CHAPTER 5

DataReader and DataSet

In Chapters 3 and 4, you saw how to build and pass queries to a database, both by using a `SqlDataSource` and by writing code to connect to the database and retrieve the required results using the `ExecuteReader()` and `ExecuteScalar()` methods. Although we've only scratched the surface of what is possible, we've actually looked at the basics of everything you need to do, and you should now be able to build data-access pages that display data to the user.

This isn't the end of the story by any means. You have a large number of options for displaying data, but they all boil down to whether you're going to work with the data directly from the database or store it on the Web server as *disconnected data*. You'll look at both options in this chapter: using the DataReader to work with the results of a query directly from the database or storing query results in a `DataSet` object on the Web server away from the database. You'll also see how to create a `DataSet` locally with your own data, independent of an external data source.

Along the way, you'll look in some detail at the DataReader and `DataSet` objects, their makeup, and their differences. At least one of these two objects will feature in every data-driven page you create, so it's good to be up to speed on how they work. Indeed you've actually looked at both the `DataSet` and DataReader objects. The examples in Chapter 4 returned data from the database as a DataReader, and the `SqlDataSource` you used in Chapter 3 actually uses a `DataSet` by default when retrieving data from its data source.

This chapter is only the first part of five in your journey through data handling. For now, we'll assume that the data you request doesn't need to be displayed on screen. In Chapters 6 and 7, we'll assume that the data will be displayed on screen but is read-only and won't need to be updated. In Chapter 8, you'll continue with building pages that allow you to create, modify, and delete data and reflect those changes back to the data source. In Chapter 9, we'll take an in-depth look at three of the new Web controls introduced in ASP.NET 2.0: `GridView`, `FormView`, and `DetailView`.

Why not just look at the DataReader and `DataSet` as you go along? Why put this interlude first? These are good questions, but they have a simple answer (borrowed from the world of Perl). For every data-related task you'll be looking at over the coming chapters, you can follow this motto: There's More Than One Way to Do It. But all of these ways stem from how the DataReader and `DataSet` work. If you don't look at these objects now and see the situations in which they're useful, you'll be less likely to choose the right option when building data-driven pages of your own.

This chapter covers the following topics:

- How to iterate through a DataReader
- Some useful properties and methods of the DataReader
- How the `DataSet` works with a DataAdapter
- How to iterate through a `DataSet`
- How to build a `DataSet` from scratch
- How to set the `SqlDataSource` to access a database as a DataReader or a `DataSet`
- Differences between the DataReader and `DataSet`
- Tips for coding DataReader and `DataSet` access

The DataReader Object

The key to the whole topic of data handling is the DataReader object, or if you prefer to be data provider-specific, the `SqlDataReader`, `OleDbDataReader`, and `OdbcDataReader` objects. True, they're optimized as appropriate for their associated technology, but their method calls and properties are, for all intents and purposes, identical.

The DataReader is a strange object. You may use it all the time, but it's intangible, representing only a pipeline in memory between the database and the page waiting for the data. In functional terms, it works much like a phone connection. While the phone connection is open, the page can communicate queries to the database, which in turn can communicate its results back to the page, but once the connection is closed, there's no trace of it or record of the data returned from the database, except in the page itself. Only if you use another object, such as the `DataSet`, can you maintain an in-memory record of the results from the query. If you like, the `DataSet` is the equivalent of an answering machine or phone-tapping mechanism.

The upshot of a DataReader being only a conduit in memory, rather than a permanent place of storage, is that when you access the data in a DataReader, the data is read-only. It also means you can access the results only one row at a time, and once you finish with a row and move on to the next one, you can't go back to it. You can go only forward. Of course, this means there are pros and cons to using only DataReader in your page. On the plus side, you have the following:

- Using a DataReader is quick and efficient, as it doesn't need to worry about keeping track of every bit of data.
- A DataReader doesn't need to store data in memory, so it uses fewer resources in creating a page.

The disadvantages are as follows:

- You can't do anything with the data, such as sending changes back to the database. This would mean referring to data already passed through the reader, which isn't possible; DataReaders work only from database to client. If you need to send changes back to the database, you'll need to make a separate query to the database, as you'll see in Chapter 8.
- DataReaders require exclusive access to a connection. Once a DataReader is open, nothing else can use a connection until the DataReader is closed.

A DataReader isn't picky about the amount of data passed through it. You could request a single item of information from a column or the entire contents of the database. As long as you understand how to access the DataReader, it won't complain.

As you'll recall, a DataReader is the resulting object from a call to `ExecuteReader()` on a Command object.

```
SqlDataReader myReader = myCommand.ExecuteReader();
```

The general practice at this point is to assign (or *bind*) the values in a DataReader to Web controls on the page, and indeed, that's what you've already done in the examples in Chapter 4. You've created a `GridView` on the page, bound the data to it, and let ASP.NET take care of the display:

```
GridView1.DataSource = myReader;
GridView1.DataBind();
```

So far, all you've seen is the data displayed as a table thanks to the `GridView`, but you can bind information to several more data-aware Web controls. For example, you can use a drop-down list, a set of radio buttons, or a calendar. We'll spend all of Chapters 7 and 8 on data binding, but there's another way to work with DataReader objects that you'll look at here, and that's to iterate through them row by row.

How to Read Through a DataReader

It may seem a waste of time to work through the results of a query row by row and work with each when you can just bind it to a Web control and let the Web control take care of it all, but consider that data isn't always for display. You may be using a database table to store user information and site preferences. Rather than displaying it on the screen, information from these tables may be assigned directly to Web controls' properties or stored in a business object for use across the whole site. For example, you may create a Preferences object to store theme information for the whole site, store values from the database in its properties, and save it as a session-level variable. Rather than accessing the database again, you just access the session variable. If any preferences are changed during the session, they're saved to the session variable, and when the session is over, the changes are sent back to the database. This minimizes both database access for this purpose and also the overhead of using many session variables at a time. You just use one with a lot of information, rather than several containing individual pieces of information.

To iterate through the contents of a DataReader, you use its `Read()` method. If you haven't worked with reader objects in general before, the idea is simple. A reader has a pointer that you use to keep track of where you are in the information coming through your reader. If you like, it's the same kind of thing that happens when you use your finger to keep your place on a book page. Until you open the book and start to read, you can't see anything. The same thing applies in code. You can't access anything until you call `Read()` the first time, and each time you call `Read()` after that, the DataReader lets another row through for you to use. `Read()` will also return a Boolean value each time you call it: true if there's another row for you to work with and false if you've reached the end of the query results, as shown in Figure 5-1.

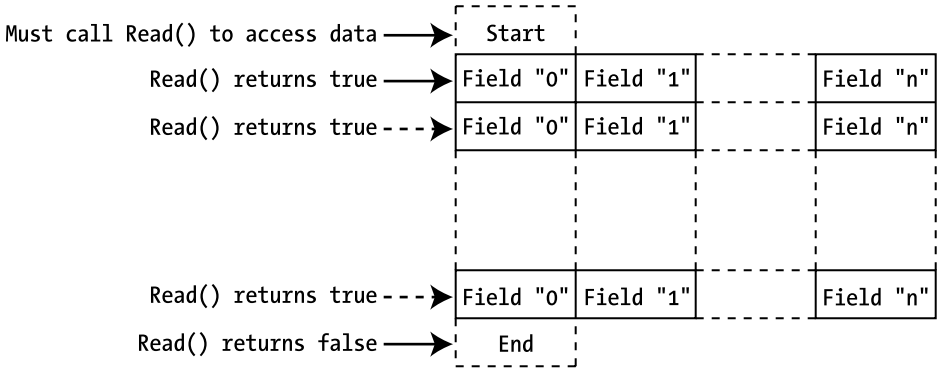

Figure 5-1. *Working through rows in a DataReader using the Read() method*

This means that you can use the call to `Read()` as the condition in a `while` loop. If your query returns no results, the first call to `Read()` ends the loop before you do anything. If not, the code will keep looping until there are no more results. In short, your page needs to have this skeleton code in it:

```
// create the connection
SqlConnection myConnection = new SqlConnection();

try
{
  // configure the connection

  // create the command

  // open the database connection

  // run the query
  SqlDataReader myReader = myCommand.ExecuteReader();

  // parse the results
  while (myReader.Read() == true)
  {
```

```
    // processing instructions for each row in DataReader
  }

  // close the reader
  myReader.Close();
}
finally
{
  // close the database connection
  myConnection.Close();
}
```

Take care not to call `Read()` in the `while` statement and then again within the loop—say, in a method call—or the code could skip some of the results. It's easy to do but hard to track down later in the code.

Besides the actual data processing, it's important that you close the DataReader once you've finished with it. Once a DataReader has been opened through a connection, nothing else can use that connection until the DataReader is closed. You can close the DataReader by either closing the `SqlConnection` (which has the effect of closing the DataReader if it is open) or by explicitly closing the DataReader, as in the previous code fragment:

```
myReader.Close();
```

If an error were to occur on the page before the call to the `Close()` method of the DataReader, the database connection is still isolated until the .NET garbage collector comes to dispose of the open DataReader. You also have a maximum number of database connections that can be open at any one time, so under heavy loads, not closing your connections could actually generate errors, which is definitely not a good thing.

Therefore, you enclose all of the code that interacts within the database, as you saw in Chapter 4, in a `try..catch..finally` block, so that you can always close the open database connection within the `finally` section:

```
finally
{
  // close the database connection
  myConnection.Close();
}
```

Try It Out: Iterating Through a DataReader

In this example, you'll see that you can do more than just fill a `GridView` with the results of a database query by passing the results into the grid and calling `DataBind()`. Here, you'll write a custom `Manufacturer` class, create an instance of it, and use a row of the Manufacturer table to populate it. In real life, you would probably then use it in the business rules tier of your Web site, but as this is a straightforward example, you'll define a simple method on the `Manufacturer` object that neatly prints the values of its properties to the page.

1. Start Visual Web Developer and create a new Web site in the `C:\BAND\Chapter05` folder. Delete the auto-created `Default.aspx` file.

2. Add a new `Web.config` file and add a new setting to the `<connectionStrings />` element:

```
<add name="SqlConnectionString"
  connectionString="Data Source=localhost\BAND;Initial Catalog=Players;
  User ID=band;Password=letmein" />
```

3. Add a new class called `Manufacturer.cs` by selecting Add New Item from the folder's context menu and selecting the Class option. When you click the Add button, you're presented with the dialog box shown in Figure 5-2. Click Yes to create the `App_Code` folder and add the `Manufacturer.cs` file to the new folder.

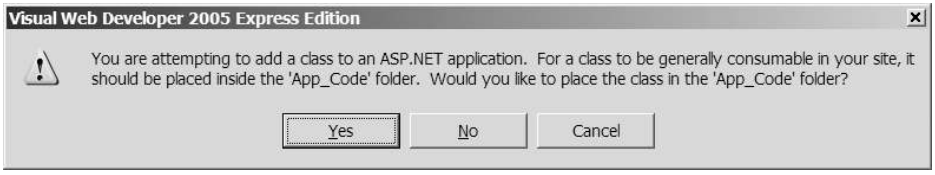

Figure 5-2. *Class files belong in the App_Code folder.*

4. Replace the code in `Manufacturer.cs` with the following:

```
using System.Text;

public class Manufacturer
{
  public string Name;
  public string Country;
  public string Email;
  public string Website;

  public Manufacturer()
  {
  }

  public override string ToString()
  {
    StringBuilder sbDescription = new StringBuilder();

    // add the name
    sbDescription.Append("Name: ");
    sbDescription.Append(this.Name);
    sbDescription.Append("<BR/>");
```

```
    // add the city
    sbDescription.Append("Country: ");
    sbDescription.Append(this.Country);
    sbDescription.Append("<BR/>");

    // add the email
    sbDescription.Append("Email: ");
    sbDescription.Append("<a href='mailto:");
    sbDescription.Append(this.Email);
    sbDescription.Append("'>");
    sbDescription.Append(this.Email);
    sbDescription.Append("</a>");
    sbDescription.Append("<BR/>");

    // add the website
    sbDescription.Append("Website: ");
    sbDescription.Append("<a href='");
    sbDescription.Append(this.Website);
    sbDescription.Append("'>");
    sbDescription.Append(this.Website);
    sbDescription.Append("</a>");
    sbDescription.Append("<BR/>");

    return (sbDescription.ToString());
  }
}
```

5. Add a new Web Form to the site called `DataReader_Iterating.aspx`. In the Source view, change the name of the page to **Iterating through a DataReader**.

6. In the Design view, add a `Label` to the blank page. You'll use this to demonstrate that your objects have been created. Set its `Text` property to an empty string.

7. In the Source view of the page, make sure the correct data provider is included at the top of the page, like so:

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data.SqlClient" %>
```

8. The second piece of code to add is for the `Page_Load` handler:

```
protected void Page_Load(object sender, EventArgs e)
{
  // create the connection
  SqlConnection myConnection = new SqlConnection();
```

```
    try
    {
      // configure the connection
      string strConnectionString = ConfigurationManager.
        ConnectionStrings["SqlConnectionString"].ConnectionString;
      myConnection.ConnectionString = strConnectionString;

      // create the command
      string strCommandText = "SELECT ManufacturerName, ➥
        ManufacturerCountry, ManufacturerEmail, ManufacturerWebsite ➥
        FROM Manufacturer ORDER BY ManufacturerName";
      SqlCommand myCommand = new SqlCommand(strCommandText, myConnection);

      // open the database connection
      myConnection.Open();

      // run the query
      SqlDataReader myReader = myCommand.ExecuteReader();

      // parse the results
      while (myReader.Read())
      {
        // create the manufacturer object
        Manufacturer objManufacturer = new Manufacturer ();
        objManufacturer.Name = Convert.ToString(myReader["ManufacturerName"]);
        objManufacturer.Country =
          Convert.ToString(myReader["ManufacturerCountry"]);
        objManufacturer.Email =
          Convert.ToString(myReader["ManufacturerEmail"]);
        objManufacturer.Website =
          Convert.ToString(myReader["ManufacturerWebsite"]);

        // output the manufacturer object details
        Label1.Text += objManufacturer.ToString() + "<BR/>";
      }

      // close the reader
      myReader.Close();
    }
    finally
    {
      // close the database connection
      myConnection.Close();
    }
  }
```

9. Save the page, and then run it. When the page loads, you'll see that the `Label` contains details of all the Manufacturers in the Manufacturer table written out, as in Figure 5-3, but not in tabular form. You have hyperlinks that work and an easier-to-read collection of data instead.

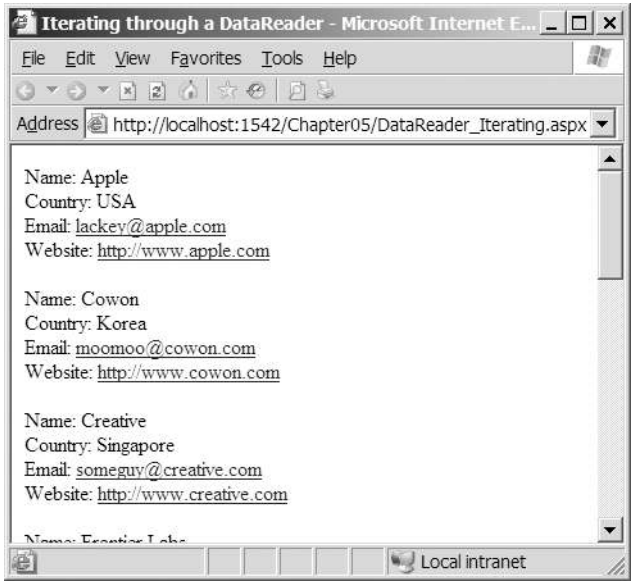

Figure 5-3. *Iterating through a DataReader*

How It Works

The aim of this page is to demonstrate that you can use a DataReader to provide values for any objects you create, so you start by defining a new object, `Manufacturer`, to use. This is created in the `App_Code` special folder, as it's not specific to a particular page. If you look at the `DataSet` code you'll use later in this chapter or the MySQL 5.0 and Microsoft Access versions of this page, you'll see that they use the same `Manufacturer` object.

The `Manufacturer` object has properties cunningly mirroring the information stored in the Manufacturer table.

```
public class Manufacturer
{
  public string Name;
  public string Country;
  public string Email;
  public string Website;

  public Manufacturer()
  {
  }
```

To demonstrate that you've achieved this aim, you can add a method that presents the information in a `Manufacturer` object neatly on the screen:

```
  public override string ToString()
  {
    StringBuilder sbDescription = new StringBuilder();

    // add the name
    sbDescription.Append("Name: ");
    sbDescription.Append(this.Name);
    sbDescription.Append("<BR/>");

    // country, email, website removed for brevity

    return (sbDescription.ToString());
  }
}
```

All objects within the .NET Framework inherit from `System.Object`, and they all have a `ToString()` method that returns a string representing the current object. You override the `ToString()` method so that the override version is called rather than the version on `System.Object`—after all, simply outputting the name of the class doesn't really show that you've achieved the aim of mirroring the database in the `Manufacturer` class.

All that's left is the `Page_Load` event, and apart from the following section, it's the same as the previous examples. Instead of plugging the results of `myCommand.ExecuteReader()` into a `GridView`, you can access the DataReader directly, like so:

```
// run query
SqlDataReader myReader = myCommand.ExecuteReader();
```

The following is the `while` loop where you work through each row in turn. The `Read()` method of a DataReader returns false if there isn't a row to retrieve and true if there is. So, if there is a row to be returned from `myReader`, the `while` loop will execute.

```
while (myReader.Read())
{
```

Inside the loop, you create a new `Manufacturer` object and give each of its properties values from the corresponding columns in the DataReader. The properties of the `Manufacturer` object are named the same as the columns in the database, so the process of constructing the `Manufacturer` object is relatively easy.

You can access columns in the current row from the DataReader in any order, and you don't need to access all the information in a particular row. Just don't forget that you can't come back to it later. You construct each `Manufacturer` object as follows:

```
  Manufacturer objManufacturer = new Manufacturer();
  objManufacturer.Name = Convert.ToString(myReader["ManufacturerName"]);
  objManufacturer.Country = Convert.ToString(myReader["ManufacturerCountry"]);
  objManufacturer.Email = Convert.ToString(myReader["ManufacturerEmail"]);
  objManufacturer.Website = Convert.ToString(myReader["ManufacturerWebsite"]);
```

You use the name of the column you're after and pass this as the required value to the `myReader` indexer. The indexer allows you to specify the name of the column you want to retrieve, and returns it as a generic `System.Object`. Therefore, you call `Convert.ToString()` to convert the object returned to a string to pass to the `Manufacturer` properties.

When you've finished creating the object, you display its details in the `Label` by calling the `ToString()` method of the `Manufacturer` object. If there's more information in the DataReader, the `while` loop will start creating another object. If not, the `while` loop finishes, and you close the DataReader by calling `Close()`, like so:

```
  Label1.Text += objManufacturer.ToString() + "<BR/>";
}

// close the datareader
myReader.Close();
```

■ **Note** Remember to close a DataReader object using `Close()` when you're finished with it. Until you do, you can't use your Connection object for any other queries or purpose. A DataReader has exclusive access to a connection until it's closed. This is true in all data providers. However, DataReaders rely on their connections to work, so make sure that the connection isn't closed before the DataReader is finished, or the results won't be pretty.

DataReader Properties and Methods

The DataReader also provides some handy support properties and methods to help you process its contents with fewer errors and more intelligence. Table 5-1 describes the DataReader properties, and Table 5-2 describes the DataReader methods.

Table 5-1. *DataReader Properties*

Name	Type	Description
`FieldCount`	`int`	Returns the number of columns in the current row
`HasRows`	`bool`	Returns true if the DataReader contains any rows
`IsClosed`	`bool`	Returns true if the DataReader is closed
`Item`	`Object`	Returns the contents of a column in a row*

** Never use `Item` by name. Instead, this is used in the background to access DataReader columns with, for instance, `myReader["columnname"]`.*

Table 5-2. *DataReader Methods*

Name	Type	Description
`Close()`	`void`	Closes the DataReader object
`Read()`	`bool`	Moves to the next row in the DataReader; returns true if a row exists, or false if at the end of the DataReader
`Get`*`XXX`*`(int)`	Varies*	Returns and casts the contents of a column at index `int` in the row**
`GetOrdinal(string)`	`int`	Returns the column index for the specified column name
`IsDBNull(int)`	`bool`	Returns true if the column at index `int` contains a null value pulled from the database, or false otherwise
`NextResult()`	`bool`	Moves to the next table in the DataReader; returns true if the next table exists, or false otherwise

** The `GetXXX()` methods return a type corresponding to the request. For example, the `GetString()` method returns a string, and `GetInt32()` returns an integer.*

*** You must use the appropriate method for the type of object you want to retrieve. There are 37 different `GetXXX()` methods for the `SqlDataReader` and 25 for the `OleDbDataReader` and `OdbcDataReader`, so check the .NET documentation.*

You've already seen the `Read()` method in action in the example, and we'll look at `NextResult()` in Chapter 12. The `NextResult()` method is used when you send a group of SQL queries to the database in one go, and the resulting DataReader contains more than one result set to scan.

You can use the remainder of these properties and methods to extend the previous example. Let's start with the `Get`*`XXX`*`(int)` methods. These methods allow the DataReader to be queried and the requested data returned in the correct format, so there are, for example, `GetString(int)`, `GetInt32(int)`, and `GetBoolean(int)` methods. In the previous example, you could construct the `Manufacturer` object as follows:

```
Manufacturer objManufacturer = new Manufacturer();
objManufacturer.Name = myReader.GetString(0);
objManufacturer.Country = myReader.GetString(1);
objManufacturer.Email = myReader.GetString(2);
objManufacturer.Website = myReader.GetString(3);
```

This does indeed do what you need it to do, and you don't have to do any casting, as the row you require is returned in the correct format. However, using this method has two drawbacks. One is that the code is a lot less readable. In the example, you can see that the `Email` property of the `Manufacturer` class is set to the `ManufacturerEmail` column from the database. Using a value of `2` for the `GetString()` method means you have to look at the SQL query you're executing to know which column you're actually returning.

Another problem with using the `Get`*`XXX`*`(int)` methods is that the code is directly tied to the specific way that the SQL query is constructed, as the ordering of the columns in the `SELECT` query is fixed. Suppose you were to change the SQL query to change the ordering of the columns returned, like so:

```
SELECT ManufacturerName, ManufacturerCountry,
  ManufacturerWebsite, ManufacturerEmail
FROM Manufacturer
ORDER BY ManufacturerName
```

Then the e-mail and Web site values for the Manufacturer would be incorrect. This is because column 2 is now the Web site instead of the e-mail address, and column 3 is the e-mail address instead of the Web site.

For these two reasons alone, it's worth taking a little extra time to use column names rather than the index, removing the possibility of the order of the `SELECT` query causing errors that may be extremely tricky to track down. You've already looked at one way of doing this in the previous example—casting the object from the DataReader to the correct type:

```
objManufacturer.Name = Convert.ToString(myReader["ManufacturerName"]);
```

You also have an alternative method that combines the `Get`*`XXX`*`()` methods with the `GetOrdinal()` method to return the correct type from the DataReader. The `GetOrdinal()` method returns the column index for a named column. Here's how you can combine these two methods:

```
Manufacturer objManufacturer = new Manufacturer();
objManufacturer.Name =
  myReader.GetString(myReader.GetOrdinal("ManufacturerName"));
objManufacturer.Country =
  myReader.GetString(myReader.GetOrdinal("ManufacturerCountry"));
objManufacturer.Email =
  myReader.GetString(myReader.GetOrdinal("ManufacturerEmail"));
objManufacturer.Website =
  myReader.GetString(myReader.GetOrdinal("ManufacturerWebsite"));
```

The `HasRows` property returns a Boolean value that's true if a DataReader does contain some information and false if it doesn't. Now, you can already detect this using `while(DataReader.Read())`, but `HasRows` allows you to be a bit neater and gives you an alternate check for a positive query if you aren't going to run straight through the `while` loop. You can add it to the earlier code, like so:

```
if (myReader.HasRows)
{
  while (myReader.Read())
  {
    ...
  }
}
else
{
  Label1.Text = "No rows returned.";
}
```

Once you know there's some information, you can make sure it's safe to retrieve the data by using a combination of `FieldCount` and `IsDBNull()`. Before retrieving data from the row, you can scan it for any columns containing null values, like so:

```
while (myReader.Read())
{
   for (int i=0; i<=(myReader.FieldCount-1); i++)
   {
      if (myReader.IsDBNull(i))
      {
         Label1.Text += "Warning: Column " + i + " is NULL.";
      }
   }

   // create the manufacturer object
}
```

Finally, you can verify that the DataReader is closed when you finish with it by checking its `IsClosed` property. As with the Connection object, telling a closed DataReader to close itself will not cause any problems. But if you need to, you can check before closing the DataReader, like so:

```
if (myReader.IsClosed == false)
{
  myReader.Close();
}
```

That about covers everything for DataReaders by themselves. You know how to iterate through them, and you'll learn how to bind data from them to Web controls in the next chapter. You've even looked at some useful properties such as `HasRows` and `IsClosed`.

One problem is that once you move past a row in a DataReader, you can't go back to it again, because a DataReader is forward-only. An option is to persist the data in a business object, as you've already seen. Another option for accessing the same data more than once is to use a `DataSet`.

The DataSet Object

DataReaders are quick and fast, but they're much like pay-per-view television. The only way to watch a film again once you've finished watching it is to go back to the channel and request it again. A `DataSet`, on the other hand, works like a video recorder; you can record the film off the television and watch it as many times as you like, rewinding and fast-forwarding through it as much as you like.

With a `DataSet`, you can store any data that you may have use for throughout the lifetime of a page. This idea of persisting data away from the database is known as *disconnected data*. In fact, it's even better than a video recorder, because once you have data inside a `DataSet`, you can alter that data, add to it, delete from it, and send all the changes back to the database relatively easily. This is handy (don't you wish you could do that with some movies?).

Of course, there's no reason why you can't use a DataSet just for displaying data in a page as well. In Chapter 6, you'll see how to use both a DataSet and a DataReader to supply read-only information to a page.

■ **Caution** A DataSet may not rely on a connection to a database, but it still lasts only for the lifetime of the page. If the page posts back and must be reassembled, so, too, must the DataSet. Either that, or it must be persisted somehow for retrieval by the next page. As a result, take care to query only for the data that will be needed on the page. A DataSet is resident in memory, so the smaller it is, the fewer resources required to keep it there, and the better the page performs and scales.

How to Fill a DataSet

The basic code to use a DataSet as a data source still follows the same three steps you saw back in Chapter 1, but in a slightly different way than creating a DataReader.

First, you set up the Connection and Command objects as usual. You also need to create the DataSet object, like so:

```
// create the connection
SqlConnection myConnection = new SqlConnection();

// create the DataSet object
DataSet myDataSet = new DataSet();

try
{
  // configure the connection
  string strConnectionString = ConfigurationManager.
    ConnectionStrings["SqlConnectionString"].ConnectionString;
  myConnection.ConnectionString = strConnectionString;

  // create the command
  string strCommandText = "SELECT ManufacturerName, ➥
    ManufacturerCountry, ManufacturerEmail, ManufacturerWebsite ➥
    FROM Manufacturer ORDER BY ManufacturerName";
  SqlCommand myCommand = new SqlCommand(strCommandText, myConnection);
```

Now it's time for something new. You use a DataAdapter as the intermediary between the database and the DataSet itself, so you need to set this up before you can populate the DataSet itself, like so:

```
  // create a DataAdapter
  SqlDataAdapter myAdapter = new SqlDataAdapter();
  myAdapter.SelectCommand = myCommand;
```

Next, you open the connection and use the DataAdapter's `Fill()` method to transfer the query results from the database to the `DataSet`, like so:

```
  // open the database connection
  myConnection.Open();

  // populate the DataSet
  myAdapter.Fill(myDataSet);
}
finally
{
  // close the database connection
  myConnection.Close();
}
```

At this point, the `DataSet` is ready for work. You can iterate through it as you did with the DataReader earlier, or simply bind the information it contains to a `GridView`:

```
// bind the data
GridView1.DataSource = myDataSet;
GridView1.DataBind();
```

■ **Note** You can find the code for this page in the `Chapter05` directory of the code download for this book (available from the Downloads section of the Apress Web site, `http:www.apress.com`). It's called `DataSet_Simple.aspx`.

This code is the simplest `DataSet` example possible, so you'll now add some more detail. There are two new data-aware objects in the code, and you need to learn more about them.

The DataAdapter Object

The eagle-eyed among you may have spotted what looks like an error in the previous code. It appears that it left out the prefix for the `DataSet` object that identifies which data provider it is a part of:

```
DataSet myDataSet = new DataSet();
```

However, this isn't an error. The `DataSet` (and the family of objects it contains) are independent of any data provider. You can find their definitions in the `System.Data` namespace. In the grand scheme of things, this makes a lot of sense. Data providers are there to provide optimized access to a data source and nothing more. The `DataSet` just stores data in memory and so should be optimized as best for .NET, rather than for the database that it personally never contacts.

The key, as you may have guessed, is the DataAdapter object—or the `SqlDataAdapter`, `OleDbDataAdapter`, and `OdbcDataAdapter` objects, if you prefer. These are the objects that translate the data from the format associated with that particular data provider to the generic .NET

format that the DataSet uses. These *are* data-provider-specific. However, their basic mechanisms are the same across the board. Their Fill() method causes data to be pulled from the database into a DataSet, and their Update() method pushes any changes made to the DataSet back to the database, as shown in Figure 5-4.

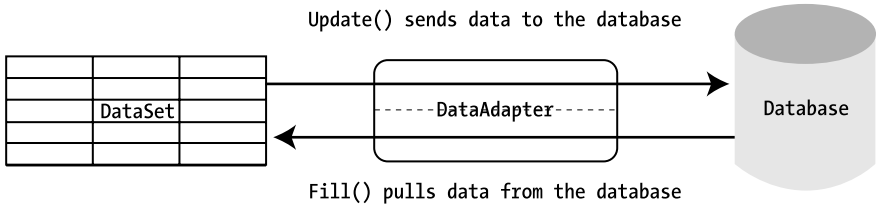

Figure 5-4. *A DataAdapter object plays the middleman between a DataSet and the database.*

Fill() and Update() are, unfortunately, not psychic, so you need to provide a DataAdapter with details of the Connection object it should use to access the database and the various SQL queries it should run when using Fill() and Update(). In the DataSet_Simple.aspx example, this takes place in two easy lines of code. First, create a SqlDataAdapter object; second, assign the SqlCommand object you've already built (which holds a SELECT query to the Manufacturer table) to its SelectCommand property. The Command object is already associated with a Connection object, so the DataAdapter is also by proxy.

```
SqlDataAdapter myAdapter = new SqlDataAdapter();
myAdapter.SelectCommand = myCommand;
```

Using the alternate constructor for the SqlDataAdapter, you could write this in a single line, like so:

```
SqlDataAdapter myAdapter = new SqlDataAdapter(myCommand);
```

Indeed, two other versions of the SqlDataAdapter constructor (and of the OleDbDataAdapter and OdbcDataAdapter, too) lead to providing the same information. The first takes two arguments, like so:

```
public SqlDataAdapter(queryString, SqlConnection);
```

In this case, the string is the SQL SELECT query written out in full, and the Connection object is as you would expect. In the final variant, the SqlConnection object is replaced by another string parameter containing the connection string written out in full, like so:

```
public SqlDataAdapter(queryString, connectionString);
```

The SQL query in these constructors is always the SELECT query that will be sent to the data source when Fill() is called. You can find it in the DataAdapter's SelectCommand property. You'll also need to provide its UpdateCommand, InsertCommand, and DeleteCommand properties with the respective queries for updating, inserting, and deleting data in the database before you can call Update() on the DataAdapter. You'll work with these three properties and Update() in Chapter 8.

■ **Note** Each of these four *xxx*Command properties of a DataAdapter object contains a Command object, rather than just a string containing the relevant SQL query.

Both Fill() and Update() can open a database connection if it's closed when they're called and will close it again once they're finished. If a connection is already open, it will remain open. If you want to close the connection, you must call Close() on the Connection object as you have in the code. In the Fill() method's case, you can use one of its many overloaded variations, which allows you to specify that it must close the connection after it has finished.

```
myAdapter.Fill(DataTable, myCommand, CommandBehavior.CloseConnection);
```

This particular variation of Fill() brings up another question. What's the first DataTable parameter? Well, it turns out there's a lot more to a DataSet than meets the eye.

DataSet Components

The DataSet is much more than a simple receptacle for query results. A DataSet is, more technically, a container for one or more DataTable objects that contain the data you retrieve from the database.

- A DataSet contains a DataTableCollection of DataTable objects. Each DataTable is referenced as myDataSet.Tables["TableName"] or myDataSet.Tables[index].
- Each DataTable contains a DataColumnCollection of DataColumn objects to represent the different pieces of information stored in the table. Each column can be referenced as myDataSet.Tables["TableName"].Columns["ColumnName"]. Properties such as AllowDBNull, Unique, and ReadOnly mimic those available in SQL Server 2005, MySQL 5.0, and Microsoft Access.
- Each DataTable also contains a DataRowCollection of DataRow objects to represent individual rows stored in the DataTable. Each row can be referenced as myDataSet.Tables["TableName"].Rows[RowNumber].
- Individual columns in a DataRow object can be referenced as myDataSet.Tables["TableName"].Rows[RowNumber]["ColumnName"].
- A DataSet also contains a DataRelationCollection of DataRelation objects that models the relationships between tables. Each DataRelation object contains the parent and child columns that are related. By default, a UniqueConstraint object is applied to the parent column, and a ForeignKeyConstraint object is applied to the child column. Thus, it mimics the way in which databases handle relationships. DataRelation objects can be referenced as myDataSet.Relations["RelationName"].

So, where was the DataTable in the previous example, DataSet_Simple.aspx? Looking at the code, there was no mention of a DataTable anywhere when you called the following:

```
myAdapter.Fill(myDataSet);
```

True, but by default, the `Fill()` method will create a `DataTable` called `Table` if one isn't specified and add the data to this. Also, when you set the `DataSource` property of the `GridView` to just the `DataSet`, by default, this means it will be bound to the first table in the `Table` collection. This can lead to a lot of problems with binding to the wrong `DataTable`, so it's better to not leave the default values.

You can name the `DataTable` to be filled and bound with the following lines of code:

```
myAdapter.Fill(myDataSet, "Manufacturer");
GridView1.DataSource = myDataSet.Tables["Manufacturer"];
```

The `Tables` property can also be accessed using an integer specifying the position in `Tables` like so:

```
GridView1.DataSource = myDataSet.Tables[0];
```

Unless you're iterating through the collection (as you'll soon see), you should always use the table name version of the indexer. Then changes to what is contained within the `Tables` collection by other parts of the code (such as adding an extra table before the table you're after) won't cause problems.

■ **Note** For what seems a simple method, `Fill()` has many variations and rules. The online documentation at `http://msdn2.microsoft.com/system.data.common.dataadapter.fill.aspx` is complete and should be the first place to look for more information about it.

Now, let's see how the components of the `DataSet` fit together by re-creating the first example and iterating through a `DataTable` to create custom objects.

Try It Out: Iterating Through a DataSet

In this example, you'll take what you've learned about the `DataSet`, `DataTable`, and the other objects in the group and replicate the previous example of iterating through a DataReader. Follow these steps:

1. In Visual Web Developer, create a new Web Form in the `Chapter05` Web site called `DataSet_Iterating.aspx`. In Source view, change the name of the page to **Iterating through a DataSet**.

2. Add a `Label` to the view of the page and set its `Text` property to an empty string.

3. In the Source view of the page, make sure the correct data provider is included at the top of the page, like so:

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data " %>
<%@ Import Namespace="System.Data.SqlClient" %>
```

4. Add a `Page_Load` event handler to the page. First, add the code to populate the `DataSet` with the contents of the Manufacturer table.

```
protected void Page_Load(object sender, EventArgs e)
{
  // create the connection
  SqlConnection myConnection = new SqlConnection();

  // create the DataSet object
  DataSet myDataSet = new DataSet();

  // configure the connection
  string strConnectionString = ConfigurationManager.
    ConnectionStrings["SqlConnectionString"].ConnectionString;
  myConnection.ConnectionString = strConnectionString;

  // create the command
  string strCommandText = "SELECT ManufacturerName, ➥
    ManufacturerCountry, ManufacturerEmail, ManufacturerWebsite ➥
    FROM Manufacturer ORDER BY ManufacturerName";
  SqlCommand myCommand = new SqlCommand(strCommandText, myConnection);

  // create a DataAdapter
  SqlDataAdapter myAdapter = new SqlDataAdapter();
  myAdapter.SelectCommand = myCommand;

  // populate the DataSet
  myAdapter.Fill(myDataSet, "Manufacturer");
```

5. Add the following code that iterates through the `DataTable` containing the data and populates the `Manufacturer` objects:

```
  // now iterate through the rows in the table
  for (int i = 0; i <= myDataSet.Tables["Manufacturer"].Rows.Count - 1; i++)
  {
    Manufacturer objManufacturer = new Manufacturer();
    objManufacturer.Name = Convert.ToString(
      myDataSet.Tables["Manufacturer"].Rows[i]["ManufacturerName"]);
    objManufacturer.Country = Convert.ToString(
      myDataSet.Tables["Manufacturer"].Rows[i]["ManufacturerCountry"]);
    objManufacturer.Email = Convert.ToString(
      myDataSet.Tables["Manufacturer"].Rows[i]["ManufacturerEmail"]);
    objManufacturer.Website = Convert.ToString(
      myDataSet.Tables["Manufacturer"].Rows[i]["ManufacturerWebsite"]);
```

```
        Label1.Text += objManufacturer.ToString() + "<BR/>";
      }
    }
```

6. Save this code, and then run it. When the page loads, you'll see the same results as the previous example, as shown earlier in Figure 5-3.

How It Works

You see a few more lines of code in this example than in the DataReader example. We reviewed most of the code in the "How to Fill a DataSet" section, where you discovered how to pull information from a database into a `DataSet`. Indeed, the only thing that has changed in the first half of `Page_Load` is to name the `DataTable` in which the results of the query will be saved.

```
myAdapter.Fill(myDataSet, "Manufacturer");
```

You may be wondering where the error-handling code that you've come to expect has gone. If you look back at the code in the "How to Fill a DataSet" section, you'll see that the reason for the error handling was to always close the database connection. Rather than manually opening and closing the database connection, you've taken advantage of the ability of the DataAdapter to open and close the connection automatically. You've removed the explicit calls to the `Open()` and `Close()` methods of the `SqlConnection` object, so you don't need to worry about the error handling in order to close the database connection.

Beyond that, the only new code in the example is for pulling individual columns into the respective properties of a `Manufacturer` object. For example, you use the following code to retrieve the name of the Manufacturer:

```
objManufacturer.Name = Convert.ToString(
  myDataSet.Tables["Manufacturer"].Rows[i]["ManufacturerName"]);
```

You start with the `DataSet` you created called `myDataSet`. You saved the contents of the Manufacturer table from the sample database into a `DataTable` called `Manufacturer`, which you can reference as `myDataSet.Tables["Manufacturer"]`. A `DataTable` contains a `DataRowCollection`, which you can query using its `Count` property to see how many rows you need to iterate through in code.

```
for (int i=0; i<=myDataSet.Tables["Manufacturer"].Rows.Count-1; i++)
```

You can access the rows inside the collection using their index number rather than their name, so you can reference each row as `myDataSet.Tables["Manufacturer"].Rows[i]`. You can then reference each column in a row either by name, as in the example, or by index. If you aren't sure how many columns are in a row (users of wildcards take heed!), you can use the `Count` property of the row's `DataColumnCollection` and use another `for` loop to iterate through them again, like so:

```
for (int i=0; i<=myDataSet.Tables["Manufacturer"].Rows[i].Columns.Count-1; i++)
```

One awkward thing about using the `DataSet` and `DataTable` is the syntax, which can get quite long. However, you can make it easier to read by accessing the `DataTable`, `DataRow`, and `DataColumn` objects directly rather than through the `DataSet` collections every time:

```
// get the manufacturer table
DataTable ManufacturerTable = myDataSet.Tables["Manufacturer"];

// now iterate through the rows in the table
for (int i = 0; i <= ManufacturerTable.Rows.Count - 1; i++)
{
  DataRow rowManufacturer = ManufacturerTable.Rows[i];

  Manufacturer m = new Manufacturer();
  objManufacturer.Name = Convert.ToString(
    rowManufacturer["ManufacturerName"]);
  objManufacturer.Country = Convert.ToString(
    rowManufacturer["ManufacturerCountry"]);
  objManufacturer.Email = Convert.ToString(
    rowManufacturer["ManufacturerEmail"]);
  objManufacturer.Website = Convert.ToString(
    rowManufacturer["ManufacturerWebsite"]);

  Label1.Text += objManufacturer.ToString() + "<BR/>";
}
```

This is a bit more manageable. You extract the table you're after from the `DataSet` as `ManufacturerTable`, and then extract the row you're after from the `ManufacturerTable`.

In the next section, you'll go one step further and build everything manually, even to the point of adding the data manually. This is a little extreme, but it demonstrates that the life of a `DataSet` isn't wholly dependent on a call to `DataAdapter.Fill()`.

Creating a DataSet from Scratch

In this section, you'll walk through building a `DataSet` that mirrors the sample database in terms of tables, strongly typed tables, and relationships. The point is to give you a feeling for the child objects and collections that a `DataTable` contains. Although you'll repeat the same tasks a few times, you'll try to look at several different ways of achieving them. Also, this will help you to understand a bit more about relationships between tables.

■ **Note** You can find the complete example in the `Chapter05` directory of the code download for this book. It's called `DataSet_Building.aspx`.

The actual page generated is nothing fancy. It contains four `GridView` controls, one for each table in the sample database. They're there purely to demonstrate that the `DataSet` does indeed mimic the database.

```
<body>
  <form id="form1" runat="server">
  <div>
    <asp:GridView ID="grdManufacturer" runat="server">
    </asp:GridView>
    <asp:GridView ID="grdPlayer" runat="server">
    </asp:GridView>
    <asp:GridView ID="grdFormat" runat="server">
    </asp:GridView>
    <asp:GridView ID="grdWPWF" runat="server">
    </asp:GridView>
  </div>
  </form>
</body>
```

As usual, the action takes place in the `Page_Load` event handler. However, you'll see a fair amount of code, so rather than have it all in one place, it's split into several methods. Inside `Page_Load` itself, it's pretty straightforward. You start by creating the `SqlConnection` object, as follows:

```
void Page_Load(object sender, EventArgs e)
{
  // create the connection
  string strConnectionString = ConfigurationManager.
    ConnectionStrings["SqlConnectionString"].ConnectionString;
  SqlConnection myConnection = new SqlConnection(strConnectionString);
```

Then you create a new `DataSet` object and build it to match the sample database, like so:

```
  // create a new DataSet
  DataSet myDataSet = new DataSet();

  // create the data
  GenerateDataSet(myDataSet, myConnection);
```

Finally, you bind each table in the `DataSet` to its own `GridView` and call `DataBind()`, like so:

```
  // bind each to table to a grid
  grdManufacturer.DataSource = myDataSet.Tables["Manufacturer"];
  grdPlayer.DataSource = myDataSet.Tables["Player"];
  grdFormat.DataSource = myDataSet.Tables["Format"];
  grdWPWF.DataSource = myDataSet.Tables["WhatPlaysWhatFormat"];

  // data bind the page
  Page.DataBind();
}
```

The key is the `GenerateDataSet()` method, but again, you're just marshaling your forces in this method. All you do here is call the methods that do the real work, like so:

```
void GenerateDataSet(DataSet dset, SqlConnection conn)
{
   // add four tables
   AddPlayerTable(dset);
   AddManufacturerTable(dset);
   AddFormatTable(dset);
   AddWhatPlaysWhatFormatTable(dset);

   // add the relationships
   AddRelationships(dset);

   // fill the tables
   FillManufacturerTable(dset, conn);
   FillPlayerTable(dset, conn);
   FillFormatTable(dset, conn);
   FillWhatPlaysWhatFormatTable(dset, conn);
}
```

■ **Note** Strictly speaking, you should never need to model an entire database in a `DataSet` for the purposes of data binding, especially considering the resources it consumes. However, for demonstration purposes, you can live with it.

Adding DataTables to a DataSet

Adding a `DataTable` object to a `DataSet` object in code may seem new, but the methods you need to call and the properties you need to set mirror almost exactly the actions you took back in Chapter 2 when you built the sample database against an actual database server. Those actions are as follows:

- Create and name the table.
- Create and name the columns within the table.
- Set the column's data type.
- Set any other properties the column should have.
- Establish the table's primary key.

Variations exist in how you do this; indeed, you don't actually need to perform all of these tasks to have a valid and working `DataTable`. The only mandatory steps are the first two: creating the table and creating the columns. The remaining three steps are optional, but they do give you more control over the type of table that you're creating.

Now, let's look at the AddPlayerTable() method. You start by creating a new DataTable object that you'll name Player. You don't have to give a name to the DataTable constructor right away; you can set it later in the TableName property, but there's less code to work through this way.

```
void AddPlayerTable(DataSet dset)
{
  // create the table
  DataTable PlayerTable = new DataTable("Player");
```

Every DataTable has a Columns collection object containing a DataColumnCollection, so to add a new DataColumn, you simply call the collection's Add() method. This will add a DataColumn object that you've already defined to the table or create a new one, add it to the collection, and return it as its result. As demonstrated, you can either set the new DataColumn to a variable for later reference or ignore the return value and just refer to the new DataColumn through the Columns collection, like so:

```
  // create the columns
  DataColumn PlayerID =
    PlayerTable.Columns.Add("PlayerID", typeof(Int32));
  PlayerTable.Columns.Add("PlayerName", typeof(String));
  PlayerTable.Columns.Add("PlayerManufacturerID", typeof(Int32));
  PlayerTable.Columns.Add("PlayerCost", typeof(Decimal));
  PlayerTable.Columns.Add("PlayerStorage", typeof(String));
```

Notice that the Add() method specifies a .NET data type as the second parameter. This allows you to constrain what is stored within the column. However, it isn't necessary to always specify the type of the column. The default data type of a column in a DataTable is String, so if you don't specify a data type, the column will contain strings. So the following declaration of the PlayerStorage column is functionally identical to the one that you're actually using:

```
  PlayerTable.Columns.Add("PlayerStorage");
```

■ **Note** If you're specifying a data type for a DataColumn, you must specify a .NET base type; thus, varchar(255) has no meaning here and would create an error. For a list of supported data types, refer to http://msdn2.microsoft.com/system.data.datacolumn.datatype.aspx.

With the columns in the table established, you can attend to their behavior. Should their contents be unique in each column, can they be null, and so on? Each DataColumn object has a set of properties that match those you saw in Chapter 2. When a column is created, the most common properties have the following default values: AllowDBNull is true, Unique is false, and ReadOnly is false. Also, for String types, MaxLength equals –1 by default, which implies there's no maximum length for the column. For a database column containing an autonumber, you must also set the AutoIncrement property to true, along with the AutoIncrementSeed property for a start value, such as 1. This latter property doesn't have a default value, but the AutoIncrementStep property does: 1.

You need to make the following adjustments:

```
  // set the properties
  PlayerTable.Columns["PlayerName"].MaxLength = 50;
  PlayerTable.Columns["PlayerName"].AllowDBNull = false;
  PlayerTable.Columns["PlayerManufacturerID"].AllowDBNull = false;
  PlayerTable.Columns["PlayerCost"].AllowDBNull = false;
  PlayerTable.Columns["PlayerStorage"].MaxLength = 50;
  PlayerTable.Columns["PlayerStorage"].AllowDBNull = false;
```

Finally, you need to set PlayerID to be the table's primary key. This will automatically set its `AllowDBNull` property to false, and its `Unique` property will be true. Note that the `PrimaryKey` property actually requires an array of `DataColumn` objects in case the table's primary key is a composite one and contains more than one column. You'll see this at work when you build the WhatPlaysWhatFormat table.

```
  // set the primary key
  PlayerTable.PrimaryKey = new DataColumn[] { PlayerID };
  PlayerTable.Columns["PlayerID"].AutoIncrement = true;
  PlayerTable.Columns["PlayerID"].AutoIncrementSeed = 1;
```

Last, but not least, you add the whole `DataTable` to the `Tables` collection of the `DataSet`, like so:

```
  // add the table
  dset.Tables.Add(PlayerTable);
}
```

So, as long as you stick to the same methodical way of adding columns to a table in a database, adding a `DataColumn` to a `DataTable` will remain a straightforward process in code. You'll now look at a couple of variations in the other `Add`*xxx*`Table()` methods.

`AddManufacturerTable()` neatly shows that you don't need to add a `DataTable` to a `DataSet` once it has been fully defined. Like the `Add()` method for a `DataTable`'s `Columns` collection, the `Add()` method for a `DataSet`'s `Tables` collection allows you to create and add a blank `DataTable` with a given name, as well as add an already established one. Thus, you can make the following call and use the `DataTable` returned by `Add()` to define the table:

```
void AddManufacturerTable(DataSet dset)
{
  // create and add the table
  DataTable ManufacturerTable = dset.Tables.Add("Manufacturer");

   ...
}
```

`AddWhatPlaysWhatFormatTable()` also demonstrates how you deal with composite primary keys. You simply add all the `DataColumn` objects in the primary key to the `DataTable`'s `PrimaryKey` array, like so:

```
void AddWhatPlaysWhatFormatTable(DataSet dset)
{
```

```
  // create the table
  DataTable WhatPlaysWhatFormatTable = new DataTable("WhatPlaysWhatFormatTable");

  // add the columns
  WhatPlaysWhatFormatTable.Columns.Add("WPWFPlayerID", typeof(Int32));
  WhatPlaysWhatFormatTable.Columns.Add("WPWFFormatID", typeof(Int32));

  // set the primary key
  WhatPlaysWhatFormatTable.PrimaryKey = new DataColumn[] {
    WhatPlaysWhatFormatTable.Columns["WPWFPlayerID"],
    WhatPlaysWhatFormatTable.Columns["WPWFFormatID"] };

  // add the table
  dset.Tables.Add(WhatPlaysWhatFormatTable);
}
```

Note that if more than one `DataColumn` is added to the `PrimaryKey` array, only their `AllowDBNull` properties will be changed from their default to true. Their `Unique` property remains false, in contrast to the situation where the primary key is only one column when `Unique` is set to true.

Setting Up Relationships in a DataSet

In Chapter 2, you learned how a relationship between two columns was first established and then clarified by a constraint. A unique constraint would ensure that the parent column contained unique values in each column, and a foreign key constraint would cover what happened to all the entries in a child table when the corresponding entry in the parent table was altered somehow. The same is true of relationships between the `DataTable` objects in a `DataSet`, as you'll see in this section.

If you recall, the sample database has three relationships between tables, as shown in Figure 5-5. Each is backed by a foreign key constraint that says that a change in the parent table cannot be made if there are corresponding entries in the child table.

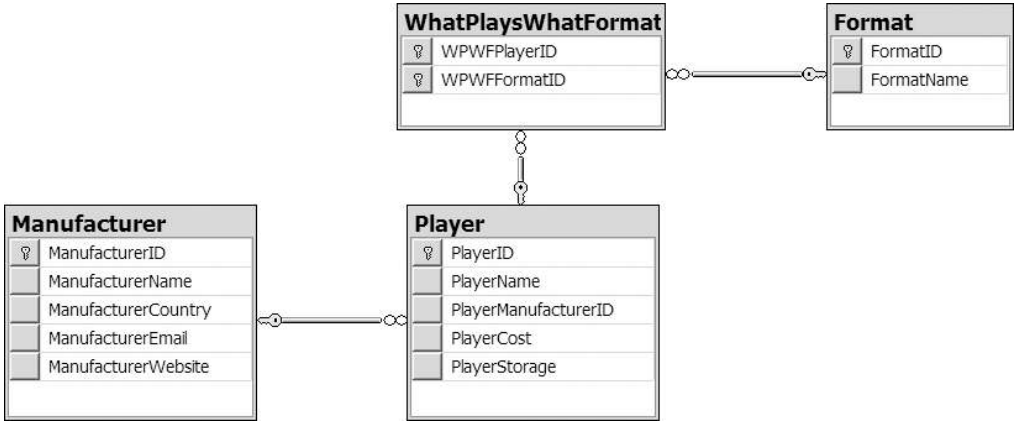

Figure 5-5. *Three relationships to create*

All three relationships are built in the same way in the AddRelationships() method, so you'll look at just one: the relationship between the Player and Manufacturer tables. The relationship between the tables is modeled using a DataRelation object and any constraints on the relationship by accessing either the ParentKeyConstraint or ChildKeyConstraint properties of the DataRelation.

To establish a relationship, you need to create a new DataRelation object, specifying its name (ManufacturerToPlayer) and the parent and child columns that form the relationship:

```
// create the Manufacturer to Player relationship
DataRelation ManufacturerToPlayerRelation = new DataRelation(
  "ManufacturerToPlayer",
  dset.Tables["Manufacturer"].Columns["ManufacturerID"],
  dset.Tables["Player"].Columns["PlayerManufacturerID"]);
```

Once the DataRelation object has been created, it can be added to the Relations collection using the Add() method:

```
dset.Relations.Add(ManufacturerToPlayerRelation);
```

Note that unlike all the Add() functions you've seen in this section, this Add() method won't create a blank DataRelation for you to use. You *must* provide the related columns for the DataRelation to be created.

With the relation established, you can set constraints on the parent or child column by assigning one to the ParentKeyConstraint or ChildKeyConstraint property, respectively. In this case, you need to establish a ForeignKeyConstraint on the child column. You need to block DELETE and UPDATE queries that would cause an orphan row in the Player table, like so:

```
ForeignKeyConstraint ManufacturerToPlayerConstraint =
  ManufacturerToPlayerRelation.ChildKeyConstraint;
ManufacturerToPlayerConstraint.DeleteRule = Rule.None;
ManufacturerToPlayerConstraint.UpdateRule = Rule.None;
```

Setting the DeleteRule and UpdateRule properties to Rule.None stops any deletions or updates to the Manufacturer table that would cause orphan rows in the Player table.

And that's it. The rest of AddRelationships() essentially repeats this code to put the other two relationships in place.

Creating DataRows

With each DataTable and the relationships between them established, all that's left is to add some data to the tables. You've already seen how to use Fill() to fill tables, but that's not the only way to add information to a DataTable. You can also create and populate DataRow objects, adding them individually to the corresponding DataTable. The FillManufacturerTable() method demonstrates this.

■ **Note** You may be wondering why you create the Player table first but populate the Manufacturer table first. You need to do this as there is a relationship between the two tables, and you can't have a Player without a Manufacturer (you'll get an `InvalidConstraintException`), so you must add Manufacturers before you can add Players. The order that you add data to the database is important, but the order in which you add unrelated (at least when they're created) tables to the database is irrelevant.

The first option to work with a `DataRow` (and the recommended one) is to create an empty `DataRow` object by calling `NewRow()` on the `DataTable` object for which you want to create the row.

```
void FillManufacturerTable(DataSet dset, SqlConnection conn)
{
  DataRow NewRow = dset.Tables["Manufacturer"].NewRow();
```

The advantage with this method is that the `DataRow` object will know what each column is called and the type of value it should contain, having ascertained it from the `DataTable` object to which the row is being added. It will therefore generate an exception if you try to add values that go against the rules on the table. When you have added values as appropriate, you use `Add()` to add it to the `Rows` collection for that `DataTable`, like so:

```
  // create a row on the table
  NewRow["ManufacturerID"] = 1;
  NewRow["ManufacturerName"] = "Apple";
  NewRow["ManufacturerCity"] = "USA";
  NewRow["ManufacturerEmail"] = "lackey@apple.com";
  NewRow["ManufacturerWebsite"] = "http://www.apple.com";
  dset.Tables["Manufacturer"].Rows.Add(NewRow);
```

The second option to add data to a `DataTable` is to create an array of generic objects that matches the columns in the table, rather than creating a `DataRow` object. The disadvantage here is that you can create an illegal value for a column that will be picked up only when you try to `Add()` it to the `Rows` collection.

```
  // create a row from an array
  Object[] NewRowColumns = new Object[5];
  NewRowColumns[0] = 2;
  NewRowColumns[1] = "Cowon";
  NewRowColumns[2] = "Korea";
  NewRowColumns[3] = "moomoo@cowon.com";
  NewRowColumns[4] = "http://www.cowon.com";
  dset.Tables["Manufacturer "].Rows.Add(NewRowColumns);
```

Several of the examples in the .NET Software Development Kit (SDK) build up rows of data using loops to generate values, which is handy for examples, but in general, you'll probably end up using a DataAdapter to fill a `DataTable` once you've created it. You should know the following about using `Fill()` in this situation:

- If you call `Fill()` on a `DataTable` with no columns, as you did in the earlier "Iterating Through a DataSet" section, the `DataTable` will be filled with the data from the table, and none of the extra details that you added in the last example—none of the `DataTable` schema definition—will be created. The `Fill()` method will assign each column in the `DataTable` a name and data type as best it can from the columns in the query results it's storing. However, properties such as `AllowDBNull` and `ReadOnly` will remain at their defaults, and the `PrimaryKey` for the table won't be set.

- In contrast, if you call `Fill()` on a `DataTable` whose details you've defined, as you have in this example, the DataAdapter will try to match `DataColumn` names with column names in the query results and fill in the values accordingly. If it can't match a column name with a `DataColumn`, it will create a new `DataColumn` with the same name as the column and use that instead. Make sure that the column and `DataColumn` names match up, or use aliases in your SQL query.

That said, one version of `Fill()` you didn't try earlier allows you to specify a subsection of the results from a query to add to a `DataTable`. This fits in nicely with the problem you now have with the `DataTable` copy of the Manufacturer table in `FillManufacturerTable()`. Using code to create the first two rows manually means that the versions of `Fill()` you've used so far would try to duplicate those two rows if you called them now. Moreover, this would cause an error because values in the primary key column would be duplicated, which isn't allowed.

You first need to create the Command and DataAdapter objects that return the information:

```
  // create the Command and DataAdapter
  SqlDataAdapter ManufacturerAdapter = new SqlDataAdapter();
  SqlCommand ManufacturerCommand = new SqlCommand(
    "SELECT * FROM Manufacturer ORDER BY ManufacturerID", conn);
  ManufacturerAdapter.SelectCommand = ManufacturerCommand;
```

This new version of `Fill()` allows you to say which row in the results you start filling from and how many rows you want to add to the `DataTable`. The first parameter identifies the `DataSet` you're working with, the second is the index number of the row in the results of the `SelectCommand` to start filling with, and the third is the number of rows (with 0 meaning all), to add to the `DataTable`, which is identified by the fourth parameter.

```
  // fill the DataTable
  ManufacturerAdapter.Fill(dset, 2, 0, "Manufacturer");
}
```

Note that the `SelectCommand` still retrieves all the rows from the Manufacturer table, even though you don't use the first two. If you wanted to retrieve only the seven rows required, you would need to alter the `SELECT` query rather than use this variant of `Fill()`.

SqlDataSource—DataSet or DataReader?

As you've seen in this chapter, you have two ways to access data in the database:

- As a DataReader, allowing forward-only access to the data
- As a `DataSet`, allowing full control over the disconnected data

This still doesn't explain how the `SqlDataSource` accesses the database. Does it do it using a DataReader or a `DataSet`? Well, actually, it can do both!

By default, the `SqlDataSource` connects to the database and stores the data internally as a disconnected `DataSet`. However, you can tell the `SqlDataSource` to access the database using a DataReader by setting its `DataSourceMode` property, which has two possible values:

- `DataReader`: Retrieves data from the database using a DataReader. The type of DataReader (`SqlDataReader`, `OleDbDataReader`, or `OdbcDataReader`) is determined from the connection used for the `SqlDataSource`.
- `DataSet`: Retrieves the data from the database into a `DataSet`. This is the default value.

So why would you want to change the default behavior and use a DataReader to connect to the database? The main reason is speed. As you've learned, the DataReader is the fastest way of talking to a database; a `DataSet` adds overhead.

If you're simply using a `SqlDataSource` to get a set of results from the database to display to the user, you should set the `DataSourceMode` property to `SqlDataSourceMode.DataReader`. You've already looked at this use of the `SqlDataSource` in Chapter 3, when you populated the list box containing the list of Manufacturers in the database. In that case, you just show this data to the user, so you should have used a DataReader.

For every other use of the `SqlDataSource`, you should leave the `DataSourceMode` as the default value of `SqlDataSourceMode.DataSet`. Although so far, you've looked at only displaying data in a `GridView` from the `SqlDataSource`, it can do a lot more. As you'll see in Chapter 9, a `SqlDataSource` in association with a `GridView` can allow paging, sorting, and filtering of the data that you've retrieved. A `SqlDataSource` can also allow the user to modify the data and have the changes propagated back to the underlying database. In these cases, you need to access the data using a `DataSet`.

DataSet vs. DataReader

Now that you have a rough idea of how a `DataSet` works, it's time to take a look at how it compares with a DataReader. The two have some obvious differences. Table 5-3 lists the differences you've seen so far and a few related ones.

You'll also see a comparison of the `DataSet` and DataReader objects at the end of Chapter 7, with respect to the theory and techniques you learn there. The intention is that by the end of Chapter 7, you'll be able to make a sound judgment as to which object should be used as the source of data for any ASP.NET pages you're writing.

Table 5-3. *Characteristics of DataReaders and DataSets*

DataReader	DataSet
A DataReader is specific to a data provider (for example, `SqlDataReader`, `OdbcDataReader`, and `OleDbDataReader`).	The `DataSet` class isn't a part of any data provider. It's specific to .NET only. However, the DataAdapter used to fill the `DataSet` with `Fill()` is specific to a data provider (for example, `SqlDataAdapter`, `OdbcDataAdapter`, and `OleDbDataAdapter`).
The data retrieved through a DataReader is read-only.	The data retrieved through a `DataSet` is read-write.
The data retrieved through a DataReader is forward-only. Once the data has been cycled through, the DataReader must be closed and re-created in order to reaccess the data.	You can work with data in a `DataSet` in any order you choose as many times as you like.
A DataReader presents data through a direct connection to the data source. Only one row of data is stored in memory at any one time.	A `DataSet` stores all the data from the data source in memory at once.
A DataReader takes up few IIS and memory resources but annexes the database connection until it's closed.	A `DataSet` takes up a lot more IIS and memory resources to store all the data, but it doesn't hold up a database connection until it's closed. The connection needs to be open only when `Fill()` is called.
A DataReader lasts as long as the connection to the database is open. It can't be persisted in a cookie or a session variable.	A `DataSet` lasts only until the page is reloaded (posted back), unless it's somehow persisted (for example, in a session variable).
Columns in a DataReader are referenced by index or name.	You can reference columns in a `DataSet` by name, but you must also name the `DataTable` and identify the row (index) that contains the column.
A DataReader has no concept of primary keys, constraints, views, or any other relational database management system concepts, except rows and columns.	A `DataSet` contains a collection of `DataTable` objects. A primary key may be set for each `DataTable`, and relationships and constraints may be established between them.
You can't update a data source through a DataReader.	You can make changes to data in a `DataSet`, and then send those changes back to the data source.
A DataReader connects to only one data source.	A `DataSet` can be filled with `Fill()` from multiple data sources but, once the data is retrieved, is not connected to any of them.

Good Practices

The next chapter looms, but before you start putting data on the screen, let's quickly recap some useful coding tips covered in this chapter.

- Query only for the information you want to use. For example, don't query for three columns per row if you're using only two. Likewise, use a `WHERE` clause in a `SELECT` query to retrieve only the rows of information that are required, rather than every row in the database.
- If you're using a DataReader, make sure you close it with `Close()` as soon as you can. Similarly, make sure you use `Close()` for your Connection as well.
- Use the DataReader's `HasRows` and `IsDBNull` properties to avoid any unwanted error messages when working with data.
- If you're using a `DataSet`, be aware of how calling `Fill()` will work with the `DataSet` you're using. Will it create new columns in a `DataTable` or use the other ones there? Make sure the columns you're querying for in the database match those in the `DataTable` you're targeting.
- Don't forget that primary keys and relationships in a `DataSet` won't be copied over from a database. You must create them in code.

Summary

In this chapter, you looked in detail at the DataReader and `DataSet` objects. These are the two objects most commonly used as the receptacle for query results by data-driven pages.

You learned that the DataReader is a read-only, forward-only, data-provider-specific window on the results of the query sent by a page, and that you can iterate through those results a row at a time using the DataReader's `Read()` method. Individual pieces of information can be identified in the current row in a DataReader by name and by index and can be vetted before being used with the DataReader's `HasRows` and `IsDBNull` properties.

In contrast, you saw that the `DataSet` is data-provider-independent. It's a container for a group of objects that can describe with some accuracy the table structure and relationships in a database, and because it's all in memory, the query results stored in a `DataSet` are read-write and can be accessed in any order. You can either build a complete data structure in code from scratch or `Fill()` it using a DataAdapter object.

In the next chapter, you'll begin your exploration of data binding, beginning with inline and list binding.