basic Web page with the words *Hello, World!* on the page. When you click on the words, a secret message pops up.

1. Open Visual Basic, and select New from the File menu. Choose the DHTML Application icon to create a new DHTML application (see Figure 14.1).

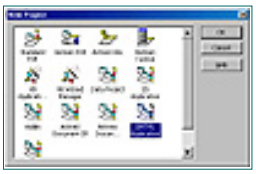

Figure 14.1 Select DHTML Application from the file template types to create a Web program.

2. If it's not already visible, open Project Explorer by choosing View|Project Explorer from the menu or typing Ctrl+R. Open the Designers folder to display DHTMLPage1. This is roughly analogous to a form in a traditional Visual Basic Program (see Figure 14.2).

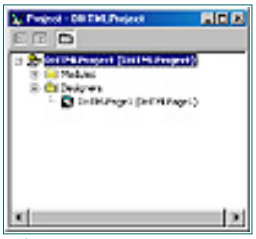

Figure 14.2 DHTML pages can be found in the Designers folder.

3. Double-click on the DHTMLPage1 icon to bring up the DHTML Editor (see Figure 14.3). This small application lets you create simple Web pages from within Visual Basic, although you will probably want to use a more sophisticated editor for most of your real-world projects.

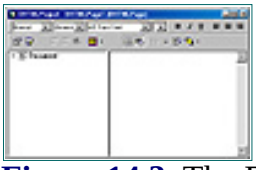

Figure 14.3 The DHTML Editor is a basic Dynamic HTML editor, and it's useful for navigating through the structural elements of a Web page.

4. Figure 14.4 shows labels for the various buttons and features of the editor. Type the word "Hello", press the Enter key, and type "World!". Select *Hello*, and choose Heading 2. Select *World*, and select Heading 1. Observe what happens in the outline view pane.

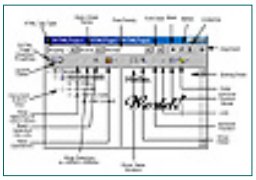

Figure 14.4 Labels showing the major features of the DHTML Editor.

5. Click on the node H2 (Hello) to select it, and press F4 to open the standard Properties box (see Figure 14.5). Set the ID property to Hello—this creates a corresponding ID tag in the H2 node (if you were to look at the HTML code at this point, the H2 code fragment would be more or less **<H2 ID=Hello>Hello</H2>**). The ID is important, because *any element that has an ID in an HTML document will have a corresponding event handler in Visual Basic.*

Figure 14.5 The standard Properties dialog box for the H2 element, with ID set to *Hello*.

TIP
Accessing The Properties List
If you're a veteran VB user, you may have tried right-clicking to bring up the context menu and select the Properties option there. Unfortunately, this will bring up the property page for the element, which will usually be blank unless the object is an image or similar item. Pressing the F4 key will always bring up the Properties list.

---

6. Open the toolbox (View|Toolbox), and click on the HTML button. The normal toolbox controls will be replaced with HTML equivalents (see Figure 14.6). Click on the Button icon (the first control after the pointer arrow), and drag a rectangle in the editor window below the *World!* text. Press F4 to bring up the button's standard Properties dialog box, and set the name, ID, and value of the control to *Greetings*. The ID identifies the button to Visual Basic, the name identifies it for internal consistency with the ID, and the value changes the button's label.

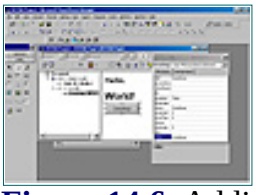

Figure 14.6 Adding a button to your Web page with the DHTML Editor.

7. You are actually creating a Web page in the background. To save this Web page, click on the DHTML Page Designer Properties icon on the editor's toolbar. This will bring up the Properties dialog box (see Figure 14.7) if you haven't saved the page previously. For now, choose Save HTML As Part Of The VB Project. This will keep the HTML internal to the project, rather than saving it to an external file.

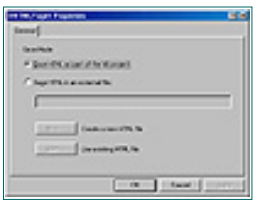

Figure 14.7 The DHTML Page Designer Properties dialog box will let you save your document internally to the project or in a separate file.

8. Close the editor, and open the Project Explorer if it's not already open. Right-click on the DHTMLPage1 entry to bring up the Context menu, and choose View Code to show the familiar Visual Basic code window. If you

click on the Object view (the left combo box), you should see that both Greetings (the button) and Hello (the text) are treated as objects (see Figure 14.8).

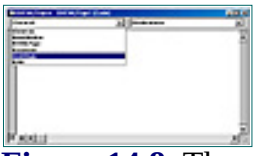

Figure 14.8 The code editor in Visual Basic 6 is the same for working with DHTML applications as it is for traditional forms.

9. Select Greetings, and choose the **onclick** event in the right combo box. Then, enter the following code into the event handler:

```
Private Function Greetings_onclick() As Boolean
    Dim Msg(5) As String
    Dim R As Integer

    Msg(0) = "Hello!"
    Msg(1) = "Hi,there!"
    Msg(2) = "How ya doin'?"
    Msg(3) = "Greetings"
    Msg(4) = "G'day!"
    R = Int(Rnd() * 5)
    ' The document object will be covered later
    Document.parentWindow.alert Msg(R)
End Function
```

The final line in the routine involves the **Document** object, a central part of DHTML programming. The **Document** object is covered extensively later in this chapter. For now, the purpose of the second to last line is to raise an alert box displaying a message that varies each time you click on the Greetings button.

10. Click on the Run button (or press F5) to start the debugger. Visual Basic will display the Project Properties dialog box and you'll see the Debugging tab, which determines how objects are instantiated and displayed (see Chapter 15 for more information about this dialog box). Throughout this chapter, you will probably want to leave the Start Component property set to its default value: DHTMLPage1 (see Figure 14.9).

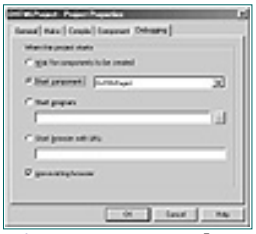

Figure 14.9 The Debugging tab determines when and how the **HTMLPage** object is instantiated.

11. Click on OK to accept the debugging option and start the program. Clicking on the Greetings button will bring up a random greeting (see

Figure 14.10). Notice that the document's URL, displayed in the Title bar, is C:\WINDOWS\TEMP\DHTMLProject_DHTMLPage1.html (your drive and Windows folder name may vary, of course). This page is generated internally by the DHTML application designer.

Figure 14.10 The Hello, World! application in action, after clicking on the Greetings button.

So, what have we actually created here? Examining the temporary file created by the application will give you a major clue. Listing 14.1 shows the HTML code generated by the DHTML application (the file has been edited slightly to improve legibility).

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Listing 14.1 HTML code generated by the DHTML application.

```
<HTML>
<HEAD>
<META NAME="GENERATOR" CONTENT="Microsoft Visual Studio 6.0">
<META CONTENT="text/html" HTTP-EQUIV=Content-Type>
<TITLE></TITLE>
</HEAD>
<BODY>
<!--METADATA TYPE="MsHtmlPageDesigner" STARTSPAN-->
<OBJECT ID="DHTMLPage1"
CLASSID="clsid:CA8B9767-4184-11D2-8BF0-20E650C10000"
WIDTH=0
HEIGHT=0>
</OBJECT>
<!--METADATA TYPE="MsHtmlPageDesigner" ENDSPAN-->
<H2 ID=Hello>Hello</H2>
<H1 ID="">World</H1>
<DIV>
<INPUT ID=Greetings
    NAME=Greetings
    STYLE="LEFT: 33px;
        POSITION: absolute;
        TOP: 130px;
        Z-INDEX: 100"
    TYPE=button
    VALUE=Greetings
    >
</DIV>
</BODY>
</HTML>
```

The highlighted code in Listing 14.1 shows how Visual Basic performs its magic. The DHTML application is an ActiveX control. Specifically, it's an ActiveX DLL embedded within a framework Web page. The **CLASSID** should be familiar to developers as a GUID, which is automatically created and registered when the temporary DLL is created. Although the control can be manipulated from the Web page via scripts (as demonstrated in Chapter 15), for the most part, the control handles the real action on the page. Theoretically, you could build a completely separate page around the control. As long as the items with IDs stay in the page, the control will work.

Obviously, your interest in this book transcends building lame "Hello, World!" pages. However, to make DHTML applications work for you, it's necessary to have a basic understanding of the Document Object Model around which Internet Explorer is built. It's likely that you'll want to create capable code, so you may also want to read the section "Don't Throw Away FrontPage Yet."

Exploring The Internet Explorer Object Model

In 1993, when the Web was just beginning to impinge upon public consciousness, working with HTML was a fairly simple undertaking. Not surprisingly, a great number of people who had only a basic inkling of programming principles took to this page-description language, setting out their shingles as HTML designers. Perhaps because of that, working with HTML and scripting languages has gained an undeserved reputation as being simple, slight, and beneath the attentions of serious application developers.

Five years later, anyone who works with building Web sites in today's supercharged world of e-commerce, DHTML, and Chrome can tell you that the browser object models are some of the most complex pieces of software currently available in the public market. There are dozens of distinct objects, each with their own unique properties and methods, just in the core Internet Explorer application alone. Start adding in the DirectAnimation API, NetShow, NetMeeting, Chat, and so forth, and, pretty soon, you're starting to talk about an application with hundreds of objects and thousands of properties, methods, and events. Yeah, Web programming is easy!

Don't Throw Away FrontPage Yet

Visual Basic 6 has a really cool DHTML Editor that you can use for all your application needs. So, who needs FrontPage, or Visual Studio, or any of those other HTML editors, right? Well, hold off on freeing up that disk space. The VB Editor falls into the same basic category as Notepad or Paint. It's handy if you need something really fast and don't want to open a full-blown editor, but it's not something you'd want to use for real production work. Microsoft recognizes this and has wisely included a Launch Editor button in the VB DHTML Editor. To set this to your favorite editor, select Preferences from the Tools menu, and click on the Editor tab. At the bottom of the Editor tab, you can change the external editor to your tool of choice. This is highly recommended, by the way, because the default choice is that Web editor of champions—Notepad!

The DHTMLPage Object

Although a DHTML application can (and usually will) be made up of many different classes and designers, the actual Web page itself is controlled by a **DHTMLPage** class. As with the IIS **WebClass** object, the **DHTMLPage** class contains a handful of subordinate classes, as shown in Table 14.1.

Table 14.1 Classes belonging to the DHTMLPage class.

Class	Description
BaseWindow	Theoretically, this returns a reference to the current window object in the Web page, with all of its associated properties and methods (including alert, open, status, and so forth). If you have trouble with this class, try using **Document.parentWindow** instead. The **BaseWindow** is of type **HTMLWindow2**.
DHTMLEvent	This contains detailed information about the last event to occur, and its properties are summarized in Table 14.2. Use the **DHTMLEvent** to find where the mouse clicked, which key was depressed, which element was last selected, and so forth. The **DHTMLEvent** is of type **IHTMLEventObj**.
Document	The document embodies the structure of an HTML document, so its properties and methods essentially span nearly all of DHTML.

The **DHTMLPage** object also has two associated events—the **Load** event, called when all the components within the ActiveX control have finished loading, and the **Unload** event, called when all the components have completed unloading. In general, as long as the page doesn't include other ActiveX components, the **Load** event corresponds to the **Load** event of a form—variables, properties, and objects can be initialized, but the page itself won't display until the **Load** event finishes processing.

Note that for any given **DHTMLPage** object (that is, **DHTMLPage1**, **DHTMLPage2**, and so forth), **DHTMLPage** (with no number) is a static class for that page. In other words, the following syntax is correct within a **DHTMLPage** object

```
DHTMLPage.Document
```

but this is not:

```
DHTMLPage1.Document
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The DHTMLEvent Object

If you peruse the event handlers for the Greetings button in the Hello World client program, one thing might strike you after a bit. There are no parameters for any event, even ones like **onmousemove**, which correspond to the Visual Basic **mouseMove** event. This reflects the different language origins of Visual Basic and JavaScript (see “To Write With Coffee” for more information). JavaScript’s implementation of the event model is to work with an event object that contains not only the relevant ID of the event but also related information, such as mouse locations, key press events, and the like. Visual Basic 6 adopts this procedure by creating the **DHTMLEvent** object.

To Write With Coffee

The first Web browser to include a scripting language was Netscape Navigator 2. It combined some (very) limited control over elements on the page—mainly form information—with a language derived from Java, called first LiveScript, then JavaScript. Although it is syntactically similar to Java, JavaScript is a purely interpreted language and has little to no typecasting, whereas Java is tightly typecast.

When Internet Explorer 3 was released, a version of Visual Basic (VBScript) was included with the program, and, because of the need to maintain cross platform support at even a minimal level, so was JavaScript (though to muddy the waters even further, Microsoft called their version Jscript). In the following years, even Microsoft’s browser-development staff realized that JavaScript was becoming the accepted standard scripting language on the Web. Recently, this unofficial status was changed to an official one by a European standards review committee (with the Swiss acronym of ECMA), which ironed out the subtle differences between Netscape’s and Microsoft’s implementation of the language to produce a new version of JavaScript, giving it the dubious name of ECMAScript.

However, in the competitive world of Internet software, even such a standard as this was only destined to last until such time as the engineers could get home to their respective companies. Although it is likely the Internet Explorer 5 and Netscape Navigator 5 versions of JavaScript (few use ECMAScript unless they absolutely have to) will be closer than the 4 versions of the same language, there is little doubt that they will not be completely compatible.

The **DHTMLEvent** object should be placed within an event handler (or a function called from an event handler) because it contains a record of only the last event in the event queue. A simple example of how this can be used would be to determine where the mouse was clicked on a page. Add the following code to the **Document_onclick()** event in the Hello World client program created earlier:

```
Private Function Document_onclick() As Boolean
    Document.parentWindow.alert "Mouse click at " + _
       CStr(DHTMLEvent.x) + "," + CStr(DHTMLEvent.y)
End Function
```

The **DHTMLEvent.x** and **DHTMLEvent.y** properties give the horizontal and vertical components, respectively, of the position of the mouse within the Web page (that is to say, the upper-left corner of the Web page window is the origin). When run, this program produces an alert box indicating where the mouse was clicked.

The events that can be picked up with the **DHTMLEvent** object are covered in more detail in Table 14.2. This object is also functionally identical to **Document.parentWindow.Event**.

Table 14.2 Properties of the **DHTMLEvent** class.

Property	Type	Description
AltKey	Boolean	True if the Alt key is depressed; false otherwise.
Button	Boolean	Retrieves which mouse button was pressed (0—no button pressed, 1—left button, 2—right button, 3—middle button).
CancelBubble	Boolean	Set to true to cancel events bubbling up to next layer; set to false to let them continue (default).
ClientX	Long	Location of the horizontal position (in pixels) of the mouse click event relative to the upper-left corner of the Web page.
ClientY	Long	Location of the vertical position (in pixels) of the mouse click event relative to the upper-left corner of the Web page.

CtrlKey	Boolean	True if the Ctrl key is depressed; false otherwise.
FromElement	IHTMLElement	Retrieves the object the mouse is exiting during **onmouseover** and **onmouseout** events.
KeyCode	Long	Numeric code sent by keyboard indicating which key was pressed. Does not correspond exactly one-to-one with ASCII, but it is close for numbers and letters.
OffsetX	Long	Location of the horizontal position (in pixels) of the mouse click event relative to the upper-left corner of the object in which the event occurred.
OffsetY	Integer	Location of the vertical position (in pixels) of the mouse click event relative to the upper-left corner of the object in which the event occurred.
Reason	Long	In a database transaction, returns a number indicating the success or failure of the operation (0—data transferred successfully, 1—operation aborted, 2—data sent erroneously).
ReturnValue	Variant	The return value provided by a dialog when it closes. See Chapter 15 for more information about dialogs.
ScreenX	Long	Location of the horizontal position (in pixels) of the mouse click relative to the upper-left corner of the computer screen.
ScreenY	Long	Location of the vertical position (in pixels) of the mouse click relative to the upper-left corner of the computer screen.
ShiftKey	Boolean	True if the Shift key is depressed; false otherwise.
SrcElement	IHTMLElement	A reference to the object that fired the event.
SrcFilter	Object	A reference to the filter that fired an **onfilterchanged** event.
ToElement	IHTMLElement	A reference to the element being entered on an **onmouseover** or **onmouseout** event.

Type	String	The type of event that occurred. This corresponds to the appropriate event handler (**type=click** for **onclick** events, **type=mousedown** for **onmousedown** events, and so forth).
X	Long	Same as the **ClientX**.
Y	Long	Same as the **ClientY**.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The distinctions between the various positional properties (**ClientX**, **ScreenX**, and so forth) may not be all that straightforward. A quick example can illustrate the difference as well as show how to use the **DHTMLEvent** object in a more varied circumstance:

1. In the (General) code section of the DHTMLPage Designer, create the **GetMousePosition** subroutine:

```
Public Sub GetMousePosition()
    Dim Buffer As String
    Buffer = ""
    Buffer = Buffer + "ClientX=" + _
      CStr(DHTMLEvent.clientX) + ";" + "ClientY=" + _
      CStr(DHTMLEvent.clientY) + vbCrLf
    Buffer = Buffer + "OffsetX=" + _
      CStr(DHTMLEvent.offsetX) + ";" + "OffsetY=" + _
      CStr(DHTMLEvent.offsetY) + vbCrLf
    Buffer = Buffer + "ScreenX=" + _
      CStr(DHTMLEvent.screenX) + ";" + "ScreenY=" + _
      CStr(DHTMLEvent.screenY) + vbCrLf
    Buffer = Buffer + "X=" + _
      CStr(DHTMLEvent.x) + ";" + "Y=" + _
      CStr(DHTMLEvent.y)
    Document.parentWindow.alert Buffer
End Sub
```

2. In the **onmousedown** event handler for the Greeting button, the *Hello* text, and the **document**, call the **GetMousePosition** routine:

```
Private Sub Document_onmousedown()
    GetMousePosition
End Sub
```

```
Private Sub Greeting_onmousedown()
    GetMousePosition
End Sub

Private Sub Hello_onmousedown()
    GetMousePosition
End Sub
```

When you run the program and click, you'll get a dialog box showing you where you clicked. Clicking on the button is most informative (see Figure 14.11). In Figure 14.11, the mouse was clicked in the upper-left corner of the button. The **ClientX** position shows the distance from the edge of the client (the browser). The **OffsetX** position, on the other hand, shows the distance from the edge of the element (the button), and the **ScreenX** position shows the distance from the left side of the screen. The final position, **X**, is just a synonym for **ClientX** and was included for cross-compatibility purposes.

Figure 14.11 Clicking on the button in this application demonstrates the differences between coordinate systems.

The Document Object

Fortunately, when dealing with the core DHTML set at least, almost everything in a Web page derives in some fashion from one of two objects: the **Document** object and the document's corresponding **Window** object. Just as the **Server** object is automatically exposed in an IIS application, the **Document** object is freely available in the DHTML application—you don't need to explicitly declare it to use it.

In a book on client/server programming, it is simply not possible to go into depth about the IE Object Model. But, to get an idea of the scope of the model, open the Object Browser, and set the type library (the top-left combo box) to MSHTML, which is the Internet designation of the scripting library. In the latest version of IE (the beta build for Internet Explorer 5), there are 220 distinct classes, many with 40 or more properties, methods, and events. The classes can readily be identified as they all start with either HTML or IHTML.

The **Document** class itself is of type **HTMLDocument**. From the document, you can get a reference to the body of the document with **Document.Body**, and a reference to the background color of the body with **Document.Body.BgColor**. In short, the structure echoes the tag and attribute structure of HTML itself.

In addition to direct tag references, the **Document** class also owns a number of

collections, many of which are covered in greater detail later in this chapter as well as in the next chapter. From the standpoint of Visual Basic, these collections work similarly to enumerated dictionaries in the previous chapter—you can reference an element out of the collection by either its ordinal position on the page (that is, in which order it occurs in the HTML) or by the element's ID, if it's assigned one. Thus, you can retrieve the third image displayed on the page (not counting background graphics) by saying **document.images(2)—2** because all arrays in HTML are zero-based. Similarly, if that particular graphic had an ID of **"myImage"**, then you could also reference it as **document.images("myImage")**.

In order to give a proper treatment to the **Document** object, you almost need to review all of Dynamic HTML, which is unfortunately beyond the scope of this book. This chapter and Chapter 15 both explore specific elements within the document, but to get a better handle on the subject, I recommend any of the books mentioned at the end of Chapter 15.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The Window Object

The **Window** belonging to the **Document** object is especially important because it handles most of the non-Web page related items, such as events, status messages, and alert and dialog boxes. You can get a handle to the document's window through the **document.windowParent property**, which has a type of **HTMLWindow2**. The original **HTMLWindow** occurred only in the Internet Explorer 3 object model and was superceded by **HTMLWindow2** in subsequent versions.

As with the **Document** object, the **Window** object is reasonably complex, and its full coverage is beyond the scope of this book. However, certain window methods can be of use to client/server developers in particular, so these are summarized in Table 14.3. Examples of most of the elements in Table 14.3 are shown throughout the book, so they aren't covered here.

TIP
Workaround For A BaseWindow Problem
In theory, the document's window should be the **BaseWindow** object that the **DHTMLPage** class exposes. However, at least in the beta, using the **BaseWindow** property for anything other than determining when the Web page was completely loaded generates all kinds of errors. Although this may change with the final version of Visual Basic, if you run into problems with **BaseWindow**, stick with using the **windowParent** property of the document to get the active window instead.

Table 14.3 Selected **Window** properties and methods.

Property Or Method	Example	Description
Alert(msg as String)	**Alert("This is a test")**	Pops up a standard dialog box with the text message and a single OK button.
Blur()	**Blur()**	Causes the window to lose its focus.

ClearInterval(timerID as Long)	**ClearInterval(id)**	When passed the ID generated by a **SetInterval** call, **ClearInterval** turns off the timer and clears it from memory (see **SetInterval**).
ClearTimeout(timerID as Long)	**ClearTimeout(id)**	When passed the ID generated by a **SetTimeout** call, **ClearTimeout** turns off the timer and clears it from memory (see **SetTimeout**).
Close()	**Close()**	Closes a Web page or dialog box. If a Web page, this will also prompt users to let them know that code is attempting to close the page.
Confirm(msg as String)	**Confirm("Are you sure you want to leave?")**	Pops up a standard dialog box with the text message and two buttons: Yes and No. Pressing Yes returns the value **true**, and pressing No returns the value **false**.

DefaultStatus	**DefaultStatus="Roll over any button to highlight an option."**	The default status is the message displayed at the bottom of the browser window when it is not over a "live" element. Compare **Status**.
Document	**Document**	Returns a handle to the document object.
Event	**Event**	Returns a handle to the event object. Same as the **DHTMLEvent** object in a **DHTMLPage** class.
ExecScript(commandStr as String,lang as String)	**ExecScript("Terminate()", "JavaScript")**	Attempts to perform the command given by the **commandStr** parameter in the language specified. Useful, but potentially dangerous.
Focus()	**Focus()**	Restores focus to the current window.
Frames()	**Frames(3)**	Returns a reference to subordinate frames collection. Not recommended for use in DHTML applications.
Open(URL as String,name as String,features as String, replace as Boolean)	**Open(http://www.microsoft.com, "MSWindow", "width:100;height:100")**	Opens a new window, and loads a URL into it.

Prompt(msg as String, defaultStr as String)	**Prompt("What is your favorite color?","Blue ")**	Pops up the equivalent of a VB Input box, with the **msg** as the prompt and the **defaultStr** as the default value.
Scroll(X as Long,Y as Long)	**Scroll(20,30)**	Moves the browser window's client area by the amount indicated.
SetInterval(cmdStr as String, msec as Long, language as String)	**ID=SetInterval("Check-Clock()",1000, "JavaScript")**	Causes a command to be evaluated repeatedly at intervals of milliseconds. Returns a handle ID, which can be passed to **ClearInterval** to stop the process.
SetTimeout(cmdStr as String, msec as Long, language as String)	**ID=SetTimeout("Times-Up()",3000,"JavaScript")**	Causes a command to be evaluated once after milliseconds have passed. Returns a handle ID, which can be passed to **ClearTimeout** to abort the process.
ShowModalDialog(dlg as String,inArg as Variant, varOption as Variant)	**ShowModalDialog("getData.htm","Big and yellow","dialogWidth=400 dialogHeight=300")**	Creates a dialog form rather than a standard window. This is covered in detail in Chapter 15.

Status	**Status="Now Loading…"**	Sets the status text at the bottom of the browser.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Text Techniques

When you get right down to it, HTML is text. This might sound trite and obvious, but keeping that fact in mind becomes essential when you want to do anything with Dynamic HTML. One direct consequence of HTML being text is that you can change any facet of your Web page simply by modifying the text for that section. In the simplest case, that means changing the contents of a **<DIV>** or similar element when a button is clicked, whereas more complex examples would include modifying the background graphic and all system colors, changing a list of items to a tree hierarchy, or perhaps even animating selections of text in multiple colors as their content changes.

NOTE
I do not recommend animating text in multiple colors as their content changes. DHTML in excess can make the **<BLINK>** tag of years past seem positively the height of Web design in comparison.

With the exception of form elements, nearly every visible element in a Web page is derived from **HTMLBlockElement**. A *block* is simply a container of other elements, whether text or other block elements. You can modify the contents of these blocks with a number of text-related methods and properties, which are summarized in Table 14.4. These are not the only ways that you can manipulate text in DHTML. By using such objects as the **Selection** object and the **Text Range** object, you can actually create a functional, albeit somewhat minimalistic, Dynamic HTML editor.

Table 14.4 Text and HTML editing properties and methods.

Property Or Method	Example	Description

ClassName=rule	**ClassName="H1"**	Sets the style to the new style rule (covered in Chapter 15), or retrieves the name of the current rule.
InnerHTML=htmlString	**InnerHTML="<B>I'm a bold statement</B>"**	**InnerHTML** retrieves or sets the contents of the block element to the HTML expression, with HTML notation getting converted into elements in the DHTML structure.
InnerText=textString	**InnerText="This example will show the brackets in the <B>"**	**InnerText** converts the string into an HTML expression, replacing characters with their character equivalents (that is, <becomes **<**). Useful for getting the text from a highly formatted HTML block.
InsertAdjacentHTML (htmlText,position)	**InsertAdjacentHTML "<LI>Another List Element</LI>", "BeforeEnd"**	Inserts HTML text or elements into a block. Position can be **"BeforeBegin"**, **"AfterBegin"**, **"BeforeEnd"**, or **"AfterEnd"**, indicating whether the text appears before or after the enclosing tags at the beginning or end of the block.
InsertAdjacentText (text,position)	**InsertAdjacentText "Some More Text", "AfterBegin"**	Similar to **InsertAdjacentHTML**, except that text is converted implicitly into a safe HTML expression first.
OuterHTML=htmlString	**OuterHTML=" This will completely replace the old element "**	**OuterHTML** replaces the entire block, not just its contents, with the replacement string (or retrieves the whole block). Otherwise, it is the same as **InnerHTML**.

OuterText=textString	**OuterText="This will completely replace the old element."**	**OuterText** either retrieves the content of the selected block or replaces the whole block (including surrounding tags) with the text. It's a good way to eliminate a reference from a page.
Style.attribute=value	**Style.position="absolute"**	Sets or retrieves individual elements of the style attribute (covered in Chapter 15).
ToString()	**MyElement.toString**	Converts an HTML element into a string representation of itself. Only supported in Internet Explorer 5.

Of all the methods used for text manipulation, you will probably end up using the pair **innerHTML** and **innerText** most often. The Hello, World client program can easily be modified to demonstrate their use.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Integrating Dynamic Text

The best way to see Dynamic HTML in action is to try manipulating the text. In this simple Hello, World exercise, you'll create a button that will generate a random greeting.

1. It is easier to add a **<DIV>** from an external editor, but you can only do that if the HTML file is itself kept in a distinct document rather than part of the project. Open the DHTML Editor, and click on the DHTML Page Designer Properties. Select the Save HTML In An External File option, and enter "HelloWorld-Client.htm" in the text field (see Figure 14.12). Click on the New button to save this to your working folder, and click on OK.

Figure 14.12 Save your HTML in an external file to edit outside of the DHTML Editor.

2. Click on the Launch Editor button to bring up your editor of choice (see the sidebar "Don't Throw Away FrontPage Yet" for more information), and create a new **<DIV>** called **MouseData** by modifying the code as shown here (the code was reformatted slightly for legibility):

```
<HTML>
<HEAD>
<META NAME="GENERATOR" CONTENT="Microsoft Visual Studio
6.0">
<META CONTENT="text/html" HTTP-EQUIV=Content-Type>
<TITLE></TITLE>
</HEAD>
<BODY>
<H2 ID=Hello STYLE="relative">Hello</H2>
<H1 ID="">World</H1>
<DIV>
```

```
<INPUT ID=Greetings
    NAME=Greetings
    STYLE="LEFT: 33px;
           TOP: 130px;
           Z-INDEX: 100"
    TYPE=button
    VALUE=Greetings
    >
</DIV>
<DIV ID="MouseData" STYLE="position:relative">
   Mouse Data</DIV>
</BODY>
</HTML>
```

The **STYLE**="**position:relative"** attribute that was added converts the position attribute from static to dynamic, making it updatable. If you don't do this, there is a chance that your program won't work correctly.

3. Press Alt+Tab to get back to Visual Basic and bring the editor into focus, if it is not already. When prompted whether you want to overwrite the current file, click on Yes.

4. Open the project window, and right-click on the DHTMLPage1 icon to select the View Code option. From the (General) section, open **GetMousePosition**, and modify it so that it generates HTML into the newly created <**DIV**>, as shown here:

```
Public Sub GetMousePosition()
    Dim Buffer As String
    Buffer = ""
    Buffer = Buffer + "ClientX=" + _
      CStr(DHTMLEvent.clientX) + ";" + "ClientY=" + _
      CStr(DHTMLEvent.clientY) + "<BR>"
    Buffer = Buffer + "OffsetX=" + _
      CStr(DHTMLEvent.offsetX) + ";" + "OffsetY=" + _
      CStr(DHTMLEvent.offsetY) + "<BR>"
    Buffer = Buffer + "ScreenX=" + _
      CStr(DHTMLEvent.screenX) + ";" + "ScreenY=" + _
      CStr(DHTMLEvent.screenY) + "<BR>"
    Buffer = Buffer + "X=" + CStr(DHTMLEvent.x) + ";" + _
      "Y=" + CStr(DHTMLEvent.y)
    MouseData.innerHTML = Buffer
End Sub
```

5. In a similar manner, change the event handler for the Greetings button so that it outputs the randomly generated message to the Hello handler. Here, you want to use **innerText**, because you only want to change the text of the message, not its style as a heading:

```
Private Function Greetings_onclick() As Boolean
    Dim Msg(5) As String
    Dim R As Integer

    Msg(0) = "Hello!"
    Msg(1) = "Hi,there!"
    Msg(2) = "How ya doin'?"
```

```
        Msg(3) = "Greetings"
        Msg(4) = "G'day!"
        R = Int(Rnd() * 5)
        ' The document object will be covered later
        Hello.InnerText=Msg(R)
    End Function
```

6. Run the application, and play with it. Now when you click anywhere within the page, the **MouseData** will change rather than an obvious alert box.

This is obviously just a glimmer of what is possible by modifying internal text. Dynamic HTML eliminates the need for frames (one of the more frustrating elements for Web developers and users both) and has considerably more flexibility than that technology. Now, it's time to do something a little more useful with it—connect it to data via tables.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Building Tables

There are a lot of ways of representing data—bar graphs, record listings, even three-dimensional sheets representing seas of data (something I absolutely, positively *won't* get into here). But for all the graphical magic available, the simple table is still one of the best ways to go. You can represent relational data concisely, you can view data as records, and you can see with a glance when data is questionable or aberrant. It's perhaps no wonder that the spreadsheet (which is really nothing more than a glorified table manager) was one of the earliest software innovations and still makes up the biggest use of computers outside of word processing.

Tables appeared fairly late in HTML, not showing up until well into the development cycle of HTML 2, yet for many, they provide not only the ability to show data but also a means of positioning elements prior to the advent of DHTML. Given that a table cell can hold images, movies, ActiveX or Java components, and more in addition to text, the ability to create and manipulate tables is one of the major features that Internet Explorer offers, and Visual Basic provides a certain level of support for these tabular functions.

This section will work through an example of how to implement tables using DHTML Web classes. In the example, the HTML document will display a listing of city weather records, including temperatures in degrees Fahrenheit. When you roll over a temperature with your mouse, it will switch to degrees Celsius, reverting when the mouse moves away. In addition to showing how to work with tables (and with the IE object model), this gives an example of how to conserve previous screen real estate by "hiding" functionality until it's needed. But first, you should look at how to get the table into the page in the first place.

A Useful Buffer Class

As you may have discovered, one of the problems with trying to generate HTML files from another source, such as Visual Basic or JavaScript, is that the notation for doing so can be rather cumbersome. For example, the code shown in Listing 14.2 will generate a table in HTML, assuming that the record set contains a collection of weather records.

Listing 14.2 A sample of creating a table using a buffer.

```
Public Sub WriteTable(RS as RecordSet,ID as string)
    Dim buffer as string
    Buffer="<TABLE border='2'>"
    Buffer=Buffer+"<TR>"
    Buffer=Buffer+"<TH>State</TH>"
    Buffer=Buffer+"<TH>City</TH>"
    Buffer=Buffer+"<TH>Hi</TH>"
    Buffer=Buffer+"<TH>Lo</TH>"
    Buffer=Buffer+"<TH>Skies</TH>"
    Buffer=Buffer+"<TH>Forecast</TH>"
    Buffer=Buffer+"</TR>"
    RS.MoveFirst
    While not RS.EOF
        Buffer=Buffer+"<TR>"
        Buffer=Buffer+"<TD>"+RS("State")+"</TD>"
        Buffer=Buffer+"<TD>"+RS("City")+"</TD>"
        Buffer=Buffer+"<TD>"+RS("Hi")+"</TD>"
        Buffer=Buffer+"<TD>"+RS("Lo")+"</TD>"
        Buffer=Buffer+"<TD>"+RS("Skies")+"</TD>"
        Buffer=Buffer+"<TD>"+RS("Forecast")+"</TD>"
        Buffer=Buffer+"</TR>
        RS.MoveNext
    Wend
    Buffer=Buffer+"</TABLE>"
    Document.all(ID).innerHTML=Buffer
End Sub
```

Although such code works perfectly well, the **Buffer=Buffer+** type notation can sometimes prove tedious, especially if you are used to working with more C++ capabilities, such as streams. After some time tracking down erroneous string statements, I wrote such a stream class. It's used somewhat extensively later in this chapter, so it's worth noting that this class is not a native class in Visual Basic, although you can load it in from this book's CD-ROM.

One of the other reasons for writing the stream class was to demonstrate that you can still use traditional classes with DHTML applications as you would with any other Visual Basic project type. Indeed, there are several benefits to doing so. You can encapsulate most of the business logic and utility routines in external classes while using the DHTML Web class strictly as a way of communicating with the HTML document.

The **COStream** class creates an internal buffer that accumulates strings of HTML or scripting code. You can actually assign to it any block element, such

as a **<DIV>** or paragraph element, accumulate strings into an internal buffer in the class, and then make a call to the **Output()** method to cause the string to replace the current contents of that block. You can also specify whether you want to output the data as HTML (the default) or text. In the latter case, HTML components, such as brackets (<>), get replaced by their character equivalents, in this case **<** and **>**.

Creating a class for a DHTML application is the same as creating a class for a standard executable or an ActiveX DLL, as demonstrated in the following exercise.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Adding A COStream Class Module

The code for the **COStream** class module is both a demonstration of how you can add additional class and general modules to your DHTML application, and a handy class whenever you need to process and output a large amount of text.

1. Open a DHTML application (or create a new one). From the Project menu, select Add Class Module. From the list of options, select Class Module. Set the name of the new class to **COStream** (for Output Stream Class).

2. Double-click on the class icon in the Project window to bring up the code window. Define an enumeration for handling the type of output, and add the following local variables into the **(General)_(Declarations)** pane:

```
Option Explicit

Public Enum OStreamTargetConstants
    ostText = 0
    ostHTML = 1
End Enum

'local variable(s) to hold property value(s)
Private mvarTarget As HTMLBlockElement 'local copy
Private mvarOutputType As OStreamTargetConstants
'local copy
Private buffer As String
```

3. Define the methods within the class (their uses will be covered later), as shown in Listing 14.3.

Listing 14.3 Method definitions in the COStream class.

```
Public Function WriteC(TextData As String) As String
```

```
    Dim expn As String
    expn = TextData
         buffer = buffer + expn
    WriteC = expn
End Function

Public Function Length() As Long
         Length = Len(buffer)
End Function

Public Function Output(Optional outString As Variant) _
   As String
    On Error GoTo OutputErr
    If Not IsMissing(outString) Then
        buffer = outString
    End If
    If mvarOutputType = ostHTML Then
        mvarTarget.innerHTML = buffer
    Else
        mvarTarget.innerText = buffer
    End If
    Output = buffer
    Clear
    Exit Function
OutputErr:
    Call RaiseError(MyUnhandledError, _
       "COstream:Output Method")
End Function

Public Function Flush() As String
    On Error GoTo FlushErr
    Flush = buffer
    Clear
    Exit Function
FlushErr:
    Call RaiseError(MyUnhandledError, _
       "COstream:Flush Method")
End Function

Public Sub Clear()
         buffer = ""
End Sub

Public Function WriteSafeQ(TextData As String) As String
    On Error GoTo WriteSafeQErr
    Dim expn As String
    expn = "\'" + TextData + "\' "
    buffer = buffer + expn
    WriteSafeQ = expn
    Exit Function
```

```
WriteSafeQErr:
    Call RaiseError(MyUnhandledError, _
       "COstream:WriteSafeQ Method")
End Function

Public Function WriteTag(Tagname As String, _
   TextData As String) As String
    On Error GoTo WriteTag
    Dim expn As String
    Dim EndTag as string

    EndTag=Split(Tagname," ")(0)
    expn = "<" + Tagname + ">" + TextData + _
       "</" + EndTagname + ">"
    buffer = buffer + expn
    WriteTag = expn
    Exit Function
WriteTag:
    Call RaiseError(MyUnhandledError, _
       "COstream:Tag Method")
End Function

Public Function WriteQ(TextData As String) As String
    On Error GoTo WriteQErr
    Dim expn As String
    expn = """+textdata+"" "
    buffer = buffer + expn
    WriteQ = expn
    Exit Function
WriteQErr:
    Call RaiseError(MyUnhandledError, _
       "COstream:WriteQ Method")
End Function

Public Function WriteS(TextData As String) As String
    On Error GoTo WriteSErr
    Dim expn As String
    expn = TextData + " "
    buffer = buffer + expn
    WriteS = buffer
    Exit Function
WriteSErr:
    Call RaiseError(MyUnhandledError, _
       "COstream:WriteS Method")
End Function

Public Function WriteLn(TextData As String) As String
    On Error GoTo WriteLnErr
    Dim expn As String
    expn = TextData + vbCrLf
```

```
    buffer = buffer + expn
    WriteLn = expn
    Exit Function
WriteLnErr:
    Call RaiseError(MyUnhandledError, _
       "COstream:WriteLn Method")
End Function

Public Function WriteBreak(TextData As String) As String
    Dim expn As String
    expn = TextData + "<br>"
    On Error GoTo WriteLnErr
    buffer = buffer + TextData + "<br>"
    WriteBreak = expn
    Exit Function
WriteBreakErr:
    Call RaiseError(MyUnhandledError, _
       "COstream:WriteBreak Method")
End Function
```

4. Create two public properties: **OutputType**, which can take one of the two values from the enumeration you defined earlier, and **Target**, which contains the destination HTML block element that the stream can output to (see Listing 14.4).

Listing 14.4 Property definitions in the COStream class.

```
Public Property Let OutputType(ByVal vData _
   As OStreamTargetConstants)
     mvarOutputType = vData
End Property

Public Property Get OutputType() _
   As OStreamTargetConstants
    OutputType = mvarOutputType
Exit Property
OutputTypeGetErr:
    Call RaiseError(MyUnhandledError, _
       "COstream:OutputType Property Get")
End Property

Public Property Set Target(ByVal vData As HTMLBlockElement)
    Set mvarTarget = vData
End Property

Public Property Get Target() As HTMLBlockElement
    Set Target = mvarTarget
End Property

Public Property Let StreamBuffer(expr as String)
         Buffer = expr
End Property
```

```
Public Property Get StreamBuffer() as string
        StreamBuffer=buffer
End Property
```

5. Select the Procedure Attributes entry in the Tools menu, and click on the Advanced button to open additional options. Set the Name combo box entry to StreamBuffer, then, in the Procedure ID combo box, set the option to (Default). This makes the **StreamBuffer** the default property, such that if an instance of **COStream** is defined as **Ost**, then **Ost="a string"** is the same as **Ost.StreamBuffer="a string"**.

6. Finally, in the **Class_Initialize** section, initialize the buffer (not strictly necessary but good form) and set the default state for the **OutputType** property to **ostHTML**, as shown:

```
Private Sub Class_Initialize()
    buffer = ""
    mvarOutputType = ostHTML
End Sub
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The **COStream** class has a simple interface, as outlined in Table 14.5.

Table 14.5 Properties and methods for the **COStream** class.

Property Or Method	Example (Assumes Ost=new COStream)	Description
Clear	**Ost.Clear**	Empties the buffer without updating the target.
Flush	**Ost.Flush**	Returns the contents of the buffer then clears it. **Flush** does not update the target.
Length	**Ost.Length**	Returns the number of characters in the string (note that this uses the **Len()** rather than **LenB()** function).
Output	**Ost.Output Ost.Output "This is a test."**	Sends the current contents of the buffer into the HTML block element defined by **Target**, if used with no parameters, or outputs the string passed to it to the element, if used with a string parameter. In the latter case, the string replaces the previous contents of the buffer.
OutputType	**Ost.OutputType=ostHTML Ost.OutputType=ostText**	Determines whether buffered output is displayed as HTML (the default) or text.

StreamBuffer	**Ost.StreamBuffer="This is buffered text" Ost="This is buffered text ?Ost"**	References the internal buffer of the object. You can set or retrieve the contents of the buffer through this object. It's also the default property, which means you can assign text to it directly. Note that using **StreamBuffer** keeps the contents of the buffer intact.
Target	**Set Ost.Target=document. all("myDiv")**	Sets the destination of the stream to a block element, such as a division, span, paragraph, or the like.
WriteBreak	**Ost.WriteBreak "This is the first line." Ost.WriteBreak "This is the next line."**	Places an HTML ** ** tag after the line.
WriteC	**Ost.WriteC "This is a te" Ost.WriteC "st"**	Writes an expression to the end of the string buffer, without inserting any spaces or special characters.
WriteLn	**Ost.WriteLn "This is the first line." Ost.WriteLn "This is the next line."**	Places a carriage return after the expression. This is useful for writing out lines of scripting code.
WriteQ	**Ost.WriteBreak "This is quoted text."**	Places quote marks around the expression. Note that you can also use double quotes ("") within any expression with this class to embed quotes into a string.
WriteS	**Ost.WriteS "This is a" Ost.WriteS "test"**	Writes an expression to the end of the buffer then places a space after it. This is useful for outputting large blocks of continuous text in HTML without having to worry about spaces between strings.
WriteSafeQ	**Ost.WriteSafeQ "This is safely quoted text"**	Places character-equivalent quotes around the expression (for example, "This is safely quoted text").
WriteTag	**Ost.WriteTag "H1","This is a header"**	Wraps opening and closing tags specified in the first parameter around the expression in the second parameter.

You can create a new output stream with the **new** keyword, and you can have more than one output stream defined at any given time. For example, the **WriteTable** handler defined at the beginning of this section can be rewritten, as shown in Listing 14.5.

Listing 14.5 An example of how to use two COStream objects.

```
Public Sub WriteTable(RS as RecordSet,ID as string)
    Dim tableOS as COStream
```

```
    Dim rowOS as COStream
    'Set the destination of the output to the
    'indicated block element
    'Create a table stream and a row stream, and clear them
    Set tableOS=new COStream
    Set rowOS=new COStream
    Set tableOS.Target=Document.all(ID)
    tableOS.Clear
    rowOS.Clear
    'Add header cells to the row stream
    RowOS.WriteTag "TH","State"
    RowOS.WriteTag "TH","City"
    RowOS.WriteTag "TH","Hi"
    RowOS.WriteTag "TH","Lo"
    RowOS.WriteTag "TH","Skies"
    RowOS.WriteTag "TH","Forecast"
    'Wrap the row within a row tag and add to the table
    TableOS.WriteTag "TR",RowOS.Flush
    'Move to the first entry in the database
    RS.MoveFirst
    While not RS.EOF
        'Clear the row stream and write data into the row
        RowOS.Clear
    RowOS.WriteTag "TD",RS("State")
        RowOS.WriteTag "TD", RS("City")
    RowOS.WriteTag "TD", RS("Hi")
        RowOS.WriteTag "TD", RS("Lo")
    RowOS.WriteTag "TD", RS("Skies")
    RowOS.WriteTag "TD", RS("Forecast")
    ' Wrap the current row stream in a new row in the table
        TableOS.WriteTag "TR",RowOS.Flush
    RS.MoveNext
    Wend
    ' Convert the table to a stream,
    ' and wrap a table header around it.
    TableOS.WriteTag "TABLE border='2'",TableOS.Flush
    ' Update the HTML page to display the table
    TableOS.Output
End Sub
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Manipulating Tables

A great deal can be done by modifying DHTML through the various text methods. However, you might be wondering whether this approach really takes advantage of Visual Basic's object language. After all, one of the central premises of the IE Document Object Model is that everything in the page is an object. If that's the case, shouldn't it be possible to work with a **Table** object, a **Row** object, or a **Cell** object instead of writing text out via the **innerHTML** property?

Tables and grids are critical components for any database application—the existence of so many different types of grid controls in the OCX vendor market is a testimony to that. The inclusion of the **Table** object in HTML makes creating and manipulating tabular data fairly easily, especially since a given table cell can itself display HTML code in a manner similar to frames or **DIV**s.

The Document Object Model exposed to Visual Basic provides a robust set of objects, methods, properties, and events for manipulating Dynamic HTML tables. Of these, the objects in Figure 14.5 are perhaps the most useful. The root of all of these is the **HTMLTable** class. An **HTMLTable** object represents a table in HTML, and you can control all the elements in a table, from the number of rows or columns to the contents, alignment, or size of any given cell.

It's worth noting that HTML tables are not simple row-by-column grids. A column can span more than one row, just as a row can span more than one column. Because of this, it's not possible to simply use a two-dimensional array to access elements, as much as that would make programming tables vastly simpler. However, there are actually a number of different ways that you *can* grab the contents of a given cell (that is to say, a single pane in a table).

Most people will probably be working with collections of records, so the simplest way of getting access to a cell is through the cell's row. The **rows** collection gives you access to the table one row at a time. Each row in turn is made up of a collection of **cells**. For

example, if you wish to retrieve the third cell in the fifth row from the table with ID **"myTable"**, you'd use the expression **myTable.rows(4).cells(2)** (remember that everything is zero-based here).

You can also query rows and cells in much the same way that you get information from any other HTML or ASP collection. For example, to determine the number of rows in the table, you'd use the expression **myTable.rows.length**, whereas the number of cells in the fifth row would be found using **myTable.rows(4).length**. It's worth noting here that the **length** property is read-only.

Getting to the actual content of a cell is simple—a cell supports most of the same interface properties that a block element like **<DIV>** does. To retrieve the text from row 5, cell 3, you'd use the expression **myTable.rows(4).cells(2).innerText**. If you wanted to set the contents to an HTML expression (for example, to load an image named Graphic.jpg into the cell), you'd write:

```
MyTable.rows(4).cells(2).innerHTML="<IMG SRC='Graphic.jpg'>"
```

You can similarly use other methods, including **insertAdjacentHTML** and **insertAdjacentText**, to add to the contents of cells. You can also use the selection object and its associated **textRange** object to perform more advanced functions (most of which are beyond the scope of this book).

Note that although you can use the same rows-and-cells system, most data access is strictly tabular—a table will always have the same number of cells in each row. Furthermore, most tables use the first row to display headers for each column. You can take advantage of this to write a general routine for accessing table elements by either position or header name, as shown in Listing 14.6. **GetCell** should be placed in a module along with your other general purpose routines.

Listing 14.6 The GetCell function should be placed in a general Visual Basic module.

```
Public Function GetCell(table As HTMLTable, _
   Row As Variant, Col As Variant) As HTMLTableCell
    Dim index As Integer
    If IsNumeric(Col) Then
        Set GetCell = tbl.rows(Row).cells(Col)
    Else
        For index = 0 To tbl.rows(0).cells.Length - 1
            If Col = tbl.rows(0).cells(index).innerText Then
                Set GetCell = tbl.rows(Row).cells(index)
                Exit Function
            End If
        Next
    End If
End Function
```

You can then use this function to retrieve items by heading as well as by position. For example,

```
Dim index as integer
For index=1 to myTable.rows.length-1
```

```
    Debug.print GetCells(myTable,index,"City").innerText
Next
```

will print all the cities in the table to the debug window. It should be noted that the **GetCell** function doesn't return the actual contents of the cell, only a reference to the cell object—you still need to use the **innerText** or **innerHTML** property to read or write the specific values for the cell in question.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Tabular Events

Dynamic HTML is, well, dynamic, and being able to manipulate the contents of a table can add considerably to how responsive your Web site appears. One advantage of working with Visual Basic Web classes is that you can capture events that take place on the Web page in a manner similar to the way that you can in a form. This is certainly obvious with graphical buttons (as demonstrated in the section “Image Handling” in Chapter 15), but the same concepts can be applied just as readily to cell elements.

A simple example may suffice to show what can be done with table events. The temperatures that are given in the database have units of degrees Fahrenheit. What would be useful, rather than providing a second column for the low and high temperatures in the table, would be a way to change the units into Celsius whenever the mouse enters the appropriate cell. Then, the cell contents could return to Fahrenheit when the mouse leaves the cell.

The actual implementation of this is not quite as straightforward as it could be, because the table is essentially generated on the fly (as will typically be the case with data tables). However, simply because a table doesn’t exist at the beginning of the session doesn’t mean that it can’t be modified at some later point after it is created.

It’s worth playing with the **WriteTable** subroutine discussed in the section “A Useful Buffer Class” earlier in this section. Originally, the routine was meant as a demonstration to show how the output stream class is used, but you can integrate it into a Web page easily enough, as shown in the following exercise.

Creating A Fahrenheit/Celsius Converter

Real estate on the screen is always at a premium, even with the scrolling capabilities intrinsic to browsers. The Fahrenheit/Celsius Converter is a good example of how to use DHTML to fold additional data into your output. It also provides several examples of how to access individual table elements:

1. Open the same project that contains the output stream class, and, in an external editor, create the HTML file shown here:

```
<HTML>
<HEAD>
<TITLE>Weather Display Page</TITLE>
</HEAD>
<BODY>
<H1>Weather Listed by City</H1>
<P>The following lists weather information by city.
To see a temperature in Celsius rather than Fahrenheit,
roll over that entry.</P>
<DIV ID=WeatherTableDiv> </DIV>
</BODY>
</HTML>
```

Save this file as WeatherDisplay.htm in your current work directory.

2. Click on the DHTML Page Designer Properties button on the editor toolbar, set the radio button option to Save HTML In External File, and click on Open: Use Existing HTML File (see Figure 14.6). Select the WeatherDisplay.htm file to use it as a template, and then click on OK. This will load a copy of the WeatherDisplay file.

3. Open the Project window, select the DHTMLPage1 Web designer, and right-click to display the pop-up menu. Choosing View Code will display the Visual Basic code for the designer. In the general declarations section, add the lines:

```
Dim WithEvents WeatherTable as HTMLTable
Dim WithEvents OldCell as HTMLTableCell
```

This will create a reference to a table object that can be set later to the table made here, as well as a reference that will be necessary later for updating the table on a rollover.

4. Create a Data Environment by right-clicking on **DHTMLProject** in the project window and selecting Add Data Environment. If you have not yet added the Weather database (found on the CD-ROM) to your ODBC databases, you can do so now. Click on the Connection tab, and connect the Data Environment to the Weather database. This new connection will be called Connection1.

5. Create the **WriteStateTable** function in DHTMLPage1, as shown in Listing 14.7.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Listing 14.7 WriteStateTable will place a table containing weather for any or all states into the document.

```
Public Function WriteStateTable(State As String, _
   TargetID As String, TableID As String) As HTMLTable
    Dim tableOS As COStream
    Dim rowOS As COStream
    Dim Conn As Connection
    Dim RS As Recordset

    ' Grab the defined connection from the
    ' data environment and open it
    Set Conn = DataEnvironment1.Connection1
    Conn.Open
    ' Set the record set to contain all of the
    ' listings from one state
    ' or, if the state entry is blank, from all states.
    If State = "" Then
        Set RS = Conn.Execute("SELECT * FROM Weather")
    Else
        Set RS = Conn.Execute("Select * FROM Weather " & _
           "where State='" + State + "';")
    End If
    'Create a table stream and a row stream, and clear them
    Set tableOS = New COStream
    Set rowOS = New COStream
    'Set the destination of the output to the
    'indicated block element
    Set tableOS.Target = Document.All(TargetID)
    tableOS.Clear
    rowOS.Clear
    'Add header cells to the row stream
    rowOS.WriteTag "TH", "State"
    rowOS.WriteTag "TH", "City"
    rowOS.WriteTag "TH", "Hi"
```

```
    rowOS.WriteTag "TH", "Lo"
    rowOS.WriteTag "TH", "Skies"
    rowOS.WriteTag "TH", "Forecast"
    'Wrap the row within a row tag and add to the table
    tableOS.WriteTag "TR", rowOS.Flush
    'Move to the first entry in the database
    RS.MoveFirst
    While Not RS.EOF
        'Clear the row stream and write data into the row
        rowOS.Clear
        rowOS.WriteTag "TD", RS("State")
        rowOS.WriteTag "TD", RS("City")
        rowOS.WriteTag "TD", CStr(RS("Hi")) + "&deg; F"
        rowOS.WriteTag "TD", CStr(RS("Lo")) + "&deg; F"
        rowOS.WriteTag "TD", RS("Skies")
        rowOS.WriteTag "TD", RS("Forecast")
    ' Wrap the current row stream in a new row in the table
        tableOS.WriteTag "TR", rowOS.Flush
        RS.MoveNext
    Wend
    ' Convert the table to a stream, and
    ' wrap a table header around it.
    tableOS.WriteTag "TABLE border='2' ID=" + _
       TableID, tableOS.Flush
    ' Update the HTML page to display the table
    tableOS.Output
    Set WriteStateTable = Document.All(TableID)
    Conn.Close
End Function
```

6. In the **BaseWindow_onload** event handler, place the following line:

```
   Set WeatherTable = WriteStateTable("", "WeatherTableDiv", _
      "WeatherTable")
```

This will call the **WriteStateTable** function to display all the states in the database, place the table into the **WeatherTableDiv** division, and name the new table WeatherTable.

7. If you run the program at this point, it should display all the data from the weather database in a table. Note that the **WriteStateTable** function sets the temperatures to degrees Fahrenheit (using the character equivalent **°** to produce the ° symbol). Stop the program.

8. So far, the table is relatively static. The next several steps will cause temperatures displayed in Fahrenheit to switch to Celsius whenever the mouse rolls over them. It also will echo the contents of the cell to another **<DIV>—WeatherStatus**. Within the DHTMLPage1 code editor, switch to **WeatherTable_onmousemove**, and insert the code shown in Listing 14.8 (the details of this section are covered in much greater depth later in this chapter).

Listing 14.8 The onmousemove event handles the detection and switching of temperature units, as well as a status indicator.

```
Private Sub WeatherTable_onmousemove()
    Static ct As Integer
    Dim obj As Variant
    Dim evt As IHTMLEventObj
```

```
    Dim cell As HTMLTableCell
    Dim row As HTMLTableRow

    Set evt = Document.parentWindow.event
    Set obj = Document.elementFromPoint(evt.x, evt.y)
    If TypeName(obj) = "HTMLTableCell" Then
        Set cell = obj
        Set row = cell.parentElement
        'WeatherTable.row
         WeatherStatus.innerHTML = CStr(cell.innerHTML)
         If Not (cell Is oldCell) Then
            If Not (oldCell Is Nothing) Then
                oldCell.innerHTML = _
                   ConvertToFahrenheit(oldCell.innerHTML)
                Set oldCell = Nothing
            End If
        End If
        If row.rowIndex > 0 Then
            If cell.cellIndex = 2 Or cell.cellIndex = 3 Then
                If Right(cell.innerHTML, 1) = "F" Then
                    cell.innerHTML = _
                       ConvertToCelsius(cell.innerHTML)
                End If
                Set oldCell = cell
            End If
        End If
    End If
End Sub
```

9. The preceding code will cause a cell that displays Celsius temperatures to revert to Fahrenheit when the mouse moves out of the cell, *provided that the mouse stays within the table*. When the mouse moves outside of the table, you need to catch this event with an **onmouseout** handler, as shown in the following code snippet:

```
   Private Sub WeatherTable_onmouseout()
       If Not (oldCell Is Nothing) Then
           oldCell.innerHTML = _
              ConvertToFahrenheit(oldCell.innerHTML)
           Set oldCell = Nothing
       End If
       Document.parentWindow.status = ""
   End Sub
```

10. Finally, you need to add the conversion routines **ConvertToFahrenheit** and **ConvertToCelsius**. These take as arguments the temperature in the format **“25° F”** or **“-5&deg: C”**, which is how the **onmousemove** function displays the temperature. These functions should be placed in a basic module, with the default **modDHTML** module the one most recommended. Listing 14.9 shows the code for the routines used to convert Celsius from Fahrenheit and vice versa.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Listing 14.9 Routines to convert to Celsius from Fahrenheit and vice versa.

```
Public Function ConvertToFahrenheit(Expression As String) _
   As String
    Dim pos As Integer
    Dim tempC As Single
    Dim tempF As Integer
    Dim tempStr As String

    pos = InStr(Expression, "&deg; C")
    ConvertToFahrenheit = "N/A"
    If pos > 0 Then
        tempC = CSng(Left(Expression, pos - 1))
        tempF = (tempC * 9 / 5) + 32
        ConvertToFahrenheit = CStr(tempF) + "&deg F"
    End If
End Function

Public Function ConvertToCelsius(Expression As String) _
   As String
    Dim pos As Integer
    Dim tempF As Single
    Dim tempC As Integer
    Dim tempStr As String

    pos = InStr(Expression, "&deg; F")
    ConvertToCelsius = "N/A"
    If pos > 0 Then
        tempF = CSng(Left(Expression, pos - 1))
        tempC = (tempF - 32) * 5 / 9
        ConvertToCelsius = CStr(tempC) + "&deg C"
```

```
    End If
End Function
```

11. Run the program again. This time, when you roll over a cell, the contents of the cell will be displayed above the table itself. However, what's even more interesting is that when you roll over a temperature in Fahrenheit, it will convert automatically to Celsius, and convert back when you move off the cell.

So, what exactly is going on here? This project provides a good example of event handling in general as well as how to mix the event handling mechanisms of Visual Basic and Internet Explorer, which at their root are not all that similar. The **WeatherTable_onmousemove** event especially is worth reexamining.

The **onmousemove** event is called whenever the mouse moves over the table, but it doesn't provide any explanation about what is beneath the mouse at the time. In order to do that, it's necessary to dip into the IE document model and use the **elementFromPoint** method of the document.

Unfortunately, you need a point. You can't get this from Visual Basic, and, unlike in Visual Basic, the **onmousemove** event doesn't pass an X,Y pair. Instead, you need to use an *event object*. If you've not done much JavaScript work (or haven't worked with Java), the idea of an event object might seem a little counterintuitive. However, the principle is fairly simple. IE keeps track of various events and passes the information about these events to the event object. For example, in a **mousemove**, the x and y position are passed to the event object.

The event object is a property of the document's window, but, for some reason, I had trouble using the Visual Basic's **BaseWindow** object to generate this particular class. As a consequence, the **evt** object is set using the **parentWindow** of the document object. This shouldn't make a lot of difference in the final outcome; however, I'm working on a beta, so it might behoove you to try experimenting with the **BaseWindow**:

```
Private Sub WeatherTable_onmousemove()
    Dim obj As Variant
        Dim evt As IHTMLEventObj
..
        Set evt = Document.parentWindow.event
        Set obj = Document.elementFromPoint(evt.x, evt.y)
        If TypeName(obj) = "HTMLTableCell" Then
           Set cell = obj
            Set row = cell.parentElement
            Document.parentWindow.status =
                    CStr(cell.innerText)
```

The problem with tables from a rollover standpoint is that not all the parts of a table are cells. In particular, the frames around cells are actually not even considered table elements by the **elementFromPoint** method. Because of this, the element from this method gets placed into a variant object call **obj**, and it's then tested to see if it is indeed a cell (has a type name of **HTMLTableCell**). If it is a cell, then the object is assigned to a variable called **cell** that has been cast to **HTMLTableCell**. Once you know the cell, you can also get the row to which that cell belongs by using the **parentElement** property (because a cell belongs to a row).

While on the subject of the **parentWindow** object, the status indicator at the bottom of the browser also belongs to the parent window. You can set this by using **document.parentWindow.status** and setting it to whatever text you want—in this case the **innerText** (not the **innerHTML**) of the cell contents. This can provide you with a secondary means of providing feedback, because its not always easy to tell at first glance which cell the mouse pointer is in.

There is no clean way of determining from a cell itself whether the contents contain a temperature or a forecast, so you need to place some bounds to make sure that you are within columns 2 or 3 (the columns that contain the temperatures). You should also check to ensure that you are not in the row containing the headings. A cell contains a **cellIndex**, which indicates which cell it is in a row, whereas a row has a **rowIndex** to determine its position in the collection of rows that makes up the table:

```
If row.rowIndex > 0 Then
    If cell.cellIndex = 2 Or cell.cellIndex = 3 Then
        If Right(cell.innerHTML, 1) = "F" Then
          cell.innerHTML = ConvertToCelcius(cell.innerHTML)
        End If
        Set oldCell = cell
    End If
End If
```

In addition to determining whether a cell is valid to convert to Celsius, the **onmousemove** routine needs to make sure that the last cell converted is returned back to Fahrenheit. This is done by means of the **oldCell** variable. Every time a cell is changed, the program assigns it **oldCell**. Before the cell is changed, the old cell is automatically reverted back to its former Fahrenheit condition, and then the variable is cleared:

```
If Not (cell Is oldCell) Then
    If Not (oldCell Is Nothing) Then
        oldCell.innerHTML =
           ConvertToFahrenheit(oldCell.innerHTML)
        Set oldCell = Nothing
    End If
End If
```

The **onmouseout** event does much the same thing for the special case when the mouse moves outside the table. It also clears the status indicator so that some other process can use it (**onmouseout** is called only once, while **onmousemove** is called repeatedly as the mouse moves within the table).

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Tables In Internet Explorer 5

Internet Explorer 5 will offer some interesting additions to the already robust table handling provided by its predecessor. For starters, it will be possible to add rows and cells using method calls rather than writing the tables via strings. This makes it a little easier to manipulate these objects from a language like Visual Basic. For example, the **MakeIE5WeatherTable** works in a similar method to the **WriteWeatherTable**, but it uses IE5 methods to manipulate the objects directly, as shown in Listing 14.10.

Listing 14.10 MakeIE5WeatherTable demonstrates how to use Internet Explorer 5 functions.

```
Public Function MakeIE5WeatherTable(State As String, _
   TargetID As String, TableID As String) As HTMLTable
    Dim WeatherTable As HTMLTable
    Dim Row As HTMLTableRow
    Dim Cell As HTMLTableCell
    Dim Conn As Connection
    Dim RS As Recordset
    Dim index As Integer

    ' Grab the defined connection from the data environment
    ' and open it
    Set Conn = DataEnvironment1.Connection1
    Conn.Open
    ' Set the record set to contain all of the listings
    ' from one state
    ' or, if the state entry is blank, from all states.
    If State = "" Then
        Set RS = Conn.Execute("SELECT State,City,Hi,Lo," & _
          "Skies,Forecast FROM Weather")
```

```
    Else
        Set RS = Conn.Execute("Select State,City,Hi,Lo," & _
          "Skies,Forecast FROM Weather WHERE State='" + _
           State + "';")
    End If
    Document.All(TargetID).innerHTML =
       "<table id=" + TableID + " border=2></table>"
    Set WeatherTable = Document.All(TableID)
    Set Row = WeatherTable.insertRow
    For index = 0 To RS.Fields.Count - 1
        Set Cell = Row.insertCell
        Cell.innerHTML = RS.Fields(index).Name
        Cell.Tagname = "TH"
    Next
    RS.MoveFirst
    While Not RS.EOF
        Set Row = WeatherTable.insertRow
        For index = 0 To RS.Fields.Count - 1
            Set Cell = Row.insertCell
            Cell.innerHTML = RS(index)
               If RS.Fields(index).Name = "Hi" Or _
               RS.Fields(index).Name = "Lo" Then
               Cell.insertAdjacentHTML "BeforeEnd", "&deg; F"
            End If
        Next
        RS.MoveNext
    Wend
    Set MakeIE5WeatherTable = WeatherTable
End Function
```

The **insertRow** and **insertCell** methods add a new row or cell respectively to their parent objects, and return a reference to the newly created items. This means that you can actually work with the items as they are created rather than having to wait for the table to be completed before you can touch cells or rows individually. You can also modify the name of a cell's tag, as is done with the statement **cell.tagname="TH"** to change the cell's style to a heading cell.

NOTE
At this stage, you still need to insert the table framework itself (the <**TABLE**></**TABLE**>) by hand, although this may change with the final release of the Internet Explorer 5 beta.

You can also make use of behaviors in Internet Explorer 5 to handle much of this manipulation. To a certain extent, the interaction with Visual Basic makes behaviors somewhat moot, although their performance is marginally better than DHTML Applications.

Although tabular data provides the most obvious form of output in a DHTML application, getting information from the user is just as obviously the domain of form elements.

Understanding Input

Text boxes and areas, buttons, checkboxes, and list boxes offer interface elements that differ little from their form-based counterparts. In most cases, using HTML input devices varies only a little from using VB form tools. Again, a working example can prove to be instructive. The weather table code in the last section lets someone see weather information, but it would be easy enough to change this into a weather editor. From here, you could choose a new state (or view all the states), edit a given entry, or add a new city or state to the database. The weather editor covered in this section illustrates how such an editor can be written, and gives some insight about why DHTML forms offer advantages over their forms-based counterparts. It also shows how effectively data access can be integrated into Web programming in a fairly transparent manner.

NOTE
Demonstration exercises of any sort should be viewed from a “How’s this done?” perspective. The example I’ve given here is meant to highlight specific techniques and elements, and I would not recommend using these algorithms as is for your own complex Web applications.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Making A Weather Editor

This is a book of client/server programming, rather than Web programming, so it's worthwhile to explore an application that can actually change data in addition to simply viewing it. A note of caution here—in attempting to use the ADO Data Environment, I repeatedly ran into problems getting the database to update, so the code here makes a Project reference to ADO directly. You may also want to try working with SQL commands explicitly and bypass the ADO functionality if it gives you problems (an example of this can be seen in Chapter 15, where entries are added into a calendar database).

1. In an external HTML editor, create the file WeatherEditor.htm with the code shown in Listing 14.11. Note that many WYSIWYG editors will set the name of **INPUT** fields with the **NAME** attribute—if so, you should add or change the attributes so that they use IDs instead.

Listing 14.11 WeatherEditor.htm template code.

```
<HTML>
<HEAD>
<TITLE>Weather Display Page</TITLE>
</HEAD>
<STYLE>
td {cursor:hand;}
</STYLE>
<BODY ID="">
<H1>Weather Editor</H1>
State <SELECT NAME="States" ID="States">
      <OPTION VALUE="" selected>All
      <OPTION VALUE="new">New
  State</OPTION>
</SELECT><BR>
City <SELECT ID="Cities">
```

```
     <OPTION VALUE="" selected></OPTION>
</SELECT><BR>
High
Temperature:<INPUT NAME="" SIZE="6" ID="Hi" >&deg; F<BR>
Low Temperature:<INPUT NAME="" SIZE="6" ID="Lo" >&deg; F<BR>
Skies:<SELECT ID="Skies">
      <OPTION VALUE="Clear" selected>Clear
      <OPTION VALUE="Hazy">Hazy
      <OPTION VALUE="Partly Cloudy">Partly Cloudy
      <OPTION VALUE="Cloudy">Cloudy
      <OPTION VALUE="Rain">Rain
      <OPTION VALUE="Fog">Fog
      <OPTION VALUE="Snow">Snow</OPTION>
</SELECT><BR>
<TEXTAREA NAME="Forecast" ID="Forecast"></TEXTAREA><BR>
<HR>
<P>The following lists weather information by city. To see
a temperature in Celsius rather than Fahrenheit, roll over
that entry.</P>
<DIV ID=WeatherStatus><INPUT ID=UpdateBtn NAME=UpdateBtn
STYLE="HEIGHT: 36px; LEFT: 200px; POSITION: absolute;
TOP: 188px; WIDTH: 108px" TYPE=button VALUE=Update></DIV>
<DIV ID=WeatherTableDiv></DIV>
</BODY>
</HTML>
```

2. Create a new DHTML Designer in your project file, and associate it with the WeatherEdit.htm file.

3. The first step in creating the editor is to fill the combo list box with a list of all the states in the database. **FillStates** does that by iterating through the database and adding a state only if it hasn't been added before. **FillStates** calls the function **FillCities** before it terminates. The method should go in the (General) section of the DHTMLPage designer, as shown in Listing 14.12.

Listing 14.12 FillStates populates the States combo box.

```
Public Sub FillStates()
    Dim conn As Connection
    Dim RS As Recordset
    Dim index As Integer
    Dim opt As HTMLOptionElement
    Dim statesCol As Dictionary
    Dim State As String
    Dim lbCtrl As HTMLSelectElement

    Set lbCtrl = Document.All("States")
    Set conn = DataEnvironment1.Connection1
    conn.open
    Set RS = conn.Execute("Select State FROM Weather")
    For index = 0 To lbCtrl.Length - 1
        lbCtrl.Remove 0
```

```
    Next
    Set opt = Document.createElement("OPTION")
    opt.Value = ""
    opt.Text = "(All)"
    opt.Selected = True
    lbCtrl.Add opt
    Set opt = Document.createElement("OPTION")
    RS.MoveFirst
    Set statesCol = New Dictionary
    While Not RS.EOF
        State = RS("State")
        If Not statesCol.Exists(State) Then
            statesCol.Add State, State
            Set opt = Document.createElement("OPTION")
            opt.Value = State
            opt.Text = State
            lbCtrl.Add opt
        End If
        RS.MoveNext
    Wend
    RS.Close
    conn.Close
    FillCities ""
End Sub
```

4. **FillCities** queries the database and populates the Cities combo box with all the cities for a given state after a state is selected from the States combo box (or it populates the Cities combo box with all the cities if no state is provided). Listing 14.13 shows the code for the **FillCities** routine.

Listing 14.13 The FillCities routine displays all the cities in a given state or all the cities in the database if no state is provided.

```
Public Sub FillCities(whichState As String)
    Dim conn As Connection
    Dim RS As Recordset
    Dim index As Integer
    Dim opt As HTMLOptionElement
    Dim cityCol As Dictionary
    Dim City As String
    Dim lbCtrl As HTMLSelectElement

    Set lbCtrl = Document.All("Cities")
    Set conn = DataEnvironment1.Connection1
    conn.open
    If whichState = "" Then
        Set RS = conn.Execute("Select City FROM Weather")
    Else
        Set RS = conn.Execute("Select City From Weather " & _
           "Where State='" + whichState + "';")
    End If
```

```
    For index = 0 To lbCtrl.Length - 1
        lbCtrl.Remove 0
    Next
    RS.MoveFirst
    Set cityCol = New Dictionary
    While Not RS.EOF
        City = RS("City")
        If Not cityCol.Exists(City) Then
            cityCol.Add City, City
            Set opt = Document.createElement("OPTION")
            opt.Value = City
            opt.Text = City
            lbCtrl.Add opt
        End If
        RS.MoveNext
    Wend
    RS.Close
    conn.Close
    lbCtrl.value=lblCtrl(1).value
End Sub
```

5. After a city is selected from the Cities list box, you need some way to populate the remainder of the **INPUT** controls with data. The **DisplayData** routine does this. Note that if a blank string is passed as a parameter, then a routine looks in the database until it finds the appropriate city, retrieves the state, and commences processing. For this reason, if you have two or more cities that have the same name (such as Springfield, which is one of the most common names in the United States), then you should have some method of differentiating between them. Listing 14.14 shows the code for the **DisplayData** routine.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Listing 14.14 The DisplayData routine puts the weather information for the given city into the INPUT fields.

```
Public Sub DisplayData(State As String, City As String)
    Dim conn As Connection
    Dim RS As Recordset
    Dim index As Integer
    Dim opt As HTMLOptionElement
    Dim cityCol As Dictionary
    Dim strSkies As String
    Dim lbCtrl As HTMLSelectElement

    Set lbCtrl = Document.All("Cities")
    Set conn = DataEnvironment1.Connection1
    conn.open
    If State = "" Then
        Set RS = conn.Execute("Select * FROM Weather " & _
           "WHERE City='" + City + "';")
        State = RS("State")
        RS.Close
    End If
    Set RS = conn.Execute("Select * FROM Weather " & _
      "WHERE State='" + State + "' AND City='" + _
      City + "';")
    Hi.innerHTML = CStr(RS("Hi"))
    Lo.innerHTML = CStr(RS("Lo"))
    Forecast.Value = RS("Forecast")
    strSkies = RS("Skies")
    For index = 0 To Skies.Length - 1
        If Skies(index).Value = strSkies Then
            Skies.selectedIndex = index
```

```
            Exit For
        End If
    Next
    RS.Close
    conn.Close
End Sub
```

6. You can change the contents of the database by changing one or more values in the **INPUT** controls, and then clicking on the update button. Note that while it is certainly possible to change individual entries, presenting an update button gives you a means to consolidate all your error checking and validation algorithms. The button simply calls the **UpdateCity** routine, which should, like the preceding routines, be put in the (General) section of the **DHTMLPage**. Listing 14.15 shows the code for the **UpdateCity** routine.

Listing 14.15 UpdateCity updates the database to reflect changes made, then redraws the table.

```
Public Sub UpdateCity()
    Dim State As String
    Dim City As String
    Dim conn As Connection
    Dim RS As Recordset

    'Get the current state and city from the popup boxes
    State = States.Value
    City = Cities.Value
    ' If no city is selected, nothing can be updated
    If City = "" Then
        ' Note Document.parentWindow.alert brings up an
        ' alert box in the Web page. Don't use MsgBox.
        Document.parentWindow.alert _
          "No city has been selected to update."
        Exit Sub
    End If
    ' Create a new connection to the Weather Database
    ' (Needs to be a system database).
    ' Note that DataEnvironments are read only with Access.
    Set conn = New Connection
    conn.open "Weather"
    ' If a city has been selected but not a state
    ' (such as when the (all) option is chosen for states,
    ' then get all cities from the list and
    ' retrieve the first.
    If State = "" Then
        Set RS = conn.Execute("Select * FROM Weather " & _
           "WHERE City='" + City + "';")
        ' Assign to variable State the state associated
        ' with the city
        State = RS("State")
        ' And close the connection
```

```
        ' (opened implicitly by Execute)
        RS.Close
    End If
    ' Create a new recordset
    Set RS = New Recordset
    ' Get city and state information
    RS.open "SELECT * FROM Weather WHERE State='" + State + _
       "' and City='" + City + "';", "Weather", _
       adOpenKeyset, adLockPessimistic
    ' Move to the first record
    RS.MoveFirst
    ' Update appropriate entries in the record
    RS("Hi") = Hi.Value
    RS("Lo") = Lo.Value
    RS("Skies") = Skies.Value
    RS("Forecast") = Forecast.Value
    ' Update the record
    RS.Update
    ' and close the recordset and connection
    RS.Close
    conn.Close
    ' Replace the displayed table with the one with new data
    ' Make sure you assign it to the WeatherTable object
    ' so that the onmouseover events continue to track.
    Set WeatherTable = WriteStateTable(State, _
      "WeatherTableDiv", "WeatherTable")
End Sub
```

Note that in the very last step, the Weather Table is redrawn (to reflect the changed data) and is set to the **WeatherTable** variable. This last step is very important, because you are in essence creating a new table in the HTML; even if it has the same name, the internal references that Internet Explorer uses don't recognize this as being the same object. By setting the new table to the variable, you circumvent this problem. This is crucial. If the reference is invalid, none of the rollovers or other routines that involve the table will work properly.

7. The **WriteStateTable** routine is essentially the same as that in the WeatherDisplay project, except that the table headers are generated from the SQL statement rather than explicitly listed (see Listing 14.16). Because the database does not store its temperatures with a °F appended, it makes more sense to explicitly list the actual data so that the correct units can be added.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Listing 14.16 The WriteStateTable routine is similar to that found in the WeatherDisplay project.

```
Public Function WriteStateTable(State As String, _
   TargetID As String, TableID As String) As HTMLTable
    Dim tableOS As COStream
    Dim rowOS As COStream
    Dim conn As Connection
    Dim RS As Recordset
    Dim Index as Integer

    ' Grab the defined connection from the data environment
    ' and open it
    Set conn = DataEnvironment1.Connection1
    conn.open
    ' Set the record set to contain all of the listings
    ' from one state
    ' or, if the state entry is blank, from all states.
    If State = "" Then
        Set RS = conn.Execute("SELECT State,City,Hi,Lo," & _
           "Skies,Forecast FROM Weather")
    Else
        Set RS = conn.Execute("Select State,City,Hi,Lo," & _
           "Skies,Forecast FROM Weather where State='" + _
           State + "';")
    End If
    'Create a table stream and a row stream, and clear them
    Set tableOS = New COStream
    Set rowOS = New COStream
    'Set the destination of the output to the
    'indicated block element
```

```
    Set tableOS.Target = Document.All(TargetID)
    tableOS.Clear
    rowOS.Clear
    'Add header cells to the row stream
    For Index=0 to RS.Fields.Count-1
        rowOS.WriteTag "TH",RS.Fields(index).Name
    Next
    'Wrap the row within a row tag and add to the table
    tableOS.WriteTag "TR", rowOS.Flush
    'Move to the first entry in the database
    RS.MoveFirst
    While Not RS.EOF
        'Clear the row stream and write data into the row
        rowOS.Clear
        rowOS.WriteTag "TD", RS("State")
        rowOS.WriteTag "TD", RS("City")
        rowOS.WriteTag "TD", CStr(RS("Hi")) + "&deg; F"
        rowOS.WriteTag "TD", CStr(RS("Lo")) + "&deg; F"
        rowOS.WriteTag "TD", RS("Skies")
        rowOS.WriteTag "TD", RS("Forecast")
    ' Wrap the current row stream in a new row in the table
        tableOS.WriteTag "TR", rowOS.Flush
        RS.MoveNext
    Wend
    ' Convert the table to a stream,
    ' and wrap a table header around it.
    tableOS.WriteTag "TABLE border='2' ID=" + _
       TableID, tableOS.Flush
    ' Update the HTML page to display the table
    tableOS.Output
    Set WriteStateTable = Document.All(TableID)
    conn.Close
End Function
```

8. In the **BaseWindow_onload** code window, you should call **FillStates** after the table is loaded to populate the combo boxes.

```
    Private Sub BaseWindow_onload()
        Set WeatherTable = WriteStateTable("", _
           "WeatherTableDiv", "WeatherTable")
        FillStates
    End Sub
```

9. When the user of your application selects a new state, this will automatically invoke the States combo box **onchange** event. This in turn rewrites the Weather Table and fills the Cities combo box with the cities for that state:

```
   Private Sub States_onchange()
       Dim row As HTMLTableRow
       Dim cell As HTMLTableCell
       Dim State As String
       Dim City As String
```

```
        State = States.Value
        Set WeatherTable = WriteStateTable(States.Value, _
           "WeatherTableDiv", "WeatherTable")
        FillCities (States.Value)
    End Sub
```

10. The **Cities_onchange** event in turn simply calls **DisplayData** to load the **INPUT** fields in the editor with the appropriate weather information. Note the use of **States.Value** and **Cities.Value** to retrieve the selected values of these combo boxes. This is similar to the way a combo box works in a standard Visual Basic form:

```
Private Sub Cities_onchange()
    DisplayData States.Value, Cities.Value
End Sub
```

11. The **UpdateBtn_onclick** handler simply invokes the **UpdateCity** routine to update the database and redisplay the weather table:

```
Private Function UpdateBtn_onclick() As Boolean
    UpdateCity
End Function
```

12. Finally, the **WeatherTable_onmousemove** and **WeatherTable_ onmouseout** handlers perform the Fahrenheit/Celsius conversion for rollovers. This routine is identical to that found in the Weather-Display DHTMLPage (see Listing 14.17).

Listing 14.17 The remaining event handlers are the same as for the WeatherDisplay section.

```
Private Sub WeatherTable_onmousemove()
    Static ct As Integer
    Dim obj As Variant
    Dim evt As IHTMLEventObj
    Dim cell As HTMLTableCell
    Dim row As HTMLTableRow

    Set evt = Document.parentWindow.event
    Set obj = Document.elementFromPoint(evt.x, evt.y)
    If TypeName(obj) = "HTMLTableCell" Then
        Set cell = obj
        Set row = cell.parentElement
        Document.parentWindow.Status = CStr(cell.innerText)
        If Not (cell Is oldCell) Then
            If Not (oldCell Is Nothing) Then
                oldCell.innerHTML =
                  ConvertToFahrenheit(oldCell.innerHTML)
                Set oldCell = Nothing
            End If
        End If
        If row.rowIndex > 0 Then
            If cell.cellIndex = 2 Or cell.cellIndex = 3 Then
                If Right(cell.innerHTML, 1) = "F" Then
                    cell.innerHTML =
```

```
                    ConvertToCelsius(cell.innerHTML)
                End If
                Set oldCell = cell
            End If
        End If
    End If
End Sub

Private Sub WeatherTable_onmouseout()
    If Not (oldCell Is Nothing) Then
        oldCell.innerHTML =
          ConvertToFahrenheit(oldCell.innerHTML)
        Set oldCell = Nothing
    End If
    Document.parentWindow.Status = ""
End Sub
```

13. Run the project. Visual Basic will display a dialog box querying which DHTMLPage to run. In this case, you should run the second.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Playing With Controls The HTML Way

Working within the HTML environment is a little different than dealing with forms. The elements that HTML supports are in some respects more primitive than their counterparts in VB forms. For example, to get the functionality of a typical VB text box, you actually need to use both a text box and a text area box, although at the same time the selection box in HTML can serve double duty as a list box or a combo box in Visual Basic.

Understanding the distinction between HTML controls and VB controls ahead of time will help you when it comes time to implement these controls in your DHTML page. The differences are fairly general:

- Forms make use of the property name to identify elements, while Internet Explorer uses the property ID to do the same thing (see the sidebar “Of Names And Values”).
- The events for forms usually pass some form of parameter to the handler (for example, a **mousemove** handler will pass the cursor position and buttons states). With an HTML control, no parameters get passed, and, instead, it needs to be retrieved through the event object.
- With both standard forms and HTML pages, the value of a control can be retrieved through the **Value** property.
- Properties in forms can be set in design time through the use of the property sheet (set with F4). In a DHTML Web page, the same properties are set via attribute pairs (for example, **SRC=“myPage.htm”** in an **<IMG>** tag). However, with the element selected in the editor, pressing F4 will bring up a property sheet for the item as well. Notice that because HTML objects only technically exist at runtime, there is little-to-no error checking that goes on when you edit an HTML property.

NOTE
To see the property page for a given DHTML element, select the object in the DHTML template view (the right pane of the DHTML Editor), and press F4. Setting the ID to something besides an empty string (the default) will make that object "live" in VB, while setting it back to the empty string will remove it from the event queue.

- Variables from IE4 are implicitly variants, unless explicitly cast to another data type in Visual Basic. This means that you need to be careful with parameter types in Visual Basic routines that handle DHTML.
- In DHTML, you can only place a control to a predefined location if the style property of that control is set for **"position:absolute"** or **"position:relative"**. Because of the wrapping capability that HTML provides, you should avoid absolutely positioning elements whenever possible so that the browser can resize the page automatically during a resize (or at worst, the control should be defined with a relative offset from a containing element so that when the site itself moves, the control also moves relative to the container.
- Finally, HTML input elements can work independently of an HTML **<FORM>** object. The primary purpose of a **<FORM>** object is to wrap the **NAME:VALUE** pairs of elements together into a set of header data for transmission to the server. Because of this, Visual Basic doesn't really have anything analogous to an HTML form object, and the Submit and Clear buttons don't have VB counterparts. However, it you want to submit HTML form data to a server and still manipulate form elements in Visual Basic, you should make sure that these elements include *both* a **NAME** and an **ID** attribute.

A list of HTML form objects and their Visual Basic form counterparts is given in Table 14.6. The more advanced controls found in VB, such as a TreeView control, can be emulated in Internet Explorer with DHTML or, in some circumstances, can be added directly as ActiveX components. Both for ease-of-use and downloading, DHTML-based controls or scriptlets are preferable to rolling your own ActiveX controls, especially because Internet Explorer contains a number of built-in controls that can be modified to do nearly anything that can be done with many ActiveX components (see Chapter 15 for details).

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Table 14.6 Visual Basic 6 standard objects and their Internet Explorer equivalents.

Visual Basic Form Control	Internet Explorer Control	Comments
Checkbox	**<INPUT TYPE=CHECKBOX>**	The methods and properties of checkboxes don't vary dramatically from VB to IE. To set a checkbox initially in Internet Explorer, use the **SELECTED** attribute, but you can set the **Value** of a checkbox to true or false to set or reset the checkbox.
Command Button	**<INPUT TYPE=BUTTON>**	The button object is supported by IE4/5 only. You should use this rather than a **<SUBMIT>** or **<RESET>** button unless you also plan to submit an HTML form containing the button to the server.
Frame	**<FIELDSET><LEGEND>**	**<FIELDSET>** is exclusive to Internet Explorer, and it's meant to duplicate the functionality of the Frame box used to group buttons or checkboxes. The **<LEGEND>** container provides the label for the fieldset.

Grid	**<TABLE>**	This is covered in detail in the section “Building Tables” in this chapter.
Image	**<IMG>**	The image control corresponds loosely to the **<IMG>** control—both can display images but are not containers. (See the “Image Handling” section in Chapter 15, for more information).
Label	**<DIV>**	Of course, the **<DIV>** is a considerably more versatile element, because it can essentially hold any HTML construction, anywhere. In general, whenever you have complex functionality within a region, look first to the **<DIV>** as your primary display element.
List Box, Combo Box	**<SELECT><OPTION>**	To duplicate a list box in IE, set the **SIZE** attribute of a **<SELECT>** item to a value greater than 1. Setting **SIZE** to 1 will make the box act like a drop-down combo box instead. See the “Reviewing Your Options” section to get more information about working with **<SELECT>** controls.
Option Button	**<INPUT TYPE=OPTION>**	Option buttons work in Internet Explorer, but their syntax differs dramatically from Option Buttons in Visual Basic. (See the “Reviewing Your Options” section for more information about working with them.)
Picture	**<DIV>**	This association may not make much sense at first, until you realize that the **PICTURE** element is actually a Container class in Visual Basic.

Scrollbars	Built into containers	There are no separate scrollbar controls in Internet Explorer, although you can use a **<DIV>** that's just wide enough to display a scrollbar, with **style="overflow:scroll"**.
Text Box (Multiline)	**<TEXTAREA>**	IE differentiates between a single line and a multiline text field. The **<TEXTAREA>** control has some significant limitations, although in IE4/5 it can be used to display HTML-formatted text.
Text Box (Single Line)	**<INPUT TYPE=TEXT>**	A single line text box in VB has most of the same properties as the text field in IE.
Timer	JavaScript timer functions (**setTimeout**, **clearTimeout**, **setInterval**, **clearInterval**)	One of the advantages that JavaScript has over Visual Basic is the support of a timer— in VB, you need to create a timer control, which makes it difficult to create standalone classes that have a periodic component.
TreeView	No equivalent	TreeView controls are enormously useful, but they're also extremely complex. While there are no native TreeView controls, you can make a collapsible hierarchical list with a **<DIV>** (shown in the next chapter).

Visual Basic 6 lets you reference HTML objects in a way that is consistent with any other object in VB. If you set the ID of the element to a specific value when you load a template (or set the **ID** property in the element's property page), then that object will appear within the list of event-receiving objects in the code window for the DHTML page.

Of Names And Values

One of the great joys of dealing with HTML (and I'm speaking sarcastically here) has to do with the issue of names. The matter, put simply, is that when Netscape introduced JavaScript, it followed the convention of using *NAME* as a way of identifying components. While this ran counter to the SGML concept on which HTML is based (SGML uses *ID* to identify its elements), HTML was evolving so fast at the time that no one really thought that it would be an issue.

Microsoft, for unknown reasons, chose to designate its elements in HTML with the term *ID*, possibly to make their version of HTML more compliant with SGML. Ironically, this runs counter to the usage in Visual Basic, where *Name* is used as a handle for components. With the advent of XML, which like SGML uses IDs to designate its elements, this strategy looks like it might prove to be a sound one, but it means that you have to be careful when using third-party HTML generation tools. They all too frequently will generate HTML form data that uses NAMEs rather than IDs, which can prove a problem.

To state it clearly, you can add any form element to a Web page and have it treated as an element in your DHTML application if you give that element an ID. This has a lot of useful implications, because, by making the element "live" in Visual Basic, you can script it without having to load it and assign it. You can see this in more detail in the exercise "Making A Weather Editor."

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Previous | Table of Contents | Next

Reviewing Your Options

Although most controls in the hybrid VB6/DHTML environment are straightforward—their values can be retrieved or set with the **Value** property—certain controls are more complicated. Two controls that can cause real headaches to DHTML application developers are without a doubt the Select control and the Option Button controls.

The Select control does double duty as a list box control and a combo box. In its simplest form, the <**SELECT**> object in DHTML acts as a list box displaying all of the elements entered into it in the form of **OPTION**s. For example, the following bit of HTML would show four states in a list:

```
<SELECT ID="States">
<OPTION SELECTED VALUE="washington">Washington
<OPTION VALUE="california">California
<OPTION VALUE="oregon">Oregon
<OPTION VALUE="newMexico">New Mexico
</SELECT>
```

The **SELECTED** property in the first <**OPTION**> tag indicates that this is the element that is selected in the list, and it is highlighted within the list box. It's worth noting that what gets displayed in the option box is the text that follows the <**OPTION**> tag, rather than the contents of the **VALUE** attribute.

To turn this into a combo box, all you would need to do is set the **SIZE** attribute of the <**SELECT**> tag to **1**:

```
<SELECT ID="States" SIZE=1>
<OPTION SELECTED VALUE="washington">Washington
<OPTION VALUE="california">California
<OPTION VALUE="oregon">Oregon
```

```
<OPTION VALUE="newMexico">New Mexico
</SELECT>
```

Programmatically, you can access each element in the selection box through the **HTMLSelectElement** collection, which is the interface property associated with selection boxes. If the selection box had an **ID** of States, then the code in Listing 14.18 would display each entry in the list with its corresponding value.

Listing 14.18 Using the HTMLSelectElement you can get the number of items (through the length property) and index each option element.

```
Dim index as Integer
Dim lblStates as HTMLSelectElement
Set lblStates=document.all("States")
' You could also just use the variable States itself,
' since including the ID makes
' the States control visible to Visual Basic.
For Index=0 to lblStates.length-1
    Debug.Print lblStates(Index).Text+":"+ _
      lblStates(Index).Value
Next Index
```

Each indexed element of an **HTMLSelectElement** collection has a data type **HTMLOptionElement**. You can modify the preceding code fragment to work with individual option elements rather than a full index, as shown in Listing 14.19.

Listing 14.19 The same routine, except the option elements are explicitly referenced as HTMLOptionElements.

```
Dim index as Integer
Dim lblStates as HTMLSelectElement
Dim Opt as HTMLOptionElement
Set lblStates=document.all("States")
' You could also just use the variable States itself,
' since including the ID makes
' the States control visible to Visual Basic.
For Index=0 to lblStates.length-1
    Set Opt=lblStates(Index)
    Debug.Print Opt.Text+":"+Opt.Value
Next Index
```

For the most part, this doesn't differ dramatically from list boxes or combo boxes in Visual Basic. However, things get a little more complicated when you want to add an element to an already defined list box. The **HTMLSelectElement** supports a default collection called *options*, which contains the elements in question (that is, **lblStates(Index)** is equivalent to **lblStates.Options(Index)**). In order to add an option to the options collection, you use the *add* method. However, there is a bit of a problem here. For **HTMLSelecteElement** options, the **add** command can only add pre-existing **Options** objects, it doesn't create them from scratch.

A little searching reveals that, in the IE object model, you can create a new option by using the **document.createElement()** method (this same method is also used to create **AREA**s for image maps). For example, to add a new state to the list, you'd use the following code:

```
Set Opt=document.createElement("OPTION")
Opt.Text="Alaska"
Opt.Value="alaska"
LblStates.Add Opt
```

This method was essentially used to populate the Cities and States combo boxes in the Weather Editor demonstration, as shown in Listing 14.20.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Listing 14.20 The FillStates routine from the Weather Editor makes extensive use of <SELECT> and <OPTION> elements.

```
Public Sub FillStates()
    Dim conn As Connection
    Dim RS As Recordset
    Dim index As Integer
    Dim opt As HTMLOptionElement
    Dim statesCol As Dictionary
    Dim State As String
    Dim lbCtrl As HTMLSelectElement

    Set lbCtrl = Document.All("States")
    Set conn = DataEnvironment1.Connection1
    conn.open
    Set RS = conn.Execute("Select State FROM Weather")
    For index = 0 To lbCtrl.Length - 1
        lbCtrl.Remove 0
    Next
    Set opt = Document.createElement("OPTION")
    opt.Value = ""
    opt.Text = "(All)"
    opt.Selected = True
    lbCtrl.Add opt
    Set opt = Document.createElement("OPTION")
    RS.MoveFirst
    Set statesCol = New Dictionary
    While Not RS.EOF
        State = RS("State")
        If Not statesCol.Exists(State) Then
            statesCol.Add State, State
```

```
            Set opt = Document.createElement("OPTION")
            opt.Value = State
            opt.Text = State
            lbCtrl.Add opt
        End If
        RS.MoveNext
    Wend
    RS.Close
    conn.Close
    FillCities ""
End Sub
```

In the **FillStates** subroutine, the **SELECT** box was first emptied using the **Remove** method by counting the number of items in the list, and then removing the first element that many times. This ensured that the combo box would not contain any spurious information from previous sessions. Then, the routine iterated through all the records and extracted the name of the state from each record, comparing it against a dictionary collection. If the state wasn't yet in the dictionary, it was added both to the collection and the list, otherwise it was skipped. Much the same thing happens in the **FillCities** subroutine, save that all cities for a given state are added.

The Other Options

Options occur in two places in HTML—the <**OPTION**> tags associated with the <**SELECT**> object and as an input type <**INPUT TYPE=OPTION**>. The latter elements are actually radio buttons, not list elements. Their implementation in Visual Basic 6 presents some frustration for anyone who has used them in JavaScript. With JavaScript or Jscript, the best way of working with a collection of radio buttons is to set all of the **ID**s to the same name. JavaScript implicitly converts collections that have more than one element but are named the same into arrays. Thus, if you had the following HTML code

```
<INPUT TYPE="OPTION" ID="Grp" NAME="Grp"
   VALUE="Eastern">Eastern States
<INPUT TYPE="OPTION" ID="Grp" NAME="Grp"
   VALUE="Midwest">MidWest States
<INPUT TYPE="OPTION" ID="Grp" NAME="Grp"
   VALUE="Western">Western States
```

then the expression **document.Grp[2].Value** would have the value **Western**. For some reason, though, the developers of the DHTML class template decided that no element could have a common **ID**, so when you import an HTML template, such as the preceding HTML code, VB6 will convert the **ID**s into **"Grp"**, **"Grp1"**, **"Grp2"**, and so forth. This makes coding them considerably more complicated and, as a consequence, makes radio buttons less attractive in building DHTML applications.

Where To Go From Here

As mentioned earlier, the Internet Explorer client is a remarkably complex beast to program. Like any good multimedia engine (which is what the browser essentially is), what appears simple on the surface often hides all types of considerations beneath it—certainly, however, the client aspects are made more powerful and richer as a consequence.

Text manipulation, tables, and forms make up three legs of the Web application. However, without the last leg—graphics—Web pages are not terribly appealing. A good grasp of image methods and properties can turn an otherwise bland client/server Web page into visual dynamite. In Chapter 15, we continue to explore aspects of Internet client-side development, including working with images and complex data types, style sheets, and integrating ActiveX components.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Chapter 15
Power Tools

Key Topics:

- Image handling
- Style sheets programming
- Dialog development
- ActiveX components
- A look at the future

Last chapter! This is always the fun one for an author—the one in which he or she gets to deal with things to come, advanced concepts, and cool tips and techniques. In this chapter, I expand on DHTML programming to cover more advanced topics, such as working with images, manipulating style sheets, creating dialog boxes, and dealing with third-party components. Although none of these are strictly necessary for building client/server applications, they dramatically enhance the interactivity of the client and extend the functionality of your applications beyond the basic text-entry stage.

Image Handling

You may have noticed in the last two chapters that the Web pages that were discussed were, well, dull. They accomplished the tasks at hand but lacked a lot in the way of aesthetic appeal. Yet one of the strengths of the Web browser is the ease with which it integrates graphics—you can have a graphical background (tiled or untiled), inline graphical images, floating pop-up pictures, and animated icons. Indeed, without images, Web pages lose much of their impact (if you've been around the Web long

enough, you may even remember text-only browsers, such as Lynx, which were functional but not particularly inspiring).

This section deals with basic image manipulation methods. If you are familiar with Dynamic HTML or JavaScript, many of these methods may seem like old friends; although, like everything else dealing with Visual Basic, even these old friends can look different under the guise of a typed programming language like VB.

Deceptive Debugging Dialogs

There is a fairly serious design flaw in the implementation of DHTML applications. When you first run your applications, they don't start where you might expect them to. On my system, for example, pressing the Play button on the toolbar launched a perfectly functioning application—in my Windows/Temp folder. In the exercises of Chapter 14, this didn't really matter that much, since the whole page was self-contained. However, if you want to include any external resources (such as images, video, sound, or resource files), then every reference that you create has to be absolute: you have to include the protocol and server location of your resource, rather than just providing a relative path. For example, if you want a background graphic named bg1.jpg for your Web page, the only way (apparently) that you can get the browser to find it is to reference its absolute path

```
<BODY BACKGROUND="http://www.myserver.com/images/bg1.jpg">
```

instead of:

```
<BODY BACKGROUND="images/bg1.jpg">
```

Modifying The Debug Mode

As it's likely that you won't be deploying your Web pages in the same folder that you develop the page (and it's almost certain that you won't be deploying your site in the Windows/Temp folder), this limitation appears fairly profound. However, it turns out that you can get your Web page to play in the correct folder by changing the Debug mode.

1. Before you begin, back up your template DHTML files, since changing the Debug option will cause Visual Basic to modify your templates for its own needs.

2. From the Project menu, select the bottom entry. This contains the Project Properties, although the specific text of the menu item changes depending upon what you named your Project. Selecting the Properties menu will bring up the Project Properties dialog box.

3. Select the Debugging tab, and choose Wait For Components To Be Created. At least in the beta, the default Start Component was set to DHTMLPage1 (see Figure 15.1). Then, press OK.

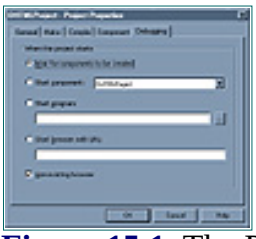

Figure 15.1 The Project Properties dialog box Debugging tab.

4. When you run the project, Visual Basic won't automatically create a new DHTML page. You will need to manually open a browser to the Web page that you loaded. Once the page loads, however, all of your relative links should point to the correct images or resources.

Making A Button

The majority of graphics that end up on Web pages (unless it's a gallery of some sort) will almost certainly be buttons. Over the years, graphical buttons have evolved into three-state creations: base, highlighted, and pressed.

A button has a base state that is typically consistent with the background—it doesn't really stand out, other than having enough of a graphical design to indicate its functionality as a button. When the mouse moves over a button, many graphical designers have the button highlight in some fashion. This highlight state can consist of having the button become lighter, move, or even animate (it's not uncommon to use animated GIFs). Finally, a button has a pressed state, in which the button is shown recessed, moved, or otherwise pushed behind the plane of action of the page (see Figure 15.2).

Figure 15.2 Three different button stages.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Creating A VB Button

You can create a button using the DHTML Application Wizard, as outlined here. You'll also want to read the "Combining VB And Scripting" section later in this chapter.

1. Create a new folder in your work directory called Images, and place the three graphical buttons Enter.jpg, EnterHi.jpg, and Enter-Pressed.jpg into it (these files are stored on this book's CD-ROM, or you can make your own with the same names). In the root directory, create a new Web page called ButtonTest.htm and add the code shown in Listing 15.1.

Listing 15.1 HTML text for ButtonTest.htm.

```
<HTML>
<HEAD>
<TITLE>Button</TITLE>
</HEAD>
<BODY
    BACKGROUND="images/islands.jpg"
    BGCOLOR="#ff8000"
    TEXT="white"
    LINK="yellow"
    VLINK="red"
    ALINK="white"
    >
<IMG
    ID=Enter
    SRC="images/enter.jpg"
    WIDTH=100
    HEIGHT=40
    ALT="Enter"
    BORDER="0"
```

```
    HSPACE=0
    >
</BODY>
</HTML>
```

2. Create a new DHTML application project in VB. Open the DHTML Editor, and choose Open|Use Existing HTML File in the DHTML Page Properties dialog, and open ButtonTest.htm.

3. Ensure that your debug mode is set to the first option (Wait For Components To Be Created) as outlined in "Modifying The Debug Mode," presented earlier in this chapter.

4. Open the code window for the DHTML page, and select the Enter object. This is the image. Choose the method **Enter_onmouseover**, and enter the following code:

```
private Function Enter_onmouseover
    Enter.src="images/EnterHi.jpg"
End Function
```

5. Choose the **onmousedown** event, and add the following code:

```
private Function Enter_onmousedown
    Enter.src="images/EnterPressed.jpg"
End Function
```

6. In the **onmouseup** event, notice that the graphic is set to the highlight state, not the base state. This is because the mouse is still over the button. Add the following code to support the Enter button's **onmouseup** event:

```
private Function Enter_onmouseup
    Enter.src="images/EnterHi.jpg"
End Function
```

7. The **onmouseout** event is where the button is restored to its initial state. Add the following code to support the Enter button's **onmouseout** event:

```
private Function Enter_onmouseout
    Enter.src="images/Enter.jpg"
End Function
```

TIP
Staying Alert
Notice that an alert box is called, rather than a **msgbox**. Calling a message box would create an alert *behind* the Web page rather than in front of it, and it would halt all processing of the Web page until the reader figures out that there is something there (probably by closing the Web page window). Also, notice that the alert is generated by the document **parentWindow** object. At least in the beta, the **baseWindow** object usually threw an exception whenever this was attempted, so it is necessary to get a reference to the window belonging to the document.

---

8. Finally, the **onclick** event handles the actual action of the button. For a number of reasons, it is usually best to separate the mechanics of the button press (**mouseover**, **mousedown**, **mouseup**, and **mouseout**) with the trigger. By keeping the code distinct, you can offload some of the mechanics to other environments (as shown in the "Combining VB And Scripting" section, later in this chapter). It also makes it

easier to debug the code when you can see functionality clearly delineated. Add the following code to support the Enter button's **onclick** event:

```
private Function Enter_onclick
  document.parentWindow.alert "You just clicked on a _
     button!"
End Function
```

9. Run the application, then open Internet Explorer and display the Web page that you initially loaded.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The image object in DHTML is represented in Visual Basic by **HTMLImg** (note the alternate spelling here). This object is remarkably robust (see Table 15.1 for a partial list). Microsoft currently uses it not only for images themselves (in either JPG, GIF, PNG, or BMP formats), but also as another way of displaying AVIs and VRML (Virtual Reality Markup Language) worlds.

Table 15.1 Image object properties and methods.

Property Or Method	Example	Description
align	**Img.align="Left"**	Sets or retrieves the current alignment of inline images. Can also be set via the **text-align** property in CSS.
alt	**Img.alt="This is a button"**	Sets the alternate text displayed when the mouse rolls over a graphic. The **alt** attribute also appears while graphics are loading.
border	**Img.border=2**	Sets the width of the border around a graphic when that graphic is used as the hotspot for a link. Setting **border=0** turns the border off.

className	**Img.className="Hilite"**	Sets the CSS style rule for the image. For more information about styles and classes, see "Doing It With Style" later in this chapter.
dataFld	**Img.dataFld="DataImage"**	Sets the default field value of an image (used with Remote Data Services). With Visual Basic, it is better to use ADO directly.
dataSrc	**Img.dataSrc="WeatherDB"**	Sets the data source for **dataFld** (used with Remote Data Services). With Visual Basic, it is better to use ADO directly.
dynsrc	**Img.dynsrc="WeatherMap.avi"**	Sets the image to an AVI or similar digital movie. NetShow is recommended instead.
height	**Img.height=480**	Sets the height of the image in pixels. Original image is scaled to match.
hspace	**Img.hspace=10**	Sets the number of pixels from the horizontal edge of a graphic to the text.
Id	**Img.Id="myImage"**	Makes an image available as an object in Visual Basic.
isMap	**Img.isMap=True**	Determines whether an image is a server-side image map (**isMap=True**).

loop	**Img.loop=3**	Indicates the number of times a **dynsrc** movie, GIF animation, or other animated resource plays. Setting **loop** to **0** stops it, and setting it to **-1** causes it to repeat indefinitely.
lowsrc	**Img.lowsrc="images/thumbs/**	Loads and displays a low-resolution **mypic.jpg"**version of the image. If **src** is also specified, then the **src** image will eventually replace the **lowsrc** image.
offsetHeight	**Height=Img.offsetHeight**	Returns the total height of an image, including that not necessarily visible save through scrolling (read-only).
offsetWidth	**Width=Img.offsetWidth**	Returns the total width of an image, including that not necessarily visible save through scrolling (read-only).
offsetTop	**Top=Img.offsetTop**	Returns the position of the top of the image relative to its containing object (in pixels).
offsetLeft	**Left=Img.offsetLeft**	Returns the position of the left of the image relative to its containing object (in pixels).
parentElement	**Var ctnr=Img.parentElement**	Returns a reference to the object that contains the image.

readyState	**If img.readyState=4 then AssignResource**	Indicates the loading status. **ready-State=3** indicates the image is in a loading state, while **readyState=4** indicates that it has completed loading.
scrollHeight	**Sh=img.scrollHeight**	Specifies the total visible height of the picture that can be seen without scrolling.
scrollLeft	**Sl=img.scrollLeft**	Specifies the distance in pixels between the left edge of the image and the left edge of the container.
scrollTop	**Img.scrollTop=Img.scrollTop+20**	Specifies the distance in pixels between the top edge of the image and the top edge of the container.
scrollWidth	**Sw=Img.scrollWidth**	ScrollWidth is the total visible width of the picture that can be seen without scrolling.
Src	**Img.Src="images/newPicture"**	Returns the URL of the image when used as a read property. As a write property, used to set a new URL, which will cause the picture to change to a new one.
start	**Img.start**	Begins an animation if the **Loop** property had been set initially to **0**.
style	**Img.style="margin:4px;"**	Sets some aspect of the **style** property, the entry point for CSS. See "Doing It With Style" later in this chapter.

useMap	**Img.useMap="#MyMap"**	Provides the local URL for the **MAP** object as a client-side image map.
vspace	**Img.vspace=10**	Sets the number of pixels minimum distance between the top or bottom of an image and any flowing text.
width	**Img.width=400**	Sets the width in pixels of the image, stretching it, if needed.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The attributes of most importance when dealing with images are probably the **src**, **height**, and **width** properties. The **src** property points to a URL, either local or complete, that contains an image. The image itself can be in JPG, GIF, PNG, or BMP formats (although the last one is not recommended). For example, the following bit of code will load two images, one from a local images folder contained in the same directory as the HTML movie, the other from another server:

```
Picture1.src="images/myPicture/gif"
Picture2.src="http://www.microsoft.com/images/BillG.jpg"
```

You can also set the **<IMG>** tag's **dynSrc** property to a URL hosting an AVI or VRML file. This was the primary way of displaying AVIs in Internet Explorer 3. In later versions of Internet Explorer, the NetShow control has pretty much superceded the **dynSrc** property, because it gives finer control to Web designers. Likewise, while you can display VRML movies using the **<IMG>**tag, you are better off using either a Java or ActiveX control-based VRML component for all but the simplest 3D worlds. The **<IMG>** control doesn't have the hooks necessary to really manipulate these worlds from JavaScript.

NOTE
There is some talk in the W3C (the group that administers Web formats) of eventually folding the **<IMG>** tag itself into the **<OBJECT>** tag, but given the current usage of **<IMG>** on the Web, that's likely to happen later than sooner.

Combining VB And Scripting

You may have noticed, especially if you have a slower machine, that your Web application wasn't exactly responsive. Indeed, on both a 150 MHz machine and a 400 MHz Pentium Pro, the speed at which this application worked was rather disappointing. However, a moment's thought will indicate why.

When you roll over a graphic, the system checks to see if there is an event attached. In this particular case, the DHTML Web class slips in its own event drivers, so the event gets passed up to the DLL. The DLL calls the associated handler, grabs a reference to the image object in the IE shell, associates a new source URL to the graphic, releases the image back to IE, and closes out. Even as an in-process server, the DLL is not likely to be anywhere near as fast as working native in the browser.

There's another problem here too. The way a button works is pretty consistent within an application—it is only the resulting action that changes from one button to the next. Yet, if you had 10 buttons on your page, you'd need to fill 50 different event handlers. It would be far better if you could have one set of routines that would handle the mechanics of the buttons (which only requires changing the names of the button graphics) and then 10 handlers for the resulting actions of the 10 buttons.

Writing A JavaScript Button Handler

Just as there are times that you need to extend Visual Basic's capabilities with the judicious use of C++, so

there are times (especially with Web work) where JavaScript provides a better solution to problems than Visual Basic. The language's lightweight features and portability (both Internet Explorer and Netscape Navigator support it) make it the de facto scripting language for Web page development, and it incorporates its own rather quirky brand of object-oriented design. In the following exercise, the various mouse events (**onmouseover**, **onmousedown**, and so on) are intercepted in JavaScript and assigned actions. Because this scripting is contained in-process to Internet Explorer, it actually runs slightly more efficiently than the corresponding Visual Basic would.

1. In a text editor, use the code in Listing 15.2 to create a file in your working directory, and name the file ButtonScripts.js.

Listing 15.2 Button manipulation routine (ButtonScripts.js).

```
// Returns the current browser as either NS (Netscape) or
// IE (Internet Explorer)
function getBrowser(){
    var nav="";
    if (navigator.appName.indexOf("Netscape")>-1){
        nav="NS";
        }
    if (navigator.appName.indexOf("Explorer")>-1){
        nav="IE";
        }
    return nav;
    }

// Returns the browser's version number rounded down to
// the nearest integer
function getVersionNumber(){
    var num=0;
    return parseInt(navigator.appVersion,10);
    }
// Called by the image, makeButton assigns event handlers to
// the image and determines the type of the image (gif, jpg,
// etc.). It also ensures that on rollover a hand icon is    // displayed.

function makeButton(me){
    me.extension=me.src.substring(me.src.length-4, _
       me.src.length);
    me.onmousedown=btnPressed;
    me.onmouseover=btnOver;
    me.onmouseup=btnOver;
    me.onmouseout=btnRestore;
    me.handleButton=handleButton;
    if ((getBrowser()=="IE") && (getVersionNumber()>3)){
        me.style.cursor="hand"
        }
    this.btnOver();
    return true;
    }

// Event called when the mouse is pressed on the graphic.
// If the name of the graphic is Enter.jpg then this routine
// assumes the depressed state is called EnterPressed.jpg.

function btnPressed(){
    if (getBrowser()=="IE"){
       this.src="images/"+(this.id)+"Pressed"+this.extension;
       ExecFunction(this.id)
        }
```

```
    else {
       this.src="images/"+(this.name)+"Pressed"+ _
          this.extension;
       ExecFunction(this.name)
        }
    this.pressed=true;
    }

// Event called when the mouse moves over the graphic.
// If the name of the graphic is Enter.jpg then this routine
// assumes the highlight state is called EnterHi.jpg.

function btnOver(){
    if (getBrowser()=="IE"){
        this.src="images/"+(this.id)+"Hi"+ _
          this.extension;
        Describe(this.id);
        }
    else {
        this.src="images/"+(this.name)+"Hi"+this.extension;
        Describe(this.name);
        }
    if (this.pressed){
        this.pressed=false;
        this.handleButton();
        }
    }

// Event called when the mouse moves off the graphic.
// This restores the graphic to its original state

function btnRestore(){
    if (getBrowser()=="IE"){
        this.src="images/"+(this.id)+this.extension;
        }
    else {
        this.src="images/"+(this.name)+this.extension;
        }
    }

// ExecFunction acts as a switchboard.
// Associate the id of the image
// with the action you want completed
// (Default is to do nothing)

function ExecFunction(id){
    switch(id){
        // Put a case in for each function
        // if button name is "Enter" then
        // code here would look like:
        // "Enter":
        // DoEnterAction(); // This is user defined
        // break; // This terminates the choice
        }
    }
// Describe is called when the mouse moves
// over or off of the button.
// You can customize actions by specifying
// the ID of the image and
```

```
// associating code as with ExecFunction.
// Default action is to set the
// status window to the name of the button.

function Describe(id){
    switch(id){
        default:
            if (""+id=="undefined"){
                window.status=window.defaultStatus;
                }
            else {
                window.status=id;
                }
            break;
        }
    }
```

2. Modify the ButtonTest.htm file as shown in Listing 15.3.

Listing 15.3 ButtonTest modified to enable buttons in JavaScript.

```
<HTML>
<HEAD>
<TITLE></TITLE>
<SCRIPT SRC="ButtonScripts.js"></SCRIPT>
<SCRIPT>
function notify(){
    window.report.innerText=DHTMLPage1.GetInfo());
    }
</SCRIPT>
</HEAD>
<BODY
    BACKGROUND="islands/islands.jpg"
    BGCOLOR="#ff8000"
    TEXT="white"
    LINK="yellow"
    VLINK="red"
    ALINK="white">
<IMG ID=Enter
    SRC="images/enter.jpg"
    WIDTH=100
    HEIGHT=40
    ALT=""
    BORDER="0"
    ONLOAD="makeButton(this)"
    ONCLICK="notify()">
<DIV ID=report> </DIV>
</BODY>
</HTML>
```

3. In your DHTML application, remove the **onmousedown**, **onmouseover**, **onmouseup**, and **onmouseout** handles of the Enter object.

TIP
Providing Security
A DHTML application involves the use of an ActiveX control. Because such a control has access to the entire machine, it represents a considerable security risk. So after you have completed your control, you will need to code sign it to provide some security validation. Otherwise, it will only run in low security environments and will raise alert boxes even there.

---

4. In the **(General)** section of DHTMLPage1, add the **GetInfo()** function:

```
Public Function GetInfo() as String
    GetInfo = "You pressed this button at " + Time$()
End Function
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

This example actually demonstrates two concepts: adding JavaScript into a VB application and calling a Visual Basic function from JavaScript. The first concept is straightforward—just because VB hooks into the Internet Explorer event queue doesn't mean that you can't still run scripts in Web pages. The script here illustrates that concept quite well. In general, if you have both a JavaScript event handler and a Visual Basic event handler for the same event, the JavaScript will be performed first.

This is demonstrated by the second concept—you can call a VB function from JavaScript. There are two event handlers for the Enter image's **onclick** event. The JavaScript handler calls a JavaScript function called **Notify**. **Notify** in turn references the DHTMLPage1 object, which has defined a **GetInfo** routine. That routine returns a string indicating when the button was pressed to the nearest second, which is displayed in a **<DIV>** below the button. After you get past the ActiveX alert (which will be covered momentarily), then the Visual Basic **onclick** event is called.

Although there is a lot that can be done with image manipulation in Visual Basic, in most circumstances you will be better off encapsulating mechanical functionality (such as buttons) in JavaScript code. By and large, these don't need to be secured anyway. The danger of people looking through your source code comes in exposing your business logic, not multimedia functions.

Preloading Images

One problem that any Web developer faces when dealing with graphics is latency—it takes time to download images. Although this is an unavoidable situation, it gets worse when you start introducing button-state graphics. Ideally, you don't want to have an obvious wait when a user presses a button for the down state to show up. If the image is cached on the client, it takes very little time to display it, but if it's not cached, then the lack of responsiveness could make your user press the same button several times.

One way of getting around this is to preload you graphics. To do this, it's necessary to create an image object without displaying it, and preferably without letting it disturb the flow of output. Although it's not directly possible from Visual Basic, you can force the IE engine to create a generic image file without having to display it. To pull this off, you need to call the **execScript** function to run some JavaScript, as follows:

```
Public Sub PreloadGraphic(URL)
Call Document.parentWindow.execScript( _
   "var Img=new Image();Img.src='"+URL+"';")
End Sub
```

You would call this function after the page has initially loaded but at some point prior to needing the graphic specified in the URL (for example, in the **BaseWindow_onLoad()** event handler). Because the image is cached, when next you call **myImage.src=URL**, the browser uses the cached copy rather than the remote copy, significantly speeding up the time to draw the graphic.

The only caveat here is that preloading graphics still takes time. You still need to factor in total download time of graphics. All that you've done by preloading is made the image available sooner.

Doing It With Style

HTML started out being rather bland. The primary heading on a page was supposed to be **<H1>**, the secondary headings **<H2>**, and so forth. Paragraphs were differentiated by **<P>** tags, but nowhere was there any specification that said what a paragraph (or a header tag, for that matter) should look like. This seemingly glaring omission was, of course, deliberate. By deliberately divorcing content from description, the original HTML specification was essentially trying to maintain an appearance agnosticism. In other words, it was the responsibility of the browser rather than the Web page to set the styles used in a page.

However, somewhere along the line, the focus of most Web pages began to change. Rather than being produced to display the latest in physics or mathematical research, Web pages became vehicles for companies to market their products or for individuals to express themselves. As a consequence, the small number of tags mushroomed as specialized needs arose. Some of them (such as the irritating **<BLINK>** tag) disappeared into obscurity, while others, including the **<FONT>** tag, became heavily integrated into Web development

tools.

The **<FONT>** tag demonstrates both the utility and danger of taking the approach of adding a tag for each specialized function. Using the **<FONT>** tag, you can set the font style and size by setting the **STYLE** and **SIZE** attributes. The drawback to using the **<FONT>** tag is that it provides absolutely no information to the document about how the currently enclosed text fits in with the rest of the document. If the relevant text is within a paragraph tag but you still set the size of the **SPAN**ned text to, say, 24pt bold, then is the text contained therein still a paragraph? In other words, if

```
<DIV><FONT SIZE="7">This Is A Header?!</FONT></DIV>
```

is visually equivalent to

```
<H1>This Is a Header?!</H1>
```

then how can the user distinguish between a header tag and a paragraph that looks like a header tag? This may seem a fairly minor distinction if all you are doing is marking up the page, but once you have programmatic control of the elements on the page, then the distinction is every bit as important as the distinction between a **long** and a **double** in Visual Basic.

In Chapter 12, the notion of the data layer and the presentation layer was given, with the focus on XML as the data layer. Although it may be awhile before widespread acceptance of XML lets you work with that element, it is already possible using Cascading Style Sheets and DHTML to create at least a basic data/presentation layer split, something that is perhaps more important to client/server programmers than to most Web developers.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Cascading Style Sheets (also referred to as CSS) are an attempt to separate the data (or context) layer of the Web page from the presentation layer. The style sheets are cascading because any style that is applied to an HTML element is also applied to all contained elements, unless they are explicitly overridden. For example, if you apply a new **<FONT SIZE>** to a paragraph that contains the bold tag (**<B></B>**), then the **<FONT SIZE>** also modifies the contents of the bold tag. In this way, if you set the style of the body to a given attribute (such as a <**FONT SIZE**> or <**FONT COLOR**>), then you also set the style of everything inside the document to that same attribute, unless an element explicitly sets its own attribute to something else.

There are two ways that you can modify the style of an element. In the first, you set the style attribute of the tag to a name-value pair, with the CSS name being followed by a colon, and the value following the colon. You can also have more than one CSS element within a style tag. In this case, each pair of attributes is separated by a semicolon. For example, if you want to set the color of a text block to red and the font size to 18 points, you'd use the following expression:

```
<P style="color:red;font-size:18pt;font-family:san-serif;">
This is a warning!</P>
```

You only need to alter those values that you need to overrule. All the other CSS properties inherit their values from the containing object (ultimately the **<BODY>** of the document).

This approach can set the individual styles of an object, but any change that is made to the style of a block affects only that block and any block that it contains. In an HTML document, it's actually pretty likely that you will have more than one instance where you'd like a particular style (such as, a "warning" style—red, larger, san-serif) of block. By specifying the block attributes as styles, you are in the ugly position of having to manually change each style should you want to make a global change (changing the warning style's font size to 14 points instead of 18 points, for instance).

Fortunately, CSS lets you create rules, also called *classes*, that specify style information for similar elements. As an example, you can create a **.Warning** class that embodies the

style rules that you need, as shown in Listing 15.4.

Listing 15.4 Style and rule class example.

```
<HTML><HEAD><TITLE>CSS Sample</TITLE>
<STYLE>
P           {font-family:serif;font-size:11pt;}
H1          {color:blue;}
.Warning    {color:red;font-family:san-serif;}
</STYLE>
</HEAD>
<BODY>
<H1>Warning Test</H1>
<P>Here is the basic message.</P>
<P Class="Warning">Warning, Will Robinson! Warning!</P>
</BODY>
</HTML>
```

Three classes were declared in the <**STYLE**> section of the document in Listing 15.4. The first two redefined current HTML tags <**H1**> and <**P**> respectively. Note that whenever an already-existing tag has its style redefined through a rule, it doesn't take a period.

The **.Warning** class, on the other hand, starts with a period to indicate that it is essentially a user-defined class. In practice, this means that an HTML element calls the class through the *class* attribute. The class then overrides the tag's implicit declarations with any explicit declaration, leaving the remainder unchanged. For example, the paragraph tag is redefined so that its **font-size** is 11 points. The **.Warning** tag subclasses the paragraph tag, setting the **color** to red and the **font-family** to san-serif, but it retains the 11 point **font-size**. If the **Warning** property had sub-classed the heading 1 style (that is, <**H1 CLASS=Warning**>) then the heading would be 24 points tall and colored red with a san-serif font.

NOTE
It should be noted that a CSS class is completely unrelated to a Visual Basic class, except in the very loosest definitions of object-oriented programming.

There are a number of different ways that you can integrate styles and CSS classes into your DHTML applications. The first (and simplest) method is to define the CSS attributes in the <**STYLE**> section of your template document's <**HEAD**> declaration, as demonstrated in Listing 15.4. While this works reasonably well for simple Web pages, this approach has the problem of requiring the same code to appear identically in multiple pages, which increases the chance of stylistic discrepancy between pages.

A second method would be to link to a common style sheet, using the <**LINK**> tag. Using this method, you extract the style sheet rules and place them in a separate document (with no enclosing <**STYLE**> tags), then reference them as an external document. Listing 15.5 shows the use of the <**LINK**> tag.

Listing 15.5 The <**LINK**> tag loads in CSS documents.

```
<!-- Contents of DocStyle.css -->
P           {font-family:serif;font-size:11pt;}
H1          {color:blue;}
```

```
.Warning     {color:red;font-family:san-serif;}

<!-- Declaration of LINK -->
<HTML><HEAD><TITLE>CSS Sample</TITLE>
<LINK REL="stylesheet" TYPE="text/css" HREF="DocStyle.css">
</HEAD>
<BODY>
<H1>Warning Test</H1>
<P>Here is the basic message.</P>
<P Class="Warning">Warning, Will Robinson! Warning!</P>
</BODY>
</HTML>
```

By removing style sheets from the HTML documents, several documents can use the same set of definitions, making for a more consistent display of information. It also takes you one step closer to an XML-based data-driven output, because, syntactically, there's not a lot of difference between

```
<DIV CLASS="Warning">This is a warning! Alert! Wake up!</DIV>
```

and

```
<WARNING>This is a warning! Alert! Wake up!</WARNING>
```

which is the format that an XML document would take.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Some Stylish Properties

In general, when you put together a Visual Basic DHTML application, you will usually be working with a Web designer to put together the "look and feel" of the HTML document. This is one of the greatest benefits of working with the **WebClass** formats, because the division of labor means that a client/server programmer can concenrate on adding functionality without having to learn all the intricacies of Web design. Manipulating styles and style sheets definitely falls into that category, although there are a few basic style properties that are essential to your job as a programmer.

There are several dozen properties associated with the style sheet, although for the purpose of client/server programming, you can get by with a fairly small subset of style properties. These are outlined in Table 15.2 and assume a **<DIV>** element called **myDiv** as the object on which they act. For a complete listing of all of the style properties, check out the books mentioned at the end of this chapter. Table 15.2 displays the properties as they would be called from Visual Basic, rather than inline as HTML code.

Table 15.2 Style attributes most useful to client/server programmers.

Property Or Method	Example	Description
border	**MyDiv.style.border="solid blue 2px"**	Determines the border style of divisions or table cells. Can be sub-classed by position or function (that is, **style.borderTopColor="blue"**, or **style.borderBottomWidth="4px"**).
clear	**MyDiv.style.clear=True**	Causes the next element or text to be displayed below the current element.
clip	**MyDiv.style.clip="rect (0 200 150 0)"**	Sets or determines the clipping region of a division. Rectangular coordinates are given as **(top right bottom left)**. See **overflow**.
color	**MyDiv.style.color="#0000FF"** or **MyDiv.style.color="blue"**	Sets the fore (or text) color. You can either use a hex triplet color pair (**#RRGGBB**) or a predefined name (**blue**).
cssText	**Debug.print myDiv.style.cssText**	Returns the entire defined CSS rule for the given element.
cursor	**MyDiv.style.cursor="hand"**	Sets or retrieves the cursor type for a given object. Some possible values include: **hand**, **crosshair**, **text**, **default**, **wait**, **move**, **help**, and the directional arrows **n-resize**, **ne-resize**, and so forth.

display	**MyDiv.style.display="none" or myDiv.style.display=""**	Controls whether an element is hidden or displayed. When set to **none**, **display** hides the element, reflowing text to fill in where the element was (cf., invisibility). When set to a blank string, display shows the element again.
font (also, **fontFamily, fontSize**, **fontStyle**, **fontVariant**, **fontWeight**)	**MyDiv.style.font="Helvetica 12pt Italic"**	Sets or retrieves the font attributes of a container.
left, **top**, **width**, **height**	**MyDiv.style.width="100px"**	Sets the positional attributes for the container. Note that position can be set in one of several coordinate systems, including pixel (**px**), point (**pt**), centimeter (**cm**), inches (**in**), and printers ems (**em**).
listStyleImage	**MyDiv.style.listStyleImage="url(images/folderIcon.gif)"**	Sets or retrieves the bullet graphic for a list item. Note the **url()** syntax.
listStylePosition	**MyDiv.style.listStylePosition="outside"**	Sets or retrieves the position of the bullet relative to the list item (**outside** is outdented, and **inside** is indented).
listStyleType	**MyDiv.style.listStyleType="square"**	Sets the type of bullet in **<UL>** elements or the numbering scheme in **<OL>** elements. For **<UL>**, values include: **none**, **circle**, **disc**, and **square**. For **<OL>**, values include: **none**, **decimal**, **lower-alpha**, **lower-roman**, **upper-alpha**, and **upper-roman**.
margin (also **marginLeft**, **marginTop**, **marginRight**, **marginBottom**)	**MyDiv.style.margin="3px"**	Sets the space between the boundaries of the container and the content of the container.
overflow	**MyDiv.style.overflow="hidden"**	Determines how content overflows the prescribed boundary box of the container, with values including **hidden**, **auto**, **scroll**, and **visible**. **hidden** will display just the clipped region if **clip** is defined.
padding	**MyDiv.style.padding="3px"**	Sets the width of the padding space between the container boundaries and the surrounding content.
position	**MyDiv.style.position="absolute"**	Sets the way that the element interacts with the rest of the page. See "Absolute Position Corrupts Absolutely" later in this chapter.
pixelLeft, **pixelTop**, **pixelWidth**, **pixelHeight**	**MyDiv.style.pixelLeft= myDiv.style.pixelLeft+10**	Sets or retrieves the requested attribute in pixels as a pure number rather than as a string-based number and units.

posLeft, **posTop**,**posHeight**, **posWidth**	**MyDiv.style.posLeft=20**	Sets or retrieves the requested attribute in the last units that the container was referenced, as a pure number. So, if a previous statement has set the left position as **myDiv.style.left=3in**, then the **myDiv.style.posLeft** is **3**.
styleFloat	**MyDiv.style.styleFloat=left**	Converts a span into an inline floating element that can be positioned either to **left**, **right**, or **none** to turn float off.
visibility	**MyDiv.style.visibility="visible"**	Makes the element visible on the screen (**visible**) or invisible (**hidden**). Unlike **display**, **visibility** still keeps the element's flow intact even when invisible.
zIndex	**MyDiv.style.zIndex=5**	Sets the order in which absolutely positioned elements appear on the page, with the higher **zIndex** appearing toward the user.

Previous | Table of Contents | Next

Products | **Contact Us** | **About Us** | **Privacy** | **Ad Info** | **Home**

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Please note that the CSS equivalents for these properties might differ slightly in syntax—usually, capitalization of the properties is replaced with a dash and lower case equivalent. Thus, **listStyleImage** has a CSS equivalent form of **list-style-image**.

Absolute Position Corrupts Absolutely

In order for DHTML to work, it's necessary to create at least two distinct types of containers, although Internet Explorer thankfully defines three. HTML works by creating a flow diagram—every element within the page has some rules about how it follows the preceding element. For example, a paragraph element will normally cause a line break so that the next element appears below the paragraph, usually with about half a line of space appended to distinguish between paragraphs. Images can partially break this ordering through the use of the **ALIGN** attribute. Setting **ALIGN="left"**, for example, will always make the image appear to the left and flush with the body of text in which it's contained.

However, in certain circumstances, you will want to remove an item from the normal flow of the page and position it absolutely on the page. For example, creating floating pop-up boxes of text in a Web page are impossible to do without absolute positioning, and the engine that handles replacement of text with **innerHTML** likewise is unable to work if the container has a static position on the page.

You can control how the element appears with the use of the **position** property. Setting **position** to **absolute** causes the element to position itself absolutely *relative to its container*. Listing 15.6 provides an example of code using absolute positioning.

Listing 15.6 An HTML code fragment that demonstrates absolute positioning.

```
<DIV ID=outerCtnr style="position:absolute;
   left:20px;top:40px;">
    <DIV ID=innerCtnr style="position:absolute;
       left:30px;top:50px;">
        This is some text
    </DIV>
</DIV>
<DIV ID=newCtnr style="position:absolute;
   left:100px;top:120px;">
    This is another container
</DIV>
<DIV ID=newCtnr2 style="position:relative;left:10px;
   top:15px;width:120px;height:80px;">
    This is yet another container
</DIV>
```

The **outerCtnr** division is positioned absolutely at 20 pixels from the left of the page boundaries and 40 pixels from the top. The **innerCtnr** division is positioned absolutely as well, but relative to the **outerCtnr**, not the page. Thus, the **innerCtnr** division starts 50px from the left and 90px from the top.

On the other hand, the **newCtnr** division is positioned absolutely, relative to the page, as **left=100px; top=120px**, because its parent container is the page itself.

Sometimes, you will want a container to relatively follow another container. To do this, use the **relative** value for the **position** attribute. For example, if **newCtnr2** had no modifiers other than **relative** in the style sheet, then it would appear directly beneath the bottom of the **newCtnr**. However, by setting **left** to **10px** and **top** to **15px**, **newCtnr2** is displaced that many pixels to the right and below **newCtnr**.

NOTE
You must set the position attribute of a container to either **relative** or **absolute** in order for it to be modified with the **innerHTML** attribute. If **position** is **static**, then the browser will not display any changes to the **innerHTML**, even though it registers them internally.

In order to turn off either **absolute** or **relative** positioning, you can set the **position** attribute to **static**—this is the default value and indicates that the element will be flowed with the normal rules of the browser.

To complicate matters, in Internet Explorer 4 you cannot change the **position** attribute after it's set in the document, at least through code. To get around it, you could retrieve the **position** property of the element and toggle its value, then set the **outerHTML** of the element to the element with the **position** property toggled, written as a string. This is a fairly ugly function, however, and won't be covered here.

CSS has an astonishingly diverse set of units for positioning elements, which are summarized in Table 15.3. Although this range of units may seem overkill for positioning items on a Web page, keep in mind that the eventual intention of CSS is to provide a single comprehensive standard for output to any number

of devices, from computer screens to printers to Braille and aural readers.

Table 15.3 Units of measure (note that all are approximately the same length).

Unit Of Measure	Example	Description
px	18px	Pixel (one dot on a computer screen—size dependent upon monitor density)
pt	18pt	Point (1/72 of an inch—note that this is the computer point, rather than the printer's point)
in	0.25in	Inch (absolute measure)
cm	1cm	Centimeter (absolute measure)
mm	10mm	Millimeter (absolute measure)
pc	1.5pc	Pica (1 pica=12pt) (absolute measure)
em	1.1em	Element's font-height
ex	1ex	Element's font x-height

Internet Explorer's positional properties provide a rich model to write code from—they are also incredibly confusing. The properties **left**, **top**, **width**, and **height** take and return strings consisting of the position followed by the unit string. For example, the left position of a **<DIV>** positioned absolutely at 1 inch from the margin could be given as 1in, 72pt, or 12pc, but if the units are not included (that is, 12) then the results could be unpredictable.

To retrieve the position in pixels, use the pixel properties (such as **pixelTop**, **pixelLeft**, and so forth). These get the positions of the objects as pure numbers, relieving you from the onerous duty of removing the units. The **posTop**, **posLeft**, **posWidth**, and **posHeight** properties will return an absolute numeric value based on the unit that was last assigned to the positional property. For example, if **myDiv.style.left="6pc"**, then **myDiv.style.posLeft** would be 6, while **myDiv.style.pixelLeft** would return 36 on a standard 640×480 monitor (6 picas is half an inch; on a 640×480 screen 1 inch is 72 pixels long on most monitors, so half of 72 is 36).

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Style Sheets And VB

When it comes to style sheets in Internet Explorer, the object model as documented by Microsoft gets ugly quick. Fortunately, most of what you need to be able to do with style sheets can be handled with a handful of properties and functions. Like everything else with Internet Explorer, getting IE to go the first 9/10 of a mile is easy—it's that last 500 feet that can kill you.

As has been hinted throughout this section, nearly every element in DHTML (with the exception of form elements, header information, and specialized tags, such as **<HR>**) has an associated style property. This is an object that provides an entry point to the CSS properties. For example, to get the position property of a **<DIV>** object named **myDiv**, you'd access the style property like this:

```
debug.print myDiv.style.position
```

If you wanted to create a placeholder shortcut, you can still get most of the style properties through Intellisense in VB by declaring the holding variable as type **HTMLStyle**:

```
dim st as HTMLStyle
set st=myDiv.style
st.position="absolute"
```

You can also change the style of a container by assigning a new **className** attribute:

```
myDiv.className="Warning"
```

Here, you need to be conscious of case, because **myDiv.className= "Warning"** is not the same as **mydiv.className="warning"** in Internet Explorer.

Manipulating class names can simplify and clarify your code. For example, if you

defined the following style sheet

```
<STYLE>
.Default {color:red;}
.Highlight {color:orange;}
.Pressed {color:maroon;}
.Hidden {visibility:hidden;}
.Visible {visibility:visible;}
</STYLE>
```

and if you had a <**DIV**> with **ID=myLink**

```
<DIV ID=mylink CLASS="Default"
    onmouseover="this.className='Highlight'"
    onmousedown="this.className='Pressed'"
    onmouseup="this.className='Highlight'"
    onmouseout="this.className='Default'"
    >
```

you could essentially implement rollover buttons with almost no real code. Likewise, you could shorten your visibility code using class names:

```
mydiv.className="Hidden"
```

Ruling Style Sheets

The next step to working with style sheets comes with the question: How can a class be changed dynamically? A good example of this would be an annotation. With Dynamic HTML, you can have a document that contains embedded annotations. Normally, you don't want to see the annotations, because they interrupt the flow of the document, but it would be nice to be able to see these annotations when you press a button. In this case, you are actually changing a class rather than simply an element. Unfortunately, it is considerably harder to access the classes to manipulate them than it is to modify a single element.

The **Document** object contains more than one style sheet. It actually has a collection of style sheets that are defined either internally through **<STYLE>** tags (with one sheet per set of tags) or externally with a **<LINK>** tag. These style sheets are zero-based. That is, the first sheet is **document.styleSheets(0)**, and each style sheet is of type **HTMLStyleSheet**.

In turn, each style sheet is made up of a collection of *rules*. A rule is essentially the same as a CSS class—a collection of CSS properties that have been explicitly defined and not surprisingly has a VB type of **HTMLStyleSheetRule**. A rule has three properties: **readOnly** (which determines whether the rule can be modified), **selectorText** (the class name), and a **style** property (which is the same as an element's **style** property). If you modify a rule's style, then all elements that have that rule as a class will get changed as well.

The function **GetRule** takes the name of a rule and returns a reference to the rule's style if the rule's name is found in the DHTML document's collection of style sheets. Listing 15.7 defines the **GetRule** function so that it returns the style object of that particular

sheet's rule.

Listing 15.7 The **GetRule** function retrieves the style object associated with a given class.

```
public Function GetRule(RuleName as String) as HTMLStyle
    Dim styleSheet as HTMLStyleSheet
    Dim rule as HTMLStyleSheetRule
    Dim styleSheetIndex as Integer
    Dim ruleIndex as Integer

    set GetRule=Nothing
For styleSheetIndex=0 to Document.StyleSheets.Length-1
        set styleSheet=Document.StyleSheets(styleSheetIndex)
        For RuleIndex=0 to styleSheet.Rules.Length-1
            set rule=styleSheet.Rules(ruleIndex)
            if rule.SelectorText=RuleName then
                Set GetRule=rule.style
                Exit Function
            end if
        Next
    Next
End Function
' Usage:
' dim st as HTMLStyle
' set st=GetRule("Annotation")
'(Make every annotation visible on the page)
' st.display=""
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Maintaining A Dialog Box

The advantages of working with Internet Explorer should be obvious by now, but there are a few drawbacks as well. One of the most obvious problems is that when you use an IE front end, your application coexists with the browser's interface. The user can press the Back and Forward buttons (or worse, the Refresh button), causing havoc with maintaining state and initiating some potentially crashing code. Also, while you can launch a new window from a Web page, the window is modeless—you can click onto any other window currently open, raising the prospect that needed data doesn't get to the right place at the right time. Fortunately, Internet Explorer 4 and higher provides a new window data type—a custom dialog box—that satisfies both requirements.

Supplying Standard Dialog Boxes

Before going into detail about creating custom dialog boxes, it's worth looking at the dialogs that you can create directly from the object model. They may very well fill your needs without you having to customize a Web page.

All standard dialog boxes are called as methods from the document's window object (usually through **document.parentWindow**). The simplest dialog box, one that just displays a line of text and an OK button, is invoked with the alert method, as follows:

```
document.parentWindow.alert "Your document has been scanned."
```

The alert box gets used far more often than it should on Web pages, because it stops the system until such time as you respond to it, and you can only respond in one way (to get rid of the alert box).

If you need the user to make a decision, you can also invoke the confirm dialog box, which has a Yes and No button (returning true and false respectively):

```
dim rslt as Boolean
```

```
rslt=document.parentWindow.confirm("Do you want some & " _
   "more information?")
if rslt then
    HelpText.innerText="Here's more information."
end if
```

Finally, you can get text information from the user with the prompt dialog box:

```
Dim Name as String
Name=document.parentWindow.prompt ("What is your name?", _
   "Nemo!")
```

This takes a prompt and a default value (which is set to **undefined**, if it's not explicitly set) and returns the resulting value. As a design note, if you need more than a single prompt, you are much better off designing a form with input boxes than using multiple prompts. Your users will thank you for it.

Launching New Browser Windows

The distinctions between standard browser windows and dialog boxes are subtle, although they can trip you up if you mix up calling standards. From Visual Basic, you can launch a window through the **window.open** command—this essentially launches another instance of the browser window, although you can control which parts of the interface appear in the window. The syntax for launching a window from a VB DHTML application is given as

```
Dim wnd as HTMLWindow2
Set wnd=Document.parentWindow.open(URL, _
   windowName,windowFeatures)
```

where **URL** is either the local or absolute URL of the document to be displayed, **windowName** is a string containing the name that a frame call can refer to (it's actually pretty meaningless in most scripting applications, because you'll probably want to work with the actual window reference, contained in the **wnd** variable in this example), and **windowFeatures** is a list of comma-delimited properties contained as a string, as summarized in Table 15.4.

Table 15.4 Properties for specifying how a regular browser window will be displayed with the **open()** method.

Attribute	Example	Description
CopyHistory	**copyHistory**	Sets the same history for the new window as the one that calls it, if the tag is included (affecting the Previous and Next buttons).
Directories	**directories**	Displays the directories (or selected links) in the browser's toolbar.
Height	**Height=400**	Displays the height of the total window in pixels.

Location	**location**	Displays the address bar when included.
MenuBar	**menubar**	Displays the menu bar below the title bar.
Resizable	**resizable**	Displays the resize box in the lower-right portion of the window.
Scrollbars	**scrollbars**	Displays scrollbars in the window frame if the document is too large for the window.
Status	**status**	Displays the status bar.
Toolbar	**toolbar**	Displays the toolbar (the bar that includes the Back, Forward, Refresh, and other buttons).
Width	**width=480**	Sets the width of the window in pixels.

Thus, if you want to display a window with no toolbar or address bar, but that does include a status bar and is resizable (original to 640×480), then for the Microsoft site the code would look like:

```
Dim wnd as HTMLWindow2
Set wnd=Document.parentWindow.open( _
   "http://www.microsoft.com","","width=640, _
   height=480,resizable,status")
```

Creating A Custom Dialog Box

Standard browser windows can be limited to simply a title bar by eliminating all the interface attributes in the **window.open** statement. However, such windows have some serious limitations:

- They are not modal. Clicking outside an invoked window will set the focus to what is clicked on, whereas true modal behavior will intercept any effort to click outside the window.
- Getting information from a window is almost impossible after it closes, without some elaborate preparatory work.
- Although you can close a window through scripting code (via the **window.close()** function), the browser will generate a warning message to the user that code is attempting to close the window. This is both distracting and makes the integrity of the code look suspect.

With Internet Explorer 4, there is a way to get around these issues. IE4 supports the **showModalDialog()** method, which will display a window with no supporting interface elements (buttons, menu items, and so forth), which address all three of these issues. The syntax is similar to that of the **open** method, although not identical:

```
Dim Answer as Variant
```

```
Answer=window.showModalDialog(URL,argumentsVar,features)
```

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

The **URL** is again a pointer to the HTML page that you want to have appear in the dialog box. The features are again contained in a string, but you need to be careful here. For some reason, the features are separated by semicolons instead of commas, and they share nothing in common with the **open()** properties. These are outlined in Table 15.5.

Table 15.5 Features list of the **showModalDialog** method.

Attribute	Example	Description
Center	**Center=yes**	Determines whether the dialog is centered on the screen. If **center=yes** then this overrides the **dialogLeft** and **dialogTop** properties.
DialogHeight	**DialogHeight=400**	Sets the height of the dialog in pixels.
DialogLeft	**DialogLeft=200**	Sets the distance from the left side of the screen in pixels.
DialogTop	**DialogTop=100**	Sets the distance from the top of the screen in pixels.
DialogWidth	**DialogWidth=400**	Sets the width of the dialog box in pixels.

The second parameter in the **showModalDialog()** method takes some explanation. You can essentially pass any string, number, or variant array in through the **argumentsVar**. Within the dialog Web page itself, this variable can in turn be referenced through the **window.dialog-Arguments** property, although it is the responsibility of the programmer to parse the arguments.

Visual Basic 6 includes two new useful functions for parsing strings: **split()** and **join()**. **Split()** takes as arguments a string and a delimiting character (such as a comma or semicolon) and then breaks the string into a variant array using the delimiting character as the divider. For example,

```
Dim CommaList as String
Dim CommaArray() as Variant
CommaList="sunny,partly cloudy,cloudy,rain,snow")
Set CommaArray=Split(CommaList,",")
Debug.Print CommaArray(0)  'Returns the first element
                           'of the newly created array
```

will cause the word “sunny” to get printed out to the debug window.

The **join()** function in turn takes a variant array and concatenates it with a delimiting character to create a string:

```
CommaList=Join(CommaArray,";")
Debug.print CommaList
```

The preceding code will print the following to the debug window:

```
"sunny;partly cloudy;cloudy;rain;snow"
```

You can use these two routines in a number of circumstances, although they really shine with parsing data. Typically, you will want to populate your dialog box with data when you create it—usually by filling fields within the dialog itself. You can do this by building a variant array of name-value pairs, then parsing this data inside the dialog box, as illustrated in Listing 15.8.

Listing 15.8 Code demonstrating how you can populate a variant array to set values within a dialog box.

```
'This is called from the originating Web document
Dim Args(5) as Variant
Args(0)="City:Olympia"
Args(1)="State:Washington"
Args(2)="Skies:Cloudy"
Args(3)="Hi:76"
Args(4)="Lo:56"
Args(5)="Forecast:Light rain interspersed with showers."
document.parentWindow.showModalDialog _
    "WeatherReport.htm",Args,"dialogWidth=600; _
    dialogHeight=400;center=yes"

'Within the WeatherReport.htm page DHTMLPage object
Private Function DHTMLPage_onload()
    Dim Index as Integer
    Dim ObjName as String
    Dim ObjValue as String
```

```
    Dim Args as Variant
    ' Create an alias variable for the dialog arguments
    Args=document.parentWindow.dialogArguments
    For Index=0 to ubound(Args)
        'Split the array at the color to get name/value
        ObjName=split(Args(Index),":")(0)
        ObjValue=split(Args(Index),":")(1)
       'Retrieve the named object and set its value
        document.all(ObjName).Value=ObjValue
    Next
End
```

If the dialog box needs to return a value, use the **window.returnValue** property within the dialog window. Note that this too can be a variant type, allowing you to pass arrays of data rather than just a single value back from the dialog box. When the dialog box is closed (using the **window.close**) method, the **window.returnValue** is passed as a return value for the **showModalDialog** function. Incidentally, one advantage to using dialogs is that you can close them without the system raising a warning message. Even if you don't make use of the argument passing, **showModalDialog** is worth remembering for that fact alone.

Accessing ActiveX And Applets

For a while, the public relations machine at Microsoft pushed the notion of ActiveX components in Web pages, but a number of factors have conspired to change this. Internet developers have been reluctant to adopt ActiveX controls, because they don't work uniformly on Netscape browsers (which still make up the lion's share of the market). Components developed in Visual Basic required a download of the control, plus such heavyweight packages as MSVBVM60.DLL. You could develop lightweight components in C++ through the use of Active Template Libraries (ATL), although the smaller size came at a cost of forcing the developer to create much of the framework for the code herself. Finally, in order to distribute these components, you needed to code-sign the control, with a cost of $400 to obtain the license and a business fitness rating that many smaller developers didn't necessarily have access to.

With the current Internet Explorer 4 browser (and even more so with IE 5), many of the initial reasons for using custom ActiveX controls have largely disappeared. Dynamic HTML can reproduce much of the interactivity that ActiveX controls originally targeted, while Internet Explorer ships with a number of specialized components, including:

- NetShow (video and sound)
- Structured graphic control (2D vector graphics)
- Path (animation positioning control)
- Sequencing (event sequencing)
- Direct Animation (3D graphics)
- Chat
- NetMeeting (collaborative services)

All of these are beyond the scope of this book, but you should check the

bibliography at the end of this chapter for some highly recommended references.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Although multimedia services are readily available, you might need other capabilities that don't ship with Internet Explorer. For example, you might want a calendar or graphic control. In general, although you can add controls to your page via the DHTML Editor in Visual Basic, I'd recommend using the ActiveX wizard or your favorite HTML editor to insert such controls.

Building A Scheduling Calendar

A scheduling calendar is a simple yet fairly powerful demonstration program that shows what type of things can be done with ActiveX controls. In it, the Microsoft Access Calendar component is coupled with a **<TEXTAREA>** box. When you click on a date, the control queries a database to determine whether an entry exists for the date and retrieves it if it does. You can also type a new entry into the box, and when you select a different date, the current entry is added to the database. This is admittedly a very crude program, but it provides the basics for ActiveX integration.

1. In an external HTML editor, create an HTML file called calendar.htm, as shown in Listing 15.8.

Listing 15.8 HTML template code for Calendar.htm.

```
<HTML>
<HEAD>
<TITLE>Calendar Test</TITLE>
</HEAD>
<BODY BGCOLOR="#ffffff">
<H1>Calendar</H1><BR>
<TEXTAREA ID=Comment COLS="42">
Press on a date to see or
set its message.</TEXTAREA>
<OBJECT CLASSID=CLSID:8E27C92B-1264-101C-8A2F-040224009C02
  HEIGHT=187 ID=Calendar1 STYLE="HEIGHT: 187px; LEFT: 20px;
  POSITION: absolute; TOP: 144px; WIDTH: 344px" WIDTH=344>
```

```
<PARAM NAME="Year" VALUE="1998">
<PARAM NAME="Month" VALUE="9">
<PARAM NAME="Day" VALUE="8">
<PARAM NAME="ShowDateSelectors" VALUE="-1">
<PARAM NAME="ShowDays" VALUE="-1">
<PARAM NAME="ShowHorizontalGrid" VALUE="-1">
<PARAM NAME="ShowTitle" VALUE="-1">
<PARAM NAME="ShowVerticalGrid" VALUE="-1">
</OBJECT>
</BODY>
</HTML>
```

The calendar object is defined in the <**OBJECT**></**OBJECT**> tag pair. Its **CLASSID** attribute is the universal identifier, or GUID, that Windows uses to identify all objects in the Registry. After the <**OBJECT**> tag, the definition includes a series of parameter tags (<**PARAM**>) which contain name-value pairs for initializing the control.

2. You will need to create a database to hold the entries. Using Access or Visual Basic's Database editor, create a one-table database that includes a text field called DateStamp and a memo field called Comment. Call the table Comments, then, use the ODBC Control Panel to create a system DSN called Calendar, with read and write privileges.

3. Create a new DHTML application project (the names are unimportant here, although Calendar is as good as any). Assign the Calendar.htm file to the class. This should give you two objects: **Calendar1** (the calendar ActiveX component) and **Comments** (a TextArea control). You should also create a reference to the ADO 2 classes (or use a database environment control).

4. In the **(General_Declarations)** section of the DHTMLPage, declare two variables: **LastDateStamp** and **LastComment**:

```
Private LastDateStamp As String
Private LastComment As String
```

These are used for updating the database.

5. In the **click** event for the calendar (invoked when you select any new element), place the code shown in Listing 15.9. This updates the database with the last element or adds a new entry into the database if the date wasn't previously selected.

Listing 15.9 Code for the calendar's click event; this adds or updates the database with the old entry before displaying the new.

```
Private Sub Calendar1_click()
    Dim Conn As Connection
    Dim RS As Recordset
    Dim DateStamp As String
    Dim commandStr As String

    'Open a connection to the Calendar database
    Set Conn = New Connection
```

```
Conn.Open "Calendar"
Set RS = New Recordset
'If the last dateStamp was previously defined then
If LastDateStamp <> "" Then
'Open the record containing the current dataStamp _
    RS.Open "SELECT * FROM Comments WHERE _
    DateStamp='" + LastDateStamp + "';", Conn, _
    adOpenKeyset
'If the last comment had contents then update it.
    If LastComment <> "" Then
    'If there are no records that match the old datestamp
       If RS.RecordCount = 0 Then
         'Close the recordset
         RS.Close
         'Replace single quote character with an "'"
         'character. This simplifies reading and
         'writing the SQL.
         LastComment = Replace(LastComment, "'", Chr(96))
         'Insert a new record into the database with the
         'appropriate dateStamp and comment.
         commandStr = "INSERT INTO Comments _
         (DateStamp,Comment) SELECT '" _
         + LastDateStamp + "' AS DateStampe,'" _
         + LastComment + "' AS Comment;"
            Conn.Execute commandStr
        Else 'Otherwise
             'Change the comment associated with
             'an already extant record.
            RS.MoveFirst
            RS("Comment") = LastComment
            RS.Update
            RS.Close
        End If
    End If
End If
'Create a DateStamp from the new data
DateStamp = CStr(Calendar1.Month) + "/" + _
   CStr(Calendar1.Day) + "/" + CStr(Calendar1.Year)
'Open the record for that date, if it exists.
Set RS = Nothing
Set RS = New Recordset
RS.Open "SELECT * FROM Comments WHERE _
   DateStamp='" + DateStamp + "';", Conn, adOpenKeyset
'If it doesn't, clear the comment box test.
If RS.RecordCount = 0 Then
    Comment.Value = ""
Else
    'Otherwise, replace the text with the entry's
    'contents, making sure that you convert the
    'acute symbo ("`") with its regular equivalent ("'").
    RS.MoveFirst
```

```
        Comment.Value = Replace(RS("Comment"), Chr(96), "'")
    End If
    RS.Close
    Conn.Close
    ' Set the old date stamp to the current one
    LastDateStamp = DateStamp
    ' Clear the last comment
    LastComment = ""
End Sub
```

6. Finally, you should catch the **keyup** event to ensure that every time the Comment box changes, the **Comment.Value** is also up-to-date:

```
Private Sub Comment_onkeyup()
    LastComment = Comment.Value
End Sub
```

7. Run the program, and try adding comments into the field to see what happens.

Previous | Table of Contents | Next

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Previous | Table of Contents | Next

The salient point about adding an ActiveX control like the calendar control is that it is similar to adding a control to the form. Visual Basic uses a multithreaded apartment approach, as does Internet Explorer. However, many older ActiveX controls were designed to work within a single-threaded environment. You can convert the component you just created to single thread by going into the Project Properties dialog box (Project|Project Properties), and set the Threading option to Single. This won't guarantee that the component will run, it just increases the likelihood—there have been some significant changes in ActiveX technology since it was first released in 1996.

Java-based applets offer an interesting challenge to the Visual Basic/DHTML programmer. If an applet has been built around a COM interface (that is, if it was created in Microsoft's Visual J++ program to expose a COM table), then it should work fine, in the same way any other ActiveX component works. If the applet wasn't built around a COM interface, then the control can probably still be manipulated from JavaScript, but you won't be able to directly control it from Visual Basic.

The Future Of Internet Programming…

is really, really bright. If the decade of the 1980s saw the rise of GUI programming and the 1990s the rise of the Internet, then the first decade of the millenium looks to be about the rise of *contextual distribution*.

One way of thinking about this is to look at the increasing application of distributed computing—the Internet, intranets, extranets—in disciplines that traditionally haven't been computer oriented, such as textiles, groceries, manufacturing, construction, and so forth. In essence, most of these deal with the process of distributing assets—getting raw materials to factories, getting finished goods to points of purchase, advertising the availability of goods for access by consumers, and the disposal of waste products in this whole train. With sophisticated access to databases, it becomes possible to track these

goods at all stages of production, which means diminishing the waste and increasing the efficiencies of all of these. Online shopping is beginning to make serious inroads against malls and other traditional retail outlets. If you look at a success story like Amazon.com, you can see that they have been effective not by replacing the media that they deal with by electronic equivalents, but rather using the electronic media and the tracking systems developed through the Internet to minimize inventories and reduce the number of extraneous distribution points.

Contextual distribution also means that data is increasingly carrying its own means of presentation. XML is significant as a data technology not for its format (which is fairly simple, all things considered) but because it can carry implicit structure and meaning within itself. Technologies such as Web classes mean that the media itself can be transformed readily through the application of database programming principles, adapting to the capabilities of the viewing tools. While DHTML applications may have somewhat less impact initially because of the relatively limited distribution of suitable browsers, even that will change over time.

A browser cannot (and should not) replace all commercial applications, but it should become a common medium upon which developers can build programs that extend beyond one machine. Unlike the dumb terminal paradigm that characterized the 1960s and 1970s (and which certain companies would love to see return with NCs in the next decade), the smart client essentially means that programs become collaborative in nature. An example of this can be seen in games, the vanguard of new technologies. With surprisingly little fanfare, most games have developed multiplayer support, because the human nature of game playing is almost always more interesting than algorithmic alternatives.

Client/server technology is following this trend as well. On one end, the data engines are moving away from proprietary technology to device compatibility (look no further than the support VB6 adds for Oracle database access). On the other end, clients are moving from proprietary solutions to generalized ones, usable by a much wider range of people because these users do not need as much training or specialized support as they would for customized clients.

Internet Explorer 5

Much of Chapter 12 was devoted to the XML capabilities of Internet Explorer 5, even though this product is still some months from hitting the market as I write this. Although XML is integral to IE5, that technology is not the only reason IE5 is the perfect match for Visual Basic programming. The DHTML application classes work as well using Internet Explorer 5 as they do with IE4, and actually, with the changes being considered for the newer version, they might work even better.

A core change in IE5 is the introduction of *behaviors*. A behavior is a code document, essentially a COM object that can be assigned to an element in the HTML code as readily as a style or CSS class can. For example, you could create code validation routines that would be invoked automatically by a certain class of elements (a **<CODE-VALIDATOR>** class, say). If this sounds like a functional version of XML, you get an idea about where

Explorer technology is heading.

This notion works closely with the concept of *namespaces*. With namespaces, you can create libraries of objects that all work together in an integrated fashion, analogous to the Java notion of packages, but working with HTML. Namespaces also provide another mechanism for data typing and conversion.

Finally, Internet Explorer 5 is scheduled to incorporate most, if not all, of the CSS2 specification, approved this summer. This gives added support to alternate device output of Web pages (most significantly to printers). One of Visual Basic's failings is that it is difficult to produce printed output from the language, as at least one of the authors can attest. Integrating Visual Basic 6 and Internet Explorer 5 would provide the best of both worlds, as data could be formatted for output to an IE engine from a Visual Basic class, without the hassle of worrying about precise placement of printed elements.

Summary

In this chapter, you learned about creating more sophisticated Dynamic HTML applications, touching on such topics as images, scripting, dialog boxes, style sheets, and ActiveX components. There's a lot to cover, and this chapter only begins to scratch the surface of Internet client/server programming. Use the exercises in this book as a jumping point to build your own applications. The game is changing, and we're all in for a really wild ride.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Bibliography

While it's easy to develop a beer belly as a programmer—the sedentary nature of the profession has turned more than one reasonably fit body into jelly—programmers typically have strong arms. This apparent paradox can be explained by the fact that in order to stay proficient in the field, you usually have to walk around with three to five books, each weighing in at four or five pounds apiece, just to keep up. In the last several chapters, I've frequently had at least 3 books by my side for reference, and one day (while in transition from one chapter to the next), I had 10 books, all open, spread out on my desk.

For the Internet developer, there are a number of superb references out there, and surprisingly there are even some that are more or less current. I would be remiss, of course, not to mention the Coriolis books in general, which provide excellent tutorials and training guides for much of the Microsoft line. I also wish to point to the Wrox line of books, which are timely, concise, and informative (and of which I mention several here).The following recommendations are organized by topic.

XML, XSL, And Style Sheets

Boumphrey, Frank: *Style Sheets for HTML and XML*. Wrox Press 1998. ISBN: 1-861001-65-7. A highly detailed look at the CSS1 and CSS2 specifications, along with the most cogent description of XML style sheet technologies I've seen yet.

Holzner, Steve: *XML Complete*. McGraw Hill 1998. ISBN: 0-07-913702-4. A detailed analysis of XML with a focus of structure and DTDs.

Light, Richard: *Presenting XML*. SamsNet 1997. ISBN: 1-57521-334-6 A good general introduction to XML and its relationship to SGML and HTML.

Server-Side Technology

Francis, Brian, et al.: *Professional Active Server Pages 2*. Wrox Press 1998. ISBN: 1-861001-26-6. A solid sequel to the first book (see the following entry), this one doesn't so much review the material in the first but expands on it.

Homer, Alex, et al.: *Professional Active Server Pages*. Wrox Press 1997. ISBN: 1-861000-72-3. Considered by many as *the* definitive reference on Active Server Pages, this provides a strong overview of the technology.

Dynamic HTML

Barta, Mike, et al.: *Professional IE4 Programming*. Wrox Press 1997. ISBN: 1-861000-70-7. One of the most complete references on programming Internet Explorer 4, including many of the component technologies, such as Direct Animation, Chat, NetMeeting, and Sprites.

Goodman, Danny: *Dynamic HTML, The Definitive Reference*. O'Reilly 1998. ISBN: 1-565924-94-0. Written by one of the greats in the computer field, this is a solid reference book summarizing the thousands of permutations of Microsoft and Netscape HTML implementations, covering HTML, CSS, the Document Object Model, and JavaScript.

Homer, Alex and Chris Ullman: *Instant IE4 Dynamic HTML*. Wrox Press 1997. 1-861000-68-5. My most dog-eared book. A handy reference to most facets of Dynamic HTML.

To access the contents, click the chapter and section titles.

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Previous | Table of Contents | Next

Part IV
Appendixes

Appendix A
Creating The Sample Database

Most of the examples in this book are from the sample database created for the book. The DDL and DML statements used to create the examples are included on the CD-ROM. Please note that different databases have slightly different syntaxes. The CD-ROM contains scripts for Oracle, Sybase SQL Anywhere, Microsoft SQL Server, and Microsoft Access. You might need to alter the statements somewhat to accommodate your own database's SQL dialect. Also, please note that dates are notoriously a pain in the neck to enter into the database. If you get an error while inserting a date value, you may need to either change the values in the SQL inserts or alter the way that your database reads dates. For instance, the default format for dates in Oracle is set when the database loads. In my own installation of Oracle, the default format is **'dd-mon-yyyy'** such as **'2-Apr-1999'**. You can usually change a session setting to alter your default date format in your individual session. In Oracle, the command is **ALTER SESSION SET NLS_DATE_FORMAT = 'mm/dd/yyyy'**.

You can either add the tables and data to an existing database or create a new database. Ask your database administrator for assistance if necessary. You should then create a user ID of **Coriolis**. You can do this with the following command:

```
GRANT CONNECT TO Coriolis IDENTIFIED BY Coriolis
```

In this example, the password is also **Coriolis**. Your database administrator might need to perform this action for you.

The actual commands are stored on the CD-ROM under the \SampData directory with file names such as Oracle.sql for Oracle and SQLAnywh.sql for SQL Anywhere. I recommend copying the files that you will use to your hard drive in case you need to edit any of them. When you are ready, log on as user ID **Coriolis** and use your database's facilities to read in and run the commands in the file. For Oracle (using SQL*Plus), the command would be

```
Start C:\Oracle.SQL
```

assuming that the file is stored on the root directory of the C: drive.

Table And Data Creation Commands

Most of the data created was randomly generated using an Excel spreadsheet, which is also included on the CD-ROM. If you examine the data, you will notice that it does not match what is on the database because each time the spreadsheet updates, all the random values change. There are 300 orders, 600 line items, 100 customers, 35 employees (19 female and 16 male), and 4 departments. The **Item** table lists 10 items in inventory but is a highly simplified design.

All the tables have appropriate integrity constraints defined.

I also included three small tables: **State**, **City**, and **Airport**. These small tables have no integrity constraints defined. Although the tables are meant to relate to one another, I purposefully gave them inconsistent data so you can experiment with defining your own integrity constraints and practice with outer joins. You will find duplicate rows as well as rows with no corresponding rows on other "related" tables (such as cities without states).

The following sample commands are for Sybase SQL Anywhere but are very similar to other databases. The most notable differences are in the DDL used to create tables: Oracle calls the **NUMERIC** data type **NUMBER** and the **VARCHAR** data type **VARCHAR2**. When you run these statements for the first time, you will get errors on the **DROP TABLE** commands since the tables do not yet exist. There is no harm.

```
DROP TABLE LOCATION ;

CREATE TABLE LOCATION
(LOC_ID         CHAR (CHAR (3) NOT NULL,
 LOC_NAME       VAR CHAR (CHAR (30) NOT NULL );
ALTER TABLE LOCATION
ADD CONSTRAINT PK_LOC_ID PRIMARY KEY (LOC_ID) ;

DROP  TABLE  DEPARTMENT ;

CREATE TABLE DEPARTMENT
(DEPT_NO           SMALLINT NOT NULL,
```

```
 DEPT_LOC_ID       CHAR (3) NOT NULL,
 DEPT_NAME         VARCHAR (30) NOT NULL );

ALTER TABLE DEPARTMENT
ADD CONSTRAINT PK_DEPT_ID PRIMARY KEY (DEPT_NO) ;

ALTER TABLE DEPARTMENT
ADD CONSTRAINT FK_DEPT_LOC FOREIGN KEY (DEPT_LOC_ID)
REFERENCES LOCATION (LOC_ID) ;

DROP TABLE EMPLOYEE ;

CREATE TABLE EMPLOYEE
(EMP_NO           SMALLINT NOT NULL,
 EMP_LNAME        VARCHAR (21) NOT NULL,
 EMP_FNAME        VARCHAR (15),
 EMP_SSN          CHAR (9),
 EMP_DOB          DATE,
 EMP_HIRE_DATE    DATE NOT NULL,
 EMP_TERM_DATE    DATE,
 EMP_SALARY       NUMERIC (9,2),
 EMP_DEPT_NO      SMALLINT,
 EMP_MGR_ID       SMALLINT,
 EMP_GENDER       CHAR (1),
 EMP_HEALTH_INS   CHAR (1),
 EMP_DENTAL_INS   CHAR (1),
 EMP_COMMENTS     VARCHAR (255) );

ALTER TABLE EMPLOYEE
ADD CONSTRAINT PK_EMP_ID PRIMARY KEY (EMP_NO)

ALTER TABLE EMPLOYEE
ADD CONSTRAINT FK_EMP_DEPT FOREIGN KEY (EMP_DEPT_NO)
REFERENCES DEPARTMENT (DEPT_NO) ;

ALTER TABLE EMPLOYEE
ADD CHECK (EMP_HEALTH_INS IN ('Y','N')) ;

ALTER TABLE EMPLOYEE
ADD CHECK (EMP_DENTAL_INS IN ('Y','N')) ;
ALTER TABLE EMPLOYEE
ADD CHECK (EMP_GENDER IN ('F', 'M')) ;

DROP TABLE CUSTOMER ;

CREATE TABLE CUSTOMER
(CUST_NO        SMALLINT NOT NULL,
 CUST_LNAME     CHAR (21) NOT NULL,
 CUST_FNAME     CHAR (15),
 CUST_ADDR1     CHAR (38),
```

```
 CUST_ADDR2     CHAR (38),
 CUST_ADDR3     CHAR (38),
 CUST_CITY      CHAR (21),
 CUST_ST        CHAR (2),
 CUST_COUNTRY   CHAR (21),
 CUST_PHONE     CHAR (10),
 CUST_FAX       CHAR (10),
 CUST_EMAIL     CHAR (24) );

ALTER TABLE CUSTOMER
ADD CONSTRAINT PK_CUST_NO PRIMARY KEY (CUST_NO) ;

DROP TABLE ITEM;

CREATE TABLE ITEM
(ITEM_NO       SMALLINT NOT NULL,
 ITEM_COST     NUMERIC (11,2) NOT NULL,
 ITEM_PRICE    NUMERIC (11,2) NOT NULL,
 ITEM_DESC     VARCHAR (30) NOT NULL) ;

ALTER TABLE ITEM
ADD CONSTRAINT PK_ITEM_NO PRIMARY KEY (ITEM_NO) ;

DROP TABLE ORDERS ;

CREATE TABLE ORDERS
(ORD_NO          SMALLINT NOT NULL,
 ORD_DATE        DATE NOT NULL,
 ORD_CUST_NO     SMALLINT NOT NULL,
 ORD_CURRENCY    CHAR (2) NOT NULL,
 ORD_EXCH_RATE   NUMERIC (12,6),
 ORD_TAX_LOCAL   NUMERIC (11,2),
 ORD_TAX_ST      NUMERIC (11,2),
 ORD_TAX_FED     NUMERIC (11,2),
 ORD_FREIGHT     NUMERIC (11,2),
 ORD_DISCOUNT    NUMERIC (11,2),
 ORD_TOTAL       NUMERIC (11,2) );
ALTER TABLE ORDERS
ADD CONSTRAINT PK_ORD_NO PRIMARY KEY (ORD_NO) ;

DROP TABLE LINE_ITEM;

CREATE TABLE LINE_ITEM
(LINE_ORD_NO     SMALLINT NOT NULL,
 LINE_NO         SMALLINT NOT NULL,
 LINE_ITEM_NO    SMALLINT NOT NULL,
 LINE_QTY        SMALLINT NOT NULL,
 LINE_PRICE      NUMERIC (11,2) NOT NULL,
 LINE_TOTAL      NUMERIC (11,2) NOT NULL );
```

```
ALTER TABLE LINE_ITEM
ADD CONSTRAINT PK_ORD_LINE_NO
PRIMARY KEY (LINE_ORD_NO, LINE_NO) ;

ALTER TABLE LINE_ITEM
ADD CONSTRAINT FK_LINE_ORD FOREIGN KEY (LINE_ORD_NO)
REFERENCES ORDERS (ORD_NO) ;

ALTER TABLE LINE_ITEM
ADD CONSTRAINT FK_LINE_ITEM FOREIGN KEY (LINE_ITEM_NO)
REFERENCES ITEM (ITEM_NO) ;

DROP TABLE STATE ;

CREATE TABLE STATE
(STATE_ID      CHAR (2) NOT NULL,
 STATE_NAME    VARCHAR (21) NOT NULL) ;

DROP TABLE CITY;

CREATE TABLE CITY
(CITY_ID       CHAR (3) NOT NULL,
 CITY_NAME     VARCHAR (21) );

DROP TABLE AIRPORT ;

CREATE TABLE AIRPORT
(AP_CITY_ID    CHAR (3) NOT NULL,
 AP_NAME       VARCHAR (21) NOT NULL );
```

Because of space constraints, the **INSERT** statements are not printed here. However, they are included on the CD-ROM.

The following constraint must be added after all of the rows are added to the **EMPLOYEE** table.

```
ALTER  TABLE  EMPLOYEE
ADD CONSTRAINT FK_EMP_MGR FOREIGN KEY (EMP_MGR_ID)
REFERENCES EMPLOYEE (EMP_NO) ;
```

After all of the rows have been inserted, the following commands will need to be run (they are on the CD-ROM as well).

```
UPDATE LINE_ITEM SET LINE_PRICE =
(SELECT ITEM_PRICE
 FROM ITEM
 WHERE ITEM_NO = LINE_ITEM_NO) ;

UPDATE LINE_ITEM
SET LINE_TOTAL = LINE_QTY * LINE_PRICE ;
```

```
UPDATE ORDERS SET ORD_TOTAL =
(SELECT SUM(LINE_TOTAL)
 FROM LINE_ITEM
 WHERE LINE_ORD_NO = ORD_NO) ;

UPDATE ORDERS
SET ORD_TOTAL = ORD_TOTAL + ORD_FREIGHT ;

UPDATE ORDERS SET ORD_FREIGHT = 0
WHERE ORD_TOTAL IS NULL ;

COMMIT WORK ;
```

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Appendix B
Differences Between Jet SQL And ANSI SQL

Microsoft Jet is mostly ANSI-89 SQL compliant but does vary from the ANSI standard in some key areas. The following sections outline most of the significant differences between Jet SQL and ANSI SQL. For the most part, Jet SQL is less restrictive than ANSI SQL. For example, when using the **BETWEEN** clause, ANSI SQL requires that the first operand be less than or equal to the second operand. Jet SQL makes no such demand.

Data Types

Table B.1 lists common ANSI SQL data types in the first column. The second column shows the Jet equivalent for each. The third column shows some of the variations you will see in other databases. For instance, although Sybase has a column data type **NUMERIC**, Oracle's equivalent is **NUMBER**. N/A indicates there is no equivalent. For example, Jet has a data type **CURRENCY** with no ANSI equivalent. If you need to guarantee portability between databases, you should avoid using data types where there are no ANSI or Jet equivalents.

Table B.1 ANSI SQL data types and their Jet equivalencies.

ANSI SQL	Jet SQL	Variations
BIT	N/A[1]	**VARBINARY**
N/A	**BIT**[1]	
N/A	**BYTE**	
N/A	**COUNTER**[2]	
DATE[3]	**DATETIME**	**TIMESTAMP**
N/A	**GUID**	

CHAR	**TEXT**	
DECIMAL	N/A	
DOUBLE PRECISION	**DOUBLE**	**FLOAT, FLOAT8, NUMBER, NUMERIC**
INTEGER	**LONG**	**INT**
N/A	**LONGBINARY**	**BLOB**
N/A	**LONGTEXT**	**LONG**
REAL	**SINGLE**	**FLOAT4**
SMALLINT	**SHORT**	**INTEGER4**

[1]*The Jet **BIT** data type is not the same as the ANSI **BIT** data type. Whereas the ANSI **BIT** is used to store binary data, Jet uses it to store No/Yes values (0 or 1).*
[2]*This **COUNTER** is used in much the same way as some databases use "Auto Incrementing" as an attribute of a column.*
[3]*Sybase uses separate **DATE** and **TIME** data types.*

The LIKE Clause

In ANSI SQL, you perform pattern or wildcard searching with the **LIKE** clause and the percent sign (%) and underscore (_) characters. The percent sign represents any number of characters, and the underscore represents exactly one character. The Jet equivalents are the asterisk (*) for any number of characters and the question mark (?) for single characters. For example, the following SQL **SELECT** statements search for any last name where the first letter is "M," the third letter is "c," and the letters "al" occur anywhere after that (such as "MacDonald").

ANSI:

```
SELECT EMP_LNAME
FROM EMPLOYEE
WHERE EMP_LNAME LIKE 'M_c%al'
```

Jet:

```
SELECT EMP_LNAME
FROM EMPLOYEE
WHERE EMP_LNAME LIKE 'M?c*al'
```

The BETWEEN Clause

The ANSI **BETWEEN** clause requires that the first comparison operand be less than or equal to the second comparison operand. Jet does not have this requirement. The following Jet **SELECT** is valid:

```
SELECT *
FROM EMPLOYEE
WHERE EMP_SALARY BETWEEN 50000 AND 30000
```

This query returns no rows using ANSI SQL.

COMMIT, ROLLBACK, And LOCK

ANSI SQL provides certain commands with no Jet equivalents. **COMMIT** is used to make permanent changes to the database, whereas **ROLLBACK** undoes all changes back to the beginning of the current transaction. In other words, Jet SQL has no transaction support. Likewise, ANSI SQL provides the **LOCK** and **LOCK TABLE** keywords to lock a table, preventing other users from updating it. Jet SQL has no equivalent.

If you peruse the Visual Basic documentation, you will undoubtedly notice that Microsoft directly contradicts my statements because it claims that Jet supports transactions through the DAO **BeginTrans**, **CommitTrans**, and **Rollback** methods. The key phrase here is "DAO methods." In other words, Visual Basic actually provides the transaction support via Jet. Is this a flaw? Nothing is a flaw if it works. However, supporting transactions in this way does put an additional burden on the client.

In Chapter 5, I discussed some of the implications of this support, such as the **Listen** method. **Listen** alerts the developer that a record currently being edited has been altered by another user. The basic sequence of events is something like the following:

1. VB client number one displays a record for edit.
2. VB client number two updates the record.
3. VB client number one looks at the database to determine if the record has been updated.
4. VB client number one determines the record has been changed and dumps the problem in the developer's lap.

This process is not really what client/server is all about. The client—and not the database—manages the transactions. Thus, I stand by my assertion that Jet has no true transaction support.

The PARAMETERS Keyword

Jet SQL supports the **PARAMETERS** keyword not supported by ANSI SQL. **PARAMETERS** allows you to create a query where search criteria are generated dynamically. You specify a variable name followed by a data type. Then, your **WHERE** clause references your variable, as shown in the following SQL **SELECT** statement:

```
PARAMETERS [LOW_SAL] CURRENCY, [HIGH_SAL] CURRENCY;
SELECT *
FROM EMPLOYEE
WHERE [EMP_SALARY] BETWEEN [LOW_SAL] AND [HIGH_SAL] ;
```

Essentially, the **PARAMETERS** statement creates variables for which your program must supply values. The following VB code shows this in action:

```
' The Parameters clause.
Dim sParm As String
sParm = "Parameters [Low_Sal] Currency, [High_Sal] Currency;"

' The SQL statement with Parameters
Dim sSelect As String
```

```
sSelect = sParm & "Select * From Employee " _
   & "Where [Emp_Salary] Between [Low_Sal] And "High_Sal];"

' The QueryDef object
' dbs is a previously created Database object
Dim qdf As QueryDef
Set qdf = dbs.CreateQueryDef ("Salary Search", sSelect)

' Supply the parameter values
qdf("Low_Sal") = 30000
qdf("High_Sal") = 50000

' Create a Recordset
Dim rstSal As RecordSet
Set rstSal = qdf.OpenRecordset(dbOpenSnapshot)

' Destroy the QueryDef object
dbs.QueryDefs.Delete "Salary Search"
```

TRANSFORM And PIVOT

Jet supports the **TRANSFORM** and **PIVOT** statements, which generate a cross-tab report that is otherwise impossible to produce in ANSI SQL. The following SQL statement is from the Northwind sample database provided with Visual Basic. It generates the report shown in Figure B.1.

```
TRANSFORM SUM(CCUR([ORDER DETAILS].[UNITPRICE] *
   [QUANTITY]*(1-[DISCOUNT])/100)*100) AS PRODUCTAMOUNT
SELECT PRODUCTS.PRODUCTNAME, ORDERS.CUSTOMERID,
   YEAR([ORDERDATE]) AS ORDERYEAR
FROM PRODUCTS
INNER JOIN (ORDERS INNER JOIN [ORDER DETAILS]
   ON ORDERS.ORDERID = [ORDER DETAILS].ORDERID)
   ON PRODUCTS.PRODUCTID = [ORDER DETAILS].PRODUCTID
WHERE (((ORDERS.ORDERDATE) BETWEEN #1/1/95# AND #12/31/95#))
GROUP BY PRODUCTS.PRODUCTNAME, ORDERS.CUSTOMERID,
   YEAR([ORDERDATE])
PIVOT "Qtr " & DATEPART("q",[ORDERDATE],1,0)
   IN ("Qtr 1","Qtr 2","Qtr 3","Qtr 4");
```

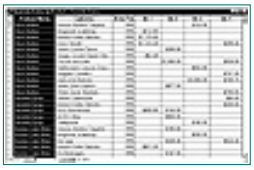

Figure B.1 A cross-tab report generated using Jet SQL.

Aggregate Functions

Jet SQL does support the standard ANSI SQL aggregate functions such as **SUM** and **MIN**. It also adds other aggregate functions such as **VAR**, **VARP**, **STDEV**, and **STDEVP**.

VAR returns the variance (the square of a standard deviation) for a set of data. **VARP** returns the variance of a given population from the database. **STDEVP** and **STDEV** calculate the actual standard deviations. **STDEVP** calculates the standard deviation of a population whereas **STDEV** evaluates the standard deviation of a population sample.

Jet does not permit the use of the **DISTINCT** operand within an aggregate function, such as **SUM(DISTINCT (Emp_Salary))**.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Appendix C
ODBC Functions

Open Database Connectivity (ODBC) is a standard by which drivers can be written to access different back-end databases. Most RDBMS vendors write their own drivers to access their own product via ODBC. Additionally, some vendors, such as Microsoft and Intersolv, market drivers that offer connectivity to a wide variety of back ends. Not all drivers are created equally. For instance, it is my experience that the Microsoft ODBC driver for Oracle is more functional than the one provided by Oracle, but that Oracle's driver is faster.

ODBC Architecture

When you use ODBC to access the back-end database, ODBC itself acts as both a traffic cop and interpreter between the database and the application. Most significant in this architecture is that ODBC is implemented as an API to which application programs speak. There may be (and usually are) intervening layers between your program and ODBC, shielding developers from the complexities of the API. For instance, VB developers can use Data Access Objects (DAO), Remote Data Objects (RDO), or ActiveX Data Objects (ADO) to avoid having to code the individual API calls. RDO is most tightly coupled with ODBC, whereas ADO relies on a Microsoft "access provider" to access ODBC.

The architecture of ODBC as it relates to the VB developer is shown in Figure C.1. When you use Data Access Objects, the DAO layer shown in the figure comes into play. Depending on whether the developer uses ODBCDirect, Jet may or may not get loaded. When you use ODBCDirect, Jet is bypassed and

RDO is loaded instead. However, when you use Remote Data Objects, only RDO is loaded. Thus, you may have either Jet or RDO acting as a layer between Visual Basic and the ODBC layer. With the ODBC API model, Visual Basic talks directly to the ODBC layer that, although somewhat more efficient than communicating through the RDO layer (and much more efficient than going through Jet), comes at the expense of the developer having to write all the ODBC API calls.

Figure C.1 The ODBC architecture.

One of the jobs of the ODBC API layer is loading and unloading various database-specific drivers. Which driver to load is determined by how the ODBC data source was defined in the ODBC Administrator applet in the Control Panel. Each driver announces to ODBC its capabilities, and ODBC translates (as much as possible) the API call to the driver. If the driver is incapable of handling the specific function, ODBC returns a “Driver Not Capable” error message to the application. ODBC is also responsible for converting data formats as necessary.

The driver itself is loaded when a **SQLConnect**, **SQLDriverConnect**, or **SQL-BrowseConnect** function is called. A driver may be *single-tiered* or *multiple-tiered*. Most nonrelational data sources (such as a dBase file) have single-tiered drivers. A single-tiered driver means that the driver directly manipulates the data in the file rather than passing control to another process. The range of ODBC functionality of these drivers is typically limited due to the nature of the data itself. For instance, you can define a plain text file as a data source, but you will typically be able to perform only **SELECT** and **INSERT** operations. More robust data sources are designed to be updated. A FoxPro data source, for instance, also supports **UPDATE** and **DELETE** operations but does not support some of the more advanced functions supported by RDBMS engines such as Oracle and Microsoft SQL Server.

A multiple-tiered driver does not interact directly with the data. Instead, it passes the request directly to the server, such as Oracle.

ODBC Conformance Levels

As stated earlier, ODBC has three levels of conformance. All drivers should support the “Core” or “Base” level functions defined by the SQL Access Group. The SQL Access Group also defines Level 1 and Level 2 extended functionalities. For a driver to be considered Level 1 (or Level 2) compliant, it must support *all* functions within that level. Sometimes a driver may support most but not all functions, omitting minor functions that may have no impact on your application. Though a driver may not be technically Level 2 compliant, the function(s) omitted may be so minor as to be irrelevant. When

choosing an ODBC driver, you should consider what level it conforms to and, if it is not wholly Level 1 or Level 2 compliant, whether the omissions are critical to your application.

In addition to the ODBC API conformance level, ODBC also defines an SQL conformance level that specifies the SQL grammar that must be supported at each level.

The following sections summarize the functional and grammatical conformance requirements of each level.

Core Level Conformance Requirements

- Allocate environment (including connection and statement handles)
- Free environment (including connection and statement handles)
- Execute SQL statements
- Retrieve a result set (query result) as well as information about that result set
- Retrieve error information
- Commit and roll back transactions
- **CREATE TABLE**
- **DROP TABLE**
- **SELECT**
- **INSERT**
- **UPDATE**
- **CHAR** and **VARCHAR** data types
- Basic functions

Level 1 Conformance Requirements

- Retrieve catalog information from the database
- Display driver capabilities information
- Provide driver-specific connection dialogs
- Retrieve a partial result set
- **ALTER TABLE**
- **CREATE INDEX**
- **DROP INDEX**
- **CREATE VIEW**
- **DROP VIEW**
- **GRANT**
- **REVOKE**
- All ANSI **SELECT** functionalities
- Subqueries
- Aggregate functions (such as **SUM**, **MIN**, **MAX**, and **AVG**)
- Numeric data types (such as **NUMERIC**, **SMALLINT**, and so on)

Level 2 Conformance Requirements

- Scrollable cursors
- Use native SQL (that is, use PL/SQL with Oracle)
- Retrieve extended catalog information (such as stored procedures and privileges information)
- List available data sources
- Send and receive arrays of parameters
- Outer joins
- **UNION**
- Positioned updates and deletes
- Scalar SQL functions (such as **LENGTH** and **SUBSTR**)
- Batch processing of SQL statements
- **DATE** and other data types

The CD-ROM includes all of the ODBC function declarations for Visual Basic, as well as comments about their purpose. Also included are the needed constants and a sample application.

Previous | Table of Contents | Next

Visual Basic 6 Client/Server Programming Gold Book
(Publisher: The Coriolis Group)
Author(s): Michael MacDonald and Kurt Cagle
ISBN: 1576102823
Publication Date: 10/01/98

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Table of Contents