9

DATABASES AND ADO.NET

If you've ever programmed internal projects for tracking customers, sales, payroll, or inventory, you've probably realized that data is the lifeblood of any company. For the Visual Basic programmer, this understanding is particularly relevant, because no other language is used as often to create database applications. In the early days of Windows programming, Visual Basic came to prominence as a simple and powerful tool for writing applications that could talk to a database and generate attractive reports. In some ways, this is still Visual Basic's most comfortable niche.

Over the years, Microsoft has given the world a confusing alphabet soup of database access technologies. Visual Basic programmers first started with something called DAO (Data Access Objects), later upgrading to RDO (Remote Data Objects) to access client-server database products such as SQL, and then migrated to ADO (Active Data Objects), which was supposed to provide the best of both worlds. In many ways, ADO fulfilled its promise, providing a flexible and powerful object model that could be used by programmers in just about any Windows-based programming language. In fact, ADO is still a bit of a newcomer, and has already fragmented into several versions. Some features, such as XML access and disconnected use, have been more or less slapped on as afterthoughts, and more letters have been added to the soup with additional technologies like RDS (Remote Data Service). However, it didn't take Microsoft long to throw in the towel once again and decide with .NET that what everyone needs is yet another entirely new way to access data.

And calling ADO.NET "entirely new" is only a modest exaggeration. While ADO.NET has some superficial similarities to ADO, its underlying technology and overall philosophy are dramatically different. While ADO was a connection-centered database technology that threw in some disconnected access features as an afterthought, ADO.NET is based on disconnected DataSets, and has no support for server-side cursors. While ADO was a "best of breed" standard Microsoft component built out of COM, ADO.NET is an inhabitant of the .NET class library, designed for managed code, and based entirely on XML for storing data. As you'll see in this chapter, ADO.NET is one more .NET revolution.

New in .NET

The .NET languages require a new database technology. ADO, the previous standard, was wedded to COM, and every interaction between .NET-managed code and ordinary code (such as that in a COM component) suffers an automatic performance hit. The surprise is that ADO.NET is not just a .NET version of ADO. Instead, ADO.NET has been redesigned from the ground up.

No Support for Cursors

In ADO, a cursor tracks your current location in a result set, and allows you to perform updates on live data. Cursors have completely disappeared in ADO.NET. They are replaced by a new disconnected model that doesn't maintain connections.

XML-Based Data Storage

In ADO, XML data access is an afterthought. ADO.NET, however, uses XML natively as its own internal format for storing data. In most cases, you won't need to explore the XML internals when dealing with relational data. On the other hand, you might put them to good use by storing data locally in an XML file, or transferring XML data to a component on another computer or operating system.

DataSets and DataReaders Replace Recordsets

Recordsets were the focal point of ADO, but they had some significant drawbacks. ADO.NET replaces the Recordset with the DataSet, which can contain more than one table of data as well as additional information about table relations. For situations where you need fast read-only access, and you don't want to hold onto information for longer than a few seconds, ADO.NET provides a special DataReader object.

New DataAdapters

In ADO, Recordsets were usually directly connected to a data source. In the connectionless world of ADO.NET, DataSets don't directly relate to any data source. Instead, you use a special DataAdapter to pull information out of the database and pop it into a DataSet. You also use the DataAdapter to submit DataSet changes back to the database when you're done.

Introducing ADO.NET

There are essentially two ways that you can use ADO.NET in your applications. First of all, you can use it entirely on its own to create tables and relations by hand. You can then easily store this information in an XML file using ADO.NET's built-in capabilities, and retrieve it later to work with it again. This way of using of ADO.NET, with a dynamically created data file, is described later in this chapter. It provides a simple, no-frills approach to creating and storing single-user information.

However, it's much more likely that you'll use ADO.NET to interact with an underlying database. This database might be a stand-alone Access database, or it might be a multi-user RDBMS (Relational Database Management System), such as Microsoft's SQL Server. No matter what your data source, the way you use ADO.NET will be essentially the same. You can still create your own tables and write an XML file representing the data you retrieve, but your ultimate goal will be to commit any modifications back to the database.

Using Relational Data

ADO.NET picks up where simple file and XML document access end. These techniques, which were examined in Chapter 8, are ideal for simple, small portions of well-contained information. ADO.NET, on the other hand, excels when it comes to dealing with *relational data*: information that is broken down into several separate tables, which are then linked together.

For example, consider an application for tracking staff work hours. In a simple form, this application might be built out of three tables: Employee, Location, and WorkLog. The Employee table would contain all the information related to a specific individual. The Location table would store the information about a work environment. The WorkLog would record a number of hours worked, along with a reference to the appropriate Employee (through an EmployeeID field) and the Location where the work was done (through a LocationID field). This relationship would make it easy to manipulate the data to find out such information as the total number of hours worked at a specific location or by a specific employee. It also wouldn't waste any space, or invite potentially conflicting information by storing employee-specific information (such as the person's name) or location-specific information (such as the address) in the WorkLog table.

The structure would look like Figure 9-1.

Figure 9-1: Table relations

This, of course, is the typical way you store information in any relational database, whether it is SQL Server, Oracle, MySQL, or something else. Usually, you use a combination of SQL statements and *stored procedures* (miniature programs added to your database) to manage this information. ADO or ADO.NET handles the processes of sending commands to the database and retrieving any resulting information, including the actual data rows, so that you can display them or work with them in your application. Of course, both ADO and ADO.NET allow you to connect to other data sources—even sources that may not be truly relational. But in the majority of cases, you will use ADO or ADO.NET to access a standard database.

The Northwind Database

Incidentally, all the examples in this chapter use tables from the Northwind database, a sample database included with SQL Server to help you test database access code. If you have SQL Server, but for some reason you are missing the Northwind database, you can use the script included with the online samples to install it automatically. You can also use this script to install the Northwind database for MSDE (Microsoft Database Engine).

MSDE is one of Microsoft's best-kept secrets—it's essentially a scaled-down version of SQL that's bundled with Visual Studio .NET. It's completely compatible with SQL Server (in fact, it *is* SQL Server), and free to use for anyone, provided they don't need to support more than five simultaneous database connections. If a client installs MSDE and then wants to upgrade to a system that supports more simultaneous users, SQL Server can be easily installed, and the MSDE databases can be imported in a snap. The only catch is that MSDE, on its own, doesn't provide the nice graphical tools that help you build tables and relations, monitor performance, and perform general database upkeep. To remedy this problem, Microsoft has added some helpful developer database utilities to Visual Studio .NET. Unfortunately, this is a chapter about programming with databases—not how to use database management applications like SQL Server

and MSDE. Although I don't have the space to tell you everything you need to know to install and configure MSDE, you can find some information at Microsoft's MSDE site (http://msdn.microsoft.com/vstudio/msde).

TIP *In the Visual Studio .NET setup, even if you select to install MSDE you must complete its setup after Visual Studio .NET is installed. To do so, manually launch the setup.exe program from the \Setup\MSDE\ subfolder under the main Visual Studio .NET installation folder.*

If you have another database product, or you don't want to install the Northwind database, you can tweak the examples to use another data source. Usually you'll only have to adjust the connection string and the names of the tables and fields. Bear in mind, however, that some data sources, such as Microsoft Access databases, don't support all the features we'll discuss (stored procedures, for instance).

SQL Server and OLE DB

Many of the ADO.NET classes are provided in two separate flavors (see Figure 9-2). The standard version works like traditional ADO, and accesses data through something called an OLE DB provider. OLE DB is the layer that lets the data source talk to ADO or ADO.NET. OLE DB providers exist for most relational data products, including Microsoft's own SQL Server.

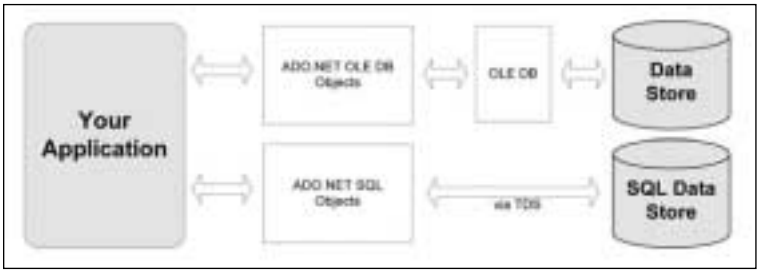

Figure 9-2: Two ways to connect with ADO.NET

NOTE *The only OLE DB provider that ADO.NET can't use is the ODBC provider. Instead, Microsoft plans to make a pre-release managed provider for ODBC available at their web site. I add this as a useful tip if you need it; everyone else should disregard this alphabet soup of database-related acronyms.*

So OLE DB is our old, trusty standard. However, SQL Server also has its own customized .NET provider that's optimized for performance. It interacts directly with SQL Server's Tabular Data Stream (TDS) and bypasses the extra COM and OLE DB layers. To summarize, you use the standard OLE DB providers for all data access in ADO.NET, except when using SQL Server. You could connect to SQL Server using OLE DB, but it wouldn't change the available features you could use, and it would probably slow down performance slightly because of the extra layers.

In the future, other database products will add their own .NET-managed providers for better performance (and some may even be included natively with the .NET framework). Don't worry: The providers for different databases derive from the same classes, implement the same procedures (with the same methods and properties), use the same DataSet object, and work almost exactly the same.

The Basic ADO.NET Objects

All of the ADO.NET features are provided through types in the System.Data branch of the .NET class library. This includes the following five namespaces:

- **System.Data** contains the fundamental classes for managing data, such as DataSet and DataRelation. These classes are totally independent of any specific type of database.
- **System.Data.Common** contains some base classes that are inherited by other classes in the System.Data.OleDb and System.Data.SqlClient namespaces. Essentially, the classes in this namespace specify the basic functionality, while the classes in the other namespaces are customized based on the data source. Thus, you don't use the System.Data.Common classes directly.
- **System.Data.OleDb** contains the classes you use to connect to OLE DB provider, including OleDbCommand and OleDbConnection.
- **System.Data.SqlClient** contains the classes you use to connect to a Microsoft SQL Server database using the optimized TDS (Tabular Data Stream) interface. This includes such classes as SqlDbCommand and SqlDbConnection, which look and act almost exactly the same as their OLE DB counterparts.
- **System.Data.SqlTypes** includes additional data types that aren't provided in .NET, but are used in SQL Server. These include SqlDateTime and SqlMoney. These types can be converted into the standard .NET equivalents, but the process introduces the possibility of a conversion or rounding error that might adversely affect data. Instead, you can create objects based on the structures defined in this class. It might even increase speed a bit, as no automatic conversions will be required.

This chapter uses SQL Server and the associated System.Data.SqlClient namespace. The techniques described here are, without exception, identical for OLE DB providers.

To get off to a good start, you should import the two namespaces you need to use:

```
Imports System.Data            ' Provides common classes like DataSet.
Imports System.Data.SqlClient  ' Use System.Data.OleDB instead if using
                               ' an OLE DB provider.
```

Fast-Forward Read-Only Access

There are two ways that you can access data with ADO.NET: as a read-only stream of information, or as a disconnected DataSet that you can examine and manipulate long after the database connection has been closed. In a sense, ADO.NET poses a difficult question that forces you to choose between two dramatically different approaches. The common middle ground found in ADO programming—a live read-write cursor that maintains a connection—just isn't an option in ADO.NET

In the first section of this chapter, we'll explore data access the easy way, and temporarily avoid the thorny issues of disconnected data. As you read ahead, you might want to refer to the diagram in Figure 9-3, which shows you the basic model for this simple type of data access.

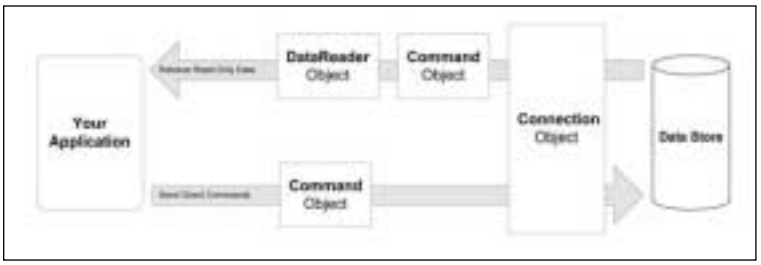

Figure 9-3: Using ADO.NET without disconnected data

Our first task is to introduce the basic ingredients for any type of data access: Connections and Commands.

Connection Objects

You use a connection object to establish a connection to a data source. The only trick is using the right connection string.

The Connection String

The connection string specifies all the information that the database needs in a single line of text. It consists of a string of named parameters and values (for example, `user id=sa`), each of which is separated by a semicolon.

Below is a sample connection string:

```
Connection = "Data Source=localhost;Initial Catalog=Northwind;user id=sa"
```

This connection string specifies a connection to the SQL Server installation on the local computer (rather than a remote network server), and identifies the database that you want to use (Northwind). It also connects with the user ID *sa*, which is the default system administrator account provided with SQL Server.

If this connection string doesn't work for you, it may be because the *sa* account has been modified since installation, and now requires a password (which is always a good practice). You can specify the password as one additional parameter in the connection string, as in `password=opensesame`.

If you are using SQL 2000, you may have disabled SQL authentication in favor of Windows integrated security, which allows you access based on the currently logged-on user. In this case, you can use a connection string with the following format:

```
Connect = "Data Source=localhost;Initial Catalog=Northwind;" & _
  "Integrated Security=SSPI"
```

In order for this to work, the currently logged-on Windows user must have the required authorization to access the SQL database.

And just for variety, here's how a connection string might look for an OLE DB provider using the OleDbConnection object. The only real difference is the addition of the Provider setting, which identifies the appropriate OLE DB provider. In the following example, the connect string is pointing to the SQL Server OLE DB provider, which you generally won't use.

```
Connection = "Provider=SQLOLEDB.1;Data Source=localhost;" & _
  "Initial Catalog=Northwind;user id=sa"
```

Other providers include MSDAORA (the OLEDB provider for an Oracle database) and Microsoft.Jet.OLEDB.4.0 (the OLEDB provider for Access).

There are several other options you can set for a connection string, and they are all documented in the .NET help files (under the SqlConnection and OleDbConnection references). These options include parameters you can use to specify a few other pieces of information, such as how long you'll wait while trying to make a connection before timing out.

Making a Connection

Once you have created the right connection string, it's easy to create a connection object and establish a live connection. For SQL Server, you use the SqlConnection object. (For an OLE DB provider, you would use the OleDbConnection object.)

It works like this:

```
Dim Connect As String = "Data Source=localhost;Integrated Security=SSPI;" & _
  "Initial Catalog=Northwind;"
Dim con As New SqlConnection(Connect)
con.Open()
```

You can close your connection at any time by calling the Close method.

NOTE *As illustrated in the preceding example,, you don't really use a Connection object. Instead, you use the appropriate derived class (SqlConnection or OleDbConnection). In cases where there is more than one flavor of a class, as with the Connection object, this chapter introduces them with a common (yet fictitious) object name.*

Command Objects

Command objects represent the SQL statements or stored procedures that you can use to retrieve data or submit changes. In order to use ADO.NET, you should have a basic understanding of the SQL language.

SQL Statements

Here's an SQL statement at its simplest. It's used to retrieve data from a single table in the current database (in this case, the Orders table). The asterisk (*) indicates that we want to retrieve all the fields from the table (including OrderID, CustomerID, and so on).

```
SQLString = "SELECT * FROM Orders"
```

Seasoned database programmers will shudder when they see this statement, because it doesn't limit the number of returned records in any way. In other words, it selects all the records in the Orders table, whether there are fifty or five million of them. This type of statement typically goes into an application that works well when it is first deployed, but gradually slows to a crawl as the number of records in the database climbs. Eventually, you may even receive timeout errors.

A safer SQL statement might look like this:

```
SQL As String = "SELECT * FROM Orders " & _
 "WHERE OrderDate < '2000/01/01' AND OrderDate > '1987/01/01'"
```

The Where clause ensures that only certain rows are retrieved. In this case, the OrderDate column must have a value between the two specified dates. (This example uses the international standard date format *yyyy/mm/dd,* which is typically a safe representation.) The values being compared are enclosed in single quotes, unless they are numbers. The And keyword allows us to use more than one criterion in our SQL statement.

Creating a Command

Provided you have a connection, a command is easy to define. If you are using an SQL Server database, you will use the SqlCommand object. OLE DB providers require the OleDbCommand object.

```
Dim SQLString As String = "SELECT * FROM Orders " & _
 "WHERE OrderDate < '2000/01/01' AND OrderDate > '1987/01/01'"
Dim cmd As New SqlCommand(SQLString, con)
```

This example creates an SqlCommand object using a constructor that lets us specify the connection that must be used, and the statement that will be used to select records. We could accomplish the same thing in the more verbose format of our next example. (Like ADO, ADO.NET often provides many paths toward accomplishing the same task.)

```
Dim cmd As New SqlCommand()
cmd.CommandText = SQLString
cmd.Connection = con
' Strictly speaking, the next line isn't required because
' CommandType.Text is the default.
cmd.CommandType = CommandType.Text
```

Notice that these examples don't actually read any data; all either of them does is define a command. In order to use the command, you have to decide whether you want to create a simple DataReader or a full-fledged disconnected DataSet.

DataReader Objects

A DataReader is the equivalent of a *firehose cursor,* which gets its name from the fact that it provides a steady stream of one-way data pouring from the database straight into your program, with few additional frills. A DataReader doesn't provide disconnected access, or any ability to change or update the original data source. You should use a DataReader whenever you need quick, read-only data, as its performance will always beat the full-fledged DataSet. On the other hand, the DataReader provides few features.

There are two types of DataReader objects: SqlDataReader (optimized for SQL Server databases) and OleDbDataReader (for OLE DB providers). Once you've made a connection and defined a Command, you can access the data through a reader.

The heart of our ADO.NET DataReader programming is contained in a single line:

```
Dim reader As SqlDataReader = cmd.ExecuteReader()
```

This defines an SqlDataReader object, and creates it using the ExecuteReader method of our Command object. In order for this to work, you must have already used the code from our earlier examples to create the connection and command that we are using here.

Once you have the reader, you can move through the rows from start to finish, interacting with each row one at a time. This is similar to the way you interacted with a Recordset in ADO, but instead of the MoveNext method you use the Read method. This method returns True as long as there is a row of data at the current position. Once you move past the last row, it changes to False.

The following loop moves through all the rows in the reader. Notice that there is no way to move backward.

```
Do While reader.Read()
    ' Process current row here.
Loop

reader.Close()
con.Close()
```

This loop invokes the reader.Read method in each pass. When the reader has read all the available information, the method will return False, the While condition will evaluate to False, and the loop will end gracefully. Keep in mind that you have to call the Read method before you start processing a row. When the reader is first created, there is no current row.

TIP *Remember, DataReaders maintain a live connection, so you should process the data and close your connection as quickly as possible.*

To access the actual data, you use a field name or an index number (which starts counting at zero). Usually, a field name will be the clearest option, but the index number provides an easy way for you to make sure you use every column in the row.

```
' Put this code inside the reader loop:
lstOrderID.Items.Add(reader("OrderID"))  ' Adds the data from the OrderID field.
lstOrderID.Items.Add(reader(0))          ' The same thing, using the field index.
```

This is just about all that a DataReader allows you to do.

A ListView Example

Our next example uses an SqlDataReader object to move through the returned records and place some basic information into a ListView control (see Figure 9-4). The code looks more complicated because the ListView control requires some special considerations.

The first step is to set up the ListView control so that it displays a multicolumn list:

```
lvOrders.View = View.Details
```

Now we can add a column for every field in the DataReader. There is no quick and easy way to get the field names, so we stick to numbers. FieldCount is one of the few properties provided by the SqlDataReader object, so we can use that to determine how many columns are required:

```
Dim i As Integer
For i = 0 To reader.FieldCount - 1
    lvOrders.Columns.Add("Column " & (i + 1).ToString, 100, _
     HorizontalAlignment.Left)
Next
```

Now that the initial setup is complete, the actual data can be added:

```
Do While (reader.Read())
    Dim NewItem As New ListViewItem()
    NewItem.Text = reader(0)

    For i = 1 To reader.FieldCount - 1
       If reader(i) Is DBNull.Value Then
            NewItem.SubItems.Add("")
        Else
            NewItem.SubItems.Add(reader(i).ToString())
        End If
    Next i

    lvOrders.Items.Add(NewItem)
Loop
```

This example looks more complicated then it actually is. The code is broken into two portions because of the way that the list control works. Every ListView control contains a collection of items. (In our example, each item represents an OrderID.) To put information into additional columns, you have to add subitems to each ListView item.

This example also checks for a null value in the field. In a database, a null value indicates only that the field is empty, and that no information has been entered. You can't change a null value into a string, however, so the Add method will fail if it's used to add a field that contains a null value. The output for this example is shown in Figure 9-4.

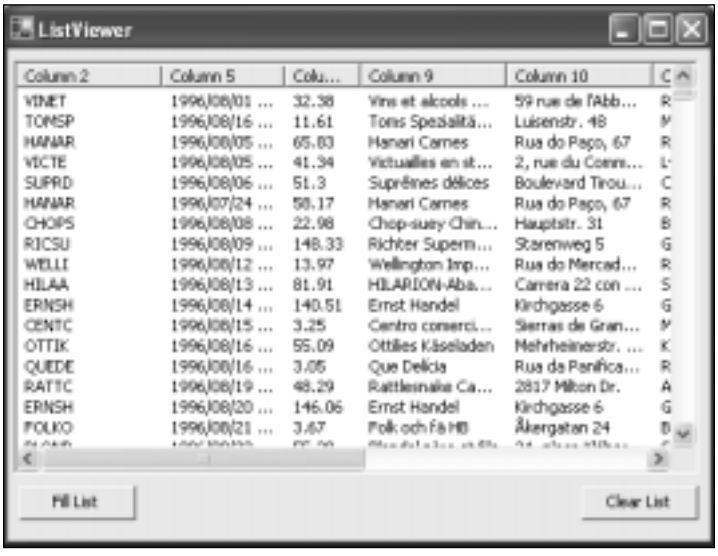

Figure 9-4: Filling a ListView

If you accomplish everything you need to with a DataReader, it's always a good choice. If you need more sophisticated data manipulating abilities, you'll need to step up to the DataSet object, which we'll explore a little later in the chapter.

Updating Data with the Command Object

The DataReader object provides a simple way to pull a stream of data out of a database, but it doesn't allow us to make any changes. What can a programmer do to modify a database? For example, maybe your program needs to record events or transactions without bothering the user with the details. In this case, you need to be able to add a new record to a table in the original database. Or, maybe your program is a more direct database application that presents a record of information, and then allows the user to specify changes. In this case, you need to modify an existing record.

Traditional ADO programming was more flexible than ADO.NET—perhaps too flexible. To update a record, a programmer would usually create a Recordset, select the one relevant row, and then use a direct cursor connection to change values as needed. To add a new record, the programmer would follow more or less the same process involved in creating a Recordset, maybe even selecting some existing records, and would then use the Recordset to add a new row. These techniques won't work in ADO.NET, and it may be worth asking if they were ever a good idea in the first place.

In ADO.NET, you have two options for updating data:

- Updating it directly with a customized Command
- Creating a DataSet, implementing the changes there, and then committing them to the original data source

The next section examines the easier of these two methods: using a Command object.

Why Use a Command Object?

If your application makes relatively straightforward changes, the best way to update the data source is to use a Command object. This object can contain a special SQL statement (such as Update or Delete), or it can run a stored procedure that exists in your database.

Generally, a Command object provides the most straightforward and uncomplicated way to make a change. The only drawback is that it can require some extra work (and code) if your application allows the user to make substantial, varied modifications. This approach—using a Command object to go straight to the data source—was available in ADO. However, ADO developers often fell back on the Recordset object because it was easier to code a solution

"on the fly"—in other words, with little planning or forethought. The advantages of the Command approach are:

- No data is returned from the database. After all, why waste time creating a Recordset, and then using a Select statement to put some information into it, if you don't need to?
- If there's a problem following your instruction, you'll know it right away. (With disconnected data and DataSets, however, several changes are usually made at once; this makes it harder to track down what's failed and identify the cause.)
- As long as you understand the SQL language (or have a helpful stored procedure), using Command objects is the most straightforward technique for making changes. Disconnected DataSets require extra planning, because they can apply changes in an unpredictable order, thus causing problems with linked tables.

A Data Update Example

Here's an example that uses a Command object to perform an update operation. For this type of operation, you use the Command object directly, with the help of its ExecuteNonQuery method:

```
' Create connection.
Dim Connect As String = "Data Source=localhost;Integrated Security=SSPI;" & _
 "Initial Catalog=Northwind;"
Dim con As New SqlConnection(Connect)
con.Open()

' Create a silly update command.
Dim SQL As String = "UPDATE Orders SET ShipCountry='Oceania' " & _
  "WHERE OrderID='10248'"
Dim cmd As New SqlCommand(SQL, con)

' Execute the command.
Dim NumAffected As Integer
NumAffected = cmd.ExecuteNonQuery()
con.Close()

' Display the number of affected records.
MessageBox.Show(NumAffected.ToString & " records updated", "Results", _
  MessageBoxButtons.OK)
```

This code creates the standard Connection and Command objects, and then executes the command directly. The update command is an SQL Update statement that finds the record with the OrderID 10248 (which exists in the default Northwind database) and changes the ShipCountry value to Oceania. This

record will be updated even if ShipCountry has already been changed, provided it can be found. The message you will see is shown in Figure 9-5.

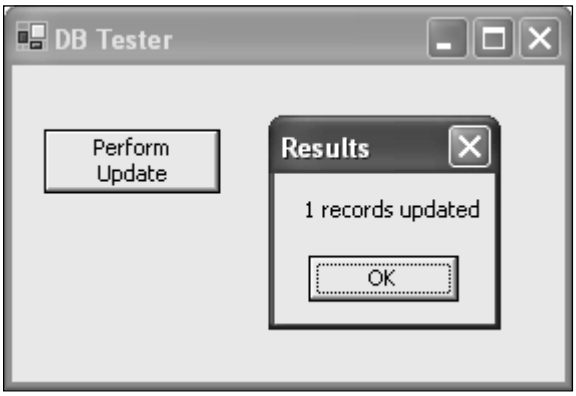

Figure 9-5: A simple update test

NOTE *The preceding example uses a fairly straightforward SQL Update statement. However, if you aren't familiar with SQL, it may take a little getting used to. Unfortunately, the SQL language is beyond the scope of this book (although the final section of this chapter will point you to some excellent resources).*

Some Real-World Changes

In a more realistic example, the Update statement would be created dynamically using another variable or a control property, as shown here:

```
Dim SQL As String = "UPDATE Orders SET ShipCountry='" & lstCountry.Text & _
 "' WHERE OrderID='" & intCurrentOrder & "'"
```

Be careful to include the single quote marks for non-numeric data! Generally, writing an SQL statement like this in your program is extremely bad form, even if it is generated dynamically. The best alternative is to go through another class that has the appropriate SQL values stored as constants, or to use a function that creates the SQL statement for you based on the corresponding parameters. Either option will make your code much more readable.

For example, a function might use this type of approach:

```
Dim cmd As New SqlCommand(GetSQLToUpdateCountry(OrderID, lstCountry.Text), con)
```

In this case, GetSQLToUpdateCountry is used right in the declaration for the command. It returns the Update string using the specified country and order ID. (You might want to streamline this example with a shorter function name.)

You might find that you are using some common SQL statements repeatedly. If so, a much better approach might be to make them enumerated values.

You could use a helper class, like this:

```
Public Class NorthwindSQL
    Public Enum Queries
        GetAllOrders
        GetAllCustomers
    End Enum

    'A shared method makes this method available even without a live instance.
    Public Shared Function GetSQL(ByVal Query As Queries) As String
        Select Case Query
            Case Queries.GetAllCustomers
                Return "SELECT * FROM Orders"
            Case Queries.GetAllOrders
                Return "SELECT * FROM Customers"
        End Select
    End Function
End Class
```

TIP *Remember, you can access the enumerations in any class by using the class name, even if you haven't created an instance of that class.*

You could use the NorthwindSQL class to create a Command that will select all the records from the Orders table, like this:

```
Dim cmd As New SqlCommand(NorthwindSQL.GetSQL(NorthwindSQL.GetAllOrders), con)
```

At this point, our code savings may not seem that great. But consider some of the advantages:

- In a real application, SQL statements may be much longer and more complex. For example, they will usually specify only the required column fields (rather than using the asterisk (*), which can slow down your application by requesting data that it doesn't need). This list of fields can be quite lengthy—and easily mistyped!
- In a real application you will probably need to set additional information—such as a limiting date range, or other options—for the Where clause. These values can be appended to the returned SQL string, or they can be added by the helper class, provided that you add some extra parameters to your function. Alternatively, you could give your class properties that record extra information (such as the required date range). Then the information would only need to be set once, and could be reused for multiple different commands.
- By having all the SQL statements in one class, you can easily update your application when the database changes.

A Transaction Example

A *transaction* allows you to execute several commands at once, and to be guaranteed that they will all succeed or fail as a unit. The basic principal of a transaction is that if any of the actions in it fails, the whole process is "rolled back" to its initial state. You can appreciate the value of this system if you have ever used an instant bank machine. If, after requesting a withdrawal, the bank machine failed and could not give you any money, you would not be happy if it still deducted the amount from your bank account. In other words, withdrawing money is a transaction made of two steps: your bank account being debited, and you receiving your money. Neither one of these steps should happen without the other.

A database often uses transactions in its stored procedures. They can save you the trouble of coding extra database logic inside your application, and can help separate the basic data management code from the rest of your application. However, in some cases this is not convenient, and you need to be able to create a transaction programmatically in your VB .NET code. To do this, you create a transaction object (either an SqlTransaction or an OleDbTransaction), and use the BeginTransaction method of the Connection object.

To start the process, you need to create and initiate the transaction:

```
' (The code to create the standard Connection and Command objects is left out.)
' Don't need to use New, as this object will be created for us.
Dim tran As SqlTransaction

' Create the transaction and assign it to our Transaction object.
tran = con.BeginTransaction()
```

Now the transaction exists, but it includes no Command objects. To make a Command object a part of this transaction, you can set its Transaction property. Assuming that we've already created two Command objects (cmdOne and cmdTwo), it works like this:

```
cmdOne.Transaction = tran
cmdTwo.Transaction = tran
```

These commands can be executed in the normal way:

```
Dim NumAffected As Integer
NumAffected = cmdOne.ExecuteNonQuery()

' Add the rows affected for the second query to find the total number of rows
' affected by both statements.
NumAffected += cmdTwo.ExecuteNonQuery()
```

However, the changes won't be permanently made to the data source until you commit them:

```
tran.Commit()
```

Alternatively, you can use the Rollback method to reverse changes, and set the data source back to its original state. Usually, you would use the Rollback method in response to an error, as shown here:

```
' Define Connection, Command, and Transaction objects here.
Try
    ' Start the Transaction, execute the Commands, and perform any other
    ' related code here.
    tran.Commit()
Catch err As Exception
    tran.Rollback()
End Try
```

An Example of a Stored Procedure

Stored procedures are miniature programs stored inside a relational database. A typical stored procedure consists of a number of SQL statements that can perform various tasks (such as selecting, updating, inserting, and deleting data).

Here's a sample stored procedure used to add a new customer record to the Customers table. It's not present in the default SQL Server Northwind database, so before you can use the procedure, you have to use the Enterprise Manager to add it. (It will already be added if you used the script provided with this chapter's examples to install the database.)

```
CREATE PROCEDURE AddNewCustomer
    @CustomerID varchar(5), @CompanyName varchar(40), @ContactName varchar(30)
AS INSERT INTO Customers(CustomerID, CompanyName, ContactName)
    VALUES(@CustomerID, @CompanyName, @ContactName)
```

This code looks quite different than anything you could write in VB .NET. The basic details are as follows:

- The procedure is called AddNewCustomer.
- The procedure uses three variables, which are defined in the second line. All SQL variables are identified with an @ symbol at the beginning of their names. The data types for these variables are also unfamiliar to the VB programmer. Varchar is the SQL-specific version of a string (and the number in brackets is its maximum character length).

- The third line features an Insert command, which adds a new row to the Customers table. It also inserts values for the three named fields (using the information on the next line, which is the list of variables). All other values will be left empty—in fact, they will have null values.

You can use the Command object to execute a stored procedure in the same way you would execute an SQL statement. However, there are two differences. First of all, for maximum efficiency, you should set the Command object's CommandType to CommandType.StoredProcedure:

```
' (Code to create a connection omitted).
Dim cmd As New SqlCommand("AddNewCustomer", con)
cmd.CommandType = CommandType.StoredProcedure
```

Secondly, if your stored procedure requires parameters with additional information (as the AddNewCustomer procedure does), you need to create a few OleDbParameter or SqlParameter objects.

Here's how you would create parameters to use with the AddNewCustomer procedure:

```
' Create an SqlParameter object. We don't use the New keyword here because this
' object will only be used to hold a reference to the parameter created with
' the Command object.
Dim param As SqlParameter

param = cmd.Parameters.Add("@CustomerID", SqlDbType.VarChar, 5)
param.Value = "*TEST"
param = cmd.Parameters.Add("@CompanyName", SqlDbType.VarChar, 40)
param.Value = "No Starch Press"
param = cmd.Parameters.Add("@ContactName", SqlDbType.VarChar, 30)
param.Value = "Matthew MacDonald"
```

When a parameter is added, you must specify information about its data type, along with the name of the corresponding stored procedure variable. You can then assign the value of the Parameter object to set the actual information that will be sent.

Once you've added the parameters and assigned the appropriate values, you can execute the stored procedure:

```
Dim NumAffected As Integer
NumAffected = cmd.ExecuteNonQuery()
con.Close()
```

The new record will be added, as shown in Figure 9-6.

Figure 9-6: The inserted record

Drawbacks to Stored Procedures

Stored procedures are a both a blessing and a curse. Their proponents point out that they let you remove database code from your application, control security (by restricting direct access to the tables, but allowing access to permitted stored procedures), and improve performance (because stored procedures can be pre-compiled, unlike the dynamically-generated SQL statements in your program). All of these advantages are true, and together they mean that you will probably want to use stored procedures in any large, professional business application.

However, stored procedures also have some drawbacks. For one thing, they are based on script-like blocks of code rather than on real objects, which are generally nicer to work with. More importantly, the parameter-based system of using stored procedures requires that you add a lot of extra information into your application about the structure of your database. In reality, your program should not care what the specific data type or string length of a parameter is, as long as it can guarantee that it is submitting a valid value. (And you may have already noticed that SQL statements have all their information in a simple string.) However, when you create a Parameter object, you have to specify all this information, essentially importing database-specific details into your application.

Once again, the best way to handle this is to use an intermediate class to "wrap" the database functions and database-specific details. For example, you could create a function that adds all the parameters for a specific stored procedure, using the values you supply, along with the database-specific information about data types stored in the class.

The following example uses a method to add the corresponding parameters and execute the command:

```
NorthwindProcedures.AddCustomer(cmd, "*TEST", "No Starch Press", "Matthew")
```

The database class looks like this:

```
Public Class NortwindProcedures

    Public Shared Function AddCustomer(cmd As SqlCommand, Customer As
String, _
      Company As String, Contact As String) As Integer

        Dim param As SqlParameter
        param = cmd.Parameters.Add("@CustomerID", SqlDbType.VarChar,
5)
        param.Value = Customer
        param = cmd.Parameters.Add("@CompanyName", SqlDbType.VarChar,
40)
        param.Value = Company"
        param = cmd.Parameters.Add("@ContactName", SqlDbType.VarChar,
30)
        param.Value = Contact

        ' Execute the Command and return the number of affected rows.
        Return cmd.ExecuteNonQuery()
    End Sub

End Class
```

Stored procedure design can become much more complicated. A stored procedure might perform multiple tasks, and may even return a result set (in which case you need to grab the results with a DataSet). A stored procedure can also send additional information to your program using output parameters. By default all the Parameter objects you add are for input parameters. However, this is easy to change:

```
param.Direction = ParameterDirection.Output
```

When using an output parameter, you don't set the Value property. Instead, you read from the Value property after the Command object has been executed to retrieve your results.

Using DataSet Objects

In many ways, the DataSet object is the focus of ADO.NET programming. Unlike the data reader, a DataSet is disconnected by nature. You can place information into a DataSet, or move information from a DataSet back into a relational database, but the DataSet itself never maintains a connection with a data source—in fact, it doesn't even have a connection property. To shuffle information back and forth between a DataSet object and a data source, you need to use a special DataAdapter object. Figure 9-7 shows how all the ADO.NET objects interact for disconnected access.

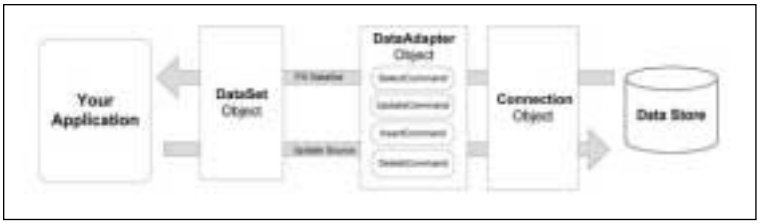

Figure 9-7: Disconnected data access with ADO.NET

When Should I Use a DataSet Object?

A DataReader provides the best possible performance. In general, you should always use a DataReader, unless you need the advanced capabilities of a DataSet. Some of these capabilities include:

- The ability to store data for long periods of time, and transfer it to other classes or components as a neatly packaged object.
- The ability to perform substantial updates and changes to the data, without needing to execute individual SQL statements or stored procedures.
- The ability to save or retrieve data as an XML file.
- Greater flexibility when reading data, such as the ability to move forward and backward through data, and the ability to jump back and forth between distinct, but related tables in the DataSet.

Filling a DataSet with a DataAdapter

As with a DataReader, you need to create a Connection and a Command object before you can retrieve the rows you need:

```
Dim Connect As String = "Data Source=localhost;Integrated Security=SSPI;" & _
  "Initial Catalog=Northwind;"
Dim con As New SqlConnection(Connect)

Dim SQL As String = "SELECT * FROM Orders " & _
 "WHERE OrderDate < '2000/01/01' AND OrderDate > '1987/01/01'"
Dim cmd As New SqlCommand(SQL, con)
```

So far, these lines are the same as those used by our DataReader.

Next, you need to create a DataAdapter. DataAdapters are another example of data source–specific objects. There are two flavors: SqlDataAdapter, shown here, and OleDbDataAdapter.

```
Dim adapter As New SqlDataAdapter(cmd)
```

This statement creates an adapter using our Command object. There are several other equivalent ways that you could accomplish the same thing. For example, you could pass the SQL and Connect strings to the SqlDataAdapter

constructor, and coax it into creating an implicit connection and command automatically on its own. However, the approach used in the preceding example is generally more flexible, particularly if you need to reuse the connection or run more than one SQL query in a row with the same adapter.

Next we'll create a DataSet, and fill it with the DataAdapter's Fill method:

```
Dim dsNorthwind As New DataSet()
con.Open()
adapter.Fill(dsNorthwind, "Orders")
con.Close()
```

The Fill method executes the command we've specified, takes the results, and inserts them into the dsNorthwind DataSet in a table named Orders. (In this case, the destination table name is the same as the table name in the data source, but it doesn't need to be.)

NOTE *The DataSet object is generic. Whether you are using an OLE DB provider or the native SQL Server classes, you always create and fill the same DataSet object.*

Accessing the Information in a DataSet

The information in a DataSet is stored in collections. This is quite different from a DataReader, which exposes only one row at a time, forcing you to use the Read method to move from row to row. A DataSet, on the other hand, has a Tables property that contains a collection of DataTable objects. Each DataTable has a Rows property that contains a collection of DataRow objects. You access these DataRows by using the corresponding field names, much as you would with a DataReader. The diagram in Figure 9-8 shows the overall object model.

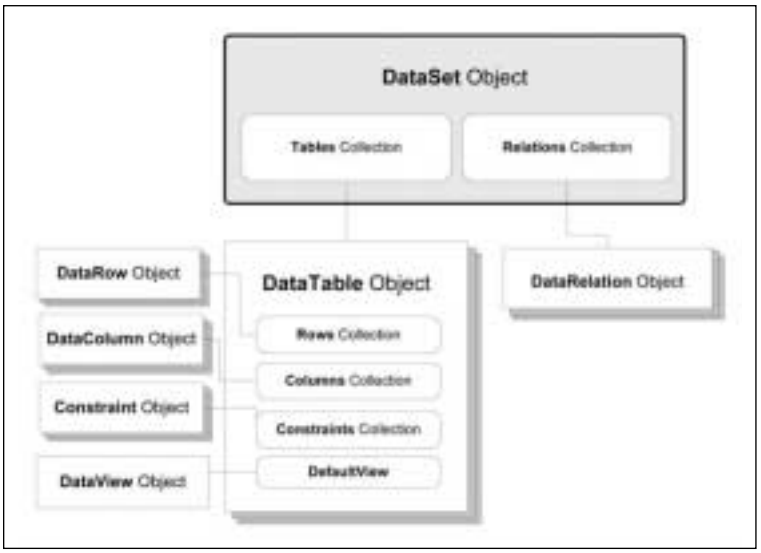

Figure 9-8: The DataSet

You'll notice that the discussion so far has left out a few of the details shown in this illustration. For example, a DataSet can also contain DataRelation objects (which link different DataTables together), and each DataTable can also contain Constraint objects (which specify restrictions on allowable column information) and Column objects (which contain information about the field name and data type of each column). These collections are generally less important, although later in the chapter we will return to the DataRelation object in more detail.

You move through the data in a table using the Rows collection, as shown here:

```
Dim row As DataRow
For Each row In dsNorthwind.Tables("Orders").Rows
    ' Here you can retrieve a value using the current row.
    lstOrderID.Items.Add(row("OrderID"))
    ' Or you can change it.
    row("ShipCountry") = "Lilliput"
Next
```

NOTE *Of course, the DataSet object is always disconnected. This means that any changes you make will appear in your program, but won't affect the original data source unless you take additional steps, which this chapter will delve into a little bit later.*

Deleting Records

You can also delete records from a DataSet using the Delete method. The process is quite straightforward:

```
Dim row As DataRow
Dim colRowsToDelete As New Collection()

For Each row In dsNorthwind.Tables("Orders").Rows
    If row("ShipCountry") <> "Brazil" Then
        ' If the ShipCountry is not Brazil, mark it for deletion.
        row.Delete()
    Else
        ' Otherwise, add it to our list.
        lstOrder.Items.Add(row("OrderID") & " to " & row("ShipCountry"))
    End If
Next
```

However, when you use the Delete method, the row is not actually removed, only marked for deletion. That's because ADO.NET needs to retain information about the record in order to be able to remove it from the original data source when you reconnect later. Your programs need to be aware of this fact, and should include steps that prevent them from trying to use deleted rows:

```
For Each row In dsNorthwind.Tables("Orders").Rows
    If row.RowState <> DataRowState.Deleted Then
        lstOrderID.Items.Add(row("OrderID"))
    End If
Next
```

NOTE *If your program tries to read a field of information from a deleted item, an error will occur. This error is meant to alert you that you are trying to access information that is scheduled for deletion.*

You can also use the Remove method to delete an item completely. However, if you use this method, the record won't be deleted from the data source when you reconnect and update it with your changes. Instead, it will just be eliminated from your DataSet object.

Adding Information to a DataSet

You can also easily add a new row using the Add method of the Rows collection. The trick is to use the NewRow method first to get a blank copy of the row you want to create:

```
Dim rowNew As DataRow
' Create the row.
rowNew = dsNorthwind.Tables("Orders").NewRow()

' Set the information in the row.
rowNew("OrderID") = 12000
rowNew("ShipCountry") = "Lilliput"
' (And so on to add more fields...)

' Add the row to the DataSet.
dsNorthwind.Tables("Orders").Rows.Add(rowNew)
```

Life might not be this easy, depending on your original data source. For example, the database may have other requirements for these columns (such as a maximum length, or a restriction against null values). In addition, most database designs use *auto-incrementing* identity columns. For example, if OrderID were an auto-numbering column, SQL Server would automatically give it a new value when you add a new record. This means that you shouldn't specify any value at all in the OrderID field while you are creating the record, or else you risk specifying a number that will conflict with an existing generated value, thus causing a problem.

Generally, ADO.NET provides one tool that can help you. It's called the FillSchema method, and you can use it before using the Fill method to retrieve a bunch of information about your database, such as column constraints, and add it into your DataTable object.

```
adapter.FillSchema(dsNorthwind, SchemaType.Mapped, "Orders")
adapter.Fill(dsNorthwind, "Orders")
```

The FillSchema method adds the DataTable and all the DataColumn objects, but it doesn't add the actual data. Unlike the Fill method, FillSchema completely configures each DataColumn object with such information as default value, nullabity, and maximum length. (For a full list, check the properties of the DataColumn object in the MSDN class library reference.) The primary key requirement is also added as a Constraint object. Foreign keys, which define relationships between tables, are not added, because ADO.NET has no way of knowing whether you have added the required linked tables. When you use the Fill method, the information streams into the ready-made columns without a problem.

The specific details of how to create and modify column constraints and default values are beyond the scope of this chapter. Most programmers will use a visual database design tool (like SQL Server's Enterprise Manager) for this task. Also, keep in mind that it's often a better idea to add a new record directly by using a stored procedure.

Working with Multiple Tables

Sadly, the Fill method can add only one table at a time. If you want to add more than one table, you have to use the Fill method more than once, including more than one Command object (or changing the Command's CommandText property in between).

```
Dim dsNorthwind As New DataSet()
adapter.Fill(dsNorthwind, "Orders")

' This command is still linked to the DataAdapter.
cmd.CommandText = "SELECT * FROM Customers"
adapter.Fill(dsNorthwind, "Customers")

cmd.CommandText = "SELECT * FROM Employees"
adapter.Fill(dsNorthwind, "Employees")
```

After these commands, there will be three tables in the DataSet, each of which can be accessed individually by specifying the appropriate table name (for example, `dsNorthwind.Table("Customers")` accesses the Customers table).

DataTable Relations

There is no way to import information about linked tables from the data source. Instead, you need to add this information manually if you want to make use of it. To link two tables together, you need to create a Relation object.

Our multi-table example uses three tables that are related. The Orders table has a CustomerID field that corresponds to a CustomerID in the Customers table. To specify this relationship in a DataSet, you can use the following code:

```
' Define the relation.
Dim relCustomersOrders As New DataRelation("CustomersOrders", _
 dsNorthwind.Tables("Customers").Columns("CustomerID"), _
 dsNorthwind.Tables("Orders").Columns("CustomerID"))

' Add the relation to the DataSet.
dsNorthwind.Relations.Add(relCustomersOrders)
```

This code defines a relationship in which the Customers table is the parent and the Orders table is the child. This is because one customer record can have multiple children (orders), but every order has only one parent (customer). This Parent-to-Child relationship is just another way of describing a One-to-Many relationship, which is a basic ingredient in database theory. Once the relationship is defined, our example adds it to the DataSet to put it to work.

As with any relational database, using a relation implies certain restrictions. For example, if you add a relation to the DataSet and then try to create a child row that refers to a nonexistent parent, ADO.NET will generate an error. Similarly, you can't delete a parent that has child records linked to it. These requirements will be enforced already by your data source, but by adding them to the DataSet, you ensure that you will catch any errors as soon as they occur, rather than waiting until an entire batch of changes is committed to the data source later on.

You can also use your relation to provide better record navigation. This technique, shown in the following code, allows you to combine information dynamically from several linked tables, without having to use a join query. It works using the GetChildRows method of a DataRow object. The results are shown in Figure 9-9.

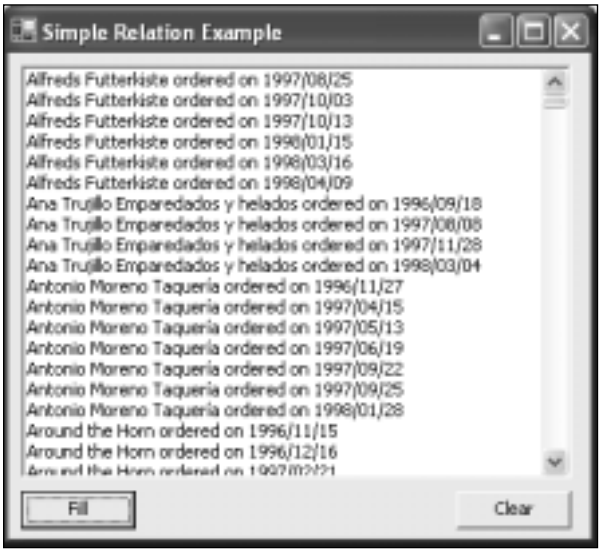

Figure 9-9: A simple example of relational data

```
Dim rowParent, rowChild As DataRow
For Each rowParent In dsNorthwind.Tables("Customers").Rows
    For Each rowChild In rowParent.GetChildRows(relCustomersOrders)
        ' Display combined information using both rows.
        lstOrder.Items.Add(rowParent("CompanyName") & _
         " ordered on " & rowChild("OrderDate"))
    Next
Next
```

An even more useful way to use this relational ability might be to construct a hierarchical TreeView display. The only difference in the code is that you need to make sure to store a reference to the current customer node so you can add order sub-nodes.

```
Dim nodeParent, nodeChild As TreeNode
Dim rowParent, rowChild As DataRow

For Each rowParent In dsNorthwind.Tables("Customers").Rows
    ' Add the customer node.
    nodeParent = treeDB.Nodes.Add(rowParent("CompanyName"))

    ' Store the disconnected customer information for later.
    nodeParent.Tag = rowParent

    For Each rowChild In rowParent.GetChildRows(relCustomersOrders)
        ' Add the child order node.
        nodeChild = nodeParent.Nodes.Add(rowChild("OrderID"))

        ' Store the disconnected order information for later.
        nodeChild.Tag = rowChild
    Next
Next
```

As an added enhancement, this code stores a reference to the associated DataRow object in the Tag property of each TreeNode. When the node is clicked, all the information is retrieved from the DataRow, and then displayed in the adjacent text box. This is one of the advantages of disconnected data objects: You can keep them around for as long as you want.

NOTE *You might remember the Tag property from Visual Basic 6, where it could be used to store a string of information for your own personal use. The Tag property in VB .NET is similar, except you can store any type of object in it.*

```
Private Sub treeDB_AfterSelect(ByVal sender As System.Object, _
  ByVal e As System.Windows.Forms.TreeViewEventArgs) Handles treeDB.AfterSelect

    ' Clear the textbox.
    txtInfo.Text = ""
    Dim row As DataRow = CType(e.Node.Tag, DataRow)

    ' Fill the textbox with information from every field.
    Dim Field As Object
    For Each Field In row.ItemArray
        txtInfo.Text &= Field.ToString & vbNewLine
    Next

End Sub
```

Figure 9-10: An advanced example of relational data

This sample program (featured in the chapter examples as the Relational-TreeView project and shown in Figure 9-10) is also a good demonstration of docking at work. To make sure all the controls stay where they should, and to allow the user to change the relative screen area given to the TreeView and text box, a Splitter control is used along with three Panel controls.

Using a DataSet Object to Update Data

One aspect of the DataSet object that you may not realize at first is that it stores additional information about the initial values of your table and the changes that you have made. You have already seen how deleted rows are left in your DataSet with a special deleted flag (DataRowState.Deleted). Similarly, added rows are given the flag DataRowState.Added, and modified rows are flagged as DataRowState.Modified. This allows ADO.NET to quickly determine which rows need to be added, removed, and changed.

However, in order to modify a changed row, ADO.NET needs to be able to select the original row from the data source. To allow this, ADO.NET stores information about the original field values, as shown in this example:

```
Dim rowEdit As DataRow
' Select the 11 row (at position 10).
rowEdit = dsNorthwind.Tables("Orders").Rows(10)

' Change some information in the row.
rowEdit("ShipCountry") = "Oceania"

' This returns "Oceania".
lblResult.text = rowEdit("ShipCountry")

' This is identical.
lblResult.text = rowEdit("ShipCountry", DataRowVersion.Current)

' This returns the last data source version (in my case, "Austria").
lblResult.text = rowEdit("ShipCountry", DataRowVersion.Original)
```

Ordinarily, you don't need to worry about this extra layer of information, other than to understand that this is what allows ADO.NET to find the original row and update it when you reconnect.

The whole process works like this:

1. Create a Connection object and define a Command object that will select the data you need.
2. Create a corresponding DataAdapter object, using your Command object.
3. Using the DataAdapter, transfer the information into a disconnected DataSet object. Close the Connection object.
4. Make changes to the DataSet (modifying, deleting, or adding rows).
5. Create a Connection object (or reuse the existing one).
6. Create Command objects for inserting, updating, and deleting data. Alternatively, to save yourself some work, you can use the special CommandBuilder class.
7. Create a DataAdapter object using your Command or CommandBuilder objects.

8. Reconnect to the data source.
9. Using the DataAdapter, update the data source with the information in the DataSet.
10. Report any concurrency errors (for example, if an operation fails because another user has already changed the row after you've retrieved it).

You can see why using a simple command with an SQL Update statement is a simpler approach than managing disconnected data!

Using the CommandBuilder Object

Assuming that you have already created the DataSet, filled it with information, and made your modifications, we can pick up with Step 5 of the preceding list. This step involves defining a connection, which is straightforward:

```
Dim Connect As String = "Data Source=localhost;Integrated Security=SSPI;" & _
  "Initial Catalog=Northwind;"
Dim con As New SqlConnection(Connect)
```

The next step is to create a Command. When we selected information from the data source, we needed only one type of command: a Select command. However, when we update the data source, three different tasks could be performed in combination, depending on the changes that we have made, including Insert, Update, and Delete commands. In order to avoid the work involved in creating these three command objects manually, you can use the special CommandBuilder object (which is provided as SqlCommandBuilder and OleDbCommandBuilder). This object has no purpose other than to save you a little manual work when updating a data source.

The CommandBuilder takes a reference to the DataAdapter object that was used to create the DataSet, and it adds the required additional three commands. Assuming that the programmer has been relatively shortsighted, and has already destroyed the Command and DataAdapter objects, we can simply recreate them:

```
' Create the Command and DataAdapter representing the Select operation.
Dim SQL As String = "SELECT * FROM Orders " & _
 "WHERE OrderDate < '2000/01/01' AND OrderDate > '1987/01/01'"
Dim cmd As New SqlCommand(SQL, con)
Dim adapter As New SqlDataAdapter(cmd)
```

At this point, the adapter.SelectCommand property refers to the cmd object. This SelectCommand is automatically used for select operations (such as Fill and ExecuteReader). However, the adapter.InsertCommand, adapter.DeleteCommand, and adapter.UpdateCommand properties are not set.

To set these three properties, you can use the CommandBuilder:

```
' Create the CommandBuilder.
Dim cb As New SqlCommandBuilder(adapter)

' Retrieve an updated DataAdapter.
adapter = cb.DataAdapter
```

Updating the Data Source

Once you have appropriately configured the DataAdapter, you can update the data source in a single line by using the DataAdapter's Update method:

```
Dim NumRowsAffected As Integer
NumRowsAffected = adapter.Update(dsNorthwind, "Orders")
```

The Update method works with one table at a time, so you'll need to call it several times in order to commit the changes in multiple tables. When you use the Update method, ADO.NET scans through all the rows in the specified table. Every time it finds a new row (DataRowState.Added), it adds it to the data source via the corresponding Insert command. Every time it finds a row that is marked with the state DataRowState.Deleted, it deletes the corresponding row from the database by using the Delete command. And every time it finds a DataRowState.Modified row, it updates the corresponding row by using the Update command.

Once the update is successfully complete, the DataSet object will be refreshed, all rows will be reset to DataRowState.Unchanged, and all the "Current" values will become "Original" values, to correspond to the data source.

Reporting Concurrency Problems

Before a row can be updated, the row in the data source must exactly match the "Original" value stored in the DataSet. This value is set when the DataSet is created, and every time the data source is updated. If however, another user has changed even a single field in the original record while your program has been working with the disconnected data, the operation will fail, the Update will be halted, and an exception will be thrown. In many cases, this prevents other valid rows from being updated.

An easier way to deal with this problem is to detect the discrepancy in the DataAdapter's RowUpdated event. This event occurs every time a single update, delete, or insert operation is completed, regardless of the result. It provides you with some special information, including the type of statement that was just executed, the number of rows that were affected, and the DataRow from the DataTable that prompted the operation. It also gives you the chance to tell the DataAdapter to ignore the error, note it, and resolve it later.

The RowUpdated event happens in the middle of DataAdapter's Update method, and so this event handler is not the place to try and resolve the problem or present the user with additional user interface, which would tie up the

database connection. Instead, you should log errors, display them on the screen in a list control, or put them into a collection so you can examine them later.

The following example puts errors in one of three shared collections provided in a class called DBErrors. The class looks like this:

```
Public Class DBErrors
    Public Shared LastInsert As Collection
    Public Shared LastDelete As Collection
    Public Shared LastUpdate As Collection
End Class
```

The event handler code looks like this:

```
Public Sub OnRowUpdated(ByVal sender As Object, ByVal e As
  SqlRowUpdatedEventArgs)

    ' Check if any records were affected.
    ' If no records were affected, the statement did not executed as expected.

    If e.RecordsAffected() < 1 Then
        ' We add information about failed operations to a table.
        Select Case e.StatementType
            Case StatementType.Delete
                DBErrors.LastDelete.Add(e.Row)
            Case StatementType.Insert
                DBErrors.LastInsert.Add(e.Row))
            Case StatementType.Update
                DBErrors.LastUpdate.Add(e.Row)
        End Select

       ' As the error has already been detected, we don't need the DataAdapter
       ' to cancel the entire operation and throw an exception, unless
       ' the failure may affect other operations.
        e.Status = UpdateStatus.SkipCurrentRow
    End If

End Sub
```

The nice thing about this approach is that it allows us the flexibility to decide how we want to deal with these errors when we execute the Update method, rather than hard-coding a specific procedure into our event handler.

To bring it all together, we need to attach the event handler before the update is performed. Our next example goes one step further, examining the error collections and displaying the results in three separate list controls in the current window.

```
' Connect the event handler.
AddHandler(adapter.RowUpdated, AddressOf OnRowUpdated)

' Perform the update.
Dim NumRowsAffected As Integer
NumRowsAffected = adapter.Update(dsNorthwind, "Orders")

' Display the errors.
Dim rowError As DataRow
For Each rowError In DB.LastDelete
    lstDelete.Items.Add(rowError("OrderID")
Next

For Each rowError In DB.LastInsert
    lstInsert.Items.Add(rowError("OrderID")
Next

For Each rowError In DB.LastUpdate
    lstUpdate.Items.Add(rowError("OrderID")
Next
```

The ConcurrencyErrors sample project shows a "live" example of this technique. It works by creating two DataSets, and simulating a multi-user concurrency problem by modifying them simultaneously in two different ways (see Figure 9-11). This artificial error is then dealt with in the RowUpdated event handler.

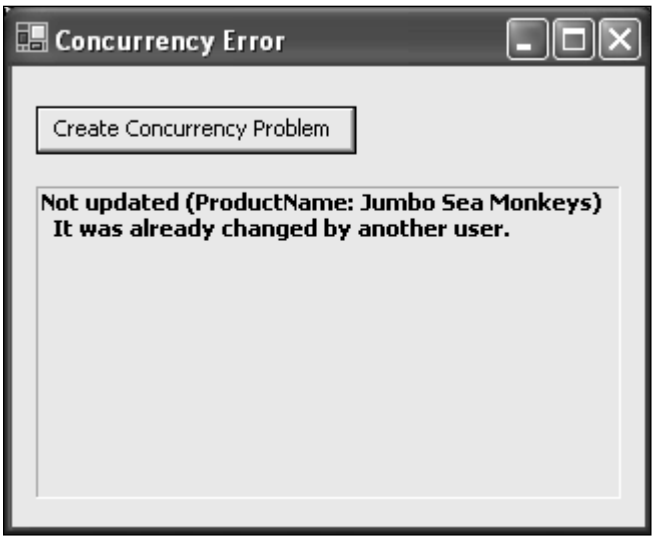

Figure 9-11: Simulating a concurrency problem

Updating Data in Stages

Concurrency issues aren't the only potential source of error when you update your data source. Another problem can occur if you use linked tables, particularly if you have deleted or added records. When you update the data source, your changes will probably not be committed in the same order in which they were performed in the DataSet. If you try to delete a Parent record while it is still being used by a Child record, an error will occur. This error will take place regardless of whether you have defined relations in your DataSet. The problem is that SQL Server will not let you remove a required Parent. Similarly, you won't be able to create a Child record that refers to a Parent that doesn't yet exist. In the case of the Northwind database, you could encounter these sorts of errors by trying to add a Product that references a non-existing Supplier or Category, or trying to delete a Supplier or Category record that is currently being used by a Product.

There is no simple way around these problems. If you are performing sophisticated data manipulation on a relational database using a DataSet, you will have to plan out the order that changes need to be implemented. However, you can then use some built-in ADO.NET features to perform these operations in separate stages.

Generally, a safe approach would proceed in this order:

1. Add any new records to all tables.
2. Modify existing records in all tables.
3. Delete records in all tables.

To perform these operations separately, you need a special update routine. This routine will create three separate DataSets, one for each operation. Then, you'll move all the new records into one DataSet, all the records marked for deletion into another, and all the modified records into a third. To perform this shuffling around, you can use the DataSet's GetChanges method:

```
' Create three DataSets, and fill them from dsNorthwind.
Dim dsNew As DataSet = dsNorthwind.GetChanges(DataRowState.Added)
Dim dsModify As DataSet = dsNorthwind.GetChanges(DataRowState.Deleted)
Dim dsDelete As DataSet = dsNorthwind.GetChanges(DataRowState.Modified)

' Update these DataSets separately, in an order guaranteed to avoid problems.
adapter.Update(dsNew, "Customers")
adapter.Update(dsNew, "Orders")
adapter.Update(dsModify, "Customers")
adapter.Update(dsModify, "Orders")
adapter.Update(dsDelete, "Customers")
adapter.Update(dsDelete, "Orders")
```

Creating a DataSet Object by Hand

Incidentally, you can add new tables and even create an entire DataSet by hand. There's really nothing tricky to this approach—it's just a matter of working with the right collections. First you have to create the DataSet, then at least one DataTable, and then add at least one DataColumn in each DataTable. After that, you can start adding DataRows.

```
' Create a DataSet and add a new table.
Dim dsPrefs As New DataSet
dsPrefs.Tables.Add("FileLocations")

' Define two columns for this table.
dsPrefs.Tables("FileLocation").Columns.Add("Folder", GetType("System.String"))
dsPrefs.Tables("FileLocation").Columns.Add("Documents", GetType("System.Int32"))

' Add some actual information into the table.
Dim newRow As DataRow = dsPrefs.Tables("FileLocation").NewRow()
newRow("Folder") = "f:\Pictures"
newRow("Documents") = 30
dsPrefs.Tables("FileLocation").Rows.Add(newRow)
```

Notice that this example uses standard .NET types instead of SQL-specific or OLE DB–specific types. That's because this table is not designed for storage in a separate data source. Instead, this miniature database stores preferences for a single user. The same information could be stored in the registry, but then it would be hard to move a user's settings from one computer to another. In XML format, these settings can be placed on an internal network, and easily made available to various workstations.

Storing a DataSet in XML

To store the custom data as an XML document, you use the built-in methods of the DataSet object:

```
' Save it as an XML file with the WriteXml method.
dsUserPrefs.WriteXml("f:\MyApp\UserData\" & UserName & ".xml")
dsUserPrefs = Nothing

' And retrieve it with the ReadXml method.
dsUserPrefs.ReadXml("f:\MyApp\UserData\" & UserName & ".xml")
```

The XML document for a DataSet is shown in Figure 9-12, as displayed in Internet Explorer.

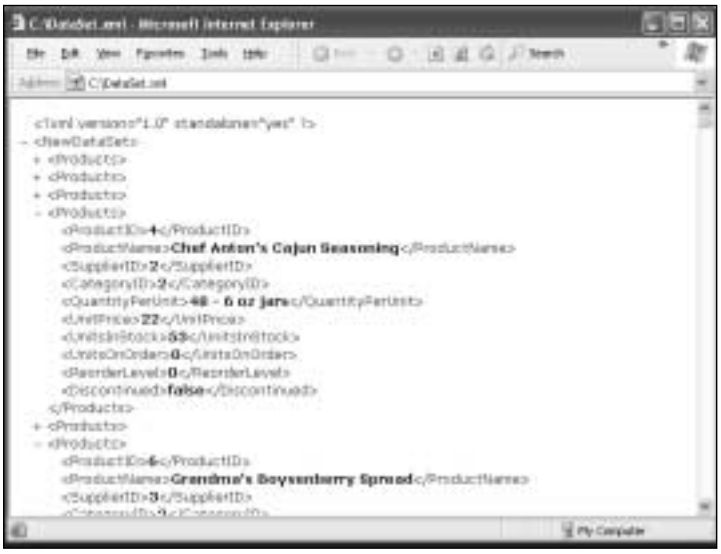

Figure 9-12: A partly collapsed view of a DataSet in XML

Of course, you will probably never need to look at it directly, because the ADO.NET DataSet object handles the XML format automatically. You can test XML reading and writing with the sample project XMLDataSet.

It really is quite easy to use ADO.NET XML in this way. However, keep in mind that this won't compensate for a true relational database. For example, there is no way to manage concurrent user updates to this file—every time it is saved, the existing version is completely wiped out.

If you need to exchange XML data with another program, or if the structure of your DataSet changes with time, you might find it a good idea to save the XML schema information for your DataSet. This document (shown in Figure 9-13) explicitly defines the format that your XML file uses, preventing any chance of confusion. Generally, it is a good safeguard, and easy to implement. All you need to remember is to read the schema into the DataSet before you load the actual data.

```
' Save it as an XML file with the WriteSchema and WriteXml methods.
dsUserPrefs.WriteSchema("f:\MyApp\UserData\" & UserName & ".xsd")
dsUserPrefs.WriteXml("f:\MyApp\UserData\" & UserName & ".xml")
dsUserPrefs = Nothing

' And retrieve it with the ReadSchema and ReadXml methods.
dsUserPrefs.ReadSchema("f:\MyApp\UserData\" & UserName & ".xsd")
dsUserPrefs.ReadXml("f:\MyApp\UserData\" & UserName & ".xml")
```

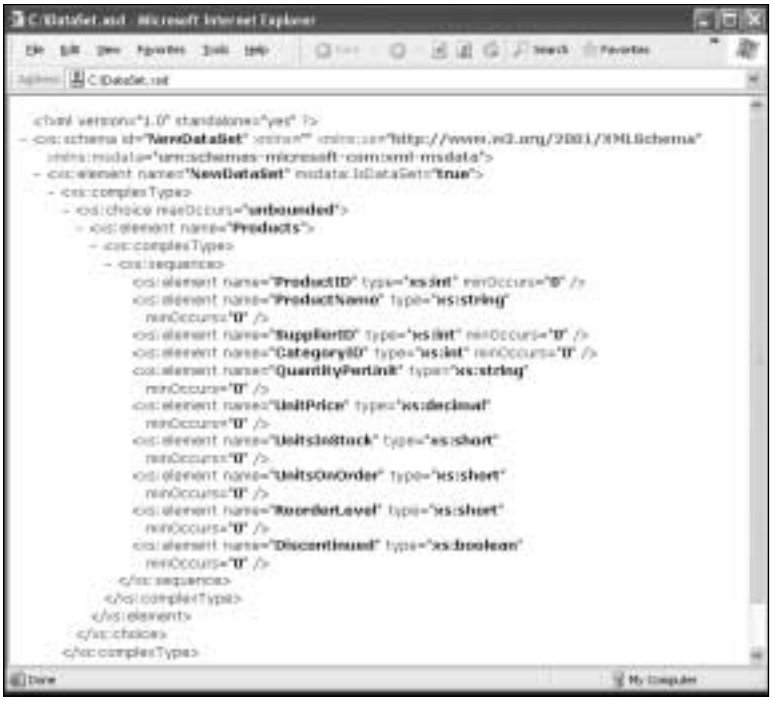

Figure 9-13: A DataSet schema

Data Binding

Data binding is a powerful way to display information from a DataSet by binding it directly to a user interface control. It saves you from needing to write simple but repetitive code to move through the database and manually copy content from a DataSet into a control. (Our ListView example used this kind of code, but in that case, we had no choice: The ListView control doesn't support data binding.)

Binding a control is usually just as easy as setting a DataSource property.

```
DataGrid1.DataSource = dsNorthwind.Tables("Products")
```

This produces a control with all the columns and data, as shown in Figure 9-14.

Figure 9-14: A data-bound grid

In its default mode, the DataGrid even allows you to edit a data value by typing in a field, and to add a new row by entering information at the bottom of the row (see Figure 9-15).

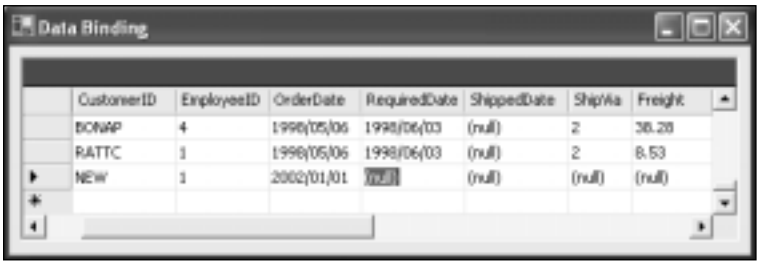

Figure 9-15: Adding a new record

When you change or add information to the DataGrid, the linked DataSet is modified automatically, providing some very convenient basic data editing features.

An interesting feature of the DataGrid is that you can use it to display an entire DataSet made up of more than one table.

```
DataGrid1.DataSource = dsNorthwind
```

The DataGrid then adds web-like navigation links that allow the user to move from table to table (Figure 9-16).

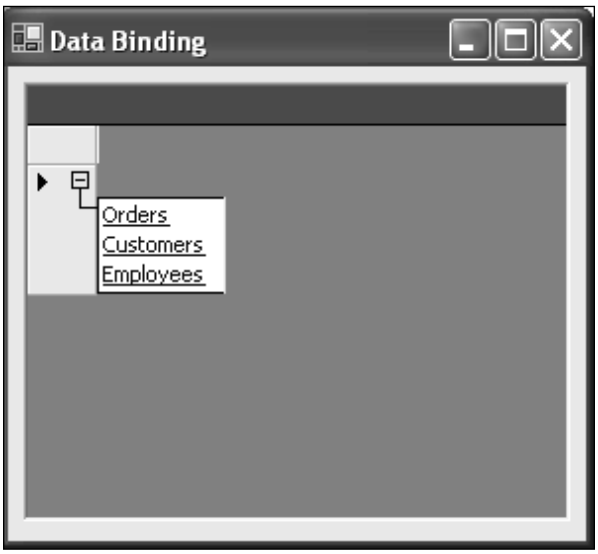

Figure 9-16: A DataGrid with multiple tables

Not all controls support data binding, and few can bind to multiple tables at once. Some, like ListBox controls, can only support binding to one field in a table. In this case, you have to specify two properties: the table data source, and the field that should be used for display purposes:

```
lstID.DataSource = dsNorthwind.Tables("Employees")
lstID.DisplayMember = "EmployeeID"
```

Just about every .NET control supports single-value data binding through the DataBindings property. This property provides a collection that allows you to connect a field in the data source with a property in the control. That means you could have a checkbox control, for example, that has several bound properties, including Text, Tag, and Checked.

The following code binds a generic text box:

```
' Bind the FirstName field to the Text property.
txtName.DataBindings.Add("Text", dsNorthwind.Tables("Employees"), "FirstName")
```

You can bind a DataSet to as many controls as you want, all at the same time (as shown in Figure 9-17). However, only one record can be selected at a time. When you select a value in the ListBox, the corresponding full record row is selected in the DataGrid, and the corresponding values are filled into other bound controls like the text box.

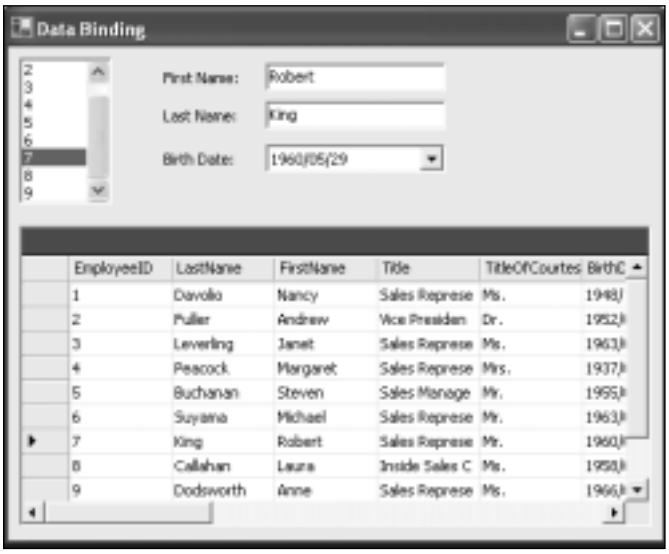

Figure 9-17: Multiple bound controls

This allows you to create windows that contain many different controls, each of which allows you to edit one property of the currently selected record. There's much more that you can do with data binding to configure advanced column display. Using such features as column mapping, you can rename or hide specific columns. ASP.NET even allows you to use templates to configure specifically how a column will look. Unfortunately, we won't get a chance to explore these topics in this chapter. Instead, refer to the MSDN help library.

What Comes Next?

This chapter has tackled a subject that can easily make up an entire book of its own. We've examined all the essentials, with a fairly in-depth look at the best way to organize database code, update information, and manage DataSets. You may want to take the time to work through this chapter again, as many of the insights contained here are the basis for "best practices" and other techniques that can ensure a robust, scalable database application.

There are still many more possibilities left for you to discover with ADO.NET. Here are some of them:

- If you don't already know the SQL language, now is the perfect time to learn. Although you don't need a sophisticated understanding to program with ADO.NET, the difference between a competent database programmer and an excellent one is often an understanding of the limitations and capabilities of SQL. Many excellent SQL resources are available online.

- It also helps to know a specific database product, such as SQL Server 2000, in order to create stored procedures and well-organized data tables. SQL Server provides its own "Books Online" help, which covers advanced tools such as stored procedures, views, column constraints, and triggers, all of which can help you become a database guru.
- Data binding was a dirty word in traditional Visual Basic programming, because it was slow, inefficient, and extremely inflexible. In .NET, data binding has been improved so much that it finally makes sense. Using data binding with the DataGrid, for example, you can automatically provide a sophisticated number of data editing features. The ASP.NET DataGrid is even more impressive, supporting such features as automatic paging (splitting results onto more than one HTML page) and sorting, and advanced selection and editing. It's poised to become the best choice for large-scale Internet applications.
- In the examples in this chapter, we updated our data source using a DataSet and the default UpdateCommand, InsertCommand, and DeleteCommand that ADO.NET generates automatically. You might be able to provide increased performance and some additional options if you learn how to customize these properties with your own commands. For example, you might create a command that can update a record even if it has been changed in the meantime, by making the selection criteria less strict. (You might look the record up just using the ID column, for example.) Or, you could configure the DataAdapter to use a specific stored procedure you have created. See the MSDN help library for more information.