CHAPTER 6

Inline and List Binding

In Chapter 5, you looked at two of the most important objects in ADO.NET: the DataReader and the `DataSet`. You saw how the DataReader is a transient object and provides forward-only, read-only access to the results of any query you send to the data source. In contrast, the `DataSet` is a read-write, random-access representation of any data source that stays in memory even when the connection to the data source closes. You learned how they both can be populated with data from a data source, and how you can pull that data into something more useful for your Web site, such as a business object.

In this and the next chapter, you'll continue your work with these two objects, as well as the `SqlDataSource`, and discover the various ways of retrieving and displaying read-only data on a page. You'll also see that the trade-off between the speed of the DataReader and the availability of the `DataSet` often makes a difference in the way you build even simple pages.

In this chapter, you'll learn the following:

- The three types of data-binding: inline binding, list binding, and table binding
- The differences between binding a DataReader or `DataSet` in code and binding using the `SqlDataSource`
- How to perform inline binding to both a DataReader and `DataSet`
- An alternative to inline binding that returns the same results
- How to perform list binding to a DataReader, `DataSet`, and `SqlDataSource`
- How to perform listing binding with Web list controls that allow multiple selections

Data-Binding Techniques

In Chapter 1, I described how a data-aware page is basically a static template into which data is added dynamically from a data source. This "plugging in" of data to a page is more commonly known as *data binding*. You saw data binding in action in Chapter 3 with the `SqlDataSource`, and then in Chapter 4, when you actually wrote code to return the results that you wanted from the database.

How you implement data binding depends on how you're accessing the database (are you using a `SqlDataSource` or writing code to access the database?), how much data you want to bind, and to which Web controls you want to bind the data. However, although there are many variations in data-binding techniques, they all fall into one of the following three categories:

- Binding single columns to the properties or value of a Web control. This is often known as *inline binding*.
- Binding a list of values (one column in a table) to a Web control. For convenience, I'll call this *list binding*.
- Binding a table of values to a Web control. For convenience, I'll call this *table binding*.

In this chapter, we'll explore inline binding and list binding. Table binding is covered in the next chapter. But before we look at the specifics for each technique, let's review the general process for data binding with code, how data binding works with the `SqlDataSource`, and the Web controls that can be data-bound.

Data Binding in Code

When accessing the database in code, the process for adding a data-bound Web control is as follows:

1. Add a data-aware Web control to the page.
2. Associate a source of data with the Web control using the `DataSource` property.
3. Call `DataBind()` on the Web control or on the `Page`.

All the examples in Chapter 4 followed these three steps with the barest minimum of code, binding the query results to a `GridView` object:

```
GridView1.DataSource = myCommand.ExecuteReader();
GridView1.DataBind();
```

Calling `DataBind()` seems straightforward, but there's one catch: on what object? Every Web control implements this interface because it must as a derivative of its parent class, `System.Web.UI.Control`. A call to `DataBind()` on a Web control will also call `DataBind()` on any Web controls contained within it. So, you could call it on a `Label`, and just that particular binding would occur. On the other hand, you could call `Page.DataBind()`, and the command would also filter down to every Web control on the page.

You need to also consider a second issue here. Should a page rebind to a data source each time a page posts back? Consider a page containing a lot of Web controls populated by binding `RadioButtonList` controls, `CheckBox` controls, and other elements to a data source with the eventual aim to update the user's answers back to the database. You don't need to bind the Web controls to the data source more than once, because its purpose is purely to set up the page,

not to record the answers given to it. It would be a huge waste of resources to rebind the data every time the page was posted back to the Web server, especially if the page were complex. It would also lose the values the user had entered onto the page if it were posted back, because the rebinding would write over them. This is obviously not ideal. We'll look at how to manage when the data binding occurs in this chapter's examples.

Data Binding and the SqlDataSource

In Chapter 3, you used a `SqlDataSource` to populate both a `DropDownList` with the list of Manufacturers in the database and a `GridView` with a filtered list of Players. The one thing you didn't do was write any code to access the database. You used the following process to add a data-bound Web control using a `SqlDataSource`:

1. Add a `SqlDataSource` to the page.
2. Add a data-aware Web control to the page.
3. Associate the `SqlDataSource` with the data-aware Web control using the `DataSourceID` property.

You'll notice that you do not make an explicit call to `DataBind()`, as you do when you write code to access the database. So how is the data binding done? Automatic data binding occurs, with a call to `DataBind()` on the Web control, after the `OnPreRender` event and before the `OnPreRenderComplete` event in the page life cycle.

Although the `DataBind()` method is called automatically when using a `SqlDataSource`, that doesn't mean that you can't call `DataBind()` on the Web control to force the data binding to occur if necessary.

Data-Aware Web Controls

All three data-binding techniques—inline binding, list binding, and table binding—apply *only* to Web controls because the whole process takes place on the server before the page is sent to the client. Technically speaking, every Web control must understand how to bind data to its properties, because it inherits the `DataBind()` method as something it must implement from its parent `System.Web.UI.Control` class. At the least, this means that every Web control understands inline binding and can set its properties to values from a database. Some Web controls also know how to bind lists and tables of data into their structure.

Table 6-1 shows which groups of Web controls support which type of binding.

■ **Note** Table 6-1 doesn't list any Web controls derived from `System.Web.UI.HtmlControl`. You can use `HtmlControl`-derived Web controls for inline binding but, in all cases, there is a `WebControl` equivalent that you should use instead.

Table 6-1. *Web Controls and the Data Binding They Support*

Control Type	Control Names	Binding Supported
Text-based controls	`HyperLink`, `Label`, `Literal`, `Localize`, `Xml`	Inline
Form items	`Button`, `CheckBox`, `FileUpload`, `HiddenField`, `ImageButton`, `LinkButton`, `RadioButton`, `TextBox`	Inline
Form lists	`BulletedList`, `CheckBoxList`, `DropDownList`, `ListBox`, `RadioButtonList`	Inline, list
Images and spaces	`AdRotator`, `Image`, `ImageMap`, `Panel`, `PlaceHolder`	Inline
Tabular	`Calendar`, `Table`, `TableCell`, `TableFooterRow`, `TableHeaderCell`, `TableHeaderRow`, `TableRow`	Inline
Validation	`CompareValidator`, `CustomValidator`, `RangeValidator`, `RegularExpressionValidator`, `RequiredFieldValidator`, `ValidationSummary`	Inline
Data-aware controls	`DataGrid`, `DataList`, `DetailsView`, `FormView`, `GridView`, `Menu`, `Repeater`, `SiteMapPath`, `TreeView`	Inline, list, table
Master page controls	`Content`, `ContentPlaceHolder`	Inline
Profile controls	`ChangePassword`, `CreateUserWizard`, `Login`, `LoginName`, `LoginStatus`, `LoginView`, `PasswordRecovery`	Inline
Wizard controls	`CompleteWizardStep`, `CreateUserWizardStep`, `MultiView`, `TemplateWizardStep`, `View`, `Wizard`, `WizardStep`	Inline
Web part catalog controls	`DeclarativeCatalogPart`, `ImportCatalogPart`, `PageCatalogPart`	Inline
Web part editor controls	`AppearanceEditorPart`, `BehaviorEditorPart`, `LayoutEditorPart`, `PropertyGridEditorPart`	Inline
Web part part controls	`ErrorWebPart`, `UnauthorizedWebPart`	Inline
Web part zone controls	`CatalogZone`, `ConnectionsZone`, `EditorZone`	Inline

As you can see in Table 6-1, a lot of the Web controls allow only inline binding, which is the same for all Web controls. With more than 70 Web controls, it may seem that only a few support list and table binding. But these Web controls are pretty powerful, and you'll be surprised at what you can actually do with them.

Associating Data to the Web Control

The following sections contain three questions to ponder:

- How much data do you need to pull from your data source?
- Which object do you stream it into?
- How do you associate it to a Web control?

How Much Data Do You Need?

You've already learned that you can use the `SELECT` query to query for as much or as little data as is required for binding to the Web controls on your page. It makes sense to query only for what you need.

For example, inline binding requires you to identify individual columns to take values from, so why take a whole table's worth? Depending on the object you're sourcing the data from (see the next section), it may not matter if it contains several rows of data, because you can specify which row and column to use. As you know, however, the DataReader presents only a row at a time, so you may want to query only for a specific row of data with a query such as the following:

```
SELECT UserCategory, PreferredColorScheme, ConnectionSpeed
FROM UserPreference
WHERE UserName = 'Damien Foggon'
```

In a similar vein, if you're interested in list binding, you need to present the Web control with a set of rows in the order you want to display them. Each row needs to contain only *two* columns: one that represents the text for items in the list and one that provides the values for items that will be passed on when selected by a user. So, for example, when you displayed the list of Manufacturers in Chapters 3 and 4, you returned only the two columns you needed from the database: the ManufacturerName column to display to the user and the ManufacturerID to make a note of the selections:

```
SELECT ManufacturerID, ManufacturerName
FROM Manufacturer
ORDER BY ManufacturerName
```

In the case of table binding, you can retrieve more data, but try to restrict your query to just what you need. It's possible to hide columns in a `GridView`, but why bother if you don't need the column in the first place? Don't forget that there's no reason a `GridView` or similar Web control can't be bound to a query whose results contain only one or two columns.

Which Object Should You Use?

Although we're restricting our discussion to the DataReader, `DataSet`, and `SqlDataSource`, it's worth noting that you can use many other objects as the source of the data to which you're binding a Web control. For list and table binding, you can use any class that implements the `IEnumerable` interface. Some examples of classes that support the `IEnumerable` interface and can be used for list and table binding are as follows:

- `ArrayList` objects
- Collections (any class that implements the `ICollection` interface)
- `DataRow` objects
- `DataTable` objects
- `DataView` objects

If you're inline binding a single value to a Web control property, you can use practically any other single value from any other object available to you, as long as you know the syntax to get the value from the object.

How Do You Create the Association?

The step to create the association comes before the call `DataBind()` is made (whether in code or automatically in the case of the `SqlDataSource`), and as such, means that a Web control or property can't be bound to data based on the values bound into some other Web control—not unless you're using a postback in the page to react to choices made by the user.

Inline binding is quite different from list or table binding at this stage, because you must associate the property with the column that will fill it in the HTML markup of the page, rather than the code. For those of you who have worked with classic ASP, inline binding is reminiscent of the way you inserted ASP code into pages.

Let's say you wanted to bind the `Text` property of a `Label`. You would use the following in the page:

```
<asp:Label id="Label1" runat="server" text="<%# expression %>">
```

The expression in the text must identify the source for the value you want bound to the `Text` property and must be surrounded by `<%# ... %>` tags.

In contrast, list and table binding can be set up in both the code and the HTML markup of the page. For both, you need to set the `DataSource` or `DataSourceID` property for the Web control you're binding to the data source. If you're binding to a Web list control, you also need to set its `DataValueField` and `DataTextField` properties to the columns in your queried data.

■ **Note** Inline binding is one of those things that you'll either love or you'll hate. Personally, it's something that I never do. ASP.NET was supposed to free us from the problems of spaghetti code, but using inline binding makes the code look like a plate of pasta. As you'll see after the Inline Binding section, you can accomplish the same task without relying on data binding.

Inline Binding

Although inline binding data from a `DataSet` may be more common, the technique is no less valid against a DataReader. In the next two examples, you'll try both approaches.

■ **Note** Although a `SqlDataSource` performs essentially the same function as a DataReader or `DataSet`, you can't use the `SqlDataSource` for inline binding. The `SqlDataSource` can be used only in list and table binding.

Try It Out: Inline Binding to a DataReader

In this example, you'll mimic the results from the "Try it Out: Iterating through a DataReader" section in the previous chapter and print the details of a single Manufacturer in your sample database. Rather than use a single Web control to present the results, you'll use two `Label` controls and two `HyperLink` controls to echo the results.

1. In Visual Web Developer, create a new Web site at `C:\BAND\Chapter06` and delete the auto-generated `Default.aspx` file.

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

```
<add name="SqlConnectionString"
  connectionString="Data Source=localhost\BAND;Initial Catalog=BAND;
    Persist Security Info=True;User ID=band;Password=letmein"
  providerName="System.Data.SqlClient" />
```

3. Add a new Web Form called `Inline_DataReader.aspx` to the Web site.

4. In the Source view, find the `<title>` tag within the HTML at the bottom of the page and change the page title to **Inline Binding to a DataReader**.

5. In the Design view, add two `Label` controls to the page. Name the first `lblName` and the second `lblCountry`. Now add two `HyperLink` controls onto the page. Name them `lnkEmail` and `lnkWebsite`. Finally, add one more `Label` called `lblError` to house any error messages should something untoward happen (oh, the horror!). Now clear the `Text` properties for all five Web controls. With a bit of added text (**Country**, **Email**, and **Website**), your page should look something Figure 6-1.

Figure 6-1. *Basic layout for Inline_DataReader.aspx*

6. In the Source view, make sure you've included the correct data provider at the top of the page.

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

7. You need to set up a DataReader to query for Manufacturer details and add your standard code for database access.

```
// must declare the DataReader globally; else the page can't see it.
SqlDataReader myReader;

protected void Page_Load(object sender, EventArgs e)
{
  // set up connection string and SQL query
  string ConnectionString = ConfigurationManager.
    ConnectionStrings["SqlConnectionString"].ConnectionString;
  string CommandText = "SELECT ManufacturerName, ManufacturerCountry, ➥
    ManufacturerEmail, ManufacturerWebsite FROM Manufacturer ➥
    WHERE ManufacturerID = 1";

  // create SqlConnection and SqlCommand objects
  SqlConnection myConnection = new SqlConnection(ConnectionString);
  SqlCommand myCommand = new SqlCommand(CommandText, myConnection);

  // use try finally clauses when the connection is open.
  try
  {
    // open the database connection
    myConnection.Open();

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

    if (myReader.Read())
    {
      // Process results here.
    }
    else
    {
      // show the error
      lblError.Text = "No results to databind to.";
    }

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

8. Now you need to set which data should be bound and to what. Scroll to the bottom of the Source view and find the <body> tag in the HTML. Modify the HTML so that it's as follows (the changed parts are shown in bold):

```
<body>
  <form id="form1" runat="server">
  <div>
    <asp:Label ID="lblName" runat="server">
      Name: <%# DataBinder.Eval (myReader, "[ManufacturerName]") %>
    </asp:Label>
    <br />
    Country:
    <asp:Label ID="lblCountry" runat="server"
      Text='<%# DataBinder.Eval (myReader, "[ManufacturerCountry]") %>'>
    </asp:Label>
    <br />
    Contact:
    <asp:HyperLink ID="lnkEmail" runat="server"
      NavigateUrl='mailto:<%# DataBinder.Eval (myReader, "[2]") %>'
      Text='<%# DataBinder.Eval (myReader, "[ManufacturerEmail]") %>'>
    </asp:HyperLink>
    <br />
    Website:
    <asp:HyperLink ID="lnkWebsite" runat="server"
      NavigateUrl='<%# DataBinder.Eval (myReader, "[3]") %>'>
      <%# DataBinder.Eval (myReader, "[ManufacturerWebsite]") %>
    </asp:HyperLink>
    <br /><br />
    <asp:Label ID="lblError" runat="server"></asp:Label><br />
  </div>
  </form>
</body>
```

9. You've added the data-aware Web controls to the page and associated them with the required data retrieved from the database. All that's left to do is call DataBind(). Scroll back to the <script> block at the start of the page and modify the Page_Load event as follows:

```
if (myReader.Read())
{
  // bind the data
  Page.DataBind();
}
else
{
  // show the error
  lblError.Text="No results to databind to.";
}
```

10. Save the code, and then view the page in a browser. When the page loads, all appears to be well, but is it? Move your cursor over the e-mail link, as shown in Figure 6-2, and you'll see that the link isn't `mailto: lackey@apple.com`. You'll have to change it.

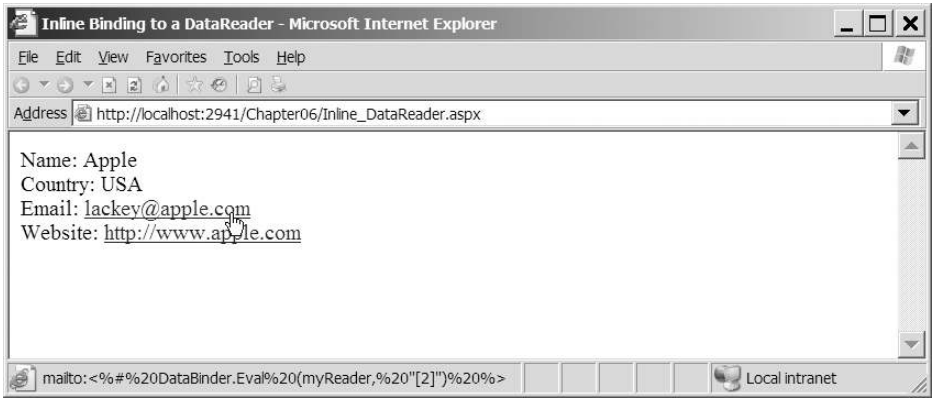

Figure 6-2. *All is not well with this link.*

11. To fix the problem, you need to use a different version of the `DataBinder.Eval()` method. An overloaded version takes a format string as its third parameter, so you can alter the `NavigateUrl` property of `lnkEmail` to be as follows:

```
NavigateUrl='<%# DataBinder.Eval (myReader, "[2]", "mailto:{0}") %>'
```

12. You've solved the problem. Save the file, and then test the code again. You'll see that the link now works, as shown in Figure 6-3.

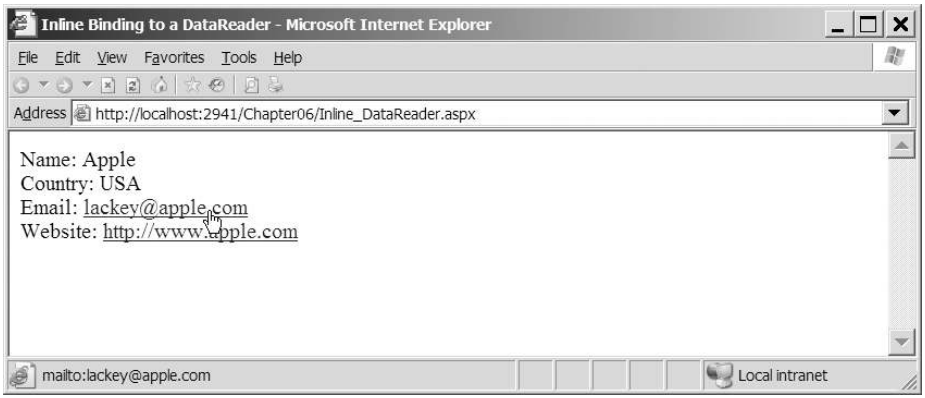

Figure 6-3. *Inline binding to a DataReader*

How It Works

The aim of the page is to display a Manufacturer's details, just as you did in the example in Chapter 5. However, here you limited your results to just a single Manufacturer (displaying a list of Manufacturers would imply list binding, which we'll cover later in this chapter):

```
SELECT ManufacturerName, ManufacturerCountry,
  ManufacturerEmail, ManufacturerWebsite
FROM Manufacturer
WHERE ManufacturerID = 1
```

Let's see how what you've done follows the three-step process for data binding outlined earlier.

Adding Data-Aware Web Controls

The first step is to add some Web controls to the page. Here, you added a Web control for each detail in the database. That way, you can experiment a bit with combinations of text and bound data to see what works and what doesn't.

Associating the Columns of Data to the Web Controls

The second step is to associate the columns of data to the Web controls. For the first `Label`, `lblName`, you've mixed the binding expression with the text value of the Web control's tag. In the expression itself, you call `DataBinder.Eval()`, which is a static method and thus always available.

```
<asp:Label id="lblName" runat="server">
  Name: <%# DataBinder.Eval (myReader, "[ManufacturerName]") %>
</asp:Label>
```

This requires two arguments: the name of the data source object (`myReader`) and a string stating from which column to take the value. Because you're using a DataReader, you can reference the column using either the name of the column or its index value in the row. You use the column name here, because it makes it a little easier to see what you're actually binding to the Web control.

■ **Tip** Using the name of the column, rather than its index, when data binding reduces the chances of errors that can sometimes be difficult to spot. If you're accessing columns by index and the columns in the `SELECT` query change, the indexes may no longer be valid, and you could swap two columns around that shouldn't be swapped. When you use the name of the column, as long as the column is still returned by the `SELECT` query, changing the columns that are returned won't cause any problems.

In the second `Label`, `lblCountry`, you're binding a value to the `Text` property. Of course, this amounts to the same thing, but it does show that you can bind to both a property and a value.

```
<asp:Label ID="lblCountry" runat="server"
  Text='<%# DataBinder.Eval (myReader, "[ManufacturerCountry]") %>'>
</asp:Label>
```

■ **Caution** You must use double quotes around the second parameter of `DataBinder.Eval()` and therefore single quotes around the binding expression as a whole. This is because ASP.NET associates single quotes with single-character values, not strings, and it will throw an error when it tries to parse `[ManufacturerCountry]` as a single character. HTML, on the other hand, isn't as picky about quotes, as long as they're paired correctly.

In the third `Label`, `lnkEmail`, you've attempted to bind to two properties, `NavigateUrl` and `Text`. However, you encountered the problem that hyperlinks require e-mail addresses to be prefixed with `mailto:` for them to be recognized as e-mail addresses rather than Web site addresses by the browser. Thus, you tried to concatenate text and binding expression inside the Web control's attribute, like so:

```
NavigateUrl='mailto: <%# DataBinder.Eval (myReader, "[2]") %>'
```

This doesn't work, because you can't mix the two inside a Web control's property. As soon as you do, ASP.NET regards the binding expression as literal text instead of as a placeholder for data. Also, *you have no way to alter the value of the column in the DataReader*. But it *is* possible to format the value of the column as it's being bound to the Web control using an alternate version of `DataBinder.Eval()`. You just pass it the format string `mailto:{0}` as its third parameter, and `DataBinder.Eval()` will retrieve the column from the DataReader, substitute it for the placeholder `{0}`, and assign the newly formatted string to the property:

```
<asp:HyperLink ID="lnkEmail" runat="server"
  NavigateUrl='mailto: <%# DataBinder.Eval (myReader, "[2]") %>'
  Text='<%# DataBinder.Eval (myReader, "[ManufacturerEmail]") %>'>
</asp:HyperLink>
```

Also notice that you've used both methods of referring to a column here. The column index is used by `NavigateUrl`, and the column name is used by the `Text` property. As noted earlier, these are interchangeable and, at least with the `SELECT` query as it stands at the moment, both refer to the same column within the row.

Finally, the fourth `Label` on the page demonstrates that you can bind to both the properties and the text value of a Web control at the same time:

```
<asp:HyperLink ID="lnkWebsite" runat="server"
  NavigateUrl='<%# DataBinder.Eval (myReader, "[3]") %>'>
  <%# DataBinder.Eval (myReader, "[ManufacturerWebsite]") %>
</asp:HyperLink>
```

Calling DataBind() on the Page

With all the placeholders for the data set up, it's just a matter of creating the DataReader, accessing the first row (you still have to call `Read()` or else there's nothing to bind to), and calling `DataBind()`:

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

if (myReader.Read())
{
  Page.DataBind();
}
else
{
  lblError.Text="No results to databind to.";
}

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

Notice that you've called `DataBind()` on the whole page rather than the individual Web controls. If you comment out this call, nothing at all will get set. As a slight extension of this example, try experimenting with binding individual Web controls just to prove that binding to one label won't affect the others unless they, too, are explicitly bound. Perhaps create a `Panel` that contains some of the Web controls and call `DataBind()` on that to prove that it's not just the `DataBind()` method in the `Page` that filters down to its children.

Try It Out: Inline Binding to a DataSet

Inline binding to values stored in a `DataSet` works in much the same way as the previous example with a DataReader. The main difference is that the binding expression is slightly different to accommodate the syntax used to identify tables, rows, and columns inside a `DataSet`. To see it in action, you'll adapt the previous example to do the same job using a `DataSet` instead of a DataReader.

1. In Visual Web Developer open `Inline_DataReader.aspx` and resave it as `Inline_DataSet.aspx`. Change the `<title>` tag of the page to be **Inline Binding to a DataSet**.

2. Add the `System.Data` `Import` statement to the top of the page:

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

3. Switch to the Source view. Rather than a DataReader, you need to alter `Page_Load` to use a `DataSet`. You'll use the same basic code you saw in Chapter 5, as follows (the changed lines are shown in bold):

```
// must declare the DataSet globally; else the page can't see it.
DataSet myDataSet = new DataSet();

void Page_Load(object sender, EventArgs e)
{
  // set up connection string and SQL query
  string ConnectionString = ConfigurationManager.
    ConnectionStrings["SqlConnectionString"].ConnectionString;
  string CommandText = "SELECT ManufacturerName, ManufacturerCountry, ➥
    ManufacturerEmail, ManufacturerWebsite FROM Manufacturer";

  // create SqlConnection and SqlCommand objects
  SqlConnection myConnection = new SqlConnection(ConnectionString);
  SqlCommand myCommand = new SqlCommand(CommandText, myConnection);

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

  // use try finally clauses when the connection is open.
  try
  {
    // open the database connection
    myConnection.Open();

    // use the DataAdapter to fill the DataSet
    myAdapter.Fill(myDataSet, "Manufacturer");
  }
  finally
  {
    // always close the database connection
    myConnection.Close();
  }

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

4. Now you set which property is bound to which column in the `DataTable`. Scroll to the bottom of the page to the `<body>` tag, and change the HTML to the following:

```
<body>
  <form id="form1" runat="server">
  <div>
    <asp:Label ID="lblName" runat="server">
    Name: DataBinder.Eval (myDataSet.Tables["Manufacturer"].Rows[0],
      "[ManufacturerName]")
    </asp:Label>
```

```
    <br />
    Country:
    <asp:Label ID="lblCountry" runat="server"
      Text='<%# DataBinder.Eval
        (myDataSet.Tables["Manufacturer"].Rows[0],
        "[ManufacturerCountry]") %>'>
    </asp:Label>
    <br />
    Contact:
    <asp:HyperLink ID="lnkEmail" runat="server"
      NavigateUrl='mailto: <%# DataBinder.Eval(
        myDataSet.Tables["Manufacturer"].Rows[0],"[2]","mailto:{0}") %>'
      Text='<%# DataBinder.Eval(
        myDataSet.Tables["Manufacturer"].Rows[0],
        "[ManufacturerEmail]") %>'>
    </asp:HyperLink>
    <br />
    Homesite:
    <asp:HyperLink ID="lnkWebsite" runat="server"
      NavigateUrl='<%# DataBinder.Eval(
        myDataSet.Tables["Manufacturer"].Rows[0], "[3]") %>'>
      <%# DataBinder.Eval (myDataSet.Tables["Manufacturer"].Rows[0],
        "[ManufacturerWebsite]") %>
    </asp:HyperLink>
    <br /><br />
    <asp:Label ID="lblError" runat="server"></asp:Label><br />
  </div>
  </form>
</body>
```

5. Save the page, and then view it in a browser (see Figure 6-4). The results are the same as for binding to a DataReader, as you saw in Figure 6-3.

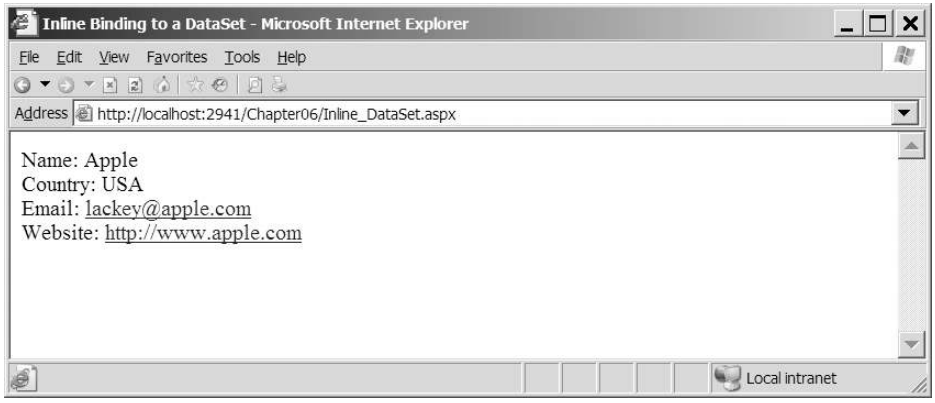

Figure 6-4. *Inline binding to a DataSet*

6. Go back to the Source view, and duplicate the Web controls and binding expressions on the page. The `ID` properties will be changed automatically to default values, but you should change them to `lblName2`, `lblCountry2`, `lnkEmail2`, and `lnkWebsite2`.

7. Change the binding expressions so that they bind to columns from a different row in the table. This is as simple as changing the `Rows` index, as this example shows:

```
<asp:Label ID="lblName2" runat="server">
  Name: DataBinder.Eval (myDataSet.Tables["Manufacturer"].Rows[4],
    "[ManufacturerName]")
</asp:Label>
<br />
Country:
  <asp:Label ID="lblCountry2" runat="server"
    Text='<%# DataBinder.Eval
     (myDataSet.Tables["Manufacturer"].Rows[4],
     "[ManufacturerCountry]") %>'>
</asp:Label>
```

8. Save the page, and then view it again (see Figure 6-5). You'll see that the `DataTable` has no problem. The DataReader would, of course, choke on this. The only way to work at random with a table with a DataReader is to keep rebuilding it.

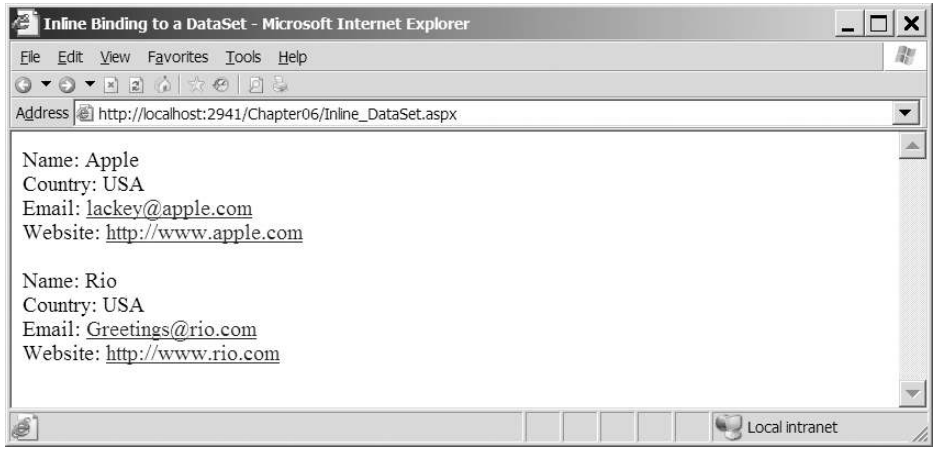

Figure 6-5. *The DataSet gives you random access through a table.*

How It Works

Only two major differences exist between this and the previous example. The first is that you're using a `DataSet` containing a `DataTable` to provide the data to bind to, and the second is the syntax you use in the binding expressions. Everything else is the same, demonstrating that there's no limit to inline binding text and properties at the same time.

This example may seem to contradict my earlier advice to query only for what you'll use in a page. You queried only the same four columns, but all of the rows from the Manufacturer table, like so:

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

However, you did this to demonstrate a point: unlike with a DataReader, you can work backward in the `DataSet` as well as forward.

Inside the binding expressions, you should recognize the slightly unwieldy syntax by now. Each expression identifies a column in a `DataSet` unambiguously. The second parameter of a call to `DataBinder.Eval()` names the specific column, and the first identifies the specific row that contains the column. If you recall from the previous example, the call when binding to a column in DataReader looked like this:

```
Name: <%# DataBinder.Eval (myReader, "[ManufacturerName]") %>
```

You can do this because a DataReader contains only one row at a time. It's therefore enough to identify the reader and then the position of the column in that row.

If you're using a `DataSet`, you need to use a call like this:

```
Name: <%# DataBinder.Eval (myDataSet.Tables["Manufacturer"].Rows[0],
  "[ManufacturerName]") %>
```

This slightly unwieldy syntax is required because you must first identify the `DataSet`, then the `DataTable` within it, and the specific row within that in one go, so that the second parameter can name the column in that row. You still have no way to alter the actual value of the column because it's bound to the Web control. However, if you need to format the value of the column, you just add the format string as the third parameter of the call to `DataBinder.Eval()`:

```
NavigateUrl='<%# DataBinder.Eval
  (myDataSet.Tables["Manufacturer"].Rows[4], "[2]", "mailto:{0}") %>'
```

The Inline Binding Alternative

I mentioned earlier that you have other options to inline binding, and that I prefer to avoid inline binding. But why not use data binding?

The main reason that you wouldn't want to use inline binding is performance. Inline binding uses reflection to determine what you're trying to show, and this is slower than using the DataReader and `DataSet` directly.

Another reason to avoid inline binding is code maintenance. If you use inline binding, you're mixing *real* code with the HTML using <% and %>. Those of you who've written ASP code in the past will remember doing this, with the result being the most horrible spaghetti code imaginable that was an absolute nightmare to debug. ASP.NET was supposed to fix all the problems with ASP, and yet here's one area where we have not moved forward.

Now, let's take a look at how you can show the results from a database query without resorting to data binding.

Try It Out: Showing Data from a DataReader

In this example, you'll display the same data from the DataReader without relying on inline binding. You'll build essentially the same pages as you've already seen, but without a call to `DataBind()` in sight.

1. Open `Inline_DataReader.aspx` and save it as `Showing_DataReader.aspx`.
2. In the Source view, scroll to the bottom of the page and change the `<title>` of the page to **Showing from a DataReader**.
3. Remove all the data-binding tags from within the `<body>` element. You should have HTML that looks similar to the following:

```
<body>
  <form id="form1" runat="server">
  <div>
    Name:
    <asp:Label ID="lblName" runat="server"></asp:Label>
    <br />
    Country:
    <asp:Label ID="lblCountry" runat="server"></asp:Label>
    <br />
    Email:
    <asp:HyperLink ID="lnkEmail" runat="server"></asp:HyperLink>
    <br />
    Website:
    <asp:HyperLink ID="lnkWebsite" runat="server"></asp:HyperLink>
    <br /><br />
    <asp:Label ID="lblError" runat="server"></asp:Label><br />
  </div>
  </form>
</body>
```

4. Within the code for the page, remove the global `SqlDataReader` definition and change the code within the `Page_Load` event to the following (the changed code is shown in bold):

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

if (myReader.Read())
{
  // set the properties on the controls
  lblName.Text = Convert.ToString(myReader["ManufacturerName"]);
  lblCountry.Text = Convert.ToString(myReader["ManufacturerCountry"]);
  lnkEmail.Text = Convert.ToString(myReader["ManufacturerEmail"]);
  lnkEmail.NavigateUrl = "mailto:" +
    Convert.ToString(myReader["ManufacturerEmail"]);
```

```
    lnkWebsite.Text = Convert.ToString(myReader["ManufacturerWebsite"]);
    lnkWebsite.NavigateUrl =
      Convert.ToString(myReader["ManufacturerWebsite"]);
  }
  else
  {
    // show the error
    lblError.Text="No results to databind to.";
  }
```

5. Save the page, and then view it in a browser (see Figure 6-6). The results are the same as for binding to a DataReader (Figure 6-3).

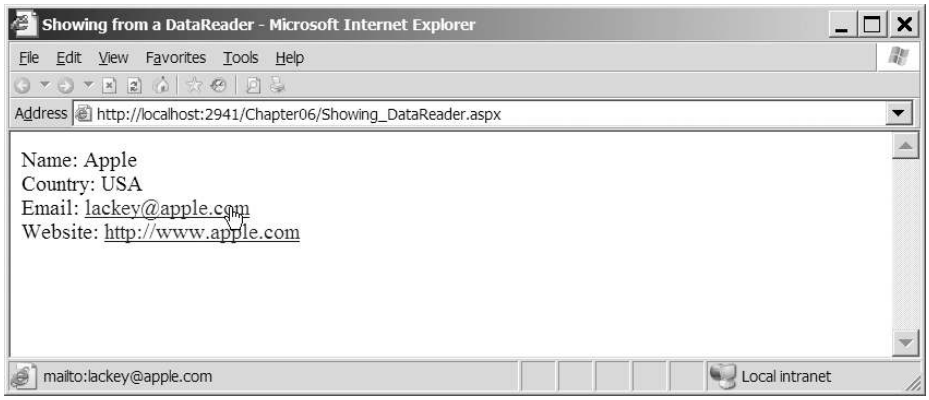

Figure 6-6. *Showing from a DataReader without binding*

How It Works

In this example, you changed quite a lot around compared to the corresponding inline binding exercise. The benefit is that these changes have made the split between what is code and what is presentation a lot easier to see.

The first change is to remove any data-binding code from the HTML. Here's how you created the `Label` controls in the data-binding example:

```
<asp:Label ID="lblName" runat="server">
  Name: <%# DataBinder.Eval (myReader, "[ManufacturerName]") %>
</asp:Label>
<br />
Country:
<asp:Label ID="lblCountry" runat="server"
  Text='<%# DataBinder.Eval (myReader, "[ManufacturerCountry]") %>'>
</asp:Label>
<br />
```

Compare this to what you have now:

```
Name:
<asp:Label ID="lblName" runat="server"></asp:Label>
<br />
Country:
<asp:Label ID="lblCountry" runat="server"></asp:Label>
<br />
```

It certainly makes the HTML a lot easier to follow, now that no data-binding code is mingled among the HTML.

You also now need to set the properties in code. You need to move the data binding from the `DataBinder.Eval()` calls within the HTML to the code. And you do this by replacing the `Page.DataBind()` with code that sets the properties directly on the four Web controls:

```
// set the properties on the controls
lblName.Text = Convert.ToString(myReader["ManufacturerName"]);
lblCountry.Text = Convert.ToString(myReader["ManufacturerCountry"]);
lnkEmail.Text = Convert.ToString(myReader["ManufacturerEmail"]);
lnkEmail.NavigateUrl = "mailto:" +
  Convert.ToString(myReader["ManufacturerEmail"]);
lnkWebsite.Text = Convert.ToString(myReader["ManufacturerWebsite"]);
lnkWebsite.NavigateUrl =
  Convert.ToString(myReader["ManufacturerWebsite"]);
```

You return the column you're after from the current row in the DataReader by passing the column name as the index to the `myReader` object. This returns the column contents as an `Object`, and this is cast to the correct type, in this case a `String`, before the properties on the Web controls are set.

The only change to the way that you set the properties is the `NavigateUrl` property of the e-mail link. You need to prepend the e-mail address with `mailto:` to ensure that it appears in the browser correctly. You can do this easily by setting `NavigateUrl` to the concatenation of the two strings.

Try It Out: Showing Data from a DataSet

As well as querying a DataReader directly, it is also possible to query a `DataSet` directly. This example will show how you can query the `DataSet` to return the correct data and use this to set properties on the Web controls on the page.

1. Open `Inline_Binding_DataSet.aspx` and save it as `Showing_DataSet.aspx`.
2. Change the `<title>` of the page to **Showing from a DataSet**.
3. Remove all the data-binding code from the `<body>` element. You should have HTML similar to the following:

```
<body>
  <form id="form1" runat="server">
  <div>
    Name:
    <asp:Label ID="lblName" runat="server"></asp:Label>
    <br />
    Country:
    <asp:Label ID="lblCountry" runat="server"></asp:Label>
    <br />
    Contact:
    <asp:HyperLink ID="lnkEmail" runat="server"></asp:HyperLink>
    <br />
    Homesite:
    <asp:HyperLink ID="lnkWebsite" runat="server"></asp:HyperLink>
    <br /><br />
    Name:
    <asp:Label ID="lblName2" runat="server"></asp:Label>
    <br />
    Country:
    <asp:Label ID="lblCountry2" runat="server"></asp:Label>
    <br />
    Contact:
    <asp:HyperLink ID="lnkEmail2" runat="server"></asp:HyperLink>
    <br />
    Homesite:
    <asp:HyperLink ID="lnkWebsite2" runat="server"></asp:HyperLink>
    <br /><br />
    <asp:Label ID="lblError" runat="server"></asp:Label><br />
  </div>
  </form>
</body>
```

4. Within the code for the page, remove the global `DataSet` definition and move it to the `Page_Load` event, as follows:

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

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

// use try finally clauses when the connection is open.
try
```

5. Replace the `Page.DataBind()` call at the end of the `Page_Load` event with the following:

```
// show the first results
DataRow myFirstRow = myDataSet.Tables["Manufacturer"].Rows[0];
lblName.Text = Convert.ToString(myFirstRow["ManufacturerName"]);
lblCountry.Text = Convert.ToString(myFirstRow["ManufacturerCountry"]);
lnkEmail.Text = Convert.ToString(myFirstRow["ManufacturerEmail"]);
lnkEmail.NavigateUrl = "mailto:" +
  Convert.ToString(myFirstRow["ManufacturerEmail"]);
lnkWebsite.Text = Convert.ToString(myFirstRow["ManufacturerWebsite"]);
lnkWebsite.NavigateUrl =
  Convert.ToString(myFirstRow["ManufacturerWebsite"]);

// show the second results
DataRow mySecondRow = myDataSet.Tables["Manufacturer"].Rows[4];
lblName2.Text = Convert.ToString(mySecondRow["ManufacturerName"]);
lblCountry2.Text = Convert.ToString(mySecondRow["ManufacturerCountry"]);
lnkEmail2.Text = Convert.ToString(mySecondRow["ManufacturerEmail"]);
lnkEmail2.NavigateUrl = "mailto:" +
  Convert.ToString(mySecondRow["ManufacturerEmail"]);
lnkWebsite2.Text = Convert.ToString(mySecondRow["ManufacturerWebsite"]);
lnkWebsite2.NavigateUrl =
  Convert.ToString(mySecondRow["ManufacturerWebsite"]);
```

6. Save the page, and then view it in a browser (see Figure 6-7). The results are the same as for binding to a `DataSet` (Figure 6-5).

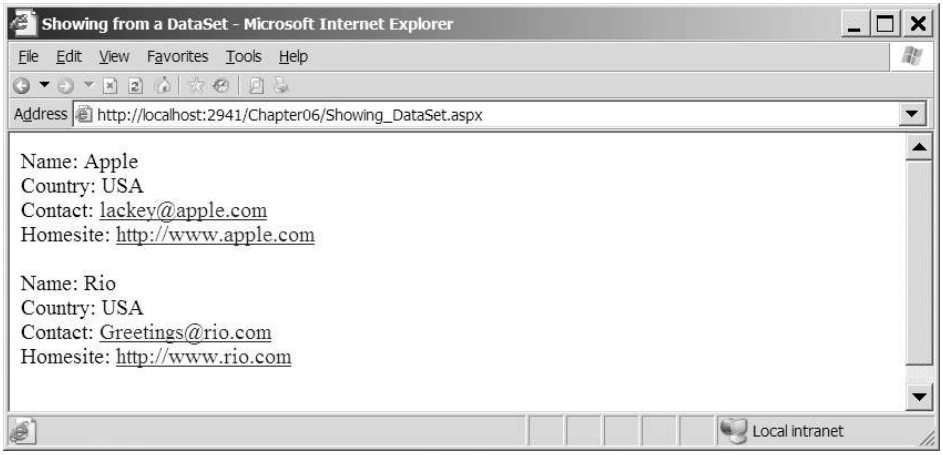

Figure 6-7. *Showing from a DataSet without binding*

How It Works

As with the DataReader example, you've removed the code from the HTML and placed it where it should be: within the code part of the page.

As you've already seen, when dealing with a DataSet, the first thing you need is a reference to the correct row within the correct table of the DataSet. Rather than having to repeat a rather unwieldy syntax every time you need to access the row, you store a reference to the row:

```
DataRow myFirstRow = myDataSet.Tables["Manufacturer"].Rows[0];
```

So, whenever you need to retrieve a column from the row, you can do so relatively easily. One of the indexes for a DataRow is the name of the column that you're after, so the syntax is remarkably similar to that for the DataReader earlier:

```
lblName.Text = Convert.ToString(myFirstRow["ManufacturerName"]);
```

After showing the data for the first row in the DataTable, you then repeat the process, but this time, show the details for the fifth row by getting a reference to a different row in the table:

```
DataRow myFirstRow = myDataSet.Tables["Manufacturer"].Rows[4];
```

List Binding

In comparison to inline binding, list binding is a somewhat less complex way to pass data into a data-bound Web control, but usually it requires more work beyond the call to DataBind() to finesse your Web control into a useful piece of the page for the user. If you consider the four Web controls you can list-bind to—the CheckBoxList, RadioButtonList, DropDownList, and ListBox—you can see that their purpose is to elicit a response from the user for information to be used elsewhere in the page.

■ **Note** There is a fifth Web list control, the BulletedList, which is more of a display-only Web control and doesn't allow the user to select options from the list. Even though we're not going to look at it here, the method for populating the Web control is the same as for the other four Web controls.

As you know from using these Web controls and working with HTML option lists, the key to finding out what the user has chosen is to establish a unique value for each option that can be retrieved from the page once the choice has been made. And, of course, you also need some text to display against each choice. In HTML, that means something like this:

```
<select name="ListBox1" size="5" id="ListBox1">
   <option value="1">Apple</option>
   <option value="2">Creative</option>
   <option value="3">iRiver</option>
   <option value="4">MSI</option>
   <option value="5">Rio</option>
</select>
```

On the screen, it looks something like Figure 6-8.

Figure 6-8. *You can build a simple check box by list binding.*

Apple, Creative, iRiver, and so on are displayed in the browser, and the choice registers in the page as 1, 2, 3, and so on. When you're list binding to a Web control, you need to establish a data source containing two columns per row: one for the text to be displayed and the other to identify the choice the user has made. Here's an example:

```
SELECT ManufacturerID, ManufacturerName FROM Manufacturer
```

With the data source established (either by setting the `DataSource` or `DataSourceID` property, in code or in the HTML markup of the Web control), you need to tell the Web control which column does what. You do this by setting the `DataTextField` and `DataValueField` properties as appropriate. You can do this either in HTML:

```
<asp:RadioButtonList id="RadioButtonList1" runat="server"
  DataTextField="ManufacturerName" DataValueField="ManufacturerID" />
```

or in code:

```
RadioButtonList1.DataTextField = "ManufacturerName";
RadioButtonList1.DataValueField = "ManufacturerID";
```

■ **Caution** If you do set the `DataTextField` and `DataValueField` properties in code, make sure you specify the data source before the other two properties, or you'll get an error.

The Web list controls have the following two relevant properties:

- `DataMember`: This property is for use with a `DataSet`, as you'll see in the example in the upcoming "Try It Out: Using Lookup Lists and Events with a DataSet" section. As a `DataSet` can contain multiple tables, this property is used to specify which table to use from the `DataSet`.
- `DataTextFormatString`: This property lets you set the format for the text displayed in the list and comes in handy when you're dealing with currency, dates, and numbers. See `http://msdn.microsoft.com/en-us/library/system.web.ui.webcontrols.listcontrol.datatextformatstring.aspx` for examples.

Once the Web list control has been configured correctly, it's time to actually perform the data binding. Using a `SqlDataSource`, the data binding takes place automatically. If you're using a `DataSet` or a DataReader as the data source, or you need the data binding to occur at a specific place in code, then all that's left to do is call `DataBind()` on the Web list control.

So, the core code for the whole three-stage binding process could be as simple as the following:

```
myConnection.Open();
myReader = myCommand.ExecuteReader();
RadioButtonList1.DataSource = myReader;
RadioButtonList1.DataTextField = "ManufacturerName";
RadioButtonList1.DataValueField = "ManufacturerID";
RadioButtonList1.DataBind();
myReader.Close();
myConnection.Close();
```

This is true of all four Web list controls.

Another thing that you'll soon discover is that you can use a DataReader as a Web list control's data source only on a one-to-one basis. Unlike inline binding, where you can bind many properties to the same column in a DataReader, you can't bind many Web list controls to the same columns in a DataReader. Actually, you can't bind more than one Web list control to the same DataReader. Once `DataBind()` has been called on one of the Web list controls, it works through all the rows in a DataReader, which, of course, is forward-only, so you can't go back to the beginning and bind the same information. The only way to bind all three Web controls from the same source is to use something other than a DataReader—a `DataSet`, for example—as the data source.

That said, you'll now look at a common application of data-bound lists: using the selection from a list to look up data from another table. First, you'll see how to do this using a DataReader as the source of the list to populate a `GridView` based on the user's selection from a `DropDownList`, a `RadioButtonList`, or a `ListBox`.

You can accomplish the same task using a `DataSet` instead of a DataReader, but in order to do that, you need to start dealing with events. So, first we'll look at the two relevant events, `DataBound` and `SelectedIndexChanged`, and then we'll extend the DataReader example before moving on to build the corresponding page using a `DataSet`.

Try It Out: Using Single-Value Lookup Lists with a DataReader

In this example, you'll build a page with two stages. In the first stage, you'll populate a `DropDownList` with information from the Manufacturer table. Specifically, you'll make the name of all the Manufacturers in the database appear on the screen and use their respective ManufacturerID values to track which Manufacturer has been clicked. The `DropDownList` allows only one value to be selected in the list, and in the second stage, you'll use the ManufacturerID of the selected Manufacturer to search the Player table for all the Players made by that Manufacturer, and then display that in a `GridView`.

1. In Visual Web Developer, add a new Web Form called `List_DataReader.aspx`.
2. In the Source view, change the title of the page to **List Binding to a DataReader**.
3. In the Design view, add a `DropDownList` and a `GridView` to the page. Rather than having the Web controls lined up underneath each other, use an HTML table to lay out the Web controls in a more pleasing manner, as shown in Figure 6-9.

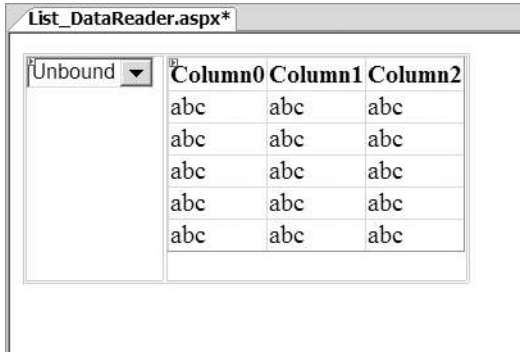

Figure 6-9. *Laying out the DropDownList and GridView*

4. Select the `DropDownList` and set its ID to `lstManufacturers`. Also set `DataTextField` to ManufacturerName, `DataValueField` to ManufacturerID, and `AutoPostBack` to True.

5. In the Source view, make sure that you've included the correct data provider at the top of the page, like so:

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

6. All the code for the page is in the `Page_Load` event handler. It begins by setting up the Connection and Command objects, like so:

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

  // create SqlCommand object
  SqlCommand myCommand = new SqlCommand();
  myCommand.Connection = myConnection;

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

7. Inside the `try` loop, you need to run two different queries: one for setting up the `DropDownList` and the other for the `GridView`. To set up the `DropDownList`, add the following code:

```
if (Page.IsPostBack == false)
{
  // If this page isn't posted back
  // you need to set up the list control
  // set up SQL query for the Manufacturer table
  myCommand.CommandText =
    "SELECT ManufacturerID, ManufacturerName FROM Manufacturer";

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

  // set up the list
  lstManufacturers.DataSource = myReader;
  lstManufacturers.DataBind();

  // close the reader
  myReader.Close();
}
```

8. To set up the `GridView`, you need the following code. Again, you're using only the `DropDownList`, so the calls against the other Web list controls are commented out.

```
else
{
  // If this page is posted back get the selected value and display
  // players made by manufacturer. You don't need to rebind the value
  // for the lists either. They are stored in the viewstate.

  // set up SQL query for the Player table
  myCommand.CommandText =
    "SELECT PlayerID, PlayerName, PlayerManufacturerID, PlayerCost, ➥
    PlayerStorage FROM Player WHERE PlayerManufacturerID = " +
    lstManufacturers.SelectedItem.Value;

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

  // setup the GridView
  GridView1.DataSource = myReader;
  GridView1.DataBind();

  // close the reader
  myReader.Close();
}
```

9. And last but not least, you need to tidy things up, like so:

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

10. Save the page, and then view it in a browser. Select one of the Manufacturers from the drop-down list. The page will post back to the server and, presto, return with the details for the Players made by that Manufacturer, as shown in Figure 6-10.

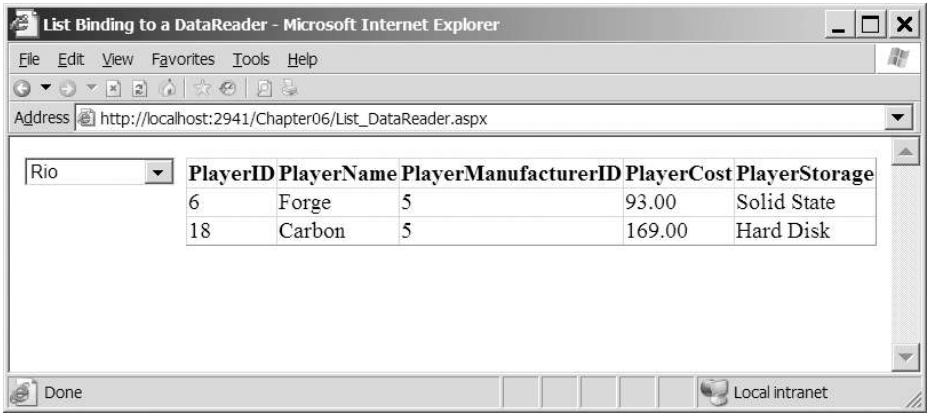

Figure 6-10. *Selecting a single value using a DropDownList*

11. Now that you've seen how this works with a `DropDownList`, you can experiment with both the `RadioButtonList` and `ListBox` to perform the same task. Switch back to Visual Web Developer and change the `<asp:DropDownList>` and `</asp:DropDownList>` tags to `<asp:RadioButtonList>` and `</asp:RadioButtonList>`. Save the page, and then view it in the browser. Selecting a Manufacturer from the radio button list will post back to the server and populate the Players automatically, as shown in Figure 6-11.

12. Switch back to Visual Web Developer and change the `<asp:RadioButtonList>` and `</asp:RadioButtonList>` tags to `<asp:ListBox>` and `</asp:ListBox>`. Save the page, and view it in the browser. Selecting a Manufacturer from the list box will post back to the server and populate the Players automatically, as shown in Figure 6-12.

13. Switch back to Visual Web Developer and revert back to using a `DropDownList` by changing the `<asp:ListBox>` and `</asp:ListBox>` tags to `<asp:DropDownList>` and `</asp:DropDownList>`.

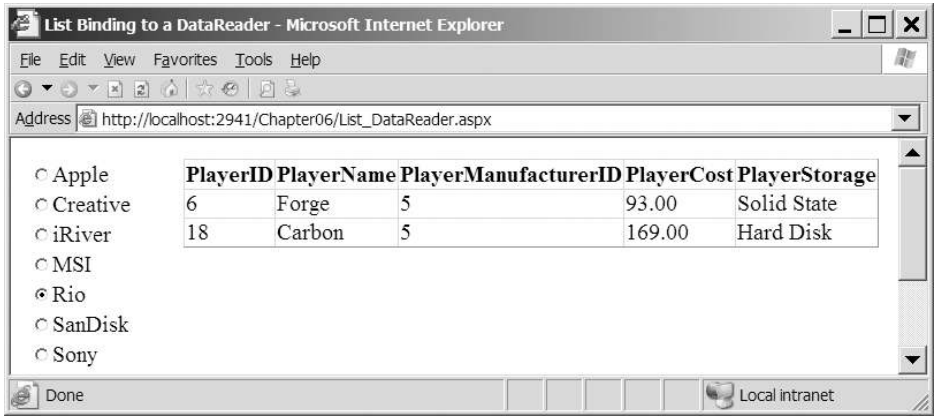

Figure 6-11. *Selecting a single value using a RadioButtonList*

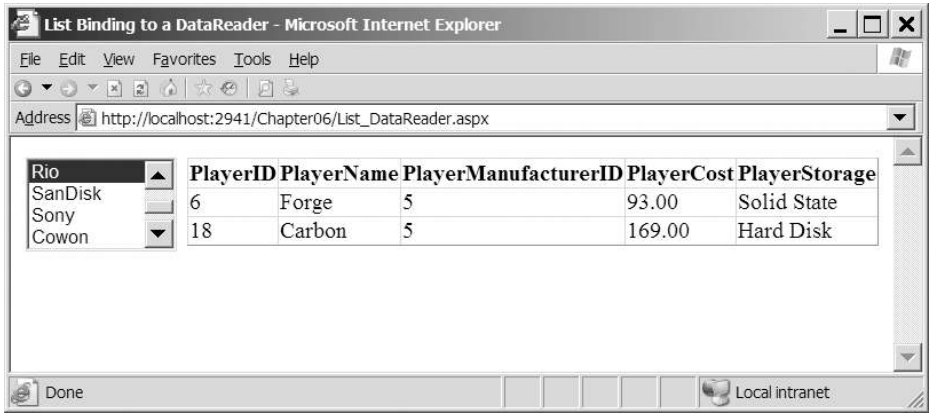

Figure 6-12. *Selecting a single value using a ListBox*

How It Works

We've discussed a great deal of this code already, but it makes a lot of difference to see it in action. Seeing how `DataTextField` and `DataValueField` translate inside a data-bound list, for example, is helpful.

When the page is first loaded, you need to populate the list from the database. However, this list will never change as you use the page, so you don't need to repopulate it each time you click the values it contains. Consequently, you set up a simple test to check whether the page has posted back. If it hasn't, you'll populate the list.

```
if (Page.IsPostBack == false)
{
```

Although the DataTextField and DataValueField properties specify what will populate the list, there's still no need to query the database for any more than those columns. You're using a DataReader, so you can't access any extra information from it anyway once you DataBind() it to the list.

```
  // set up SQL query for Manufacturer table
  myCommand.CommandText =
    "SELECT ManufacturerID, ManufacturerName FROM Manufacturer";

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

  // set up the list
  lstManufacturers.DataSource = myReader;
  lstManufacturers.DataBind();

  // close the reader
  myReader.Close();
}
```

■ **Note** You set the DataSource property within the code rather than the HTML (as you saw when we looked at inline binding), as this makes the code a little more self-contained. This way, you don't need to have a global reference to the DataReader for the page.

You set the AutoPostBack property of the DropDownList to true, so any time you select an option from the list, the page posts back, and you can update the GridView accordingly.

The DropDownList, and indeed all three Web list controls, expose the currently selected item in the list through the SelectedItem property. You can use its Text and Value properties to retrieve the exact details. In this case, you need the ManufacturerID for the Player search, so you use SelectedItem.Value because you set ManufacturerID to DataValueField in the Web list control.

```
else
{
  // set up SQL query for Player table
  myCommand.CommandText =
    "SELECT PlayerID, PlayerName, PlayerManufacturerID, PlayerCost, ➥
    PlayerStorage FROM Player WHERE PlayerManufacturerID = " +
    lstManufacturers.SelectedItem.Value;

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

```
  // set up the GridView
  GridView1.DataSource = myReader;
  GridView1.DataBind();

  // close the reader
  myReader.Close();
}
```

An alternate way to go here would be to use the Web list control's `SelectedValue` and `SelectedText` properties instead of `SelectedItem.Value` and `SelectedItem.Text`. That approach produces the same results in this scenario, where you can select only one item from the list. However, this doesn't work with multiple-selection lists, as you'll see in the "Multiple-Selection Lists" section later in this chapter. Nor can you use the `SelectedItem` property to find all the list items selected in a group. You must choose another tack.

The last two steps in the example changed the Web list control from a `DropDownList` to a `RadioButtonList` and then to a `ListBox`. You simply changed the HTML tags from `<asp:DropDownList>` to `<asp:RadioButtonList>` or `<asp:ListBox>`. Nothing else on the page changes—all Web list controls have exactly the same interface, and the same methods and properties are available.

List Binding Events

In the previous example, you saw how easy it is to build a list from a DataReader. Before you can see how to do the same thing using the `DataSet` and `SqlDataSource`, you need to be aware of the events that are exposed by the Web list controls.

The main problem with the previous example is that there is no way to know which Web list control the user used to cause the postback to the page. As you have only one visible Web list control, this isn't a problem. You can assume that if there has been a postback, the visible Web control caused the postback, and so populate the `GridView` accordingly. This is exactly what you've done:

```
if (Page.IsPostBack == false)
{
  // populate the list control
}
else
{
  // populate the GridView
}
```

However, when you start adding more Web controls to the page, such as a button or indeed another Web list control that has its `AutoPostBack` property set to true, you have a problem. How do you know which Web control has caused the postback? A button has a `Click` event, but what about a Web list control?

Thankfully, a Web list control has its own event that is raised whenever the page is posted back because of a user selection within the list: `SelectedIndexChanged`. As you'll soon see, you can use this event to modify other Web controls.

As well as the SelectedIndexChanged event, Web list controls have another very helpful event. Introduced in ASP.NET 2.0, the DataBound event is fired immediately after the Web control has been data-bound—whether this is automatic or caused by an explicit call to DataBind() for the Web control or its parent.

Try It Out: Using Lookup Lists and Events with a DataReader

In this example, you'll modify the previous DataReader example to use the SelectedIndexChanged and DataBound events, rather than relying on the fact that the page has been posted back to assume that the user has made a selection.

1. Open List_DataReader.aspx and save it as List_Binding_Events.aspx.
2. Change the <title> of the page to **List Binding with Events to a DataReader**.
3. In the Source view, change the Page_Load event handler to the following:

```
protected void Page_Load(object sender, EventArgs e)
{
  if (Page.IsPostBack == false)
  {
    // create SqlConnection object
    string ConnectionString = ConfigurationManager.
      ConnectionStrings["SqlConnectionString"].ConnectionString;
    SqlConnection myConnection = new SqlConnection(ConnectionString);

    // create SqlCommand object
    SqlCommand myCommand = new SqlCommand();
    myCommand.Connection = myConnection;

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

      // set up SQL query for Manufacturer table
      myCommand.CommandText =
        "SELECT ManufacturerID, ManufacturerName FROM Manufacturer";

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

      // set up the list
      lstManufacturers.DataSource = myReader;
      lstManufacturers.DataBind();
```

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

4. In the Design view, double-click the `DropDownList`. This will add the `SelectedIndexChanged` event handler. Change the code within the handler to the following:

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

  // create SqlCommand object
  string CommandText = "SELECT PlayerID, PlayerName, PlayerManufacturerID, ➥
    PlayerCost, PlayerStorage FROM Player WHERE PlayerManufacturerID = " +
    lstManufacturers.SelectedItem.Value;
  SqlCommand myCommand = new SqlCommand(CommandText, myConnection);

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

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

    // set up the GridView
    GridView1.DataSource = myReader;
    GridView1.DataBind();

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

5. Save the page, and then view it in the browser. Selecting one of the options will populate the results with the Players for the selected Manufacturer, similar to the results you've already seen in Figure 6-10. However, when the page is first loaded, the `DropDownList` is showing Apple, but the `GridView` doesn't appear!

6. Switch back to Visual Web Developer, and in the Design view, show the properties for the `DropDownList`. Switch to the Events view and double-click the `DataBound` event to add the event handler, as shown in Figure 6-13.

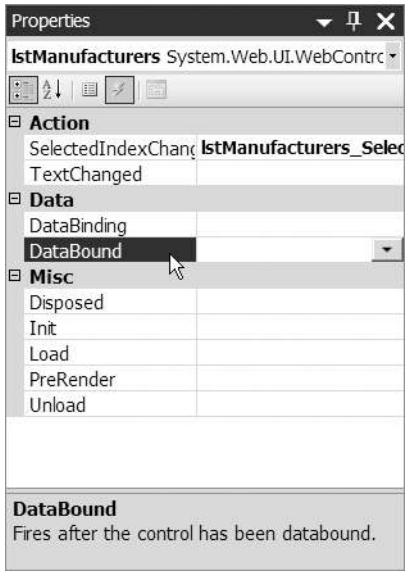

Figure 6-13. *Setting the DataBound event for a DropDownList*

7. Add the following code to the `DataBound` event:

```
protected void lstManufacturers_DataBound(object sender, EventArgs e)
{
  ListItem myListItem = new ListItem();
  myListItem.Text = "please select...";
  myListItem.Value = "-1";
  lstManufacturers.Items.Insert(0, myListItem);
}
```

8. Modify the code within the `lstManufacturers_SelectedIndexChanged` event handler as follows:

```
protected void DropDownList1_SelectedIndexChanged(object sender, EventArgs e)
{
  if (lstManufacturers.SelectedValue != "-1")
  {
    // code as it stands at the moment
  }
```

```
    else
    {
      // clear the GridView
      GridView1.DataSource = null;
      GridView1.DataBind();
    }
  }
```

9. Save the page, and then view it in the browser. This time, notice a new entry, "please select...," added to the `DropDownList`. Select one of the Manufacturers, and the Player list changes to the selected Manufacturer's Players. Select "please select...," and notice that the list of Players disappears.

10. Experiment with changing the `DropDownList` to a `RadioButtonList` and a `ListBox`. Notice again that all three Web controls will have the "please select..." entry, as well as the list of Manufacturers, and selecting an option displays the same results, regardless of the type of Web list control that you're using.

How It Works

The first thing that you'll notice about this code is that the `Page_Load` event is no longer populating the `GridView`, and this is now handled by the `SelectedIndexChanged` event of the Web list control.

The `Page_Load` event is now solely responsible for creating the list of Manufacturers for the Web list control. As you saw in the previous example, this must be done only when the page is first loaded, so the first thing that you check is that this is indeed the first load of the page. If it is, you populate the Web list control in the same way as the previous example, by querying the Manufacturer table for the ManufacturerID and ManufacturerName combinations.

Within the `SelectedIndexChanged` event, you first need to create a connection to the database, and then create the correct SQL query to execute:

```
// create SqlConnection object
string ConnectionString = ConfigurationManager.
  ConnectionStrings["SqlConnectionString"].ConnectionString;
SqlConnection myConnection = new SqlConnection(ConnectionString);

// create SqlCommand object
string CommandText = "SELECT PlayerID, PlayerName, PlayerManufacturerID, ➥
  PlayerCost, PlayerStorage FROM Player WHERE PlayerManufacturerID = " +
  lstManufacturers.SelectedItem.Value;
SqlCommand myCommand = new SqlCommand(CommandText, myConnection);
```

The SQL query is constructed in the same way as in the previous example. You retrieve the value selected from the drop-down list using the `SelectedItem.Value` property and concatenate this with the rest of the query.

Once you have a Command object, you can run the query and pass the results to the `DataSource` of the `GridView` and bind the results:

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

// set up the GridView
GridView1.DataSource = myReader;
GridView1.DataBind();

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

The only difference from the previous example is that you've changed the page to respond to a specific event, `SelectedIndexChanged`, to bind the Players to the `GridView`.

The one new piece of code that you've added here is the `DataBound` event, which you use to add the "please select..." entry to the list.

A `DropDownList` always has an item selected, since it must display something, even if the user hasn't made a selection. When you first loaded the page Apple was the selected entry, but the list of Players made by Apple wasn't shown in the list. Contrast this with both the `RadioButtonList` and the `ListBox`, which don't have a selected item when they're first loaded. So, you need some way of dealing with this, which is the purpose of the "please select..." entry to the Web list control.

All additions to a Web list control must be made once any data binding has occurred. Before ASP.NET 2.0, any additions to a Web list control had to be made after the Web list control had been data-bound, as follows:

```
// data-bind the list
DropDownList1.DataBind();

// add the "please select..." entry
ListItem myListItem = new ListItem();
myListItem.Text = "please select...";
myListItem.Value = "-1";
lstManufacturers.Items.Add(myListItem);
```

While this works without any problems, you're using an event-driven programming model, and the new `DataBound` event can be used for this purpose:

```
protected void lstManufacturers_DataBound(object sender, EventArgs e)
{
  ListItem myListItem = new ListItem();
  myListItem.Text = "please select...";
  myListItem.Value = "-1";
  lstManufacturers.Items.Insert(0, myListItem);
}
```

Once the data binding of the Web list control is complete, the `DataBound` event is fired and the new entry added to the Web list control. Either option works to do the same task, but using the event separates the different parts of the code more cleanly and will make the code easier to maintain later.

Obviously, you can't use the dummy entry "please select ..." as a Manufacturer. Therefore, you check that the selected entry is a real one when handling the `SelectedIndexChanged` event.

You know that the ManufacturerID values in the database start at 1, so you set the dummy entry to have a value that can't exist, -1. If this is the value that the user has selected, you don't attempt to bind to a set of results from the database. Instead, you remove any data binding already in place by telling the GridView to bind to a null data source:

```
protected void DropDownList1_SelectedIndexChanged(object sender, EventArgs e)
{
  if (DropDownList1.SelectedValue != "-1")
  {
    // code to show the players
  }
  else
  {
    // clear the GridView
    GridView1.DataSource = null;
    GridView1.DataBind();
  }
}
```

You'll notice that you've added the dummy entry to all Web list controls that you may use, including the RadioButtonList and ListBox, even though they can display themselves without having an entry selected. It's only the DropDownList that cannot and needs the dummy entry. Therefore, you may want to remove the DataBound event handler when using a RadioButtonList or a ListBox.

Try It Out: Using Lookup Lists and Events with a DataSet

In this exercise, you'll see how the code for list binding a DataTable in a DataSet to a Web control is almost identical to list binding to a DataReader.

1. Open List_DataReader_Events.aspx and save it as List_DataSet_Events.aspx.
2. Change the <title> of the page to **List Binding with Events to a DataSet**.
3. Add the System.Data Import statement to the top of the page:

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

4. If the page is using a Web list control other than the DropDownList, change it back to a DropDownList and, if necessary, add the DataBound event that you used in the previous example.
5. Add a RadioButtonList to the page beneath the DropDownList. Set its DataTextField property to ManufacturerName, DataValueField to ManufacturerID, and AutoPostBack to True.
6. You need to populate a DataSet from three different places, so you're going to move the code to do so to a new method called BuildDataSet(). Add the following:

```
private DataSet BuildDataSet(string commandText, string tableName)
{
  // DataSet we're going to return
  DataSet myDataSet = new DataSet();

  // set up connection string
  string ConnectionString = ConfigurationManager.
    ConnectionStrings["SqlConnectionString"].ConnectionString;

  // create SqlConnection and SqlCommand objects
  SqlConnection myConnection = new SqlConnection(ConnectionString);
  SqlCommand myCommand = new SqlCommand(commandText, myConnection);

  // Create the SqlDataAdapter
  SqlDataAdapter myAdapter = new SqlDataAdapter(myCommand);

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

    // fill the DataSet
    myAdapter.Fill(myDataSet, tableName);
  }
  finally
  {
    // always close the connection
    myConnection.Close();
  }

  // return the DataSet
  return(myDataSet);
}
```

7. Modify the `Page_Load` event to use the `BuildDataSet()` method to retrieve the data and populate the Web list control. Since this needs to be done only when the page first loads, the code needs to run only when the page hasn't been posted back.

```
protected void Page_Load(object sender, EventArgs e)
{
  if (Page.IsPostBack == false)
  {
    // set up SQL query for Manufacturer table
    string CommandText =
      "SELECT ManufacturerID, ManufacturerName FROM Manufacturer";
```

```
    // DataSet with list of manufacturers
    DataSet myDataSet = BuildDataSet(CommandText, "Manufacturer");

    // set up the DropDownList
    lstManufacturers.DataSource = myDataSet;
    lstManufacturers.DataMember = "Manufacturer";
    lstManufacturers.DataBind();

    // set up the RadioButtonList
    RadioButtonList1.DataSource = myDataSet;
    RadioButtonList1.DataMember = "Manufacturer";
    RadioButtonList1.DataBind();
  }
}
```

8. You now need to change the `SelectedIndexChanged` event handler for the `DropDownList` to use the new `BuildDataSet()` method to retrieve the results from the database. Change the `lstManufacturers_SelectedIndexChanged` as follows:

```
protected void lstManufacturers_SelectedIndexChanged(object sender,
  EventArgs e)
{
  if (lstManufacturers.SelectedValue != "-1")
  {
    // set up SQL query for Player table
    string CommandText = "SELECT PlayerID, PlayerName, PlayerManufacturerID, ➥
      PlayerCost, PlayerStorage FROM Player WHERE PlayerManufacturerID = " +
      lstManufacturers.SelectedItem.Value;

    // set up the GridView
    GridView1.DataSource = BuildDataSet(CommandText, "Player");
    GridView1.DataMember = "Player";
    GridView1.DataBind();
  }
  else
  {
    // clear the GridView
    GridView1.DataSource = null;
    GridView1.DataBind();
  }
}
```

9. Switch to the Design view and double-click the `RadioButtonList` to add its `SelectedIndexChanged` event. Add the following code to the event handler:

```
protected void RadioButtonList1_SelectedIndexChanged(object sender,
   EventArgs e)
{
  // set up SQL query for Player table
  string CommandText = "SELECT PlayerID, PlayerName, PlayerManufacturerID, ➥
    PlayerCost, PlayerStorage FROM Player WHERE PlayerManufacturerID = " +
    RadioButtonList1.SelectedItem.Value;

  // set up the GridView
  GridView1.DataSource = BuildDataSet(CommandText, "Player");
  GridView1.DataMember = "Player";
  GridView1.DataBind();
}
```

10. Save the page, and then run it in a browser. Select one of the Manufacturers from either the `DropDownList` or the `RadioButtonList`. The page will post back to the server, and the list of Players will be presented, based on the Manufacturer selected, as shown in Figure 6-14.

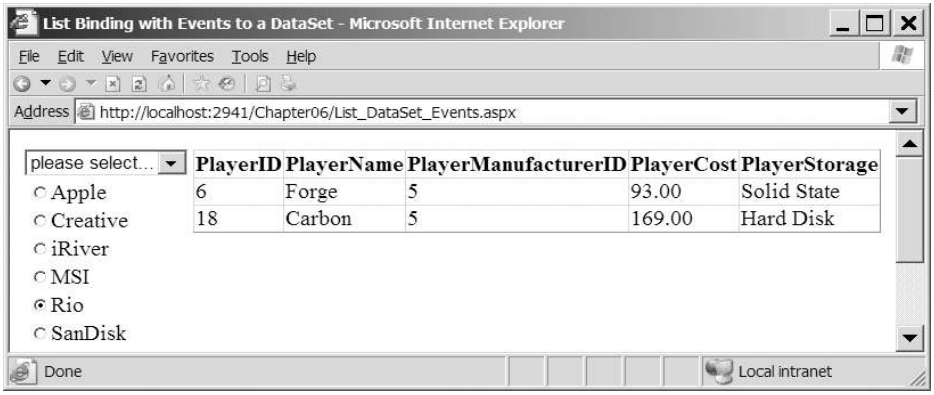

Figure 6-14. *Backing lookup lists with a DataSet*

How It Works

The most immediate difference between this example and its DataReader equivalent is that both the `DropDownList` and the `RadioButtonList` are displayed simultaneously. Because of its forward-only nature, you can bind only one Web list control to a DataReader at a time, which is why you had only one Web list control in the DataReader examples. In contrast, you can reuse the contents of a `DataTable` as many times as you like, so you can use it as a data source to as many Web controls as you like.

Beyond this fairly large functional difference, however, the code is more or less the same. You still identify a `DataTextField` and a `DataValueField` for each Web list control, and they work as usual, as does the `DataSource` property. But there's a slight variation here that applies when using a `DataSet` as the data source. For either type of binding to work, you must identify the

specific DataTable for the Web control to bind to. In the example, you've done this by specifying the DataMember property in code:

```
GridView1.DataSource = myDataSet;
GridView1.DataMember = "Player";
```

Alternatively, you could just set the DataSource to the DataTable directly:

```
GridView1.DataSource = myDataSet.Tables["Player"];
```

The event handlers are all straightforward. Because each handler is associated with a specific Web list control, you know which one to check for the newly selected Manufacturer. With that, you can derive the correct SQL query and populate the waiting GridView from the DataTable storing the results of the query.

Try It Out: Using Lookup Lists and Events with a SqlDataSource

In this example, you'll perform list binding using the SqlDataSource to provide the data. You'll see that if we use the SqlDataSource, rather than a DataReader or a DataSet, you can build quite complex pages with very little code.

1. Open List_DataReader_Events.aspx and save it as List_DataSource_Events.aspx.
2. Change the <title> of the page to **List Binding with Events to a SqlDataSource**.
3. If the page is using a Web list control other than the DropDownList, change it back to a DropDownList and, if necessary, add the DataBound event that you used in the previous example.
4. Remove all of the code from the page other than the lstManufacturers_DataBound event handler.
5. Switch to the Design view and add a SqlDataSource to the page. Select Configure Data Source from the Tasks menu.
6. Select SqlConnectionString from the drop-down list on the Choose Your Data Connection step, and then click the Next button.
7. On the Configure the Select Statement step, select Manufacturer from the Name drop-down list and check the ManufacturerID and ManufacturerName columns, as shown in Figure 6-15. Click the Next button.
8. If you want to test that the results returned are as expected, click the Test Query button. If they're not what you expect, click the Previous button to return to the previous step. Once you're happy that the results are as expected, click the Finish button.
9. Select the DropDownList and set its DataSourceID property to SqlDataSource1. Switch to the events list for the control and remove the SelectedIndexChanged event handler.
10. Add another SqlDataSource to the page, select to configure the data source, and select SqlConnectionString as the connection string to use. Click the Next button.

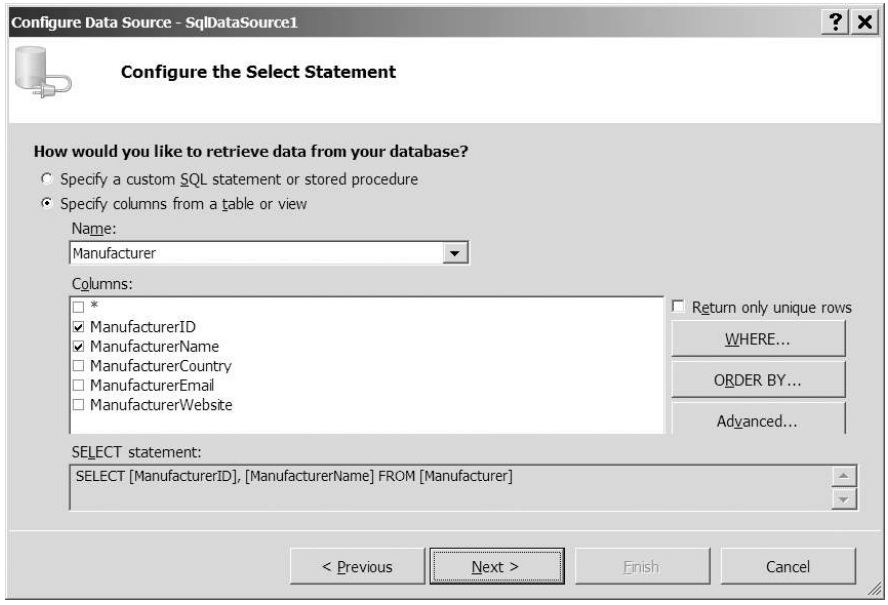

Figure 6-15. *Configuring the SELECT query for a SqlDataSource*

11. On the Configure the Select Statement step, select Player from the Name drop-down list and the * entry in the columns list. Click Next, and then click Finish to complete this part of the configuration.

12. In the Properties window for `SqlDataSource2`, click the ellipsis next to the `SelectQuery` property to open the Command and Parameter Editor dialog box.

13. Click the AddParameter button and give the new parameter a name of `ManufacturerID`. Select Control as the Parameter Source, and from the ControlID drop-down list, select lstManufacturers.

14. Click the Query Builder button to launch the Query Builder dialog box. As shown in Figure 6-16, add `@ManufacturerID` as the filter for the PlayerManufacturerID column. You can click the Execute Query option to test that the query is correct.

15. Click the OK button to close the Query Builder, and then click OK again to close the Command and Parameter Editor dialog box.

16. Select the `GridView` and change its `DataSourceID` property to `SqlDataSource2`.

17. Save the page, and then run it in a browser. Select one of the Manufacturers from the `DropDownList`. The page will post back to the server, and the list of Players will be presented based on the Manufacturer selected. Clicking the "please select..." entry will cause the `GridView` to be cleared.

18. Experiment with changing the `DropDownList` to a `RadioButtonList` and a `ListBox`. Notice again that all three Web controls will have the "please select..." entry, as well as the list of Manufacturers, and selecting an option displays the same results, regardless of the type of Web list control that you're using.

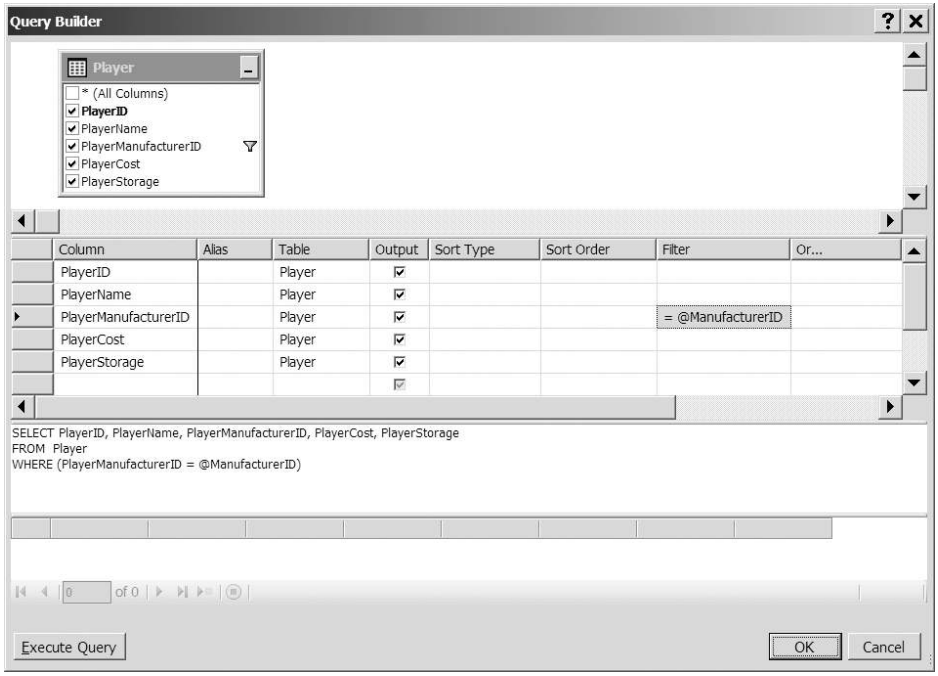

Figure 6-16. *Using the Query Builder to filter the list of Manufacturers*

How It Works

After having to write code to populate the list and grid, you've now come to an example that requires very little code in order to offer the same functionality. Instead, you've used the graphical tools provided with Visual Web Developer to build the page.

First, we'll look at the `DropDownList` and the first `SqlDataSource` that you added to the page. Then we'll look at the `GridView` and its `SqlDataSource`, as well as the parameters that it requires. Finally, we'll consider some of the limitations of the `SqlDataSource`.

The DropDownList Control

If you look at the HTML markup for `SqlDataSource1`, you'll see that apart from the `ID` and `runat` properties that all Web controls have, the `SqlDataSource` has two other properties that enable you to configure the Web control for your purposes: `ConnectionString` and `SelectCommand`.

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
  ConnectionString="<%$ ConnectionStrings:SqlConnectionString %>"
  SelectCommand="SELECT ManufacturerID, ManufacturerName FROM Manufacturer">
</asp:SqlDataSource>
```

The `ConnectionString` property sets the connection string that the `SqlDataSource` will use. This can be any valid connection string. In this case, you're using a new construct added in ASP.NET 2.0 that allows you to automatically retrieve the connection string from the

<connectionStrings> section of Web.config by specifying the name of the connection string—in this case, SqlConnectionString:

```
<%$ ConnectionStrings:SqlConnectionString %>
```

The SelectCommand property is the query that the SqlDataSource will use to query the database to return the results. In this case, you return a list with the ManufacturerID and ManufacturerName entries for all of the Manufacturers in the database.

■ **Note** SelectCommand hints that the SqlDataSource can be used for a whole lot more than just data binding. As you might guess, it also has corresponding DeleteCommand, InsertCommand, and UpdateCommand properties, which allow you to do much more. Now that I've tempted you with the possibilities, I'm going to make you wait until Chapter 9 before you get to see these in action.

That completes the configuration of SqlDataSource1. All that's left is to tell the drop-down list about its data source. You do this by setting the DataSourceID property of DropDownList1 to SqlDataSource1.

With the drop-down list, you're again making use of the DataBound event to add the dummy entry to the options available. The code is identical to the code you've used in the previous two examples.

Once you've configured the DropDownList and the SqlDataSource that it's binding to, you don't need to worry about the data binding, as it takes place automatically between the OnPreRender and OnPreRenderComplete events in the page life cycle. And it's here that ASP.NET applies some intelligence to the data binding.

In the previous examples, the first decision you had to make within the Page_Load event was whether you needed to bind the DropDownList. You'll recall that you need to do this only once, as the contents of the list are remembered across postbacks. The DropDownList is intelligent enough to know when it needs to bind to the data source and does this only the first time the page is loaded. You can see this by adding a breakpoint to the DataBound event and watch how many times it's reached—only once when the page is first loaded.

The GridView Control

In the previous examples, you've had to respond to the OnSelectedIndexChanged event to pass a SQL query to the GridView and tell it to bind itself. We'll look at the GridView in a lot more detail in the next chapter, when we cover table binding. Here, we'll look at how the SQL query that is executed is constructed.

If you take a look at the data source for the GridView, SqlDataSource2, you'll see that it's remarkably similar to the data source that you've used for the DropDownList:

```
<asp:SqlDataSource ID="SqlDataSource2" runat="server"
  ConnectionString="<%$ ConnectionStrings:SqlConnectionString %>"
  SelectCommand="SELECT PlayerID, PlayerName, PlayerManufacturerID,
    PlayerStorePrice, PlayerStorage FROM dbo.Player
    WHERE (PlayerManufacturerID = @ManufacturerID)">
```

```
  <SelectParameters>
    <asp:ControlParameter ControlID="lstManufacturers"
      Name="ManufacturerID" PropertyName="SelectedValue" />
    </SelectParameters>
</asp:SqlDataSource>
```

There's a `ConnectionString` that points to the correct database, and again, you have a `SelectCommand` that details the query that you want to execute. There's also a new child element, `SelectParameters`, which you haven't seen before.

The `SelectParameters` collection allows you to specify various parameters to the `SelectCommand` query, as shown in Table 6-2.

Table 6-2. *SelectCommand Parameter Types*

Name	Description
`ControlParameter`	Takes its value from another Web control on the same page. The Web control to use is specified by the `ControlID` attribute. The property on the Web control to retrieve the value from is indicated by the `PropertyName` attribute.
`CookieParameter`	Uses a value specified in a cookie, indicated by the `CookieName` attribute, to set the value of the parameter.
`FormParameter`	Uses a value specified in a form variable, indicated by the `FormName` attribute, to set the value of the parameter.
`QueryStringParameter`	Uses a value in the page's query string, indicated by the `QueryStringField` attribute, to set the value of the parameter.
`SessionParameter`	Uses a value in the user's session, indicated by the `SessionField` attribute, to set the value of the parameter.
`System.Data.SqlServerCe`	Provides native access to SQL Server CE for the .NET Compact Framework

By setting the parameters in the `SelectParameters` collection, you can build queries that are automatically modified depending on form values, query string values, values stored in cookies or the session, or, as in this case, the value of another Web control on the page:

```
<asp:ControlParameter ControlID="lstManufacturers"
  Name="ManufacturerID" PropertyName="SelectedValue" />
</SelectParameters>
```

The `ControlID` and `PropertyName` properties of the `ControlParameter` tell you which Web control and which property to use. In this case, you have `lstManufacturers.SelectedValue`, which returns the ManufacturerID of the selected Manufacturer. The other property that you need is `Name`, which is the property, common to all the different parameter types, that allows you to tie the parameter to the query.

If you take a look at the `SelectCommand` property, you'll see that the query actually has a `WHERE` clause added that takes a parameter:

```
WHERE (PlayerManufacturerID = @ManufacturerID)
```

The parameter you need to provide is `@ManufacturerID`, and the `ControlParameter` object has a name of `ManufacturerID`. The `@` is added automatically, and whatever value is selected in the `DropDownList` is used to modify the query to return the correct list of Players.

■ **Note** `DeleteCommand`, `InsertCommand`, and `UpdateCommand` properties also have parameter collections. We'll look at `DeleteParameters`, `InsertParameters`, and `UpdateParameters` in Chapter 9.

The SqlDataSource: Panacea?

You may now be thinking that the `SqlDataSource` is the perfect data source and gets rid of all of that horrible code that no one really likes writing. Well, to a certain extent it is. However, as you'll see shortly when we look at lists that allow multiple selections, there are times when the `SqlDataSource` doesn't quite make the grade.

You also have a little problem here that you didn't see when you used code to connect to the database. Although this page appears to have the same functionality, it doesn't. What happens when the user selects the dummy entry? In the DataReader and `DataSet` examples, you wrote code to clear the `GridView` when the user selects that entry. You're not manually data binding when you use the `SqlDataSource`, so you have no way, as it stands, of clearing the list instead of performing the query. When you select the dummy entry, a query is made to the database that returns no results:

```
SELECT PlayerID,PlayerName,PlayerManufacturerID,
  PlayerStorePrice,PlayerStorage
FROM dbo.Player
WHERE (PlayerManufacturerID = -1)
```

In the simple example, this isn't necessarily an issue, as the number of needless hits that you're going to make to the database is quite limited. But what happens when the page is being used by a hundred or a thousand users?

One solution is to stop the data binding from occurring if the user has selected an invalid entry. You can do this by responding to the `SelectedIndexChanged` event of the `DropDownList`:

```
protected void lstManufacturers_SelectedIndexChanged(object sender, EventArgs e)
{
  if (lstManufacturers.SelectedValue == "-1")
  {
    GridView1.DataSourceID = null;
  }
  else
  {
    GridView1.DataSourceID = "SqlDataSource2";
  }
}
```

This isn't the most elegant of solutions, but it does prevent needless hits on the database. Selecting the dummy value in the `DropDownList` sets the `DataSourceID` of the `GridView` to a null data source. This has the effect of turning off automatic data binding, as the `GridView` no longer has a data source to bind to. If the selected entry in the `DropDownList` isn't the dummy value, then you set the `DataSourceID` back to the correct `SqlDataSource`, and the data binding takes place as expected.

You also have the same problem when you first load the page, as the `GridView` has a valid data source even if the query that is executed will return no results. There are several ways that this can be remedied; I'll leave it up to you to experiment to find out what they are.

■ **Note** There's nothing stopping you from using the `SqlDataSource` and its properties within code. Indeed, any property that you can set within the HTML markup can be modified within code. You'll see this when you start changing the SQL query that is executed by directly modifying the `SelectCommand` property.

Connecting to Other Data Sources

In this example, you're connecting to a SQL Server database, and the definition of the `SqlDataSource` is complete, simply specifying the `ConnectionString` will connect to the database. However, this isn't the whole story. You're actually providing, by omission in this example, one further piece of information that the `SqlDataSource` needs.

The `SqlDataSource` can be used to connect to any data source, provided that there is a data provider for it, but as yet, you have not told the `SqlDataSource` what type of connection string you've provided. You need to do this using the `ProviderName` property. The `ProviderName` property can have several different values, depending on which data provider you want to use. The standard data providers are listed Table 6-3.

Table 6-3. *Standard Data Providers*

ProviderName	Description
`System.Data.Odbc`	Any data source that is accessed through an ODBC driver
`System.Data.OleDb`	Any data source that is accessed through an OLE DB provider
`System.Data.OracleClient`	Provides native access to Oracle databases
`System.Data.SqlClient`	Provides native access to SQL Server databases (default value)
`System.Data.SqlServerCe`	Provides native access to SQL Server CE for the .NET Compact Framework

As you can see, the default `ProviderName` is `System.Data.SqlClient`, so you don't need to specify it when you're connecting to a SQL Server database; the `SqlDataSource` assumes, unless you specify otherwise, that you're using a SQL Server database.

However, if you're not connecting to a SQL Server database, as in the MySQL and Access examples in the code download, you must specify that you want to use a different provider. You add the `ProviderName` attribute to the `SqlDataSource` definition, like this:

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
  ConnectionString="<%$ ConnectionStrings:OdbcConnectionString %>"
  SelectCommand="SELECT ManufacturerID, ManufacturerName FROM Manufacturer">
  ProviderName="<%$ ConnectionStrings:OdbcConnectionString.ProviderName %>"
</asp:SqlDataSource>
```

You can either specify the provider you want directly, as in `System.Data.Odbc`, or set it as part of the connection string in `Web.config`:

```
<add name="OdbcConnectionString"
  connectionString="Driver={MySQL ODBC 3.51 Driver};
    server=localhost;database=players;uid=band;pwd=letmein;"
  providerName="System.Data.Odbc" />
```

If you set it in `Web.config`, you can access the `providerName` property of the required connection string using this syntax:

```
<%$ ConnectionStrings:OdbcConnectionString.ProviderName %>
```

■ **Note** As a shortcut, you can specify the connection string, as we've done here, as just `OdbcConnectionString`. If you don't specify a specific part of the connection string, it's assumed that you mean `OdbcConnectionString.ConnectionString`.

Multiple Selection Lists

The Web list controls that you've looked at so far have allowed only a single item to be selected from the Web control. But what if you want the user to be able to select multiple entries from the same list?

The `ListBox` allows you to do this by changing its `SelectionMode` property from the default value of `Single` to `Multiple`. The one remaining Web list control that we haven't looked at yet, the `CheckBoxList`, also allows the selection of multiple values.

■ **Note** Now that you've seen how easy it is to change between a DataReader and a `DataSet`, you won't use both in the example presented here. `List_DataSet_Multiple.aspx` in the code download is the corresponding `DataSet` example.

Try It Out: Using Multiple-Value Lookup Lists with a DataReader

In this example, you'll see what alternative methods you can employ to deal with Web list controls that allow multiple selections. You can't rely on using the `SelectedItem` property, because that will return only the first item selected in the list. Instead, you must iterate through the list each time and build up a SQL query accordingly.

1. In Visual Web Developer, create a new Web Form called `List_DataReader_Multiple.aspx`.
2. Change the `<title>` of the page to **Multiple Selection Using a DataReader**.
3. Make sure you've included the correct `Import` statements at the top of the page.

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

4. In the Design view, add a `ListBox`, a `Button`, and a `GridView` to the page. Change the `Text` for the `Button` to **Select**. You can use a table to lay out the Web controls in a more user-friendly manner, as shown in Figure 6-17.

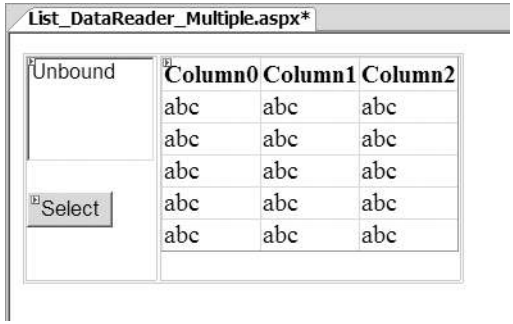

Figure 6-17. *Laying out the Web list controls*

5. For the `ListBox`, set `DataTextField` to ManufacturerName, `DataValueField` to ManufacturerID, and `SelectionMode` to Multiple.
6. The code for the `Page_Load` event is pretty much the same as in the previous DataReader example:

```
protected void Page_Load(object sender, EventArgs e)
{
  if (Page.IsPostBack == false)
  {
    // create SqlConnection object
    string ConnectionString = ConfigurationManager.
      ConnectionStrings["SqlConnectionString"].ConnectionString;
    SqlConnection myConnection = new SqlConnection(ConnectionString);

    // create SqlCommand object
    SqlCommand myCommand = new SqlCommand();
    myCommand.Connection = myConnection;
```

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

      // set up SQL query for Manufacturer table
      myCommand.CommandText =
        "SELECT ManufacturerID, ManufacturerName FROM Manufacturer";

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

      // set up the list control
      ListBox1.DataSource = myReader;
      ListBox1.DataBind();

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

7. Add a `Click` event handler for the `Button`. First, add the code to create the Command and Connection objects, and then open the connection:

```
protected void Button1_Click(object sender, EventArgs e)
{
  // create SqlConnection object
  string ConnectionString = ConfigurationManager.
    ConnectionStrings["SqlConnectionString"].ConnectionString;
  SqlConnection myConnection = new SqlConnection(ConnectionString);

  // create SqlCommand object
  SqlCommand myCommand = new SqlCommand();
  myCommand.Connection = myConnection;

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

8. To determine which items were selected in the Web list control, you must iterate through the Web control and build up the query from the results:

```
// set up SQL query for Player table
StringBuilder Query = new StringBuilder("SELECT PlayerID, PlayerName, ➥
  PlayerManufacturerID, PlayerCost, PlayerStorage FROM Player ➥
  WHERE PlayerManufacturerID IN (");

bool gotResult = false;

for (int i=0; i<ListBox1.Items.Count; i++)
{
  if (ListBox1.Items[i].Selected)
  {
    if (gotResult == true) Query.Append(",");
    Query.Append(ListBox1.Items[i].Value);
    gotResult = true;
  }
}

Query.Append(")");
```

9. Now you find out if any items were checked. If so, you run the query you built. If not, you clear the grid.

```
// get results if we have a selection
if (gotResult)
{
  // set the query to execute
  myCommand.CommandText = Query.ToString();

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

  // set up the GridView
  GridView1.DataSource = myReader;
  GridView1.DataBind();

  // close the reader
  myReader.Close();
}
else
{
  // clear the GridView
  GridView1.DataSource = null;
  GridView1.DataBind();
}
}
```

10. And again, matching the last example, you handle any errors and wrap up the code by closing the connection:

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

11. Save the page, and then run it in your browser. You'll see the `ListBox` containing the familiar list of Manufacturers. To select more than one item, hold down the Ctrl key. Clicking the Select button will cause the `GridView` to be populated with the Manufacturers you've selected, as shown in Figure 6-18.

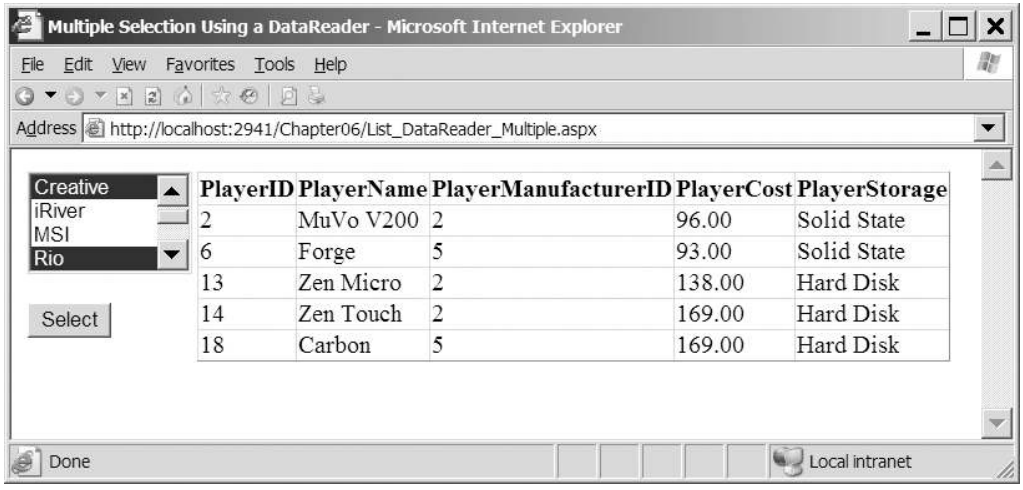

Figure 6-18. *Multiple selections from a ListBox*

How It Works

Much of the code is the same as the previous example, so we won't go over it here. The major change is to the code to generate a query for Player information for the `GridView`.

Rather than responding to the user's every selection in the list box, you've placed the code to populate the `GridView` in a button's `Click` event. This will allow the user to select several different Manufacturers before clicking the button to have the Players for the selected Manufacturers displayed.

The `SELECT` query uses the `IN` keyword to let you search for columns with one of a given set of values, so you use that to build your query. Your queries take the following shape:

```
SELECT PlayerID, PlayerName, PlayerManufacturerID, PlayerCost, PlayerStorage
FROM Player WHERE PlayerManufacturerID IN (1,2)
```

You'll need code that inserts the ID number for each selected Manufacturer into the query. You're using a `StringBuilder`, rather than normal string concatenation, as the `StringBuilder` is more efficient. You first add the start of the `SELECT` query you want to execute, like so:

```
// set up SQL query for Player table
StringBuilder Query = new StringBuilder("SELECT PlayerID, PlayerName, ➥
  PlayerManufacturerID, PlayerCost, PlayerStorage ➥
  FROM Player WHERE PlayerManufacturerID IN (";
```

The strategy here is to iterate through each item in the list and see if it has been selected. Fortunately, each `ListItem` object has a `Selected` property, which is true if it has been selected and false otherwise. If it has been selected, you pull its `Value` into the query.

```
bool gotResult = false;

for (int i=0; i<ListBox1.Items.Count; i++)
{
  if (ListBox1.Items[i].Selected)
  {
    if (gotResult == true) Query.Append(",");
    Query.Append(ListBox1.Items[i].Value);
    gotResult = true;
  }
}
```

You've included the Boolean `gotResult` to keep track of whether anything has been selected. You need to comma separate the selected entries, so you use `gotResult` to decide if you need to add a comma to the end of the query.

The `gotResult` variable is also used to decide whether you need to populate the `GridView`. If the user has selected at least one entry in the `ListBox`, you set the `CommandText` property of the Command object and execute the query to populate the `GridView`. If the user has unselected all the items in the list, you clear the `GridView`. This will ensure that the list of Players doesn't appear when you don't have a Manufacturer selected:

```
if (gotResult)
{
  myCommand.CommandText = Query.ToString();

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

  // set up the GridView
  GridView1.DataSource = myReader;
  GridView1.DataBind();

  // close the reader
  myReader.Close();
}
```

```
else
{
  // clear the GridView
  GridView1.DataSource = null;
  GridView1.DataBind();
}
```

So nothing much has changed really. You're modifying the query that you want to execute to handle multiple selected values, but the rest of the code remains the same. The same is true if you look at the `DataSet` version of the page in the code download.

However, all is not the same when you want to use a `SqlDataSource`.

Try It Out: Using Multiple-Value Lookup Lists with a SqlDataSource

In this example, you'll see how you need to write code when you want to control how the `SqlDataSource` behaves.

1. In Visual Web Developer, create a new Web Form called `List_DataSource_Multiple.aspx`.
2. Change the `<title>` of the page to **Multiple Selection Using a SqlDataSource**.
3. Make sure you've included the correct `Import` statement at the top of the page.

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

4. In the Design view, add a `ListBox`, a `Button`, and a `GridView` to the page. Change the `Text` for the `Button` to **Select**. You can use a table to lay out the Web controls in a more user-friendly manner, as shown earlier in Figure 6-17.
5. For the `ListBox`, set `DataTextField` to ManufacturerName, `DataValueField` to ManufacturerID, and `SelectionMode` to Multiple.
6. You need two `SqlDataSource` controls for this example, but rather than using the graphical tools to create them, you can add them directly to the HTML markup. Switch to the Source view and add the definitions for the two `SqlDataSource` controls at the top of the page:

   ```
   <asp:SqlDataSource ID="SqlDataSource1" runat="server"
     ConnectionString="<%$ ConnectionStrings:SqlConnectionString %>"
     SelectCommand="SELECT ManufacturerID, ManufacturerName FROM Manufacturer">
   </asp:SqlDataSource>
   <asp:SqlDataSource ID="SqlDataSource2" runat="server"
     ConnectionString="<%$ ConnectionStrings:SqlConnectionString %>">
   </asp:SqlDataSource>
   ```

7. Switch to the Design view and set the `DataSourceID` for the `ListBox` to `SqlDataSource1` and for the `GridView` to `SqlDataSource2`.

8. Add a `Click` event handler for the `Button`. Then add the following code:

```
protected void Button1_Click(object sender, EventArgs e)
{
  // set up SQL query for Player table
  StringBuilder Query = new StringBuilder("SELECT PlayerID, PlayerName, ➥
    PlayerManufacturerID, PlayerCost, PlayerStorage FROM Player ➥
    WHERE PlayerManufacturerID IN (");

  bool gotResult = false;

  for (int i = 0; i < ListBox1.Items.Count; i++)
  {
    if (ListBox1.Items[i].Selected)
    {
      if (gotResult == true) Query.Append(",");
      Query.Append(ListBox1.Items[i].Value);
      gotResult = true;
    }
  }

  Query.Append(")");

  if (gotResult)
  {
    // set the correct SelectCommand
    SqlDataSource2.SelectCommand = Query.ToString();
  }
  else
  {
    // clear the GridView
    SqlDataSource2.SelectCommand = null;
  }
}
```

9. Save the page, and then run it in your browser. You'll see the `ListBox` containing the list of Manufacturers. To select more than one item, hold down the Ctrl key. Clicking the Select button will cause the `GridView` to be populated with the Manufacturers you've selected, as shown earlier in Figure 6-18.

How It Works

Here's one good example of where the `SqlDataSource` is not the panacea you might have initially thought it was.

The `SqlDataSource` is ideal when the query that you're executing is simple—or more correctly, any parameters that you want to add to it are simple. It works well when all you need to do is find if `x = 1` or `y < 3`, and so on. But it falls down when you need to use conditional statements

such as IN, and the parameter can't just be added to the query that you want to execute. In these cases, you need to massage the query and change the SelectCommand for the SqlDataSource.

The first change that you'll notice is that the second SqlDataSource on the page has a ConnectionString, but it doesn't have a SelectCommand:

```
<asp:SqlDataSource ID="SqlDataSource2" runat="server"
  ConnectionString="<%$ ConnectionStrings:SqlConnectionString %>">
</asp:SqlDataSource>
```

This is perfectly valid, and all it means is that no data binding will take place when a Web control uses this as its data source—no SelectCommand, no automatic data binding. So the page when it first loads won't connect to the database to try to show a list of Players in the GridView.

In order for the automatic data binding to occur, you must add a SelectCommand. And this is exactly what you do in the Click event handler for the button on the page.

The query is constructed in exactly the same way as you saw in the DataReader example. If the user has made a selection from the list, you have a query that you want to run, and you set the SelectCommand property:

```
SqlDataSource2.SelectCommand = Query.ToString();
```

Once you've set the SelectCommand to the query, you can let the automatic data binding occur. As you'll recall, this happens after the OnPreRender event, so as long as you've set the SelectCommand before, you'll be able to rely on the automatic data binding.

If no selection has been made, you don't want to run a query. So, you remove the SelectCommand by setting it to null:

```
SqlDataSource2.SelectCommand = null;
```

As there's no query to execute, the automatic data binding won't occur, thus clearing the GridView.

Summary

You don't have to surf far on the Web to find parallels between the examples you've seen in this chapter and, for example, the pages of an e-commerce or a business Web site where individual pieces of information are placed all around the page, as well as in an orderly list or grid. Dealing with read-only data is a big subject, and you've learned just the basics in this chapter.

In this chapter, we've looked at several ways to handle data binding:

- You learned how to inline-bind a piece of information from the current row in the DataReader or from a DataSet to the property or the text value of a Web control on the page.
- You learned about an alternative to inline binding that involves directly accessing the data source to return the required information directly.
- You learned how to list-bind Web controls such as the DropDownList, RadioButtonList, and ListBox. This requires you to nominate a column for the text of each list item and another column to act as the value for each list item.

- You saw that the DataReader, `DataSet`, and `SqlDataSource` can be used as the data source for list binding.
- The `SqlDataSource`, though a great Web control to have in our toolbox, isn't the panacea that you may have thought it was. You saw various examples of when you need to massage it a little to get it to perform exactly as desired.

In the next chapter, we'll continue our exploration of data binding by looking at table binding. You'll learn how to show the list of results in pretty much any format you require using the `Repeater`, `DataList`, and `GridView`.

You'll also start to see the power that the `GridView` provides, as well as how you can implement pages that, in the past, would have required a lot of hard work to develop. Because the `GridView` can perform a lot of complex functions automatically, hard-working developers now have time for the finer things in life.