Mastering™ Visual Basic® .NET Database Programming

Evangelos Petroutsos; Asli Bilgin

Chapter 6: A First Look at ADO.NET

ISBN: 0-7821-2878-5

Sybex Inc.
1151 Marina Village Parkway
Alameda, CA 94501
U.S.A.
Phone: 510-523-8233
www.sybex.com

Chapter 6

A First Look at ADO.NET

- How does ADO.NET work?
- Using the ADO.NET object model
- The Connection object
- The Command object
- The DataAdapter object
- The DataReader object
- The DataSet object
- Navigating through DataSets
- Updating Your Database by using DataSets
- Managing concurrency

IT'S TIME NOW TO get into some real database programming with the .NET Framework components. In this chapter, you'll explore the Active Data Objects (ADO).NET base classes. ADO.NET, along with the XML namespace, is a core part of Microsoft's standard for data access and storage. As you recall from Chapter 1, "Database Access: Architectures and Technologies," ADO.NET components can access a variety of data sources, including Access and SQL Server databases, as well as non-Microsoft databases such as Oracle. Although ADO.NET is a lot different from classic ADO, you should be able to readily transfer your knowledge to the new .NET platform. Throughout this chapter, we make comparisons to ADO 2.*x* objects to help you make the distinction between the two technologies.

For those of you who have programmed with ADO 2.*x*, the ADO.NET interfaces will not seem all that unfamiliar. Granted, a few mechanisms, such as navigation and storage, have changed, but you will quickly learn how to take advantage of these new elements. ADO.NET opens up a whole new world of data access, giving you the power to control the changes you make to your data. Although native OLE DB/ADO provides a common interface for universal storage, a lot of the data activity is hidden from you. With client-side disconnected RecordSets, you can't control how your updates occur. They just happen "magically." ADO.NET opens that black box, giving you more granularity with your data manipulations. ADO 2.*x* is about common data access. ADO.NET extends this model and factors out data storage from common data access. Factoring out functionality makes it easier for you to understand how ADO.NET components work. Each ADO.NET component has its own specialty, unlike the RecordSet, which is a jack-of-all-trades. The RecordSet could be disconnected or stateful; it could be read-only or updateable; it could be stored on the client or on the server—it is multifaceted. Not only do all these mechanisms bloat the RecordSet with functionality you might never use, it also forces you to write code to anticipate every possible chameleon-like metamorphosis of the RecordSet. In ADO.NET, you always know what to expect from your data access objects, and this lets you streamline your code with specific functionality and greater control.

Although a separate chapter is dedicated to XML (Chapter 10, "The Role of XML"), we must touch upon XML in our discussion of ADO.NET. In the .NET Framework, there is a strong synergy between ADO.NET and XML. Although the XML stack doesn't technically fall under ADO.NET, XML and ADO.NET belong to the same architecture. ADO.NET persists data as XML. There is no other native persistence mechanism for data and schema. ADO.NET stores data as XML files. Schema is stored as XSD files.

There are many advantages to using XML. XML is optimized for disconnected data access. ADO.NET leverages these optimizations and provides more scalability. To scale well, you can't maintain state and hold resources on your database server. The disconnected nature of ADO.NET and XML provide for high scalability.

In addition, because XML is a text-based standard, it's simple to pass it over HTTP and through firewalls. Classic ADO uses a binary format to pass data. Because ADO.NET uses XML, a ubiquitous standard, more platforms and applications will be able to consume your data. By using the XML model, ADO.NET provides a complete separation between the data and the data presentation. ADO.NET takes advantage of the way XML splits the data into an XML document, and the schema into an XSD file.

By the end of this chapter, you should be able to answer the following questions:

- What are .NET data providers?
- What are the ADO.NET classes?
- What are the appropriate conditions for using a DataReader versus a DataSet?
- How does OLE DB fit into the picture?
- What are the advantages of using ADO.NET over classic ADO?
- How do you retrieve and update databases from ADO.NET?
- How does XML integration go beyond the simple representation of data as XML?

Let's begin by looking "under the hood" and examining the components of the ADO.NET stack.

How Does ADO.NET Work?

ADO.NET base classes enable you to manipulate data from many data sources, such as SQL Server, Exchange, and Active Directory. ADO.NET leverages .NET data providers to connect to a database, execute commands, and retrieve results.

The ADO.NET object model exposes very flexible components, which in turn expose their own properties and methods, and recognize events. In this chapter, you'll explore the objects of the ADO.NET object model and the role of each object in establishing a connection to a database and manipulating its tables.

Is OLE DB Dead?

Not quite. Although you can still use OLE DB data providers with ADO.NET, you should try to use the managed .NET data providers whenever possible. If you use native OLE DB, your .NET code will suffer because it's forced to go through the COM interoperability layer in order to get to OLE DB. This leads to performance degradation. Native .NET providers, such as the `System.Data.SqlClient` library, skip the OLE DB layer entirely, making their calls directly to the native API of the database server.

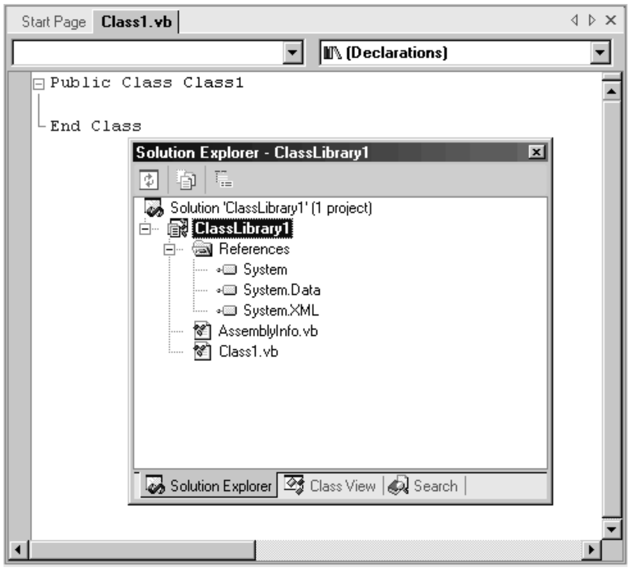
However, this doesn't mean that you should avoid the OLE DB .NET data providers completely. If you are using anything other than SQL Server 7 or 2000, you might not have another choice. Although you will experience performance gains with the SQL Server .NET data provider, the OLE DB .NET data provider compares favorably against the traditional ADO/OLE DB providers that you used with ADO 2.*x*. So don't hold back from migrating your non-managed applications to the .NET Framework for performance concerns. In addition, there are other compelling reasons for using the OLE DB .NET providers. Many OLE DB providers are very mature and support a great deal more functionality than you would get from the newer SQL Server .NET data provider, which exposes only a subset of this full functionality. In addition, OLE DB is still the way to go for universal data access across disparate data sources. In fact, the SQL Server distributed process relies on OLE DB to manage joins across heterogeneous data sources.

Another caveat to the SQL Server .NET data provider is that it is tightly coupled to its data source. Although this enhances performance, it is somewhat limiting in terms of portability to other data sources. When you use the OLE DB providers, you can change the connection string on the fly, using declarative code such as COM+ constructor strings. This loose coupling enables you to easily port your application from an SQL Server back-end to an Oracle back-end without recompiling any of your code, just by swapping out the connection string in your COM+ catalog.

Keep in mind, the only native OLE DB provider types that are supported with ADO.NET are `SQLOLEDB` for SQL Server, `MSDAORA` for Oracle, and `Microsoft.Jet.OLEDB.4` for the Microsoft Jet engine. If you are so inclined, you can write your own .NET data providers for any data source by inheriting from the `System.Data` namespace.

At this time, the .NET Framework ships with only the SQL Server .NET data provider for data access within the .NET runtime. Microsoft expects the support for .NET data providers and the number of .NET data providers to increase significantly. (In fact, the ODBC.NET data provider is available for download on Microsoft's website.) A major design goal of ADO.NET is to synergize the native and managed interfaces, advancing both models in tandem.

You can find the ADO.NET objects within the `System.Data` namespace. When you create a new VB .NET project, a reference to the `System.Data` namespace will be automatically added for you, as you can see in Figure 6.1.



```
Public Class Class1

End Class
```

FIGURE 6.1
To use ADO.NET, reference the `System.Data` namespace.

To comfortably use the ADO.NET objects in an application, you should use the `Imports` statement. By doing so, you can declare ADO.NET variables without having to fully qualify them. You could type the following `Imports` statement at the top of your solution:

```
Imports System.Data.SqlClient
```

After this, you can work with the SqlClient ADO.NET objects without having to fully qualify the class names. If you want to dimension the SqlClientDataAdapter, you would type the following short declaration:

```
Dim dsMyAdapter as New SqlDataAdapter
```

Otherwise, you would have to type the full namespace, as in:

```
Dim dsMyAdapter as New System.Data.SqlClient.SqlDataAdapter
```

Alternately, you can use the visual database tools to automatically generate your ADO.NET code for you. As you saw in Chapter 3, "The Visual Database Tools," the various wizards that come with VS .NET provide the easiest way to work with the ADO.NET objects. Nevertheless, before you use these tools to build production systems, you should understand how ADO.NET works programmatically. In this chapter, we don't focus too much on the visual database tools, but instead concentrate on the code behind the tools. By understanding how to program against the ADO.NET object model, you will have more power and flexibility with your data access code.

Using the ADO.NET Object Model

You can think of ADO.NET as being composed of two major parts: .NET data providers and data storage. Respectively, these fall under the connected and disconnected models for data access and presentation. *.NET data providers,* or *managed providers,* interact natively with the database. Managed providers are quite similar to the OLE DB providers or ODBC drivers that you most likely have worked with in the past.

The .NET data provider classes are optimized for fast, read-only, and forward-only retrieval of data. The managed providers talk to the database by using a fast data stream (similar to a file stream). This is the quickest way to pull read-only data off the wire, because you minimize buffering and memory overhead.

If you need to work with connections, transactions, or locks, you would use the managed providers, not the DataSet. The DataSet is completely disconnected from the database and has no knowledge of transactions, locks, or anything else that interacts with the database.

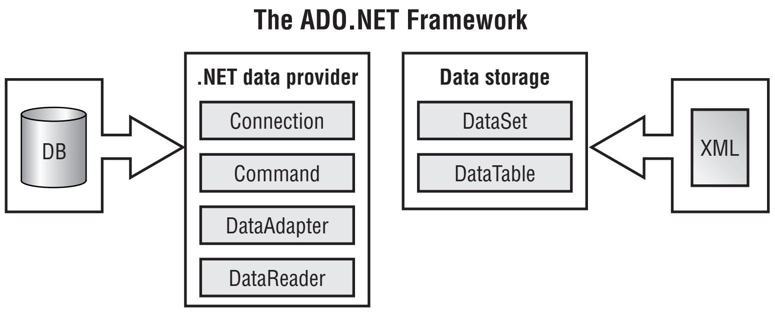
Five core objects form the foundation of the ADO.NET object model, as you see listed in Table 6.1. Microsoft moves as much of the provider model as possible into the managed space. The Connection, Command, DataReader, and DataAdapter belong to the .NET data provider, whereas the DataSet is part of the disconnected data storage mechanism.

TABLE 6.1: ADO.NET CORE COMPONENTS

OBJECT	**DESCRIPTION**
Connection	Creates a connection to your data source
Command	Provides access to commands to execute against your data source
DataReader	Provides a read-only, forward-only stream containing your data
DataSet	Provides an in-memory representation of your data source(s)
DataAdapter	Serves as an ambassador between your DataSet and data source, proving the mapping instructions between the two

Figure 6.2 summarizes the ADO.NET object model. If you're familiar with classic ADO, you'll see that ADO.NET completely factors out the data source from the actual data. Each object exposes a large number of properties and methods, which are discussed in this and following chapters.

FIGURE 6.2
The ADO Framework



NOTE *If you have worked with collection objects, this experience will be a bonus to programming with ADO.NET. ADO.NET contains a collection-centric object model, which makes programming easy if you already know how to work with collections.*

Four core objects belong to .NET data providers, within the ADO.NET managed provider architecture: the Connection, Command, DataReader, and DataAdapter objects. The *Connection object* is the simplest one, because its role is to establish a connection to the database. The *Command object* exposes a Parameters collection, which contains information about the parameters of the command to be executed. If you've worked with ADO 2.*x*, the Connection and Command objects should seem familiar to you. The *DataReader object* provides fast access to read-only, forward-only data, which is reminiscent of a read-only, forward-only ADO RecordSet. The *DataAdapter object* contains Command objects that enable you to map specific actions to your data source. The DataAdapter is a mechanism for bridging the managed providers with the disconnected DataSets.

The *DataSet object* is not part of the ADO.NET managed provider architecture. The DataSet exposes a collection of DataTables, which in turn contain both DataColumn and DataRow collections. The DataTables collection can be used in conjunction with the DataRelation collection to create relational data structures.

First, you will learn about the connected layer by using the .NET data provider objects and touching briefly on the DataSet object. Next, you will explore the disconnected layer and examine the DataSet object in detail.

NOTE *Although there are two different namespaces, one for* `OleDb` *and the other for the* `SqlClient`*, they are quite similar in terms of their classes and syntax. As we explain the object model, we use generic terms, such as Connection, rather than SqlConnection. Because this book focuses on SQL Server development, we gear our examples toward SQL Server data access and manipulation.*

In the following sections, you'll look at the five major objects of ADO.NET in detail. You'll examine the basic properties and methods you'll need to manipulate databases, and you'll find examples of how to use each object. ADO.NET objects also recognize events, which we discuss in Chapter 12, "More ADO.NET Programming."

The Connection Object

Both the `SqlConnection` and `OleDbConnection` namespaces inherit from the `IDbConnection` object. The Connection object establishes a connection to a database, which is then used to execute commands against the database or retrieve a DataReader. You use the `SqlConnection` object when you are working with SQL Server, and the `OleDbConnection` for all other data sources. The `ConnectionString` property is the most important property of the Connection object. This string uses name-value pairs to specify the database you want to connect to. To establish a connection through a Connection object, call its `Open()` method. When you no longer need the connection, call the `Close()` method to close it. To find out whether a Connection object is open, use its `State` property.

What Happened to Your ADO Cursors?

One big difference between classic ADO and ADO.NET is the way they handle cursors. In ADO 2.*x*, you have the option to create client- or server-side cursors, which you can set by using the `CursorLocation` property of the Connection object. ADO.NET no longer explicitly assigns cursors. This is a good thing.

Under classic ADO, many times programmers accidentally specify expensive server-side cursors, when they really mean to use the client-side cursors. These mistakes occur because the cursors, which sit in the COM+ server, are also considered client-side cursors. Using server-side cursors is something you should never do under the disconnected, *n*-tier design. You see, ADO 2.*x* wasn't originally designed for disconnected and remote data access. The `CursorLocation` property is used to handle disconnected and connected access within the same architecture. ADO.NET advances this concept by completely separating the connected and disconnected mechanisms into managed providers and DataSets, respectively.

In classic ADO, after you specify your cursor location, you have several choices in the type of cursor to create. You could create a static cursor, which is a disconnected, in-memory representation of your database. In addition, you could extend this static cursor into a forward-only, read-only cursor for quick database retrieval.

Under the ADO.NET architecture, there are no updateable server-side cursors. This prevents you from maintaining state for too long on your database server. Even though the DataReader does maintain state on the server, it retrieves the data rapidly as a stream. The ADO.NET DataReader works much like an ADO read-only, server-side cursor. You can think of an ADO.NET DataSet as analogous to an ADO client-side, static cursor. As you can see, you don't lose any of the ADO disconnected cursor functionality with ADO.NET; it's just architected differently.

Connecting to a Database

The first step to using ADO.NET is to connect to a data source, such as a database. Using the Connection object, you tell ADO.NET which database you want to contact, supply your username and password (so that the DBMS can grant you access to the database and set the appropriate privileges), and, possibly, set more options. The Connection object is your gateway to the database, and all the operations you perform against the database must go through this gateway. The Connection object encapsulates all the functionality of a data link and has the same properties. Unlike data links, however, Connection objects can be accessed from within your VB .NET code. They expose a number of properties and methods that enable you to manipulate your connection from within your code.

Note *You don't have to type this code by hand. The code for all the examples in this chapter is located on the companion CD that comes with this book. You can find many of this chapter's code examples in the solution file* `Working with ADO.NET.sln`. *Code related to the ADO.NET Connection object is listed behind the Connect To Northwind button on the startup form.*

Let's experiment with creating a connection to the Northwind database. Create a new Windows Application solution and place a command button on the Form; name it **Connect to Northwind**. Add the `Imports` statement for the `System.Data.SqlClient` name at the top of the form module. Now you can declare a Connection object with the following statement:

```
Dim connNorthwind As New SqlClient.SqlConnection()
```

As soon as you type the period after `SqlClient`, you will see a list with all the objects exposed by the `SqlClient` component, and you can select the one you want with the arrow keys. Declare the `connNorthwind` object in the button's click event.

Note *All projects on the companion CD use the setting* `(local)` *for the data source. In other words, we're assuming you have SQL Server installed on the local machine. Alternately, you could use* `localhost` *for the data source value.*

The *ConnectionString* Property

The `ConnectionString` property is a long string with several attributes separated by semicolons. Add the following line to your button's click event to set the connection:

```
connNorthwind.ConnectionString="data source=(local);"& _
  "initial catalog=Northwind;integrated security=SSPI;"
```

Replace the data source value with the name of your SQL Server, or keep the local setting if you are running SQL Server on the same machine. If you aren't using Windows NT integrated security, then set your user ID and password like so:

```
connNorthwind.ConnectionString="data source=(local);"& _
   "initial catalog=Northwind; user ID=sa;password=xxx"
```

Tip *Some of the names in the connection string also go by aliases. You can use* `Server` *instead of* `data source` *to specify your SQL Server. Instead of* `initial catalog`*, you can specify* `database`*.*

Those of you who have worked with ADO 2.*x* might notice something missing from the connection string: the provider value. Because you are using the `SqlClient` namespace and the .NET Framework, you do not need to specify an OLE DB provider. If you were using the `OleDb` namespace, then you would specify your provider name-value pair, such as `Provider=SQLOLEDB.1`.

Overloading the Connection Object Constructor

One of the nice things about the .NET Framework is that it supports constructor arguments by using overloaded constructors. You might find this useful for creating your ADO.NET objects, such as your database Connection. As a shortcut, instead of using the `ConnectionString` property, you can pass the string right into the constructor, as such:

```
Dim connNorthwind as New SqlConnection _
("data source=localhost; initial catalog=Northwind; user ID=sa;password=xxx")
```

Or you could overload the constructor of the connection string by using the following:

```
Dim myConnectString As String = "data source=localhost; initial
   catalog=Northwind; user ID=sa;password=xxx"
```

You have just established a connection to the SQL Server Northwind database. As you remember from Chapter 3, you can also do this visually from the Server Explorer. The `ConnectionString` property of the Connection object contains all the information required by the provider to establish a connection to the database. As you can see, it contains all the information that you see in the Connection properties tab when you use the visual tools.

Keep in mind that you can also create connections implicitly by using the DataAdapter object. You will learn how to do this when we discuss the DataAdapter later in this section.

In practice, you'll never have to build connection strings from scratch. You can use the Server Explorer to add a new connection, or use the appropriate ADO.NET data component wizards, as you did in Chapter 3. These visual tools will automatically build this string for you, which you can see in the properties window of your Connection component.

TIP *The connection pertains more to the database server rather than the actual database itself. You can change the database for an open SqlConnection, by passing the name of the new database to the* `ChangeDatabase()` *method.*

The *Open ()* Method

After you have specified the `ConnectionString` property of the Connection object, you must call the `Open()` method to establish a connection to the database. You must first specify the `ConnectionString` property and then call the `Open()` method without any arguments, as shown here (`connNorthwind` is the name of a Connection object):

```
connNorthwind.Open()
```

NOTE *Unlike ADO 2.x, the* `Open()` *method doesn't take any optional parameters. You can't change this feature because the* `Open()` *method is not overridable.*

The *Close ()* Method

Use the Connection object's `Close()` method to close an open connection. Connection pooling provides the ability to improve your performance by reusing a connection from the pool if an appropriate one is available. The OleDbConnection object will automatically pool your connections for you. If you have connection pooling enabled, the connection is not actually released, but remains alive in memory and can be used again later. Any pending transactions are rolled back.

NOTE *Alternately, you could call the* `Dispose()` *method, which also closes the connection:* `connNorthwind.Dispose()`

You must call the `Close()` or `Dispose()` method, or else the connection will not be released back to the connection pool. The .NET garbage collector will periodically remove memory references for expired or invalid connections within a pool. This type of lifetime management improves the performance of your applications because you don't have to incur expensive shutdown costs. However, this mentality is dangerous with objects that tie down server resources. Generational garbage collection polls for objects that have been recently created, only periodically checking for those objects that have been around longer. Connections hold resources on your server, and because you don't get deterministic cleanup by the garbage collector, you must make sure you explicitly close the connections that you open. The same goes for the DataReader, which also holds resources on the database server.

The Command Object

After you instantiate your connection, you can use the Command object to execute commands that retrieve data from your data source. The Command object carries information about the command to be executed. This command is specified with the control's `CommandText` property. The `CommandText` property can specify a table name, an SQL statement, or the name of an SQL Server stored procedure. To specify how ADO will interpret the command specified with the `CommandText` property, you must assign the proper constant to the `CommandType` property. The `CommandType` property recognizes the enumerated values in the `CommandType` structure, as shown in Table 6.2.

TABLE 6.2: SETTINGS OF THE `CommandType` PROPERTY

CONSTANT	DESCRIPTION
`Text`	The command is an SQL statement. This is the default `CommandType`.
`StoredProcedure`	The command is the name of a stored procedure.
`TableDirect`	The command is a table's name. The Command object passes the name of the table to the server.

When you choose `StoredProcedure` as the `CommandType`, you can use the `Parameters` property to specify parameter values if the stored procedure requires one or more input parameters, or it returns one or more output parameters. The `Parameters` property works as a collection, storing the various attributes of your input and output parameters. For more information on specifying parameters with the Command object, see Chapter 8, "Data-Aware Controls."

Executing a Command

After you have connected to the database, you must specify one or more commands to execute against the database. A command could be as simple as a table's name, an SQL statement, or the name of a stored procedure. You can think of a Command object as a way of returning streams of data results to a DataReader object or caching them into a DataSet object.

Command execution has been seriously refined since ADO 2.*x*., now supporting optimized execution based on the data you return. You can get many different results from executing a command:

- If you specify the name of a table, the DBMS will return all the rows of the table.
- If you specify an SQL statement, the DBMS will execute the statement and return a set of rows from one or more tables.
- If the SQL statement is an action query, some rows will be updated, and the DBMS will report the number of rows that were updated but will not return any data rows. The same is true for stored procedures:
  - If the stored procedure selects rows, these rows will be returned to the application.
  - If the stored procedure updates the database, it might not return any values.

TIP *As we have mentioned, you should prepare the commands you want to execute against the database ahead of time and, if possible, in the form of stored procedures. With all the commands in place, you can focus on your VB .NET code. In addition, if you are performing action queries and do not want results being returned, specify the* `NOCOUNT ON` *option in your stored procedure to turn off the "rows affected" result count.*

You specify the command to execute against the database with the Command object. The Command objects have several methods for execution: the `ExecuteReader()` method returns a forward-only, read-only DataReader, the `ExecuteScalar()` method retrieves a single result value, and the `ExecuteNonQuery()` method doesn't return any results. We discuss the `ExecuteXmlReader()` method, which returns the XML version of a DataReader, in Chapter 7, "ADO.NET Programming."

NOTE *ADO.NET simplifies and streamlines the data access object model. You no longer have to choose whether to execute a query through a Connection, Command, or RecordSet object. In ADO.NET, you will always use the Command object to perform action queries.*

You can also use the Command object to specify any parameter values that must be passed to the DBMS (as in the case of a stored procedure), as well as specify the transaction in which the command executes. One of the basic properties of the Command object is the `Connection` property, which specifies the Connection object through which the command will be submitted to the DBMS for execution. It is possible to have multiple connections to different databases and issue different commands to each one. You can even swap connections on the fly at runtime, using the same Command object with different connections. Depending on the database to which you want to submit a command, you must use the appropriate Connection object. Connection objects are a significant load on the server, so try to avoid using multiple connections to the same database in your code.

Why Are There So Many Methods to Execute a Command?

Executing commands can return different types of data, or even no data at all. The reason why there are separate methods for executing commands is to optimize them for different types of return values. This way, you can get better performance if you can anticipate what your return data will look like. If you have an `AddNewCustomer` stored procedure that returns the primary key of the newly added record, you would use the `ExecuteScalar()` method. If you don't care about returning a primary key or an error code, you would use the `ExecuteNonQuery()`. In fact, now that error raising, rather than return codes, has become the de facto standard for error handling, you should find yourself using the `ExecuteNonQuery()` method quite often.

Why not use a single overloaded `Execute()` method for all these different flavors of command execution? Initially, Microsoft wanted to overload the `Execute()` method with all the different versions, by using the DataReader as an optional output parameter. If you passed the DataReader in, then you would get data populated into your DataReader output parameter. If you didn't pass a DataReader in, you would get no results, just as the `ExecuteNonQuery()` works now. However, the overloaded `Execute()` method with the DataReader output parameter was a bit complicated to understand. In the end, Microsoft resorted to using completely separate methods and using the method names for clarification.

Selection queries return a set of rows from the database. The following SQL statement will return the company names for all customers in the Northwind database:

```
SELECT CompanyName FROM Customers
```

As you recall from Chapter 4, "Structured Query Language," SQL is a universal language for manipulating databases. The same statement will work on any database (as long as the database contains a table called `Customers` and this table has a `CompanyName` column). Therefore, it is possible to execute this command against the SQL Server Northwind database to retrieve the company names.

NOTE *For more information on the various versions of the sample databases used throughout this book, see the sections "Exploring the Northwind Database," and "Exploring the Pubs Database" in Chapter 2, "Basic Concepts of Relational Databases."*

Let's execute a command against the database by using the `connNorthwind` object you've just created to retrieve all rows of the `Customers` table. The first step is to declare a Command object variable and set its properties accordingly. Use the following statement to declare the variable:

```
Dim cmdCustomers As New SqlCommand
```

NOTE *If you do not want to type these code samples from scratch as you follow along, you can take a shortcut and load the code from the companion CD. The code in this walk-through is listed in the click event of the Create DataReader button located on the startup form for the* `Working with ADO.NET` *solution.*

Alternately, you can use the `CreateCommand()` method of the Connection object.

```
cmdCustomers = connNorthwind.CreateCommand()
```

Overloading the Command Object Constructor

Like the Connection object, the constructor for the Command object can also be overloaded. By overloading the constructor, you can pass in the SQL statement and connection, while instantiating the Command object—all at the same time. To retrieve data from the `Customers` table, you could type the following:

```
Dim cmdCustomers As OleDbCommand = New OleDbCommand _
("Customers", connNorthwind)
```

Then set its `CommandText` property to the name of the `Customers` table:

```
cmdCustomers.CommandType = CommandType.TableDirect
```

The `TableDirect` property is supported only by the OLE DB .NET data provider. The `TableDirect` is equivalent to using a `SELECT * FROM` *`tablename`* SQL statement. Why doesn't the SqlCommand object support this? Microsoft feels that when using specific .NET data providers, programmers should have better knowledge and control of what their Command objects are doing. You can cater to your Command objects more efficiently when you explicitly return all the records in a table by using an SQL statement or stored procedure, rather than depending on the `TableDirect` property to do so for you. When you explicitly specify SQL, you have tighter reign on how the data is returned, especially considering that the `TableDirect` property might not choose the most efficient execution plan.

The `CommandText` property tells ADO.NET how to interpret the command. In this example, the command is the name of a table. You could have used an SQL statement to retrieve selected rows from the `Customers` table, such as the customers from Germany:

```
strCmdText = "SELECT ALL FROM Customers"
strCmdText = strCmdText & "WHERE Country = 'Germany'"
cmdCustomers.CommandText = strCmdText
cmdCustomers.CommandType = CommandType.Text
```

By setting the `CommandType` property to a different value, you can execute different types of commands against the database.

Note *In previous versions of ADO, you are able to set the command to execute asynchronously and use the* `State` *property to poll for the current fetch status. In VB .NET, you now have full support of the threading model and can execute your commands on a separate thread with full control, by using the* `Threading` *namespace. We touch on threading and asynchronous operations in Chapter 11, "More ADO.NET Programming."*

Regardless of what type of data you are retuning with your specific `Execute()` method, the Command object exposes a `ParameterCollection` that you can use to access input and output parameters for a stored procedure or SQL statement. If you are using the `ExecuteReader()` method, you must first close your DataReader object before you are able to query the parameters collection.

Warning *For those of you who have experience working with parameters with OLE DB, keep in mind that you must use named parameters with the* `SqlClient` *namespace. You can no longer use the question mark character (?) as an indicator for dynamic parameters, as you had to do with OLE DB.*

The DataAdapter Object

The DataAdapter represents a completely new concept within Microsoft's data access architecture. The DataAdapter gives you the full reign to coordinate between your in-memory data representation and your permanent data storage source. In the OLE DB/ADO architecture, all this happened behind the scenes, preventing you from specifying how you wanted your synchronization to occur.

The DataAdapter object works as the ambassador between your data and data-access mechanism. Its methods give you a way to retrieve and store data from the data source and the DataSet object. This way, the DataSet object can be completely agnostic of its data source.

The DataAdapter also understands how to translate *deltagrams,* which are the DataSet changes made by a user, back to the data source. It does this by using different Command objects to reconcile the changes, as shown in Figure 6.3. We show how to work with these Command objects shortly.

The DataAdapter implicitly works with Connection objects as well, via the Command object's interface. Besides explicitly working with a Connection object, this is the only other way you can work with the Connection object.

The DataAdapter object is very "polite," always cleaning up after itself. When you create the Connection object implicitly through the DataAdapter, the DataAdapter will check the status of the connection. If it's already open, it will go ahead and use the existing open connection. However, if it's closed, it will quickly open and close the connection when it's done with it, courteously restoring the connection back to the way the DataAdapter found it.

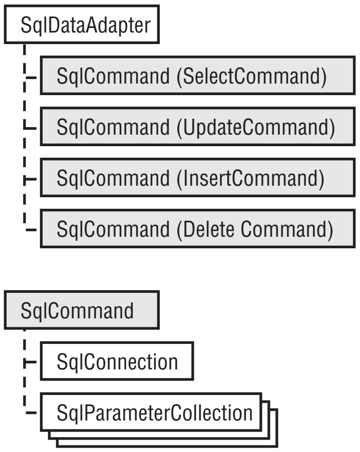

SqlDataAdapter
SqlCommand (SelectCommand)
SqlCommand (UpdateCommand)
SqlCommand (InsertCommand)
SqlCommand (Delete Command)
SqlCommand
SqlConnection
SqlParameterCollection

FIGURE 6.3 The ADO.NET `SqlClient` DataAdapter object model

The DataAdapter works with ADO.NET Command objects, mapping them to specific database update logic that you provide. Because all this logic is stored outside of the DataSet, your DataSet becomes much more liberated. The DataSet is free to collect data from many different data sources, relying on the DataAdapter to propagate any changes back to its appropriate source.

Populating a DataSet

Although we discuss the DataSet object in more detail later in this chapter, it is difficult to express the power of the DataAdapter without referring to the DataSet object.

The DataAdapter contains one of the most important methods in ADO.NET: the `Fill()` method. The `Fill()` method populates a DataSet and is the only time that the DataSet touches a live database connection. Functionally, the `Fill()` method's mechanism for populating a DataSet works much like creating a static, client-side cursor in classic ADO. In the end, you end up with a disconnected representation of your data.

The `Fill()` method comes with many overloaded implementations. A notable version is the one that enables you to populate an ADO.NET DataSet from a classic ADO RecordSet. This makes interoperability between your existing native ADO/OLE DB code and ADO.NET a breeze. If you wanted to populate a DataSet from an existing ADO 2.*x* RecordSet called `adoRS`, the relevant segment of your code would read:

```
Dim daFromRS As OleDbDataAdapter = New OleDbDataAdapter
Dim dsFromRS As DataSet = New DataSet
daFromRS.Fill(dsFromRS, adoRS)
```

WARNING *You must use the* `OleDb` *implementation of the DataAdapter to populate your DataSet from a classic ADO RecordSet. Accordingly, you would need to import the* `System.Data.OleDb` *namespace.*

Updating a Data Source from a DataSet by Using the DataAdapter

The DataAdapter uses the `Update()` method to perform the relevant SQL action commands against the data source from the deltagram in the DataSet.

TIP *The DataAdapter maps commands to the DataSet via the DataTable. Although the DataAdapter maps only one DataTable at a time, you can use multiple DataAdapters to fill your DataSet by using multiple DataTables.*

Using SqlCommand and SqlParameter Objects to Update the Northwind Database

NOTE *The code for the walkthrough in this section can be found in the* `Updating Data Using ADO.NET.sln` *solution file. Listing 6.1 is contained within the click event of the Inserting Data Using DataAdapters With Mapped Insert Commands button.*

The DataAdapter gives you a simple way to map the commands by using its `SelectCommand`, `UpdateCommand`, `DeleteCommand`, and `InsertCommand` properties. When you call the `Update()` method, the DataAdapter maps the appropriate update, add, and delete SQL statements or stored procedures to their appropriate Command object. (Alternately, if you use the `SelectCommand` property, this command would execute with the `Fill()` method.) If you want to perform an insert into the `Customers` table of the Northwind database, you could type the code in Listing 6.1.

Listing 6.1: Insert Commands by Using the DataAdapter Object with Parameters

```
Dim strSelectCustomers As String = "SELECT * FROM Customers ORDER BY CustomerID"
Dim strConnString As String = "data source=(local);" & _
  "initial catalog=Northwind;integrated security=SSPI;"
' We can't use the implicit connection created by the
' DataSet since our update command requires a
' connection object in its constructor, rather than a
' connection string
Dim connNorthwind As New SqlConnection(strConnString)
' String to update the customer record - it helps to
' specify this in advance so the CommandBuilder doesn't
' affect our performance at runtime
Dim strInsertCommand As String = _
  "INSERT INTO Customers(CustomerID,CompanyName) VALUES (@CustomerID,
@CompanyName)"
Dim daCustomers As New SqlDataAdapter()
Dim dsCustomers As New DataSet()
Dim cmdSelectCustomer As SqlCommand = New SqlCommand _
                (strSelectCustomers, connNorthwind)
Dim cmdInsertCustomer As New SqlCommand(strInsertCommand, connNorthwind)
daCustomers.SelectCommand = cmdSelectCustomer
daCustomers.InsertCommand = cmdInsertCustomer
connNorthwind.Open()
daCustomers.Fill(dsCustomers, "dtCustomerTable")
cmdInsertCustomer.Parameters.Add _
 (New SqlParameter _
 ("@CustomerID", SqlDbType.NChar, 5)).Value = "ARHAN"
cmdInsertCustomer.Parameters.Add _
 (New SqlParameter _
 ("@CompanyName", SqlDbType.VarChar, 40)).Value = "Amanda Aman Apak Merkez Inc."
cmdInsertCustomer.ExecuteNonQuery()
connNorthwind.Close()
```

This code sets up both the `SelectCommand` and `InsertCommand` for the DataAdapter and executes the insert query with no results. To map the insert command with the values you are inserting, you use the `Parameters` property of the appropriate SqlCommand objects. This example adds parameters to the `InsertCommand` of the DataAdapter. As you can see from the DataAdapter object model in Figure 6.3, each of the SqlCommand objects supports a `ParameterCollection`.

As you can see, the `Insert` statement need not contain all the fields in the parameters—and it usually doesn't. However, you must specify all the fields that can't accept Null values. If you don't, the DBMS will reject the operation with a trappable runtime error. In this example, only two of the new row's fields are set: the `CustomerID` and the `CompanyName` fields, because neither can be Null.

WARNING *In this code, notice that you can't use the implicit connection created by the DataSet. This is because the* `InsertCommand` *object requires a Connection object in its constructor rather than a connection string. If you don't have an explicitly created Connection object, you won't have any variable to pass to the constructor.*

TIP *Because you create the connection explicitly, you must make sure to close your connection when you are finished with it. Although implicitly creating your connection takes care of cleanup for you, it's not a bad idea to explicitly open the connection, because you might want to leave it open so you can execute multiple fills and updates.*

Each of the DataSet's Command objects have their own `CommandType` and `Connection` properties, which make them very powerful. Consider how you can use them to combine different types of command types, such as stored procedures and SQL statements. In addition, you can combine commands from multiple data sources, by using one database for retrievals and another for updates.

As you can see, the DataAdapter with its Command objects is an extremely powerful feature of ADO.NET. In classic ADO, you don't have any control of how your selects, inserts, updates, and deletes are handled. What if you wanted to add some specific business logic to these actions? You would have to write custom stored procedures or SQL statements, which you would call separately from your VB code. You couldn't take advantage of the native ADO RecordSet updates, because ADO hides the logic from you.

In summary, you work with a DataAdapter by using the following steps:

1. Instantiate your DataAdapter object.
2. Specify the SQL statement or stored procedure for the `SelectCommand` object. This is the only Command object that the DataAdapter requires.
3. Specify the appropriate connection string for the `SelectCommand`'s Connection object.
4. Specify the SQL statements or stored procedures for the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` objects. Alternately, you could use the `CommandBuilder` to dynamically map your actions at runtime. This step is not required.

5. Call the `Fill()` method to populate the DataSet with the results from the `SelectCommand` object.
6. If you used step 4, call the appropriate `Execute()` method to execute your command objects against your data source.

WARNING *Use the* `CommandBuilder` *sparingly, because it imposes a heavy performance overhead at runtime. You'll find out why in Chapter 9, "Working with DataSets."*

The DataReader Object

The DataReader object is a fast mechanism for retrieving forward-only, read-only streams of data. The SQL Server .NET provider have completely optimized this mechanism, so use it as often as you can for fast performance of read-only data. Unlike ADO RecordSets, which force you to load more in memory than you actually need, the DataReader is a toned-down, slender data stream, using only the necessary parts of the ADO.NET Framework. You can think of it as analogous to the server-side, read-only, forward-only cursor that you used in native OLE DB/ADO. Because of this server-side connection, you should use the DataReader cautiously, closing it as soon as you are finished with it. Otherwise, you will tie up your Connection object, allowing no other operations to execute against it (except for the `Close()` method, of course).

As we mentioned earlier, you can create a DataReader object by using the `ExecuteReader()` method of the Command object. You would use DataReader objects when you need fast retrieval of read-only data, such as populating combo-box lists.

Listing 6.2 depicts an example of how you create the DataReader object, assuming you've already created the Connection object `connNorthwind`.

LISTING 6.2: CREATING THE DATAREADER OBJECT

```
Dim strCustomerSelect as String = "SELECT * from Customers"
Dim cmdCustomers as New SqlCommand(strCustomerSelect, connNorthwind)
Dim drCustomers as SqlDataReader
connNorthwind.Open()
drCustomers = cmdCustomers.ExecuteReader()
```

NOTE *The code in Listing 6.2 can be found in the click event of the Create DataReader button on the startup form for the* `Working with ADO.NET` *solution on the companion CD.*

Notice that you can't directly instantiate the DataReader object, but must go through the Command object interface.

WARNING *You cannot update data by using the DataReader object.*

The DataReader absolves you from writing tedious `MoveFirst()` and `MoveNext()` navigation. The `Read()` method of the DataReader simplifies your coding tasks by automatically navigating to a position prior to the first record of your stream and moving forward without any calls to navigation methods, such as the `MoveNext()` method. To continue our example from Listing 6.2, you could retrieve the first column from all the rows in your DataReader by typing in the following code:

```
While(drCustomers.Read())
   Console.WriteLine(drCustomers.GetString(0))
End While
```

Note *The* `Console.WriteLine` *statement is similar to the* `Debug.Print()` *method you used in VB6.*

Because the DataReader stores only one record at a time in memory, your memory resource load is considerably lighter. Now if you wanted to scroll backward or make updates to this data, you would have to use the DataSet object, which we discuss in the next section. Alternately, you can move the data out of the DataReader and into a structure that is updateable, such as the DataTable or DataRow objects.

Warning *By default, the DataReader navigates to a point prior to the first record. Thus, you must always call the* `Read()` *method before you can retrieve any data from the DataReader object.*

The DataSet Object

There will come a time when the DataReader is not sufficient for your data manipulation needs. If you ever need to update your data, or store relational or hierarchical data, look no further than the DataSet object. Because the DataReader navigation mechanism is linear, you have no way of traversing between relational or hierarchical data structures. The DataSet provides a liberated way of navigating through both relational and hierarchical data, by using array-like indexing and tree walking, respectively.

Unlike the managed provider objects, the DataSet object and friends do not diverge between the `OleDb` and `SqlClient` .NET namespaces. You declare a DataSet object the same way regardless of which .NET data provider you are using:

```
Dim dsCustomer as DataSet
```

Realize that DataSets stand alone. A DataSet is not a part of the managed data providers and knows nothing of its data source. The DataSet has no clue about transactions, connections, or even a database. Because the DataSet is data source agnostic, it needs something to get the data to it. This is where the DataAdapter comes into play. Although the DataAdapter is not a part of the DataSet, it understands how to communicate with the DataSet in order to populate the DataSet with data.

DataSets and XML

The DataSet object is the nexus where ADO.NET and XML meet. The DataSet is persisted as XML, and only XML. You have several ways of populating a DataSet: You can traditionally load from a database or reverse engineer your XML files back into DataSets. You can even create your own customized application data without using XML or a database, by creating custom DataTables and DataRows. We show you how to create DataSets on the fly in this chapter in the section "Creating Custom DataSets."

DataSets are perfect for working with data transfer across Internet applications, especially when working with WebServices. Unlike native OLE DB/ADO, which uses a proprietary COM protocol, DataSets transfer data by using native XML serialization, which is a ubiquitous data format. This makes it easy to move data through firewalls over HTTP. Remoting becomes much simpler with XML over the wire, rather than the heavier binary formats you have with ADO RecordSets. We demonstrate how you do this in Chapter 16, "Working with WebServices."

As we mentioned earlier, DataSet objects take advantage of the XML model by separating the data storage from the data presentation. In addition, DataSet objects separate navigational data access from the traditional set-based data access. We show you how DataSet navigation differs from RecordSet navigation later in this chapter in Table 6.4.

DataSets versus RecordSets

As you can see in Figure 6.4, DataSets are much different from tabular RecordSets. You can see that they contain many types of nested collections, such as relations and tables, which you will explore throughout the examples in this chapter.

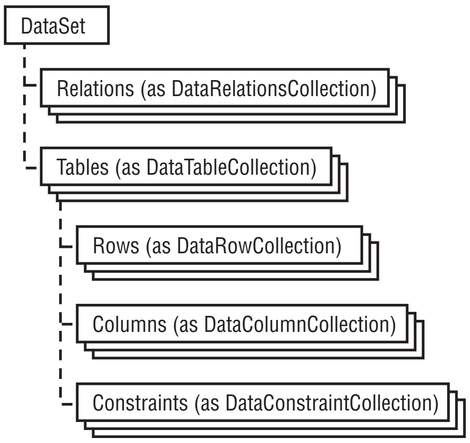

DataSet
Relations (as DataRelationsCollection)
Tables (as DataTableCollection)
Rows (as DataRowCollection)
Columns (as DataColumnCollection)
Constraints (as DataConstraintCollection)

FIGURE 6.4
The ADO.NET DataSet object model

What's so great about DataSets? You're happy with the ADO 2.*x* RecordSets. You want to know why you should migrate over to using ADO.NET DataSets. There are many compelling reasons. First, DataSet objects separate all the disconnected logic from the connected logic. This makes them easier to work with. For example, you could use a DataSet to store a web user's order information for their online shopping cart, sending deltagrams to the server as they update their order information. In fact, almost any scenario where you collect application data based on user interaction is a good candidate for using DataSets. Using DataSets to manage your application data is much easier than working with arrays, and safer than working with connection-aware RecordSets.

Another motivation for using DataSets lies in their capability to be safely cached with web applications. Caching on the web server helps alleviate the processing burden on your database servers. ASP caching is something you really can't do safely with a RecordSet, because of the chance that the RecordSet might hold a connection and state. Because DataSets independently maintain their own state, you never have to worry about tying up resources on your servers. You can even safely store the DataSet object in your ASP.NET Session object, which you are warned never to do with RecordSets. RecordSets are dangerous in a Session object; they can crash in some versions of ADO because of issues with marshalling, especially when you use open client-side cursors that aren't streamed. In addition, you can run into threading issues with ADO RecordSets, because they are apartment threaded, which causes your web server to run in the same thread

DataSets are great for remoting because they are easily understandable by both .NET and non-.NET applications. DataSets use XML as their storage and transfer mechanism. .NET applications don't even have to deserialize the XML data, because you can pass the DataSet much like you would a RecordSet object. Non-.NET applications can also interpret the DataSet as XML, make modifications using XML, and return the final XML back to the .NET application. The .NET application takes the XML and automatically interprets it as a DataSet, once again.

Last, DataSets work well with systems that require tight user interaction. DataSets integrate tightly with bound controls. You can easily display the data with DataViews, which enable scrolling, searching, editing, and filtering with nominal effort. You will have a better understanding of how this works when you read Chapter 8.

Now that we've explained how the DataSet gives you more flexibility and power than using the ADO RecordSet, examine Table 6.3, which summarizes the differences between ADO and ADO.NET.

TABLE 6.3: WHY ADO.NET IS A BETTER DATA TRANSFER MECHANISM THAN ADO

FEATURE SET	ADO	ADO.NET	ADO.NET'S ADVANTAGE
Data persistence format	RecordSet	Uses XML	With ADO.NET, you don't have data type restrictions.
Data transfer format	COM marshalling	Uses XML	ADO.NET uses a ubiquitous format that is easily transferable and that multiple platforms and sites can readily translate. In addition, XML strings are much more manageable than binary COM objects.
Web transfer protocol	You would need to use DCOM to tunnel through Port 80 and pass proprietary COM data, which firewalls could filter out.	Uses HTTP	ADO.NET data is more readily transferable though firewalls.

Let's explore how to work with the various members of the DataSet object to retrieve and manipulate data from your data source. Although the DataSet is designed for data access with any data source, in this chapter we focus on SQL Server as our data source.

Working with DataSets

Often you will work with the DataReader object when retrieving data, because it offers you the best performance. As we have explained, in some cases the DataSet's powerful interface for data manipulation will be more practical for your needs. In this section, we discuss techniques you can use for working with data in your DataSet.

The DataSet is an efficient storage mechanism. The DataSet object hosts multiple result sets stored in one or more DataTables. These DataTables are returned by the DBMS in response to the execution of a command. The DataTable object uses rows and columns to contain the structure of a result set. You use the properties and methods of the DataTable object to access the records of a table. Table 6.4 demonstrates the power and flexibility you get with ADO.NET when retrieving data versus classic ADO.

TABLE 6.4: WHY ADO.NET IS A BETTER DATA STORAGE MECHANISM THAN ADO

FEATURE SET	**ADO**	**ADO.NET**	**ADO.NET'S ADVANTAGE**
Disconnected data cache	Uses disconnected RecordSets, which store data into a single table.	Uses DataSets that store one or many DataTables.	Storing multiple result sets is simple in ADO.NET. The result sets can come from a variety of data sources. Navigating between these result sets is intuitive, using the standard collection navigation. DataSets never maintain state, unlike RecordSets, making them safer to use with *n*-tier, disconnected designs.
Relationship management	Uses `JOIN`s, which pull data into a single result table. Alternately, you can use the SHAPE syntax with the shaping OLE DB service provider.	Uses the DataRelation object to associate multiple DataTables to one another.	ADO.NET's DataTable collection sets the stage for more robust relationship management. With ADO, `JOIN`s bring back only a single result table from multiple tables. You end up with redundant data. The SHAPE syntax is cumbersome and awkward. With ADO.NET, DataRelations provide an object-oriented, relational way to manage relations such as constraints and cascading referential integrity, all within the constructs of ADO.NET. The ADO shaping commands are in an SQL-like format, rather than being native to ADO objects.

Continued on next page

TABLE 6.4: WHY ADO.NET IS A BETTER DATA STORAGE MECHANISM THAN ADO *(continued)*

FEATURE SET	ADO	ADO.NET	ADO.NET'S ADVANTAGE
Navigation mechanism	RecordSets give you the option to only view data sequentially.	DataSets have a nonlinear navigation model.	DataSets enable you to traverse the data among multiple DataTables, using the relevant DataRelations to skip from one table to another. In addition, you can view your relational data in a hierarchical fashion by using the tree-like structure of XML.

There are three main ways to populate a DataSet:

- After establishing a connection to the database, you prepare the DataAdapter object, which will retrieve your results from your database as XML. You can use the DataAdapter to fill your DataSet.
- You can read an XML document into your DataSet. The .NET Framework provides an `XMLDataDocument` namespace, which is modeled parallel to the ADO.NET Framework. You will explore this namespace in Chapter 7.
- You can use DataTables to build your DataSet in memory without the use of XML files or a data source of any kind. You will explore this option in the section "Updating Your Database by Using DataSets" later in this chapter.

Let's work with retrieving data from the Northwind database. First, you must prepare the DataSet object, which can be instantiated with the following statement:

```
Dim dsCustomers As New DataSet()
```

Assuming you've prepared your DataAdapter object, all you would have to call is the `Fill()` method. Listing 6.3 shows you the code to populate your DataSet object with customer information.

LISTING 6.3: CREATING THE DATASET OBJECT

```
Dim strSelectCustomers As String = "SELECT * FROM Customers ORDER BY CustomerID"
Dim strConnString As String = "data source=(local);" & _
  "initial catalog=Northwind;integrated security=SSPI;"
Dim daCustomers As New SqlDataAdapter(strSelectCustomers, strConnString)
Dim dsCustomers As New DataSet()
Dim connNorthwind As New SqlConnection(strConnString)

daCustomers.Fill(dsCustomers, "dtCustomerTable")
MsgBox(dsCustomers.GetXml, , "Results of Customer DataSet in XML")
```

NOTE *The code in Listing 6.3 can be found in the click event of the Create Single Table DataSet button on the startup form for the* `Working with ADO.NET` *solution on the companion CD.*

This code uses the `GetXml()` method to return the results of your DataSet as XML. The rows of the `Customers` table are retrieved through the `dsCustomers` object variable. The DataTable object within the DataSet exposes a number of properties and methods for manipulating the data by using the DataRow and DataColumn collections. You will explore how to navigate through the DataSet in the upcoming section, "Navigating Through DataSets." However, first you must understand the main collections that comprise a DataSet, the DataTable, and DataRelation collections.

The *DataTableCollection*

Unlike the ADO RecordSet, which contained only a single table object, the ADO.NET DataSet contains one or more tables, stored as a `DataTableCollection`. The `DataTableCollection` is what makes DataSets stand out from disconnected ADO RecordSets. You never could do something like this in classic ADO. The only choice you have with ADO is to nest RecordSets within RecordSets and use cumbersome navigation logic to move between parent and child RecordSets. The ADO.NET navigation model provides a user-friendly navigation model for moving between DataTables.

In ADO.NET, DataTables factor out different result sets that can come from different data sources. You can even dynamically relate these DataTables to one another by using DataRelations, which we discuss in the next section.

NOTE *If you want, you can think of a DataTable as analogous to a disconnected RecordSet, and the DataSet as a collection of those disconnected RecordSets.*

Let's go ahead and add another table to the DataSet created earlier in Listing 6.3. Adding tables is easy with ADO.NET, and navigating between the multiple DataTables in your DataSet is simple and straightforward. In the section "Creating Custom DataSets," we show you how to build DataSets on the fly by using multiple DataTables. The code in Listing 6.4 shows how to add another DataTable to the DataSet that you created in Listing 6.3.

NOTE *The code in Listing 6.4 can be found in the click event of the Create DataSet With Two Tables button on the startup form for the* `Working with ADO.NET` *solution on the companion CD.*

LISTING 6.4: ADDING ANOTHER DATATABLE TO A DATASET

```
Dim strSelectCustomers As String = "SELECT * FROM Customers ORDER BY CustomerID"
Dim strSelectOrders As String = "SELECT * FROM Orders"
Dim strConnString As String = "data source=(local);" & _
   "initial catalog=Northwind;integrated security=SSPI;"
Dim daCustomers As New SqlDataAdapter(strSelectCustomers, strConnString)
Dim dsCustomers As New DataSet()
Dim daOrders As New SqlDataAdapter(strSelectOrders, strConnString)
daCustomers.Fill(dsCustomers, "dtCustomerTable")
daOrders.Fill(dsCustomers, "dtOrderTable")
Console.WriteLine(dsCustomers.GetXml)
```

WARNING *DataTables are conditionally case sensitive. In Listing 6.4, the DataTable is called* `dtCustomerTable`*. This would cause no conflicts when used alone, whether you referred to it as* `dtCustomerTable` *or* `dtCUSTOMERTABLE`*. However, if you had another DataTable called* `dtCUSTOMERTABLE`*, it would be treated as an object separate from* `dtCustomerTable`*.*

As you can see, all you had to do was create a new DataAdapter to map to your `Orders` table, which you then filled into the DataSet object you had created earlier. This creates a collection of two DataTable objects within your DataSet. Now let's explore how to relate these DataTables together.

The DataRelation Collection

The DataSet object eliminates the cumbersome shaping syntax you had to use with ADO RecordSets, replacing it with a more robust relationship engine in the form of DataRelation objects. The DataSet contains a collection of DataRelation objects within its `Relations` property. Each DataRelation object links disparate DataTables by using referential integrity such as primary keys, foreign keys, and constraints. The DataRelation doesn't have to use any joins or nested DataTables to do this, as you had to do with ADO RecordSets.

In classic ADO, you create relationships by nesting your RecordSets into a single tabular RecordSet. Aside from being clumsy to use, this mechanism also made it awkward to dynamically link disparate sets of data.

With ADO.NET, you can take advantage of new features such as cascading referential integrity. You can do this by adding a `ForeignKeyConstraint` object to the `ConstraintCollection` within a DataTable. The `ForeignKeyConstraint` object enforces referential integrity between a set of columns in multiple DataTables. As we explained in Chapter 2, in the "Database Integrity" section, this will prevent orphaned records. In addition, you can cascade your updates and deletes from the parent table down to the child table.

Listing 6.5 shows you how to link the `CustomerID` column of your `Customer` and `Orders` DataTables. Using the code from Listing 6.3, all you have to do is add a new declaration for your DataRelation.

LISTING 6.5: USING A SIMPLE DATARELATION

```
Dim drCustomerOrders As DataRelation = New DataRelation("CustomerOrderRelation",
   dsCustomers.Tables("Customers").Columns("CustomerID"),
   dsCustomers.Tables("Orders").Columns("CustomerID"))
   dsCustomers.Relations.Add(drCustomerOrders)
```

NOTE *The code in Listing 6.5 can be found in the click event of the Using Simple DataRelations button on the startup form for the* `Working with ADO.NET` *solution on the companion CD.*

As you can with other ADO.NET objects, you can overload the DataRelation constructor. In this example, you pass in three parameters. The first parameter indicates the name of the relation. This is similar to how you would name a relationship within SQL Server. The next two parameters indicate the two columns that you wish to relate. After creating the DataRelation object, you add it to the `Relations` collection of the DataSet object.

WARNING *The data type of the two columns you wish to relate must be identical.*

Listing 6.6 shows you how to use DataRelations between the `Customers` and `Orders` tables of the Northwind database to ensure that when a customer ID is deleted or updated, it is reflected within the `Orders` table.

LISTING 6.6: USING CASCADING UPDATES

```
Dim fkCustomerID As ForeignKeyConstraint
fkCustomerID = New ForeignKeyConstraint
  ("CustomerOrderConstraint", dsCustomers.Tables
  ("Customers").Columns("CustomerID"),
dsCustomers.Tables("Orders").Columns("CustomerID"))
fkCustomerID.UpdateRule = Rule.Cascade
fkCustomerID.AcceptRejectRule = AcceptRejectRule.Cascade
dsCustomers.Tables("CustomerOrder").Constraints.Add
(fkCustomerID)
dsCustomers.EnforceConstraints = True
```

NOTE *The code in Listing 6.6 can be found in the click event of the Using Cascading Updates button on the startup form for the* `Working with ADO.NET` *solution on the companion CD.*

In this example, you create a foreign key constraint with cascading updates and add it to the `ConstraintCollection` of your DataSet. First, you declare and instantiate a `ForeignKeyConstraint` object, as you did earlier when creating the DataRelation object. Afterward, you set the properties of the `ForeignKeyConstraint`, such as the `UpdateRule` and `AcceptRejectRule`, finally adding it to your `ConstraintCollection`. You have to ensure that your constraints activate by setting the `EnforceConstraints` property to `True`.

Navigating through DataSets

We already discussed navigation through a DataReader. To sum it up, as long as the DataReader's `Read()` method returns `True`, then you have successfully positioned yourself in the DataReader. Now let's discuss how you would navigate through a DataSet.

In classic ADO, to navigate through the rows of an ADO RecordSet, you use the `Move()` method and its variations. The `MoveFirst()`, `MovePrevious()`, `MoveLast()`, and `MoveNext()` methods take you to the first, previous, last, and next rows in the RecordSet, respectively. This forces you to deal with cursoring and absolute positioning. This makes navigation cumbersome because you have to first position yourself within a RecordSet and then read the data that you need.

In ADO 2.*x*, a fundamental concept in programming for RecordSets is that of the *current row*: to read the fields of a row, you must first move to the desired row. The RecordSet object supports a number of navigational methods, which enable you to locate the desired row, and the `Fields` property, which enables you to access (read or modify) the current row's fields. With ADO.NET, you no longer have to use fixed positioning to locate your records; instead, you can use array-like navigation.

Unlike ADO RecordSets, the concept of the current row no longer matters with DataSets. DataSets work like other in-memory data representations, such as arrays and collections, and use familiar navigational behaviors. DataSets provide an explicit in-memory representation of data in the form of a collection-based model. This enables you to get rid of the infamous `Do While Not rs.EOF() And Not rs.BOF()` loop. With ADO.NET, you can use the friendly `For Each` loop to iterate through the DataTables of your DataSet. If you want to iterate through the rows and columns within an existing DataTable named `tblCustomers`, stored in a `dsCustomers` DataSet, you could use the following loop in Listing 6.7.

LISTING 6.7: NAVIGATING THROUGH A DATASET

```
For Each tblCustomer In dsCustomers.Tables
        Dim rowCustomer As DataRow
        For Each rowCustomer In  tblCustomer.Rows
            Dim colCustomer As DataColumn
            For Each colCustomer In  thisTable.Columns
                Console.WriteLine (rowCustomer (colCustomer))
            Next colCustomer
        Next rowCustomer
    Next tblCustomer
```

This will print out the values in each column of the customers DataSet created in Listing 6.3. As you can see, the `For Each` logic saves you from having to monitor antiquated properties such as `EOF` and `BOF` of the ADO RecordSet.

DataTables contain collections of DataRows and DataColumns, which also simplify your navigation mechanism. Instead of worrying about the `RecordCount` property of RecordSets, you can use the traditional `UBound()` property to collect the number of rows within a DataTable. For the example in Listing 6.7, you can calculate the row count for the customer records by using the following statement:

```
UBound(rowCustomer)
```

DataTable Capacities

In classic ADO, you could specify *paged RecordSets*—the type of RecordSets displayed on web pages when the results of a query are too many to be displayed on a single page. The web server displays 20 or so records and a number of buttons at the bottom of the page that enable you to move quickly to another group of 20 records. This technique is common in web applications, and ADO supports a few properties that simplify the creation of paged RecordSets, such as the `AbsolutePage`, `PageSize`, and `PageCount` properties.

With ADO.NET, you can use the `MinimumCapacity` property to specify the number of rows you wish to bring back for a DataTable. The default setting is 25 rows. This setting is especially useful if you want to improve performance on your web pages in ASP.NET. If you want to ensure that only 50 customer records display for the `Customers` DataTable, you would specify the following:

```
dtCustomers.MinimumCapacity = 50
```

If you have worked with paged RecordSets, you will realize that this performance technique is much less involved than the convoluted paging logic you had to use in ADO 2.*x*.

Navigating a Relationship between Tables

ADO.NET provides a navigation model for navigating through DataTables by using the relationships that connect them. Keep in mind that relations work as separate objects. When you create the relationship between the `Customers` and `Orders` tables, you can't directly jump from a customer DataRow to the related order DataRows. You must open the DataRelation separately and then pull the related rows. This is fine with one-to-many relationships; however, if you are using one-to-one relationships, you should stick with SQL `JOIN` statements.

You will explore the many techniques you can do with your retrieved data later in this chapter. First, let's review basic ways of updating your data sources by using DataSets.

Updating Your Database by Using DataSets

The two connected and disconnected models of ADO.NET work very differently when updating the database. Connected, or managed, providers communicate with the database by using command-based updates. As we showed you in "The DataSet Object" section earlier, disconnected DataSets update the database by using a cached, batch-optimistic method. DataSets work independently from a connection, working with the deltagram of data on the disconnected DataSet and committing the changes only after you call the `Update()` method from the DataAdapter. The separation between the command-based model used with managed providers and the optimistic model carried out by the DataSet objects enables the programmer to make a distinction between server-side execution and cached execution.

WARNING *In ADO 2.x, there was a good amount of confusion regarding client-side cursors. Some implementations mistakenly used server-side cursors when they meant to use client-cursors on the application server. Don't confuse disconnected, cached DataSets as user-side data. The DataSets can also be stored on your middle tier, which you should consider as a client-side cache, even though it is stored on your application server. You'll explore how to use DataSets within your ASP.NET code in Part IV, "Data Access from the Web."*

To update data, you make changes to your DataSet and pass them up to the server. Obviously, you can't use the DataReader, because its forward-only, read-only nature can't be updated. There are many ways that you can make updates to a DataSet:

- Make changes to an existing DataSet which was retrieved from a query executed on your database server(s). Pass the changes to the data source via the Data Adapter.

- Load data from an XML file by using the `ReadXml()` method. Map the resulting DataSet to your data source by using the DataAdapter.
- Merge multiple DataSets by using the `Merge()` method, passing the results to the data source via the DataAdapter.
- Create a new DataSet with new schema and data on the fly, mapping it to a data source by using the DataAdapter.

As you can see, all these options have one thing in common: your changes are not committed back to the server until the DataAdapter intervenes. DataSets are completely unaware of where their data comes from and how their changes relate back to the appropriate data source. The DataAdapter takes care of all this.

Realize that updating a record is not always a straightforward process. What happens if a user changes the record after you have read it? And what will happen if the record you're about to update has already been deleted by another user? In this chapter, you will learn the basics of updating databases through the ADO.NET DataSet, assuming no concurrency is involved. However, we discuss the implications of concurrency at the end of this chapter. In the meantime, let's set up your ADO.NET objects to insert a customer row into the Northwind database.

Updating Your DataSet by Using the DataTable and DataRow Objects

Earlier in this chapter, we showed you how to update your database by using parameterized stored procedures. Although this is efficient for making single row changes, it isn't quite useful when you have a significant number of changes to pass to the server. What happens when you want to apply changes in bulk? Consider an e-commerce application that uses an online shopping cart. The shopping cart could have multiple rows of data that would be inserted and updated as the user browsed through the site. When it comes time to push these changes to the server, it would be much easier to pass them in one single batch, rather than call the stored procedure multiple times for each row that's modified.

In ADO 2.*x*, you use disconnected RecordSets along with the `UpdateBatch()` method to pass your changes on to the server. In ADO.NET, you pass the disconnected deltagram from the DataSet object to the DataAdapter `Update()` method. Once again, ADO.NET clearly draws the line between your data and your data source. The DataSet object doesn't directly contact the data source.

First, let's see how you can manage changes within a DataSet. As the user edits the in-memory cache, the changes are stored into a buffer and not yet committed to the DataSet. You can commit modifications to a DataSet by using the `AcceptChanges()` method of the DataSet, DataTable, or DataRow objects. If you execute this method on the parent object, it will propagate down onto the children. For example, if you call `AcceptChanges()` on the DataSet object, it will cascade down onto the DataTables within the DataSet's `Table` collection (likewise for a DataTable to its relevant DataRow collection).

When you insert a row into a DataTable, you can monitor the "dirtiness" of a row by examining the `RowState` property. Let's go ahead and add a new row to your `dsCustomers` DataSet. In Figure 6.5, we continue the logic that we used in Listing 6.3 to populate your `dsCustomers` DataSet.

NOTE *Until you call the* `Update()` *method, your DataSet changes will not be committed to your data source.*

First, let's look at the code that pulls down the data that you want to work with from your database into a DataSet. Using the existing DataSet, you will add a new row directly to the DataSet by using the DataTable and DataRow collections of the DataSet.

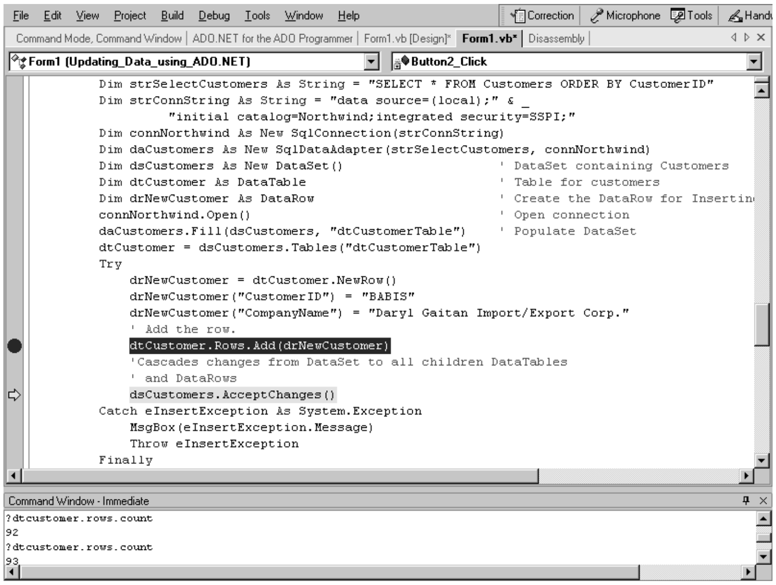
NOTE *The code depicted in Figure 6.5 can be found in the* `Updating Data using ADO.NET.sln` *solution file, within the click event of the Inserting Data With DataSets and DataTables button.*

As you see in Figure 6.5, DataSet updates are very straightforward. All you have to do is fill your DataSet, as we've shown you earlier in the chapter. Then you set up a new DataRow object with the DataTable's `NewRow()` method. The `Add()` collection of the `Rows` collection will add your new row to the collection. Finally, you call the `AcceptChanges()` method of the DataSet, which will automatically cascade all changes down to its inner DataTables and DataRows. Alternately, you could call the `AcceptChanges()` method specifically on the inner object you wish to update because the DataTable and DataRow also support the `AcceptChanges()` method.

As the note indicates, the source code for this example is available on the accompanying CD. Go ahead and load the code into Visual Studio .NET and place a breakpoint on the `Add()` method. Execute the code by pressing F5. When you get to your breakpoint, type the following in the Command window:

```
?dtcustomer.rows.count
```

FIGURE 6.5
Updating your DataSet object


File Edit View Project Build Debug Tools Window Help
Correction Microphone Tools Hand
Command Mode, Command Window | ADO.NET for the ADO Programmer | Form1.vb [Design]* | Form1.vb* | Disassembly
Form1 (Updating_Data_using_ADO.NET)
Button2_Click
Command Window - Immediate

```
        Dim strSelectCustomers As String = "SELECT * FROM Customers ORDER BY CustomerID"
        Dim strConnString As String = "data source=(local);" & _
                "initial catalog=Northwind;integrated security=SSPI;"
        Dim connNorthwind As New SqlConnection(strConnString)
        Dim daCustomers As New SqlDataAdapter(strSelectCustomers, connNorthwind)
        Dim dsCustomers As New DataSet()                   ' DataSet containing Customers
        Dim dtCustomer As DataTable                        ' Table for customers
        Dim drNewCustomer As DataRow                       ' Create the DataRow for Insertin
        connNorthwind.Open()                               ' Open connection
        daCustomers.Fill(dsCustomers, "dtCustomerTable")   ' Populate DataSet
        dtCustomer = dsCustomers.Tables("dtCustomerTable")
        Try
            drNewCustomer = dtCustomer.NewRow()
            drNewCustomer("CustomerID") = "BABIS"
            drNewCustomer("CompanyName") = "Daryl Gaitan Import/Export Corp."
            ' Add the row.
            dtCustomer.Rows.Add(drNewCustomer)
            'Cascades changes from DataSet to all children DataTables
            ' and DataRows
            dsCustomers.AcceptChanges()
        Catch eInsertException As System.Exception
            MsgBox(eInsertException.Message)
            Throw eInsertException
        Finally
```

```
?dtcustomer.rows.count
92
?dtcustomer.rows.count
93
```

WARNING *If you have difficulty working with the Command window, it might be because you are not in Immediate mode. If you see a > prompt, then this is most likely the case. Toggle the mode from Command mode to Immediate mode by typing* **`immed`** *at the prompt and pressing Enter. Now you should be able to debug your code.*

You will see the number of rows in your `Customers` table, within your DataSet, prior to making changes. Hit F11 to step into the `Add()` method. This will update your DataSet with the newly added row. Go back to the Command window and hit the Up arrow key and Enter to re-execute the row count statement. The results will show that the `Add()` method increments your row count in your DataRow by one record. However, if you compare the result to the data in the database, you will see that your data still has the same number of original rows. This is an important point. None of your changes will be committed to the data source until you call the `Update()` method of the DataAdapter object. Finish the execution of the code to commit the changes in your DataSet.

In summary, all you have to do is execute the following steps to commit updates to your DataSet:

1. Instantiate your DataSet and DataAdapter objects.
2. Fill your DataSet object from the DataAdapter object.
3. Manipulate your DataSet by using the DataRow objects.
4. Call the `AcceptChanges()` method of the DataSet, DataTable, or DataRow object to commit your changes to your DataSet.

Updating Your Data Source by Using the DataSet and DataAdapter

In this section, we show you how to insert a new row into your DataSet with the DataRow and DataTable objects. After you've updated your DataSet, we show you how you can commit those changes to the DataSet. Committing changes to a DataSet doesn't mean that they are committed to the database. To commit your changes to the database, you use the `Update()` method, which is similar to the `Fill()` method, only it works in reverse, updating your data source with the deltagram from the DataSet. Listing 6.8 contains the code that enables you to update a database with changes from a DataSet object.

NOTE *The code in Listing 6.8 can be found in the* `Updating Data Using ADO.NET` *solution on the companion CD, within the click event of the Committing Changes From Your DataSet To Your Database button.*

Although the `Update()` method is the only method you need to call to commit your changes back to the database, you must do some preparation work in advance. You must set up the appropriate action-based Command objects before you call the DataAdapter's `Update()` method. These Command objects map to the relevant insert, update, and delete stored procedures or SQL statements. Alternately, you can use the `CommandBuilder` object to dynamically generate the appropriate SQL statements for you.

LISTING 6.8: COMMITTING DATASET CHANGES TO A DATABASE

```
Dim strSelectCustomers As String = "SELECT * FROM Customers ORDER BY CustomerID"
Dim strConnString As String = "data source=(local);" & _
  "initial catalog=Northwind;integrated security=SSPI;"
Dim connNorthwind As New SqlConnection(strConnString)
Dim daCustomers As New SqlDataAdapter(strSelectCustomers, connNorthwind)
Dim dsCustomers As New DataSet()
Dim dtCustomer As DataTable
Dim drNewCustomer As DataRow
Dim custCB As SqlCommandBuilder = New SqlCommandBuilder(daCustomers)
connNorthwind.Open()
daCustomers.Fill(dsCustomers, "dtCustomerTable")
connNorthwind.Close()
dtCustomer = dsCustomers.Tables("dtCustomerTable")
Try
  drNewCustomer = dtCustomer.NewRow()
  drNewCustomer(0) = "OTISP"
  drNewCustomer(1) = "Otis P. Wilson Spaghetti House."
  dtCustomer.Rows.Add(drNewCustomer)
Dim drModified As
   DataRow() = dsCustomers.Tables("dtCustomerTable").Select(Nothing,Nothing,_
   DataViewRowState.Added)
  connNorthwind.Open()
  daCustomers.Update(drModified)
Catch eInsertException As Exception
   MsgBox(eInsertException.Message)
  Throw eInsertException
Finally
  connNorthwind.Close()
End Try
```

In summary, all you have to do is execute the following steps to update your data source from your DataSet, after you've made your changes to the DataSet:

1. Create a new row object that contains all the modified rows. You can use the `DataViewRowState` property to extract the appropriate rows. In our case, we used the `DataViewRowState.Added` value.
2. Call the `Update()` method of the DataAdapter object to send your changes back to the appropriate data source(s). Pass a copy of the DataRow containing your changes.

That's it. As you see, it's quite simple to add new rows to your database. Updates and deletes work the same way.

Managing DataSet Changes

Because the DataSet is inherently disconnected from the data source, it must manage its changes by itself. The DataSet supports several "dirty" flags that indicate whether changes have occurred. These flags come in the form of the `GetChanges()` and `HasChanges()` methods, which enable it to reconcile changes back to its data source via the DataAdapter object. These methods are used in conjunction with the `RowState` property, which we discuss next.

THE *ROWSTATE* PROPERTY

The `RowState` property enables you to track the condition of your rows. It works hand in hand with the `AcceptChanges()` method, which we discuss next. Until the `AcceptChanges()` method is called, the row state will be dirty. After `AcceptChanges()` has been called on the row, the row state will reflect a committed record that is no longer in flux. The `RowState` depends on what type of modification was made on the row, such as an insert, update, or delete. Table 6.5 shows you the possible values that the `RowState` might contain and why.

TABLE 6.5: VALUES OF THE *ROWSTATE* PROPERTY

CONSTANT	DESCRIPTION
`Added`	Occurs when a new row is first added to the `DataRowCollection`
`Deleted`	Indicates that the row was marked for deletion
`Detached`	Indicates that the row is "floating" and not yet attached to a `DataRowCollection`
`Modified`	Indicates that the row is "dirty"
`Unchanged`	Indicates that either the row was never touched in the first place, or the `AcceptChanges()` method was called, committing the changes to the row

If you want advanced information on how the `RowState` property works, please refer to Chapter 11, where we show you its importance with event-based programming.

THE *ACCEPTCHANGES ()* METHOD

Until you call this method, all the modified rows in your DataSet will remain in edit mode. The `AcceptChanges()` commits your modifications to a DataSet. The DataTable and DataRow objects also support this method. Keep in mind that this will not update your database, just your DataSet and friends. `AcceptChanges()` works incrementally, updating the DataSet with the modifications since the last time you called it. As we noted earlier, you can cascade your changes down to children objects. If you wanted to automatically accept changes for all the DataRows within a DataTable, you would need to call only the `AcceptChanges()` method on the DataTable, which automatically commits the changes for all its member DataRows.

THE *REJECTCHANGES ()* METHOD

If you decide not to commit the new row to the DataSet, call the `RejectChanges()` method. This method doesn't require any arguments. It simply deletes the newly added row or reverses the changes you made to an existing row.

THE *HASCHANGES ()* METHOD

The `HasChanges()` method queries whether a DataSet contains "dirty" rows. Generally, you would call this method before you called the `GetChanges()` method, so you don't unnecessarily retrieve changes that might not exist. This method can be overloaded by passing in the `RowState` as a parameter. By doing this, you can filter out specific change types. If you only wanted to query if the DataSet had any deletions, you would type:

```
If dsCustomers.HasChanges(DataRowState.Deleted)Then
   ' Do some logic to get the changes
End If
```

THE *GETCHANGES ()* METHOD

The `GetChanges()` method creates a DataSet containing the changes made to it since the last time you called the `AcceptChanges()` method. If you haven't called `AcceptChanges()`, then it will retrieve a copy of the DataSet with all your changes. You can optionally use the overloaded version of this method, which accepts the `DataRowState` as a parameter. This way, you can get only the changes based on a certain state. If you wanted to get only the deletions for a DataSet, you would first call the `HasChanges()` method to see if any deletions occurred and then retrieve the changes:

```
dsCustomers = dsCustomers.GetChanges(DataRowState.Deleted)
```

Merging

Another technique for working with DataSets uses the ability to merge results from multiple DataTables or DataSets. The merge operation can also combine multiple schemas together. The `Merge()` method enables you to extend one schema to support additional columns from the other, and vice versa. In the end, you end up with a union of both schemas and data. This is useful when you want to bring together data from heterogeneous data sources, or to add a subset of data to an existing DataSet. The merge operation is quite simple:

```
dsCustomers.Merge (dsIncomingCustomers)
```

Typed DataSets

There are many data typing differences between ADO and ADO.NET. In classic ADO, you have more memory overhead than ADO because the fields in a RecordSet are late-bound, returning data as the `Variant` data type. ADO.NET supports stricter data typing. ADO.NET uses the `Object`, rather than the `Variant` data type for your data. Although `Objects` are more lightweight than `Variants`, your code will be even more efficient if you know the type ahead of time. You could use the `GetString()` method to convert your column values to strings. This way, you avoid boxing your variables to the generic `Object` type. You can use similar syntax for the other data types, such as `GetBoolean()` or `GetGuid()`. Try to convert your values to the native format to reduce your memory overhead.

When you work with classic ADO, you experience performance degradation when you refer to your fields by name. You would type the following:

```
strName = rsCustomers.Fields("CustomerName").Value
```

Now, with ADO.NET, you can use strong typing to reference the fields of a `DataSet` directly by name, like so:

```
strName = dsCustomers.CustomerName
```

Because the values are strictly typed in ADO.NET, you don't have to write type-checking code. ADO.NET will generate a compile-time error if your have a type mismatch, unlike the ADO run-time errors you get much too late. With ADO.NET, if you try to pass a string to an integer field, you will raise an error when you compile the code.

Creating Custom DataSets

You don't need a database to create a DataSet. In fact, you can create your own DataSet without any data at all. The ADO.NET DataSet enables you to create new tables, rows, and columns from scratch. You can use these objects to build relationships and constraints, ending up with a mini-database into which you can load your data.

Listing 6.9 contains code that enables you to build a simple three-column online shopping cart DataSet on the fly. First, let's create a `BuildShoppingCart()` method that will create your table schema.

LISTING 6.9: CREATING A DATASET ON THE FLY

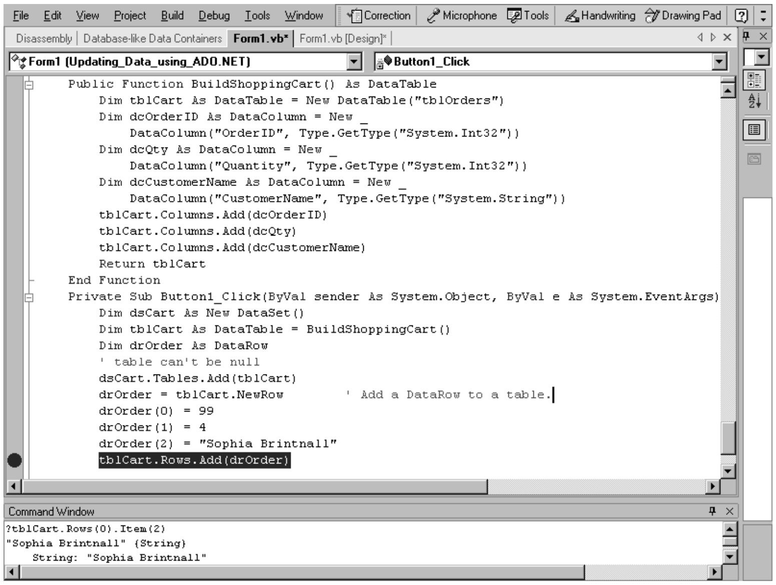
```
Public Function BuildShoppingCart() As DataTable
  Dim tblCart As DataTable = New DataTable("tblOrders")
  Dim dcOrderID As DataColumn = New
   DataColumn("OrderID", Type.GetType("System.Int32"))
  Dim dcQty As DataColumn = New
   DataColumn("Quantity",Type.GetType("System.Int32"))
  Dim dcCustomerName As DataColumn = New _
   DataColumn("CustomerName",
   Type.GetType("System.String"))
  tblCart.Columns.Add(dcOrderID)
  tblCart.Columns.Add(dcQty)
  tblCart.Columns.Add(dcCustomerName)
  Return tblCart
End Function
```

Now, all you have to do is set a DataTable variable to the results of your method and populate it. If you load the code from the companion CD, place a breakpoint on the `Add()` method of the DataRow collection, as shown in Figure 6.6. This way, you can use the Immediate mode of the Command window to see if your custom DataSet was successfully updated. With ADO.NET, it's easy to use array-like navigation to return the exact value you are looking for. In this example, you query the value of the customer name in the first row by using the `tblCart.Rows(0).Item(2)`statement. Figure 6.6 shows you the results.

TIP *Again, you can see the power of constructors. In this sample, you see how you can set your constructor to a method result.*

FIGURE 6.6

Populating your custom DataSet object


File Edit View Project Build Debug Tools Window Correction Microphone Tools Handwriting Drawing Pad
Disassembly Database-like Data Containers Form1.vb* Form1.vb [Design]*
Form1 (Updating_Data_using_ADO.NET)
Button1_Click
Command Window

```
Public Function BuildShoppingCart() As DataTable
    Dim tblCart As DataTable = New DataTable("tblOrders")
    Dim dcOrderID As DataColumn = New _
        DataColumn("OrderID", Type.GetType("System.Int32"))
    Dim dcQty As DataColumn = New _
        DataColumn("Quantity", Type.GetType("System.Int32"))
    Dim dcCustomerName As DataColumn = New _
        DataColumn("CustomerName", Type.GetType("System.String"))
    tblCart.Columns.Add(dcOrderID)
    tblCart.Columns.Add(dcQty)
    tblCart.Columns.Add(dcCustomerName)
    Return tblCart
End Function
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
    Dim dsCart As New DataSet()
    Dim tblCart As DataTable = BuildShoppingCart()
    Dim drOrder As DataRow
    ' table can't be null
    dsCart.Tables.Add(tblCart)
    drOrder = tblCart.NewRow        ' Add a DataRow to a table.
    drOrder(0) = 99
    drOrder(1) = 4
    drOrder(2) = "Sophia Brintnall"
    tblCart.Rows.Add(drOrder)
```

```
?tblCart.Rows(0).Item(2)
"Sophia Brintnall" {String}
    String: "Sophia Brintnall"
```

Being able to create your own DataSet from within your code enables you to apply many of the techniques discussed in this book. You can use these custom DataSets to store application data, without incurring the cost of crossing your network until you need to commit your changes.

Managing Concurrency

When you set up your DataSet, you should consider the type of locking, or concurrency control, that you will use. Concurrency control determines what will happen when two users attempt to update the same row.

ADO.NET uses an optimistic architecture, rather than a pessimistic model. *Pessimistic locking* locks the database when a record is retrieved for editing. Be careful when you consider pessimistic locking. Pessimistic locking extremely limits your scalability. You really can't use pessimistic locking in a system with a large number of users. Only certain types of designs can support this type of locking.

Consider an airline booking system. A passenger (let's call her Sam) makes a request to book a seat and retrieves a list of the available seats from the database. Sam selects a seat and updates the information in the database. Under optimistic locking, if someone else took her seat, she would see a message on her screen asking her to select a new one. Now let's consider what happens under pessimistic locking. After Sam makes a request for the list of available seats, she decides to go to lunch. Because pessimistic locking prevents other users from making changes when Sam is making edits, everyone else would be unable to book their seats. Of course, you could add some logic for lock timeouts, but the point is still the same. Pessimistic locking doesn't scale very well. In addition, disconnected architecture cannot support pessimistic locking because connections attach to the database only long enough to read or update a row, not long enough to maintain an indefinite lock. In classic ADO, you could choose between different flavors of optimistic and pessimistic locks. This is no longer the case. The .NET Framework supports only an optimistic lock type.

An *optimistic lock* type assumes that the data source is locked only at the time the data update commitment occurs. This means changes could have occurred while you were updating the disconnected data cache. A user could have updated the same `CompanyName` while you were making changes to the disconnected DataSet. Under optimistic locking, when you try to commit your `CompanyName` changes to the data source, you will override the changes made by the last user. The changes made by the last user could have been made after you had retrieved your disconnected DataSet. You could have updated the `CompanyName` for a customer, after someone else had updated the `Address`. When you push your update to the server, the updated address information would be lost. If you expect concurrency conflicts of this nature, you must make sure that your logic detects and rejects conflicting updates.

If you have worked with ADO 2.*x*, you can think of the `Update()` method of the DataAdapter object as analogous to the `UpdateBatch()` method you used with the RecordSet object. Both models follow the concept of committing your deltagram to the data source by using an optimistic lock type.

Understanding how locking works in ADO.NET is an essential part of building a solid architecture. ADO.NET makes great strides by advancing the locking mechanism. Let's take a look at how it changes from classic ADO in order to get an idea of how much power ADO.NET gives you.

In ADO 2.*x*, when you make changes to a disconnected RecordSet, you call the `UpdateBatch()` method to push your updates to the server. You really don't know what goes on under the covers and you hope that your inserts, updates, and deletes will take. You can't control the SQL statements that modify the database.

When you use optimistic concurrency, you still need some way to determine whether your server data has been changed since the last read. You have three choices with managing concurrency: time-date stamps, version numbers, and storing the original values.

Time-date stamps are a commonly used approach to tracking updates. The comparison logic checks to see if the time-date of the updated data matches the time-date stamp of original data in the database. It's a simple yet effective technique. Your logic would sit in your SQL statements or stored procedures, such as:

```
UPDATE Customers SET CustomerID = "SHAMSI",
 CustomerName = "Irish Twinkle SuperMart"
WHERE DateTimeStamp = olddatetimestamp
```

The second approach is to use version numbers, which is similar to using the time-date stamp, but this approach labels the row with version numbers, which you can then compare.

The last approach is to store the original values so that when you go back to the database with your updates, you can compare the stored values with what's in the database. If they match, you can safely update your data because no one else has touched it since your last retrieval. ADO.NET does data reconciliation natively by using the `HasVersion()` method of your DataRow object. The `HasVersion()` method indicates the condition of the updated DataRow object. Possible values for this property are `Current`, `Default`, `Original`, or `Proposed`. These values fall under the `DataRowVersion` enumeration. If you wanted to see whether the DataRow changes still contained original values, you could check to see if the DataRow has changed by using the `HasVersion()` method:

```
If r.HasVersion(datarowversion.Proposed) Then
' Add logic
End if
```

Summary

This concludes our discussion of the basic properties of the ADO.NET objects. After reading this chapter, you should be able to answer the questions that we asked you in the beginning:

- What are .NET data providers?
- What are the ADO.NET classes?
- What are the appropriate conditions for using a DataReader versus a DataSet?
- How does OLE DB fit into the picture?
- What are the advantages of using ADO.NET over classic ADO?
- How do you retrieve and update databases from ADO.NET?
- How does XML integration go beyond the simple representation of data as XML?

Although you covered a lot of ground in this chapter, there is still a good amount of ADO.NET functionality we haven't discussed. We use this chapter as a building block for the next few chapters. In the next chapter, we show you how to search and filter ADO.NET DataSets. You will learn about such things as data binding and data-aware controls in Chapter 8.