IN THIS CHAPTER

- The Two Managed Providers  51
- Managed Connections  51
- Managed Commands  55
- The DataReader  59
- Managed Commands with Stored Procedures  64
- The DataAdapter  67
- Table and Column Mappings  79
- Summary  82
In the last chapter, you began looking at how an ASP.NET data-driven solution works. You looked at the ADO.NET object model, and how to build a `DataSet` dynamically. In this chapter, you’ll dig into the two data access Managed Providers offered in ADO.NET: the SQL Managed Provider and the OleDB Managed Provider.

In this chapter, you’ll learn the following:

• How the .NET Managed Providers are a bridge from the application, such as an ASP.NET Web Form, to a data store, such as Microsoft SQL Server.
• How to create Managed Connections to connect to a data store.
• How to use Managed Commands to execute SQL statements on a database.
• How to use `DataAdapters` to retrieve data and populate a `DataSet`.
• How to create custom table and column mappings.

Managed Providers, as shown in Figure 3.1, are ADO.NET’s bridge from an application, such as an ASP.NET Web Form to the data source. Data sources include Microsoft’s SQL Server, Access, Oracle, or any other such data storage device.

![Figure 3.1](image)

**FIGURE 3.1**

*Managed Providers are the bridge from a data store to a .NET application.*

The Managed Providers have four core components:

• **Connection**
  
The Connection represents a unique session to a data store. This might be manifested as a network connection in a client/server database application.
• **Command**
  The Command represents a SQL statement to be executed on a data store.

• **DataReader**
  The DataReader is a forward-only, read-only stream of data records from a data store to a client.

• **DataAdapter**
  The DataAdapter represents a set of Commands and a Connection which are used to retrieve data from a data store and fill a DataSet.

**The Two Managed Providers**

ADO.NET, the successor to Microsoft’s highly successful ActiveX Data Objects (ADO), offers two Managed Providers. These providers are similar in their object model, but are chosen at design-time based on the data provider being used. The SQL Managed Provider offers a direct link into Microsoft’s SQL Server database application (version 7.0 or higher), while the OleDb Managed Provider is used for all other data providers. Following is a brief description of each of the Managed Providers. Throughout this chapter we will show you how the Managed Providers work, and specify when a particular object, property, method or event is unique to only one of the Managed Providers.

**OleDb Managed Provider**

The **OleDb Managed Provider** uses native OLEDB and COM Interop to establish a connection to a data store and negotiate commands. The OleDb Managed Provider is the data access provider to use when you are working with data from any data source that is not Microsoft’s SQL Server 7.0 or higher. To use the OleDb Managed Provider, you must import the `System.Data.OleDb` namespace.

**SQL Managed Provider**

The **SQL Managed Provider** is designed to work directly with Microsoft SQL Server 7.0 or greater. It connects and negotiates directly with SQL Server without using OLEDB. This provides a better performance model than the OleDb Managed Provider, but it’s restricted to use with Microsoft SQL Server 7.0 or higher. To use the SQL Managed Provider, you must import the `System.Data.SqlClient` namespace.

**Managed Connections**

Much like classic ADO, the OleDb and SQL Managed Connection objects (`OleDbConnection` and `SqlConnection`) provide a set of properties that might be familiar to you. These are listed in Table 3.1. Properties that apply to only one of the Managed Providers are indicated.
TABLE 3.1 Managed Connection Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectionState</td>
<td>Gets or sets the string used to open a data store.</td>
</tr>
<tr>
<td>ConnectionTimeout</td>
<td>Gets or sets the time to wait while establishing a connection before terminating the attempt and generating an error.</td>
</tr>
<tr>
<td>Container</td>
<td>Returns the IContainer that contains the object.</td>
</tr>
<tr>
<td>Database</td>
<td>Gets or sets the name of the current database or the database to be used once a connection is open.</td>
</tr>
<tr>
<td>DataSource</td>
<td>Gets or sets the name of the database to connect to.</td>
</tr>
<tr>
<td>PacketSize</td>
<td>Gets the size of the packets the data is transferred in.</td>
</tr>
<tr>
<td>Provider</td>
<td>Gets or sets the name of the OLEDB provider.</td>
</tr>
<tr>
<td>ServerVersion</td>
<td>Gets a string containing the version of the connected SQL Server.</td>
</tr>
<tr>
<td>Site</td>
<td>Gets or sets the site of the component.</td>
</tr>
<tr>
<td>State</td>
<td>Gets the current state of the connection.</td>
</tr>
</tbody>
</table>

OleDbConnection

The OleDb Managed Provider uses a ConnectionString property format identical to that of a classic ADO connection object. Listing 3.1 shows how to connect to an Access 2000 database using the OleDbConnection object.

WARNING

In the following code listing, the OleDb Managed Connection object is pointing to an Access 2000 database file using the path, C:\Program Files\Microsoft Office\Office\Samples\Northwind.mdb. This is the default path to the Northwind sample database that is installed when Access 2000 is installed. The path on your machine might vary. Alter the code as necessary.

LISTING 3.1 Connecting to an Access 2000 Database with the OleDbConnection

[VB]

01: <%@ Page Language="VB" %>
02: <%@ Import Namespace="System.Data.OleDb" %>
03: <script runat="server">
LISTING 3.1  Continued

04:  Sub Page_Load(Sender As Object, E As EventArgs)
05:   Dim myConnection As OleDbConnection
06:   myConnection = New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;"
➥   Data Source=C:\Program Files\Microsoft"
➥   Office\Office\Samples\Northwind.mdb;")
07:   myConnection.Open()
08:   ConnectionState.Text = myConnection.State.ToString()
09:   myConnection.Close()
10:  End Sub
11: </script>

[C#]
01: <%@ Page Language="C#" %>
02: <%@ Import Namespace="System.Data.OleDb" %>
03: <script runat="server">
04:  void Page_Load(Object sender, EventArgs e){
05:   OleDbConnection myConnection;
06:   myConnection = new OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;"
➥   Data Source=C:\\Program Files\\Microsoft"
➥   Office\\Office\\Samples\\Northwind.mdb;");
07:   myConnection.Open();
08:   ConnectionState.Text = myConnection.State.ToString();
09:   myConnection.Close();
10: }
11: </script>

[VB & C#]
12: <html>
13: <body>
14: <form runat="server" method="post">
15:  Connection State: <asp:Label runat="server" id="ConnectionState" />
16: </form>
17: </body>
18: </html>

In Listing 3.1, you import the System.Data.OleDb namespace on line 2. On line 5, you declare a variable for the OleDbConnection class and on line 6, you instantiate the OleDbConnection class, passing in the ConnectionString as the connection's only parameter. The ConnectionString property specifies that the OLEDB Provider is Microsoft.Jet.OLEDB.4.0, the provider necessary to connect to an Access 2000 database.

On line 7 you open the connection with the Open() method of the OleDbConnection class. On line 8 you set the Text property of an ASP.NET Label to the string representation of the State property of the OleDbConnection class.
In the C# example in Listing 3.1 you will notice that the ConnectionString property of the Managed Connection object uses a double slash (\) between the tree hierarchy of the path to the Northwind sample database file. This is because C# treats the slash (/) as an escape character in a string. Using a double slash (\) lets the compiler know that you really want to use a slash character in that spot.

**WARNING**

**NOTE**

If you are using Access 2000 with user name and password security, you might see an error indicating that Access can not find the installable ISAM. This error is related to your Access 2000 installation, and not the .NET Framework. For more information, see http://support.microsoft.com/support/kb/articles/Q209/805.ASP.

**SqlConnection**

The SQL Managed Provider uses a ConnectionString property format that’s similar to that of a classic ADO connection object. Since you know what the database application is from using the SQL Managed Provider, the Provider property isn’t required (it isn’t even allowed, for that matter). Listing 3.2 shows sample code for connecting to a Microsoft SQL Server database using the SqlConnection object.

**LISTING 3.2  Connecting to a SQL Server Database with the SqlConnection**

```vbnet
01: <%@ Page Language="VB" %>
02: <%@ Import Namespace="System.Data.SqlClient" %>
03: <script runat="server">
04:  Sub Page_Load(Sender As Object, E As EventArgs)
05:   Dim myConnection As SqlConnection
06:   myConnection = New SqlConnection("server=localhost;database=Northwind;uid=sa;pwd=")
07:   myConnection.Open()
08:   ConnectionState.Text = myConnection.State.ToString()
09:   myConnection.Close()
10:  End Sub
11: </script>
```
In Listing 3.2 you create a connection to a SQL Server database. The code in Listing 3.2 is nearly identical to that of Listing 3.1. The only two differences are on lines 2 and 6. On line 2 you import the System.Data.SqlClient namespace rather than the System.Data.OleDb namespace. This allows you access to the SQL Managed Provider classes, like the SqlConnection class. On line 6 you create an instance of the SqlConnection class and pass in the ConnectionString property as the only parameter. In the ConnectionString property you do not specify a provider since the SqlConnection is designed to connect only to a Microsoft SQL Server database.

**Managed Commands**

Managed Commands represent SQL syntax to be executed on the data store. Managed Commands can be simple SELECT statements or complex, parameterized commands.

Once a connection to a data store is established, you can retrieve, update, or insert data. One way of accomplishing this is to use a Managed Command. This is the most direct way to execute a SQL statement on a data store.
As with the Managed Connection object, there are both OleDb and SQL versions of the Managed Command—OleDbCommand and SqlCommand.

The Managed Command is similar to the classic ADO command object. In its simplest form, you create a Managed Command with the SQL statement and the connection (as either a ConnectionString or a Managed Connection object) as its parameters. In Listing 3.3 you create a SqlCommand to execute a simple SELECT statement against the Northwind database. This example will not render any output, but you will build on it in the following examples.

NOTE

For the bulk of this chapter I will be showing samples using the SQL Managed Provider. You can use the OleDb Managed Provider by changing the Managed Provider classes from SqlWidget to OleDbWidget. For example, in Listing 3.3 you could use the OleDbCommand class in replacement of the SqlCommand class. Remember that the OleDb Managed Provider uses the System.Data.OleDb namespace instead of the System.Data.SqlClient namespace.

LISTING 3.3 Creating a SqlCommand Object

[VB]

01: < %@ Page Language="VB" %>
02: < %@ Import Namespace="System.Data.SqlClient" %>
03: < script runat="server">
04: Sub Page_Load(Sender As Object, E As EventArgs)
05: Dim myConnection As SqlConnection
06: Dim myCommand As SqlCommand
07: myConnection = New SqlConnection("server=localhost;
➥ database=Northwind; uid=sa; pwd=")
08: myCommand = New SqlCommand("SELECT * FROM Customers", myConnection)
09: End Sub
10: </ script>

[C#]

01: < %@ Page Language="C#" %>
02: < %@ Import Namespace="System.Data.SqlClient" %>
03: < script runat="server">
04: void Page_Load(Object sender, EventArgs e){
05: SqlConnection myConnection;
06: SqlCommand myCommand;
07: myConnection = new SqlConnection("server=localhost;
➥ database=Northwind; uid=sa; pwd=");
08: myCommand = new SqlCommand("SELECT * FROM Customers", myConnection);
09: }
10: </ script>
In Listing 3.3 you create a Web Form that uses the SQL Managed Provider to create a
SqlConnection and a SqlCommand. To execute the SqlCommand on the database you call
one of the provided execute methods.

- **ExecuteNonQuery**: Executes a SQL statement that does not return any records.
- **ExecuteReader**: Returns a DataReader object.
- **ExecuteScalar**: Executes the SQL statement and returns the first column of the first row.
- **ExecuteXmlReader** (SQL Managed Provider only): Executes a SQL statement and
returns the results as an XML stream.

The only thing you're missing before calling one of the execute methods is an object to hold
the results that are returned. For this example you will use the ExecuteReader() method and
return the results as a DataReader object (SqlDataReader or OleDbDataReader). Once you
have a DataReader object you can work with the data. For now you will bind the DataReader
to a DataGrid server control. In Listing 3.4 you will use the ExecuteReader() method to
return the results of the command execution into a DataReader object.

### Listing 3.4 Executing a SqlCommand Object and Returning the Results in a DataReader

[VB]

```vbnet
01: <%@ Page Language="VB" %>
02: < %@ Import Namespace="System.Data.SqlClient" %>
03: <script runat="server">
04:  Sub Page_Load(Sender As Object, E As EventArgs)
05:   Dim myConnection As SqlConnection
06:   Dim myCommand As SqlCommand
07:   myConnection = New SqlConnection("
server=localhost;
→  database=Northwind; uid=sa; pwd=");
08:   myCommand = New SqlCommand("SELECT * FROM Customers", myConnection)
09:   myConnection.Open()
10:  Dim myDataGrid As SqlDataReader = myCommand.ExecuteReader()
11:  myDataGrid.DataSource = myDataReader
```
In Listing 3.4 you extend the code in Listing 3.3 to execute the SqlCommand object using the ExecuteReader() method. Since this method returns the results in a SqlDataReader object, you first create an instance of the SqlDataReader class on line 10, and set it to the returned result of the SqlCommand.ExecuteReader() method. You will notice that I create an instance of the SqlDataReader and assign it to the results of the ExecuteReader() on the same line. This is just another way of constructing an object in .NET, rather than doing the same thing across two lines of code.

Once the command has executed and the SqlDataReader has been created, you bind the SqlDataReader to a DataGrid server control (line 11) by setting the DataSource property of the DataGrid as the SqlDataReader. Once the DataSource property is set you call the DataBind() method of the DataGrid to bind the command results to the output of the DataGrid.
Figure 3.2 shows the ASP.NET Web Form with the results from the Managed Command execution in a DataGrid.

The DataReader

The DataReader provides a forward-only, read-only stream of data from the data store. The DataReader is best used when either there are many records in the result set and pulling them all in at once would use too much memory, or when you want to iterate through the records to work with the data returned. As a stream of records, the DataReader helps manage memory allocation. Rather than all of the records in the result set being returned at once and using up a chunk of memory on the server, the DataReader streams in one record at a time.

You’ve seen how to execute a command against a data store, and in the previous listings, the results weren’t too big. But imagine if your Managed Command returned a result set with over 100,000 records in it. Now imagine 1,000 users doing that all at the same time. It would use up the memory space for 100,000,000 records of data, and that could spell disaster for your Web application.

What would be ideal is a way to connect to the data store, bring the results back in a stream, and evaluate those results one record at a time. Ideally, this would only use up the memory for one record at a time.
This type of functionality is exactly what the `DataReader` provides. The `DataReader` is the classic “fire hose” of data access—a forward-only, read-only stream returned from the data store. In Listing 3.4, you bound a `DataGrid` server control to the `DataReader`. The `DataGrid` is covered in depth in Chapter 5, “Using a Basic DataGrid,” and Chapter 6, “Altering DataGrid Output.” This will set up the `DataGrid` to display the entire contents of the stream. However, with a data stream, you can step through the data that’s returned very easily and decide how to react to it.

The `DataReader` exposes a `Read()` method which advances to the next record in the stream. Using the `Read()` method you can iterate through the result set evaluating or working with the data.

```
[VB]
While myDataReader.Read
   'Do something with the current row
End While
[C#]
while (myDataReader.Read()){
   //Do something with the current row
}
```

You should be able to come up with a good reason to step through the results of a command execution and evaluate the results. This is something you’ve done relentlessly as a classic ASP developer (does `Do While Not RecordSet.EOF` sound familiar?). How often have you looped through ADO RecordSets, checking the values of a particular column and using `Response.Write` on the RecordSet row if the criterion is met? Or granted access to a page if the user name and password columns match the submitted values?

In Listing 3.5 you will use the `DataReader` class to iterate through the result set and add any record with the value “USA” to a new `DataTable`. The `DataTable` and `DataRow` classes you learned about in Chapter 2, “What Is ADO.NET?” are used in Listing 3.5.

**Listing 3.5  Evaluating Data with the DataReader**

```
[VB]
01: <%@ Page Language="VB" %>
02: <%@ Import Namespace="System.Data" %>
03: <%@ Import Namespace="System.Data.SqlClient" %>
04: <script runat="server">
05: Sub Page_Load(Sender As Object, E As EventArgs)
06:   Dim myConnection As SqlConnection
07:   Dim myCommand As SqlCommand
08:   Dim myDataTable As New DataTable
09:   Dim myRow As DataRow
```
LISTING 3.5  Continued

10:
11:  myConnection = New SqlConnection('server=localhost;
12:      database=Northwind; uid=sa; pwd=;')
13:  myCommand = New SqlCommand("SELECT * FROM Customers", myConnection)
14:  myConnection.Open()
15:  Dim myDataReader As SqlDataReader = myCommand.ExecuteReader()
16:  myDataTable.Columns.Add("CustomerID",
17:      System.Type.GetType("System.String"))
18:  myDataTable.Columns.Add("CompanyName",
19:      System.Type.GetType("System.String"))
20:  myDataTable.Columns.Add("Address",
21:      System.Type.GetType("System.String"))
22:  myDataTable.Columns.Add("City",
23:      System.Type.GetType("System.String"))
24:  myDataTable.Columns.Add("Region",
25:      System.Type.GetType("System.String"))
26:  myDataTable.Columns.Add("Country",
27:      System.Type.GetType("System.String"))
28:  While myDataReader.Read
29:    If myDataReader("Country") = 'USA' Then
30:      myRow = myDataTable.NewRow()
31:      myRow("CustomerID") = myDataReader("CustomerID")
32:      myRow("CompanyName") = myDataReader("CompanyName")
33:      myRow("Address") = myDataReader("Address")
34:      myRow("City") = myDataReader("City")
35:      myRow("Region") = myDataReader("Region")
36:      myRow("Country") = myDataReader("Country")
37:      myDataTable.Rows.Add(myRow)
38:    End If
39:  End While
40:  myDataGrid.DataSource = myDataTable
41:  myDataGrid.DataBind()
42:  myConnection.Close()
43: 
[C#]
01:  <%@ Page Language="C#" %>
02:  <%@ Import Namespace="System.Data" %>
03:  <%@ Import Namespace="System.Data.SqlClient" %>
04:  <script runat="server">
LISTING 3.5  Continued

05: void Page_Load(Object sender, EventArgs e){
06:   SqlConnection myConnection;
07:   SqlCommand myCommand;
08:   DataTable myDataTable = new DataTable();
09:   DataRow myRow;
10:   myConnection = new SqlConnection("server=localhost;
11:   database=Northwind; uid=sa; pwd=");
12:   myCommand = new SqlCommand("SELECT * FROM Customers", myConnection);
13:   myConnection.Open();
14:   SqlDataReader myDataReader = myCommand.ExecuteReader();
15:   myDataTable.Columns.Add("CustomerID",
16:     System.Type.GetType("System.String"));
17:   myDataTable.Columns.Add("CompanyName",
18:     System.Type.GetType("System.String"));
19:   myDataTable.Columns.Add("Address",
20:     System.Type.GetType("System.String"));
21:   myDataTable.Columns.Add("City",
22:     System.Type.GetType("System.String"));
23:   myDataTable.Columns.Add("Region",
24:     System.Type.GetType("System.String"));
25:   myDataTable.Columns.Add("Country",
26:     System.Type.GetType("System.String"));
27:   while(myDataReader.Read()){  
28:     if(myDataReader["Country"].ToString() == "USA"){
29:       myRow = myDataTable.NewRow();
30:       myRow["CustomerID"] = myDataReader["CustomerID"].ToString();
31:       myRow["CompanyName"] = myDataReader["CompanyName"].ToString();
32:       myRow["Address"] = myDataReader["Address"].ToString();
33:       myRow["City"] = myDataReader["City"].ToString();
34:       myRow["Region"] = myDataReader["Region"].ToString();
35:       myRow["Country"] = myDataReader["Country"].ToString();
36:       myDataTable.Rows.Add(myRow);
37:     }
38:   }
39:   myDataGrid.DataSource = myDataTable;
40:   myDataGrid.DataBind();
41:   myConnection.Close();
42: }
LISTING 3.5  Continued

[VB & C#]

42: <html>
43: <body>
44: <form runat="server" method="post">
45:  <asp:DataGrid runat="server" id="myDataGrid" />
46: </form>
47: </body>
48: </html>

In Chapter 2, you read an overview of the ADO.NET Document Object Model and saw samples of creating DataTables dynamically. Listing 3.5 makes use of that knowledge. On line 08, you create a DataTable dynamically. On lines 16–21, you add columns to the DataTable’s Columns collection. On lines 23–34, you step through the results of the SqlCommand execution with the DataReader.Read() method. If the “Country” field is “USA”, you create a new DataRow (line 25) and add the values of the current record in the DataReader stream to the row (lines 26–31). On line 32, you add the new DataRow object to the DataTable’s Rows collection. Finally, you bind the dynamically created DataTable to the DataGrid server control. You end up with a table limited to customers in the USA, as shown in Figure 3.3.

![Figure 3.3](image)

**Figure 3.3**

*You can use the DataReader.Read() method to iterate through the result set and react to the data.*
Managed Commands with Stored Procedures

Although the example in Listing 3.5 allows you to evaluate data and output only what is desired, a more economical way to get a restricted result set is to use a SQL statement with a WHERE clause. Rather than querying the data store and bringing back all of the records in the table, you can write a SQL statement or stored procedure to bring back only the data you want. Stored procedures typically give better performance to your application than a SQL statement passed via a command.

You can use the Managed Command to call a stored procedure. In many cases, the stored procedure will have the parameters you pass it to determine the result set, such as the country abbreviation in the previous examples.

Listing 3.6 is the syntax for a SQL stored procedure. In the SQL Server Enterprise Manager, create a new stored procedure in the Northwind database named GetCustomersByCountry.

**Listing 3.6**  GetCustomersByCountry Stored Procedure for SQL Server Northwind Database

```sql
CREATE PROCEDURE [GetCustomersByCountry]
@country varchar (50)
AS
SELECT * FROM Customers WHERE Country = @country
```

When using a Managed Command to execute a stored procedure you must set the CommandType property. The default value of the CommandType property is Text. To execute a stored procedure you set the CommandType property to CommandType.StoredProcedure and pass in the name of the stored procedure.

Just as classic ADO command objects have a Parameters collection for passing parameters to the stored procedure, ADO.NET Managed Command classes (SqlCommand and OleDbCommand) also have a Parameters collection. You can call to a stored procedure, passing the required parameters in the collection using the following steps:

1. Create a Managed Command object (SqlCommand or OleDbCommand).
2. Set the Managed Command’s CommandType property to CommandType.StoredProcedure.
3. Declare a parameter variable (SqlParameter or OleDbParameter).
4. Add a new instance of the parameter class to the Managed Command’s Parameters collection, passing in its name and data type (SqlDbType for SqlParameter and OleDbDbType for OleDbParameter—see Appendix B for a list of valid SqlDbType and OleDbTypes).

5. Set the parameter’s Direction property. (Optional—“Input” is the default.)

6. Set the Parameter’s Value property.

7. Repeat steps 4-6 for additional parameters.

8. Execute the Managed Command.

Listing 3.7 demonstrates these steps.

**WARNING**

When you’re using the SqlCommand object, the names of the parameters added to the Parameters collection must match the names of the markers in the stored procedure. The SQL Managed Provider will treat the parameters as named parameters, and it will look for markers in the stored procedure of the same name.

**LISTING 3.7** Calling a Parameterized Stored Procedure with a SqlCommand

```vbnet
<% @ Page Language="VB" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<script runat="server">
  Sub Page_Load(Sender As Object, E As EventArgs)
    Dim myConnection As SqlConnection
    Dim myCommand As SqlCommand
    Dim myParameter As SqlParameter
    myConnection = New SqlConnection("server=localhost; database=Northwind; uid=sa; pwd=")
    myCommand = New SqlCommand("GetCustomersByCountry", myConnection)
    myCommand.CommandType = CommandType.StoredProcedure
    myParameter = myCommand.Parameters.Add(New SqlParameter("@country", SqlDbType.VarChar, 50))
    myParameter.Direction = ParameterDirection.Input
    myParameter.Value = "USA"
    myConnection.Open()
    Dim myDataReader As SqlDataReader = myCommand.ExecuteReader()
    myDataGrid.DataSource = myDataReader
    myDataGrid.DataBind()
```
LISTING 3.7  Continued

22:   myConnection.Close()
23:  End Sub
24: </script>

[C#]
01: <%@ Page Language="C#" %>
02: <%@ Import Namespace="System.Data" %>
03: <%@ Import Namespace="System.Data.SqlClient" %>
04: <script runat="server">
05:  void Page_Load(Object sender, EventArgs e){
06:   SqlConnection myConnection;
07:   SqlCommand myCommand;
08:   SqlParameter myParameter;
09:   
10:   myConnection = new SqlConnection("server=localhost;
11:   database=Northwind; uid=sa; pwd=");
12:   myCommand = new SqlCommand("GetCustomersByCountry", myConnection);
13:   myCommand.CommandType = CommandType.StoredProcedure;
14:   myParameter = myCommand.Parameters.Add(new SqlParameter("@country",
15:     SqlDbType.VarChar, 50));
16:   myParameter.Direction = ParameterDirection.Input;
17:   myParameter.Value = "USA";
18:   myConnection.Open();
19:   SqlDataReader myDataReader = myCommand.ExecuteReader();
20:   myDataGrid.DataSource = myDataReader;
21:   myDataGrid.DataBind();
22:   myConnection.Close();
23: }
24: </script>

[VB & C#]
25: <html>
26: <head>
27: <title>Chapter 3: Managed Providers in ADO.NET</title>
28: </head>
29: <body>
30: <form runat="server" method="post">
31: <asp:DataGrid runat="server" id="myDataGrid" />
32: </form>
33: </body>
34: </html>
In Listing 3.7, you create a SqlConnection, SqlCommand, and SqlDataReader, the same as in previous listings. On line 8, you declare a SqlParameter object that will be used to create and add parameters to the SqlCommand. On line 13, you instantiate the SqlParameter object and set its name to @country, which is the same name given to the input parameter in the stored procedure in Listing 3.6. Additionally, you set the data type to SqlDbType.VarChar, 50, as required by the database table. On line 14, you set the SqlParameter’s direction to ParameterDirection.Input, and you set the SqlParameter’s Value to “USA” on line 15. Lastly, you open the connection and execute the command.

```
NOTE
In Listing 3.5 you created a series of new DataRow objects, and assigned their values after all the DataRows were created. The same approach can be used for SqlParameters.
myCommand.Parameters["@country"].Value = "USA";
```

```
NOTE
Input is the default direction for a Managed Command’s parameter. There’s no need to set a parameter’s direction to Input explicitly. Listing 3.7 demonstrated how to set the parameter’s Direction property. The possible values are ParameterDirection.Input and ParameterDirection.Output.
```

The DataAdapter

In Chapter 2, you looked at the DataSet as a collection of DataTable objects. You used the DataTables in the DataSet to populate one or more server controls on an ASP.NET Web Form. The DataAdapter is the bridge between the DataSet and the data store.

Unlike the past model of connection-based data processing, the DataAdapter works on a disconnected message-based model, revolving around and delivering chunks of information in a disconnected fashion. The DataAdapter is made up of four command methods, a TableMappings collection, a Command collection, and an Exception collection (for OleDbErrors). Like the other objects in the Managed Providers, the DataAdapter comes in two stock flavors, the SqlDataAdapter and the OleDbDataAdapter. Figure 3.4 illustrates the DataAdapter Object Model.
FIGURE 3.4
The DataAdapter Object Model.

The primary function of a DataAdapter is to retrieve data from a data store and push it into a DataTable in the DataSet. To complete this task, the DataAdapter requires two pieces of information, or parameters:

- A Managed Connection
- A Select Command

The DataAdapter constructor can accept either the command and connection values as text, or a Managed Command object as a single parameter. Listing 3.7 demonstrated constructing a DataAdapter with text values, while Listing 3.8 demonstrates constructing the DataAdapter with a single Managed Command.

LISTING 3.8 Creating a SqlDataAdapter with Connection and Command Text Values

[VB]
01: <%@ Page Language="VB" %>
02: <%@ Import Namespace="System.Data" %>
03: <%@ Import Namespace="System.Data.SqlClient" %>
04: <script runat="server">
05:  Sub Page_Load(Sender As Object, E As EventArgs)
06:   Dim myDataAdapter As SqlDataAdapter
07:   Dim myDataSet As New DataSet
08:   myDataAdapter = New SqlDataAdapter("SELECT * FROM Customers", "server=localhost; database=Northwind; uid=sa; pwd=")
09:   myDataAdapter.Fill(myDataSet, "Customers")
In Listing 3.8 you create an instance of the SqlDataAdapter class. When instantiating the class, on line 8 you pass in the command and connection values as text. The SqlDataAdapter uses these values to create SqlCommand and SqlConnection objects behind the scenes. These objects are used to connect to the database and retrieve the appropriate data.

In Listing 3.9 you achieve the same result as in Listing 3.8, using explicit SqlCommand and SqlConnection objects.

LISTING 3.9 Creating an SqlDataAdapter with Connection and Command Objects

[VB]
LISTING 3.9  Continued

04: <script runat="server">
05: Sub Page_Load(Sender As Object, E As EventArgs)
06: Dim myConnection As SqlConnection
07: Dim myCommand As SqlCommand
08: Dim myDataAdapter As SqlDataAdapter
09: Dim myDataSet As New DataSet
10: myConnection = New SqlConnection("server=localhost;
11: database=Northwind; uid=sa; pwd=;")
12: myCommand = New SqlCommand("SELECT * FROM Customers", myConnection)
13: myDataAdapter = New SqlDataAdapter(myCommand)
14: myDataAdapter.Fill(myDataSet, "Customers")
15: myDataGrid.DataSource = myDataSet.Tables("Customers")
16: End Sub
17: </script>

[C#]
01: <%@ Page Language="C#" %>
02: <%@ Import Namespace="System.Data" %>
03: <%@ Import Namespace="System.Data.SqlClient" %>
04: <script runat="server">
05: void Page_Load(Object sender, EventArgs e){
06: SqlConnection myConnection;
07: SqlCommand myCommand;
08: SqlDataAdapter myDataAdapter;
09: DataSet myDataSet = new DataSet();
10: myConnection = new SqlConnection("server=localhost;
11: database=Northwind; uid=sa; pwd=;");
12: myCommand = new SqlCommand("SELECT * FROM Customers", myConnection);
13: myDataAdapter = new SqlDataAdapter(myCommand);
14: myDataAdapter.Fill(myDataSet, "Customers");
15: myDataGrid.DataSource = myDataSet.Tables("Customers");
16: }
17: </script>

[VB & C#]
18: <html>
19: <body>
20: <form runat="server" method="post">
21: <asp:DataGrid runat="server" id="myDataGrid" />
22: </form>
23: </body>
24: </html>
In Listing 3.9 you explicitly create SqlCommand and SqlConnection objects. The SqlConnection object is used when constructing the SqlCommand object, and the SqlCommand object is passed into the SqlDataAdapter when it is instantiated.

While this might seem logical, it is more efficient to create the DataAdapter using the text values. The DataAdapter will manage the creation and destruction of the connection and command objects it requires. The explicit creation of these objects is only useful if you will be using either or both of them again, separate of the DataAdapter.

**DataAdapter.Fill() Method**

In Listings 3.8 and 3.9, you used various techniques and languages to create an instance of the Managed Provider DataAdapter. In one fashion or another, you created the DataAdapter and passed it values for the SelectCommand and Connection properties (either as inline values or as objects). Once the DataAdapter was created, the Fill() method of the DataAdapter was called.

The DataAdapter.Fill() method is like the switch that makes it go. Up until the Fill() method is called, the DataAdapter is idle. When the Fill() method is called, the connection to the database is made, and the SQL statement is executed. The results from the execution are filled into a DataSet, specified as a parameter of the Fill() method.

Specifying only a DataSet to fill the result set will cause a new DataTable object to be created in the DataSet. The DataTable is then accessible by its index value.

```csharp
DataAdapter.Fill([DataSet])
DataGrid.DataSource = DataSet.Tables(0)
```

Optionally, you can also pass in a string value representing the name you would like assigned to the DataTable that is created. This makes your code easier to follow and more readable, as you can then access the DataTable by name rather than by its index value.

```csharp
DataAdapter.Fill([DataSet], "[Table Name]")
DataGrid.DataSource = DataSet.Tables("[Table Name]")
```

On line 13 of Listing 3.9 you invoke the Fill() method of the DataAdapter and fill the results of the executed SQL statement into an empty DataSet, creating a new DataTable named “Customers”.

When the Fill() method is invoked, the bridge to the data store is extended. Then the data is retrieved and brought back to the calling application in the form of an XML file. This XML file is materialized as a DataTable in the DataSet you specified. The DataTable schema (table/column/primary key definitions) will be created automatically based on the schema of the database. Figure 3.5 illustrates the process when you’re invoking the DataAdapter.Fill() method.
Reading and Displaying Data

PART II

Data Tier
Web Forms
DataAdapter.Fill()
XML

FIGURE 3.5
Invoking the Fill() method of the DataAdapter class causes a bridge to be extended to the data store and the results to be returned to the calling application in the form of an XML file. The XML file is materialized as a DataTable in a DataSet.

While the Fill() method will create the DataTable dynamically, it can also fill an existing DataTable that you create explicitly, as shown in Listing 3.10.

LISTING 3.10  Using the DataAdapter with an Existing DataTable

[VB]
01: <%@ Page Language="VB" %>
02: <%@ Import Namespace="System.Data" %>
03: <%@ Import Namespace="System.Data.SqlClient" %>
04: <script runat="server">
05:  Sub Page_Load(Sender As Object, E As EventArgs)
06:   Dim myDataAdapter As SqlDataAdapter
07:   Dim myDataSet As New DataSet
08: 
09:   myDataSet.Tables.Add(New DataTable("Customers"))
10:   myDataSet.Tables("Customers").Columns.Add("CompanyName",
11:     System.Type.GetType("System.String"))
12:   myDataSet.Tables("Customers").Columns.Add("ContactName",
13:     System.Type.GetType("System.String"))
14:   myDataSet.Tables("Customers").Columns.Add("Region",
15:     System.Type.GetType("System.String"))
16:   myDataAdapter = New SqlDataAdapter("SELECT CompanyName, ContactName, Region FROM Customers",
17:     "server=localhost; database=Northwind; uid=sa; pwd=;"")
18:   myDataAdapter.Fill(myDataSet, "Customers")
19:   myDataGrid.DataSource = myDataSet.Tables("Customers")
20:   myDataGrid.DataBind()
21:  End Sub
22: </script>
LISTING 3.10  Continued

[C#]

01: <%@ Page Language="C#" %>
02: <%@ Import Namespace="System.Data" %>
03: <%@ Import Namespace="System.Data.SqlClient" %>
04: <script runat="server">
05:  void Page_Load(Object sender, EventArgs e){
06:   SqlDataAdapter myDataAdapter;
07:   DataSet myDataSet = new DataSet();
08:   myDataSet.Tables.Add(new DataTable("Customers"));
09:   myDataSet.Tables["Customers"].Columns.Add("CompanyName",
10:     System.Type.GetType("System.String"));
11:   myDataSet.Tables["Customers"].Columns.Add("ContactName",
12:     System.Type.GetType("System.String"));
13:   myDataSet.Tables["Customers"].Columns.Add("Region",
14:     System.Type.GetType("System.String"));
15:   myDataAdapter = new SqlDataAdapter("SELECT
16:     CompanyName, ContactName, Region FROM Customers",
17:     "server=localhost; database=Northwind; uid=sa; pwd=");  
18:   myDataAdapter.Fill(myDataSet, "Customers");
19:   myDataGrid.DataSource = myDataSet.Tables["Customers"];
20:   myDataGrid.DataBind();
21:  }
22: </script>

[VB & C#]

23: <html>
24: <body>
25: <form runat="server" method="post">
26: <asp:DataGrid runat="server" id="myDataGrid" />
27: </form>
28: </body>
29: </html>

In Listing 3.10 you create a DataTable explicitly on lines 9–12. Using the Fill() method of the
DataAdapter you fill this newly created DataTable with the results from the SQL statement execu-
tion. This is done by calling the Fill() method and passing in the DataSet and DataTable
name for the DataTable you just created. If data already exists in the DataTable, then the Fill() method will update, or add rows to the DataTable. You can use the same DataAdapter, change its SelectCommand.CommandText property, and invoke the Fill() method again. In Listing 3.11 you use the DataAdapter to return two different result sets from similar SQL statements. You use the Fill() method to add the records to the same DataTable in the DataSet.
LISTING 3.11 Using the Fill() Method to Add Records to a DataTable

[VB]
01: <%@ Page Language="VB" %>
02: <%@ Import Namespace="System.Data" %>
03: <%@ Import Namespace="System.Data.SqlClient" %>
04: <script runat="server">
05: Sub Page_Load(Sender As Object, E As EventArgs)
06: Dim myDataAdapter As SqlDataAdapter
07: Dim myDataSet As New DataSet
09: myDataSet.Tables.Add(New DataTable("Customers"))
10: myDataSet.Tables("Customers").Columns.Add("CompanyName", System.Type.GetType("System.String"))
11: myDataSet.Tables("Customers").Columns.Add("ContactName", System.Type.GetType("System.String"))
12: myDataSet.Tables("Customers").Columns.Add("Region", System.Type.GetType("System.String"))
14: myDataAdapter = new SqlDataAdapter("GetCustomersByCountry", "server=localhost; database=Northwind; uid=sa; pwd=;")
15: myDataAdapter.SelectCommand.CommandType = CommandType.StoredProcedure
16: myDataAdapter.SelectCommand.Parameters.Add(new SqlParameter("@country", SqlDbType.VarChar, 50))
17: myDataAdapter.SelectCommand.Parameters("@country").Value = "Canada"
18: myDataAdapter.Fill(myDataSet, "Customers")
20: myDataAdapter.SelectCommand.Parameters("@country").Value = "Spain"
21: myDataAdapter.Fill(myDataSet, "Customers")
23: myDataGrid.DataSource = myDataSet.Tables("Customers")
24: myDataGrid.DataBind()
26: End Sub
27: </script>

[C#]
01: <%@ Page Language="C#" %>
02: <%@ Import Namespace="System.Data" %>
03: <%@ Import Namespace="System.Data.SqlClient" %>
04: <script runat="server">
05: void Page_Load(Object sender, EventArgs e){
06: SqlDataAdapter myDataAdapter;
07: DataSet myDataSet = new DataSet();
09: myDataSet.Tables.Add(new DataTable("Customers"));
LISTING 3.11 Continued

10:   myDataSet.Tables["Customers"].Columns.Add("CompanyName",

11:   System.Type.GetType("System.String"));
12:   myDataSet.Tables["Customers"].Columns.Add("ContactName",

13:   System.Type.GetType("System.String"));
14:   myDataSet.Tables["Customers"].Columns.Add("Region",

15:   System.Type.GetType("System.String"));
16:   myDataAdapter = new SqlDataAdapter("GetCustomersByCountry",

17:   "server=localhost; database=Northwind; uid=sa; pwd=");
18:   myDataAdapter.SelectCommand.CommandType = CommandType.StoredProcedure;
19:   myDataAdapter.SelectCommand.Parameters.Add(new

20:     SqlParameter("@country", SqlDbType.VarChar, 50));
21:   myDataAdapter.SelectCommand.Parameters["@country"].Value = "Canada";
22:   myDataAdapter.Fill(myDataSet, "Customers");
23:   myDataAdapter.SelectCommand.Parameters["@country"].Value = "Spain";
24:   myDataAdapter.Fill(myDataSet, "Customers");
25: }
26: </script>

[VB & C#]

27: <html>
28: <body>
29: <form runat="server" method="post">
30: <asp:DataGrid runat="server" id="myDataGrid" />
31: </form>
32: </body>
33: </html>

In Listing 3.11 you use the DataAdapter to select a small set of records from the database (line 14) with the stored procedure from Listing 3.6. On line 15 you specify that the SelectCommand property of the DataAdapter is a stored procedure. On line 16 you add a parameter for the expected “@country” parameter in the stored procedure. On line 17 you set the value of the parameter to “Canada.” Using the Fill() method you add those records to the Customers DataTable on line 18. On line 20 you change the parameter’s value to “Spain.” Since you are using the same DataAdapter to execute the second Fill() method, you do not need to set the ConnectionString property, or recreate the parameter you are using. Using the Fill() method, on line 18, you add the new result set to the existing records in the Customers table. Figure 3.6 shows the result of executing the code in Listing 3.11.
As you’ve learned, the DataAdapter works on a disconnected message-based model. You can reuse the DataAdapter to fill additional DataTables in the same DataSet, or in other DataSets, because there’s no physical link between the DataAdapter and the DataSet or DataTable. You only need to change the SelectCommand property of the DataSet (if you want to use a new SQL statement), or change a parameter value (if you want to use the same SQL statement), and call the Fill() method, passing it the new DataSet and DataTable name (see Listing 3.12).

**Figure 3.6**
The DataAdapter.Fill() method can be used to add records to a DataTable, or update existing records.

**Listing 3.12  Adding a Second DataTable to a DataSet**

[VB]

```vbnet
01: <%@ Page Language="VB" %>
02: <%@ Import Namespace="System.Data" %>
03: <%@ Import Namespace="System.Data.SqlClient" %>
04: <script runat="server">
05: Sub Page_Load(Sender As Object, E As EventArgs)
06: Dim myDataAdapter As SqlDataAdapter
07: Dim myDataSet As New DataSet
08: myDataAdapter = New SqlDataAdapter("GetCustomersByCountry",
09: "server=localhost; database=Northwind; uid=sa; pwd=")
```
**LISTING 3.12**  Continued

```csharp
10:     myDataAdapter.SelectCommand.CommandType = CommandType.StoredProcedure
11:     myDataAdapter.SelectCommand.Parameters.Add(new
12:      SqlCommand Parameter("@country", SqlDbType.VarChar, 50));
13:     myDataAdapter.SelectCommand.Parameters("@country").Value = "Canada"
14:     myDataAdapter.Fill(myDataSet, "Canada_Customers")
15:     myDataAdapter.SelectCommand.Parameters("@country").Value = "Spain"
16:     myDataAdapter.Fill(myDataSet, "Spain_Customers")
17:     myDataAdapter.SelectCommand.Parameters("@country").Value = "Canada"
18:     myDataAdapter.SelectCommand.Parameters("@country").Value = "Spain"
19:     myDataAdapter.Fill(myDataSet, "Canada_Customers")
20:     myDataAdapter.SelectCommand.Parameters("@country").Value = "Spain"
21:     myDataAdapter.Fill(myDataSet, "Spain_Customers")
22:     myDataGrid.DataSource = myDataSet.Tables("Canada_Customers")
23:     myDataGrid.DataBind()
24:     myDataGrid.DataBind()
25:     myOtherDataGrid.DataSource = myDataSet.Tables("Spain_Customers")
26:     myOtherDataGrid.DataBind()
27:     End Sub
28: </script>

[C#]

```csharp
01: < %@ Page Language="C#" %>
02: < %@ Import Namespace="System.Data" %>
03: < %@ Import Namespace="System.Data.SqlClient" %>
04: <script runat="server">
05:     void Page_Load(Object sender, EventArgs e){
06:         SqlDataAdapter myDataAdapter;
07:         DataSet myDataSet = new DataSet();
08:         myDataAdapter = new SqlDataAdapter("GetCustomersByCountry",
09:             SqlDbType.VarChar, 50));
10:         myDataAdapter.SelectCommand.CommandType = CommandType.StoredProcedure;
11:         myDataAdapter.SelectCommand.Parameters.Add(new
12:             SqlCommand Parameter("@country", SqlDbType.VarChar, 50));
13:         myDataAdapter.SelectCommand.Parameters("@country").Value = "Canada"
14:         myDataAdapter.SelectCommand.Parameters("@country").Value = "Spain"
15:         myDataAdapter.SelectCommand.Parameters("@country").Value = "Canada"
16:         myDataAdapter.SelectCommand.Parameters("@country").Value = "Spain"
17:         myDataAdapter.SelectCommand.Parameters("@country").Value = "Canada"
18:         myDataAdapter.SelectCommand.Parameters("@country").Value = "Spain"
19:         myDataAdapter.SelectCommand.Parameters("@country").Value = "Canada"
20:         myDataAdapter.SelectCommand.Parameters("@country").Value = "Spain"
21:         myDataAdapter.SelectCommand.Parameters("@country").Value = "Canada"
22:     myDataGrid.DataSource = myDataSet.Tables("Canada_Customers")
23:     myDataGrid.DataBind();
24:     myOtherDataGrid.DataSource = myDataSet.Tables("Spain_Customers")
25:     myOtherDataGrid.DataBind();
26: } }
27: </script>

```
LISTING 3.12  Continued

[VB & C#]

25: <html>
26: <form runat="server" method="post">
27: <body>
28: <asp:DataGrid runat="server" id="myDataGrid" />
29: <asp:DataGrid runat="server" id="myOtherDataGrid" />
30: </form>
31: </body>
32: </html>

In Listing 3.12 you use the same data access code from Listing 3.11. You use the Fill() method to create a new DataTable named Canada_Customers. Then, by resetting the parameter value you retrieve another set of data from the database. Using the Fill() method you create a second DataTable named Spain_Customers. Finally you bind each of these DataTables to a separate DataGrid. The resulting page is shown in Figure 3.7.

FIGURE 3.7
The DataAdapter can be used to create multiple DataTables in one or more DataSets. This is allowed because the DataAdapter is not explicitly tied to a single DataSet or DataTable.
Table and Column Mappings

Table and column mappings enable you to alter the schema of the DataTable that’s dynamically created in the DataSet. In the previous listings you filled the DataSet, taking in the schema provided to you by the data store. There are certainly instances when you’ll want to change this. Often, database column names can be a bit cryptic, and changing them can make writing your code easier. Table and column mappings allow you to create a master mapping between the data returned from the data store and the DataTable in the DataSet. The DataSet maintains the table and column mappings and can translate them back to their original names when reconciling the data with the data store.

**NOTE**

In the previous listings, you’ve been filling a DataSet with the Customers table from the Northwind database. In the following listings, you’ll create a new TableMapping for the Authors table in the Pubs database. Pubs is a sample database installed with Microsoft SQL Server. I chose to switch to it because of its notoriously cryptic naming conventions.

In the following example, you’ll create table and column mappings for the Authors table in the Pubs database. When you add a new TableMapping, you must pass in the TableMapping source name and the DataTable name. If you specify “Table” (the default) as the source name, when you fill the DataSet you do not need to pass in the DataTable name. The data retrieved will use the TableMappings for “Table” by default, and a DataTable with the name you specified when you created the TableMappings will be created. This is demonstrated in Listing 3.13.

**LISTING 3.13** Creating Table and Column Mappings for the Default Table

```vbnet
01: <%@ Page Language="VB" %>
02: <%@ Import Namespace="System.Data" %>
03: <%@ Import Namespace="System.Data.SqlClient" %>
04: <script runat="server">
05:  Sub Page_Load(Sender As Object, E As EventArgs)
06:   Dim myDataAdapter As SqlDataAdapter
07:   Dim myDataSet As New DataSet
08:   myDataAdapter = new SqlDataAdapter("SELECT * FROM Authors",
09:       "server=localhost; database=Pubs; uid=sa; pwd=")
10: 
```
LISTING 3.13  Continued

11:   myDataAdapter.TableMappings.Add("Table", "Authors")
12:   With myDataAdapter.TableMappings("Table").ColumnMappings
13:     .Add("au_id", "ID")
14:     .Add("au_lname", "Last Name")
15:     .Add("au_fname", "First Name")
16:     .Add("phone", "Phone")
17:     .Add("address", "Address")
18:     .Add("city", "City")
19:     .Add("state", "State")
20:     .Add("zip", "Zipcode")
21:     .Add("contract", "Contract")
22:   End With
23:   myDataAdapter.Fill(myDataSet)
24:   myDataGrid.DataSource = myDataSet.Tables("Authors")
25:   myDataGrid.DataBind()
26: End Sub
27:
28: </script>

[C#]

01: <%@ Page Language="C#" %>
02: <%@ Import Namespace="System.Data" %>
03: <%@ Import Namespace="System.Data.SqlClient" %>
04: <script runat="server">
05:  void Page_Load(Object sender, EventArgs e){
06:   SqlDataAdapter myDataAdapter;
07:   DataSet myDataSet = new DataSet();
08:   myDataAdapter = new SqlDataAdapter("SELECT * FROM Authors",
09:     "server=localhost; database=Pubs; uid=sa; pwd=;");
10:   myDataSet.Tables.Add("Table", "Authors");
11:   with myDataAdapter.TableMappings("Table").
12:     ColumnMappings.Add("au_id", "ID");
13:     ColumnMappings.Add("au_lname", "Last Name");
14:     ColumnMappings.Add("au_fname", "First Name");
15:     ColumnMappings.Add("phone", "Phone");
16:     ColumnMappings.Add("address", "Address");
17:     ColumnMappings.Add("city", "City");
### Listing 3.13  Continued

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>myDataAdapter.TableMappings[&quot;Table&quot;].ColumnMappings.Add(&quot;state&quot;, &quot;State&quot;);</td>
</tr>
<tr>
<td>20</td>
<td>myDataAdapter.TableMappings[&quot;Table&quot;].ColumnMappings.Add(&quot;zip&quot;, &quot;Zipcode&quot;);</td>
</tr>
<tr>
<td>21</td>
<td>myDataAdapter.TableMappings[&quot;Table&quot;].ColumnMappings.Add(&quot;contract&quot;, &quot;Contract&quot;);</td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>myDataAdapter.Fill(myDataSet);</td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>myDataGrid.DataSource = myDataSet.Tables[&quot;Authors&quot;];</td>
</tr>
<tr>
<td>27</td>
<td>myDataGrid.DataBind();</td>
</tr>
<tr>
<td>28</td>
<td>}</td>
</tr>
<tr>
<td>29</td>
<td>&lt;/script&gt;</td>
</tr>
<tr>
<td></td>
<td>[VB &amp; C#]</td>
</tr>
<tr>
<td>30</td>
<td>&lt;html&gt;</td>
</tr>
<tr>
<td>31</td>
<td>&lt;body&gt;</td>
</tr>
<tr>
<td>32</td>
<td>&lt;form runat=&quot;server&quot; method=&quot;post&quot;&gt;</td>
</tr>
<tr>
<td>33</td>
<td>&lt;asp:DataGrid runat=&quot;server&quot; id=&quot;myDataGrid&quot; /&gt;</td>
</tr>
<tr>
<td>34</td>
<td>&lt;asp:DataGrid runat=&quot;server&quot; id=&quot;myOtherDataGrid&quot; /&gt;</td>
</tr>
<tr>
<td>35</td>
<td>&lt;/form&gt;</td>
</tr>
<tr>
<td>36</td>
<td>&lt;/body&gt;</td>
</tr>
<tr>
<td>37</td>
<td>&lt;/html&gt;</td>
</tr>
</tbody>
</table>

On line 9, you create a new SqlDataAdapter, selecting all of the fields in the Authors table of the Pubs database. On line 11, you add a new default table mapping by passing in the name “Table” as the first parameter. Any table created in the DataSet that doesn’t have a name specified in the Fill() method will use this table mapping and will be given the name “Authors”, as indicated on line 11. On lines 13–21 you create the column mappings for all of the fields in the Authors table. The first parameter passed into the ColumnMappings collection’s Add() method is the name of the field in the database. The second parameter is the name you’re giving the field in the DataTable.

In case you’d rather not alter the DataSet’s default table and column mappings, the DataAdapter allows you to add named table and column mappings. Rather than specifying “Table” as the source name, you can provide a new source name. When you call the Fill() method, you pass in the new source name as the table parameter. The DataSet checks for a table mapping for the source name passed in. If no table mapping exists, the schema is built on the fly based on the data store’s schema, the same as in earlier examples. If there’s a table mapping for the name passed in, it’s used.

11:  myDataAdapter.TableMappings.Add("BookAuthors", "BookAuthors");
24:  myDataAdapter.Fill(myDataSet, "BookAuthors");
Line 11 shows a table mapping named “BookAuthors”, specifying the name “BookAuthors” to be used when the `DataTable` is created. (Lines 12–23 did not change from Listing 3.13.) On line 24, you call the `Fill()` method, passing in the name of your table mapping (“BookAuthors”). The `DataSet` finds this table mapping and creates a `DataTable` named “BookAuthors”, using the schema you created.

**Summary**

Chapter 1 looked at what ASP.NET is, and Chapter 2 looked at what ADO.NET is. This chapter pulled the two pieces together.

In this chapter you learned that the ADO.NET Managed Providers are the bridge from a data store to your data-driven application. The Managed Providers come in two stock flavors: the SQL Managed Provider and the OleDb Managed Provider. The SQL Managed Provider is used to connect directly to a Microsoft SQL Server database (version 7.0 or higher), and bypasses OLEDB to provide better performance. The OleDb Managed Provider is used to connect to non-Microsoft SQL Server databases, such as Access, Oracle, and a host of others.

In this chapter you created connections to a database, built command objects to execute SQL statements on the database, and used the `DataReader` to iterate through data before binding it to a server control. You also learned about the `DataAdapter` class; a specific class for bridging between the Web application and the database to return the result set as a `DataTable` in a `DataSet`.

Throughout this chapter you built sample Web forms that connected to the database and returned records as either a data stream (`DataReader`), or a `DataTable` (`DataAdapter`).

By now you should be comfortable with the two Managed Providers and should be ready to start building data-driven Web applications. For the rest of this book you’ll be working with both the Managed Providers. You’ll be pulling data into an application with the Managed Command or the `DataAdapter`, and you’ll use the `DataSet` to persist the data, or the `DataReader` to iterate through the data.

If it felt as though you covered a lot in this chapter, you did. You’ll be using it repetitively throughout this book, so don’t worry. . . you’ll get lots of practice.