CHAPTER 7

Table Binding

In the previous chapter, we focused on two of the three types of data binding. You saw that inline binding allows you to bind individual columns from the database to specific properties of Web controls on the page. I also explained why I prefer to avoid inline binding, and described another solution: using the DataReader or `DataSet` directly in code, rather than requiring a `Page.DataBind()` call. We then looked at list binding, which allows you to bind a series of name/value pairs into a Web list control, such as the `DropDownList` or the `RadioButtonList`.

This chapter covers the last of the three binding varieties in ASP.NET. For simplicity's sake, I'll call it *table binding*. With table binding, you can bind any number of columns per row from a data source to a Web control that supports it.

You've already looked at table binding, although it wasn't referred to as such. In the examples in Chapters 3 and 4, you used the `GridView` to display the list of Players from the database. In Chapter 3, you were introduced to data-aware pages by using a `SqlDataSource` to display the list of Players from the database. In Chapter 4, you saw that you can accomplish the same result in code using a DataReader.

Chapter 5 introduced you to the `DataSet`, and you saw that it is possible to create a disconnected representation of the results. In Chapter 6, you saw that the `DataSet` can be used as the data source for inline and list binding, and it will come as no surprise that you can also use the `DataSet` as the data source for table binding.

This chapter covers the following topics:

- The six table-binding Web controls
- How to use the `Repeater`, which allows you complete control over the output
- The differences between inline binding and event-based binding
- An introduction to the `DataList`, which is halfway between the `Repeater` and the `GridView`
- How to table-bind to the `GridView` and add sorting and paging to the displayed results
- A comparison of the DataReader and the `DataSet`

The Table-Binding Web Controls

Six Web controls support table binding in ASP.NET 2.0: the `GridView`, `DetailsView`, `FormView`, `Repeater`, `DataList`, and `DataGrid`. These are summarized in Table 7-1. If you've built Web sites using ASP.NET 1.1, you'll recognize the `DataGrid`, `DataList`, and `Repeater`, as their ASP.NET 2.0

versions are almost identical. They've been updated to support the use of a SqlDataSource as their data source, but fundamentally nothing has changed.

Table 7-1. *Web Controls That Support Table Binding*

Control	Description
Repeater	This is the simplest of the Web controls that supports table binding. The Repeater supports the "display data and nothing else" model.
DataList	This Web control is a step up from the Repeater. It outputs its content in a table and allows inline editing of the content that it is displaying. By defining different templates, you can control how the output is rendered when viewing data and how this changes when an item is selected or being edited.
DataGrid	This is the predecessor of the GridView. Much like the GridView, the DataGrid displays its output in a tabular format, with one row in the HTML table corresponding to one row of data from the data source. It also allows inline editing of the data that it is displaying, but this tends to involve writing quite a lot of code for all but the simplest tasks.
GridView	This renders as a tabular list of the data that it's bound to, with each row in the HTML table representing a row from the database.
DetailsView	This is used to output a single row of data from a data source in a tabular format. Unlike the GridView, each row in the HTML table represents a column from the selected row. The DetailsView supports inline editing and makes an ideal partner for the GridView, allowing you to build quite powerful master/detail pages with very little code.
FormView	This is similar to the DetailsView in that it displays a single row from the data source. But whereas the DetailsView is output as an HTML table, the FormView has no constraints on its output, and you're free to display the data however you wish. The FormView also supports inline editing, and, like the DetailsView, makes an ideal partner for the GridView when building master/detail pages.

In this chapter, we'll look at three of the table-binding Web controls: the Repeater, the DataList, and the GridView. We'll defer our discussion of the DetailsView and FormView until Chapter 9. We're not going to spend any time looking at the DataGrid. Although the DataGrid is present in ASP.NET 2.0, it doesn't appear in the Toolbox. The idea is to use the GridView instead for all tabulated data.

■**Note** You can find a very good resource for all of the different table-binding Web controls on MSDN at http://msdn.microsoft.com/en-us/library/a63e36w2.aspx. This includes a discussion of the different Web controls and links to further information.

You've already seen how to associate and bind data to a GridView, and the process for binding to a Repeater or DataList is exactly the same. At its simplest, the whole operation needs only the following four lines of code:

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

If you're using a `SqlDataSource`, you need to declare the `DataSourceID` in the markup for the Web control. So, to bind a `GridView` to a data source called `SqlDataSource1`, you would need the following as a minimum:

```
<asp:GridView ID="GridView1" runat="server" DataSourceID="SqlDataSource1">
</asp:GridView>
```

The same process is repeated for all of the table-bound Web controls. However, as you can see from `TableBinding.aspx` (which you'll find in the code download for this book) shown in Figure 7-1, there is more to these three Web controls than simply binding the data source to the Web control.

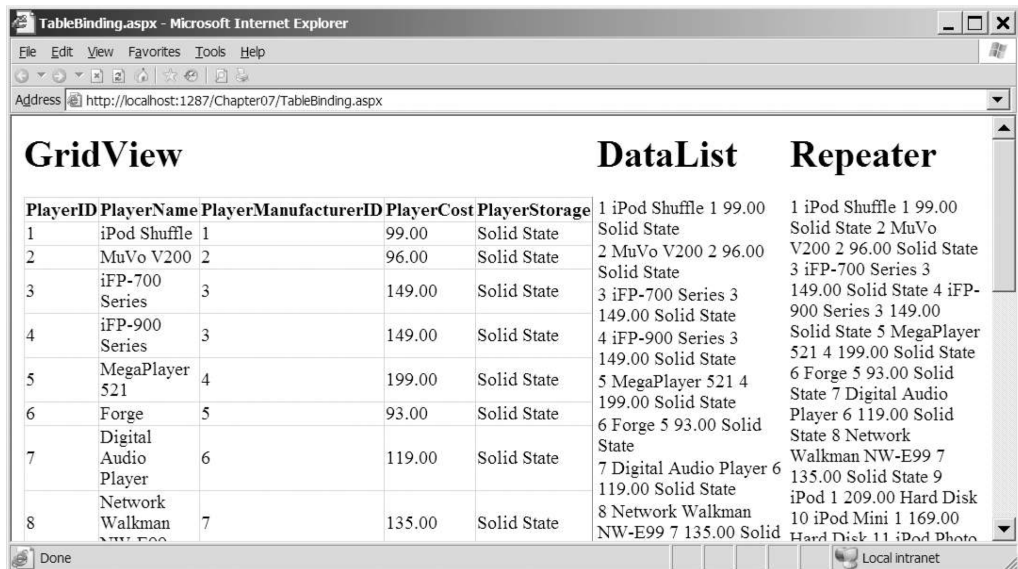


Figure 7-1. *GridView, DataList, and Repeater Web controls*

Repeater, DataList, and GridView Differences

The page shown in Figure 7-1 demonstrates binding and displaying the entire Player table. The `GridView` takes care of itself; it already understands what rows and columns are and that each column in the row should be placed in its own table cell. In contrast, a `DataList` only knows what a row is. When you bind a database table to it, a `DataList` still works row by row, but it doesn't create individual table cells for each column in a row. A `Repeater` knows even less, and simply spouts out everything bound to it in one paragraph, distinguishing neither column nor row.

Whereas the `GridView` understands rows and columns, the `DataList` and `Repeater` both rely on templates that are used to display each row in the data source. You need to define these templates. This may sound complex, but it isn't really, as you'll see in the next section.

The following summarizes the differences between the three Web controls:

- A Repeater is completely dependent on the template you provide for displaying data. For example, unless you include line breaks in the template, it will display incoming data on one line (as in Figure 7-1). You must do any customization through the templates. The Repeater supports templates for rows, alternate rows, headers, footers, and row separators. However, the data is read-only and can't readily support paging or sorting.

- A DataList displays your data, one item per row, according to a template you must provide. It won't separate individual pieces of information unless told to do so in the template. You can customize the display of the DataList extensively through various properties to match up with the templates you define. The DataList supports the editing, adding, and deleting of data but not sorting or paging. It supports templates to define the header of the list, to define the footer of the list, to highlight alternating rows in the list, and to highlight the row either currently selected or currently being edited.

- A GridView displays your data as a neatly formed grid by default. You can customize the GridView extensively through various properties, as you'll see shortly. You can even switch from the automatic generation of columns in favor of the template approach used by the DataList and Repeater.

Item Templates

Templates are defined using <ItemTemplate>. The DataList in the previous example has the following <ItemTemplate> defined:

```
<asp:DataList ID="DataList1" runat="server" DataSourceID="SqlDataSource1">
  <ItemTemplate>
    <asp:Label ID="PlayerIDLabel" runat="server"
      Text='<%# DataBinder.Eval(Container, "DataItem.PlayerID") %>'>
    </asp:Label>
    <asp:Label ID="PlayerNameLabel" runat="server"
      Text='<%# Eval("PlayerName") %>'>
    </asp:Label>
    <asp:Label ID="PlayerManufacturerIDLabel" runat="server"
      Text='<%# Eval("PlayerManufacturerID") %>'>
    </asp:Label>
    <asp:Label ID="PlayerCostLabel" runat="server"
      Text='<%# Eval("PlayerCost") %>'>
    </asp:Label>
    <asp:Label ID="PlayerStorageLabel" runat="server"
      Text='<%# Eval("PlayerStorage") %>'>
    </asp:Label>
  </ItemTemplate>
</asp:DataList>
```

If you look at the source for the page, you'll see that the Repeater has exactly the same <ItemTemplate> defined, yet it displays the data differently on the page. As the DataList understands rows, it shows each item template on its own row (we'll look at how it does this shortly).

However, the Repeater doesn't really understand anything, so it simply puts each item template one directly after another, and you can't see where one row ends and the next one begins.

You can define several kinds of row templates for the DataList and Repeater, and we'll get to them in a minute. The basic one is the ItemTemplate, as shown here. You can think of each template as a mini-page where you can place any combination of text, HTML, Web controls, and, most important, instructions on how and where to bind the columns in the data source in that row. When the call to DataBind() is made (either directly in code or automatically as part of the page life cycle), the mini-page template is repeated for each row in the data.

The ItemTemplate in the example is simple. It declares that each row will contain five Label controls, one for each column in the Player table. To associate a column in the data with the Label, you inline-bind it with the following binding expression:

```
<%# DataBinder.Eval(Container, "DataItem.PlayerID") %>
```

You saw the static DataBinder.Eval method when we looked at inline binding in the previous chapter. The first parameter locates the data source, and the second locates the column that will be bound. Container in this case means that the data source is the container for this template, or, rather, the DataList or Repeater. You can use this nomenclature in any template. As you may have noticed, a row corresponds to an item here, and you can refer to the column you want by name through the DataItem object.

When using table binding, you also have a slightly different way of inline binding, as shown in the sample ItemTemplate:

```
<%# Eval("PlayerName") %>
```

This is a shorthand way of saying the following:

```
<%# DataBinder.Eval(Container, "DataItem.PlayerName") %>
```

These two inline-binding statements perform exactly the same task, but one of them involves a whole lot less typing! Feel free to use whichever method you prefer. The only word of warning is that you can use the shorthand version only within a Web control that supports table binding. If you're inline binding to Web controls that exist directly on a page, you need to use the longhand version.

Now, we'll take a closer look at using the Repeater. It provides an ideal introduction to using templates without any of the added complications of the other table-binding Web controls.

The Repeater Web Control

As you saw in Figure 7-1, when using the Repeater to display data, you're completely responsible for the layout of that data. The Repeater is the most flexible of the table-binding Web controls, and it allows you to show data in whatever format you desire. However, the flexibility of the Repeater comes at a price—without specifying, as a minimum, an ItemTemplate, the Repeater will not display anything.

We'll first look at the different templates that can be defined for the Repeater, and then you'll see how to use inline binding to extract the data that you want to display. Next, we'll look at the events that the Repeater exposes and when they might be useful.

The Repeater Templates

As explained in the previous section, you must supply at least an ItemTemplate in order for the data that you're trying to show to be visible. As shown in Table 7-2, this isn't the only template that the Repeater supports.

Table 7-2. *Templates Supported by the Repeater*

Name	Description
AlternatingItemTemplate	The AlternatingItemTemplate, if specified, is output alternatively with the ItemTemplate. It can be used to change the formatting that is applied to the row (such as changing the background color of alternating rows) to provide a visible division between the different data items.
FooterTemplate	The HTML content of the FooterTemplate is output once all of the data presented to the Repeater has been displayed.
HeaderTemplate	The HTML content of the HeaderTemplate is output before the Repeater outputs any data via the ItemTemplate and AlternatingItemTemplate.
ItemTemplate	The ItemTemplate is used to display the data returned from the data source in the format that you specify. As a minimum, this template must be specified in order for the Repeater to output the returned data.
SeparatorTemplate	The SeparatorTemplate is output between instances of the ItemTemplate and AlternatingItemTemplate. It allows you to add any HTML that you wish between the data that the Repeater generates. If only the ItemTemplate is specified, the Repeater will output ItemTemplate, SeparatorTemplate, ItemTemplate. If both an ItemTemplate and an AlternatingItemTemplate are specified, the output will be ItemTemplate, SeparatorTemplate, AlternatingItemTemplate.

Each of the templates can contain any combination of HTML and Web controls, but only the ItemTemplate and AlternatingItemTemplate can use data binding to actually output the data that is given to the Repeater. You use the FooterTemplate, HeaderTemplate, and SeparatorTemplate to present the data in the format that you require.

Try It Out: Using the Repeater to Display the Manufacturers

In this first example, you'll use a Repeater to output the list of Manufacturers in the database. You'll use a DataReader to query the database for the Manufacturers and see how to use the different templates to present the data in a usable format.

1. In Visual Web Developer, create a new Web site at C:\BAND\Chapter07 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=Players;
    Persist Security Info=True;User ID=band;Password=letmein"
  providerName="System.Data.SqlClient" />
```

3. Add a new Web Form to the Web site called `Repeater_DataReader.aspx`. Make sure that the Place Code in Separate File check box is unchecked and Language is set to Visual C#.

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

5. In the Design view, add a `Repeater` to the page. Switch back to the Source view and make sure you've included the correct data provider at the top of the page.

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

6. You need to set up the DataReader to query for the Manufacturers and add your standard code for database access.

```
protected void Page_Load(object sender, EventArgs e)
{
  if (Page.IsPostBack == false)
  {
    // 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);

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

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

      // set the data source and bind
      Repeater1.DataSource = myReader;
      Repeater1.DataBind();

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

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

7. Save the page, and then view it in a browser. As you didn't define any templates, the `Repeater` won't output anything to the page.

8. Switch back to Visual Web Developer and, in the Source view, add the `HeaderTemplate` inside the `<asp:Repeater>` and `</asp:Repeater>` tags:

```
<asp:Repeater ID="Repeater1" runat="server">
  ...
  <HeaderTemplate>
    <div style="background-color:Bisque">
      <font size="+2">Manufacturers</font>
    </div>
    <hr style="color:blue"/>
  </HeaderTemplate>
  ...
</asp:Repeater>
```

9. Add the `ItemTemplate` to the `Repeater` as follows:

```
<asp:Repeater ID="Repeater1" runat="server">
  ...
  <ItemTemplate>
    <div style="background-color:Ivory">
      <b>
      <asp:Label ID="lblName" runat="server"
        Text='<%# Eval("[ManufacturerName]") %>'>
      </asp:Label>
      </b>
      <br />
      Country:
      <asp:Label ID="lblCountry" runat="server"
        Text='<%# Eval("[ManufacturerCountry]") %>'>
      </asp:Label>
      <br />
      Email:
      <asp:HyperLink ID="lnkEmail" runat="server"
        NavigateUrl='<%# Eval("[ManufacturerEmail]", "mailto:{0}") %>'
        Text='<%# Eval("[ManufacturerEmail]") %>'>
      </asp:HyperLink>
      <br />
```

```
      Website:
      <asp:HyperLink ID="lnkWebsite" runat="server"
        NavigateUrl='<%# Eval("[ManufacturerWebsite]") %>'>
        <%# Eval("[ManufacturerWebsite]") %>
      </asp:HyperLink>
    </div>
  </ItemTemplate>
  ...
</asp:Repeater>
```

10. Add the SeparatorTemplate to the Repeater as follows:

```
<asp:Repeater ID="Repeater1" runat="server">
  ...
  <SeparatorTemplate>
    <hr style="color:blue"/>
  </SeparatorTemplate>
  ...
</asp:Repeater>
```

11. Add the AlternatingItemTemplate to the Repeater as follows:

```
<asp:Repeater ID="Repeater1" runat="server">
  ...
  <AlternatingItemTemplate>
    <div style="background-color:Azure">
      <b>
      <asp:Label ID="lblName" runat="server"
        Text='<%# Eval("[ManufacturerName]") %>'>
      </asp:Label>
      </b>
      <br />
      Country:
      <asp:Label ID="lblCountry" runat="server"
        Text='<%# Eval("[ManufacturerCountry]") %>'>
      </asp:Label>
      <br />
      Email:
      <asp:HyperLink ID="lnkEmail" runat="server"
        NavigateUrl='<%# Eval("[ManufacturerEmail]", "mailto:{0}") %>'
        Text='<%# Eval("[ManufacturerEmail]") %>'>
      </asp:HyperLink>
      <br />
      Website:
      <asp:HyperLink ID="lnkWebsite" runat="server"
        NavigateUrl='<%# Eval("[ManufacturerWebsite]") %>'>
        <%# Eval("[ManufacturerWebsite]") %>
      </asp:HyperLink>
    </div>
```

```
  </AlternatingItemTemplate>
  ...
</asp:Repeater>
```

12. Finally, add the `FooterTemplate` to the `Repeater`:

```
<asp:Repeater ID="Repeater1" runat="server">
  ...
  <FooterTemplate>
    <hr style="color:blue"/>
    <div style="background-color:Bisque">
      <br />
    </div>
  </FooterTemplate>
  ...
</asp:Repeater>
```

13. Save the page, and then view it in a browser. When the page loads, you'll see that the list of Manufacturers is presented in a rather unattractive color scheme. Figure 7-2 shows the output.

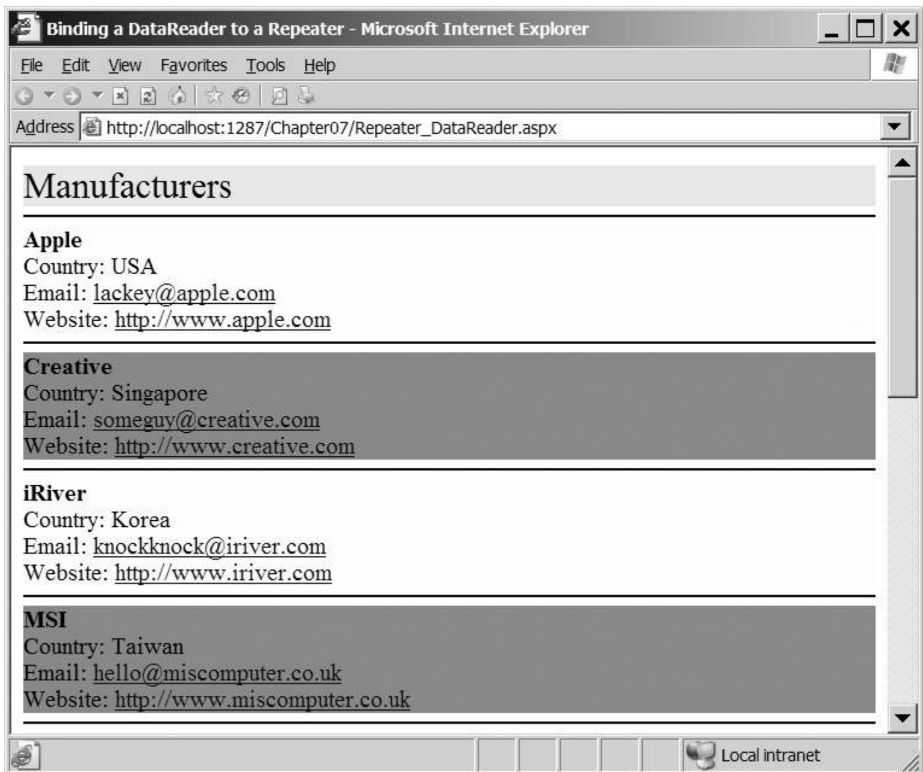


Figure 7-2. *Showing the list of Manufacturers in a Repeater*

How It Works

In this example, you used a DataReader to return the list of Manufacturers from the database and then used this as the data source for the `Repeater`. You could have used a `DataSet` to return the list of Manufacturers, and, as you've learned in the previous chapter, very little of the page would change. (`Repeater_DataSet.aspx` in the code download contains this page using a `DataSet` rather than a DataReader.)

Whether you're using a DataReader or a `DataSet`, the code that you use to query the database is the same as you've used in the previous examples. You set up the connection to the database, create the command, open the connection, and close the connection in the same way as you've already seen.

Only three lines of code are a little different:

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

// set the data source and bind
Repeater1.DataSource = myReader;
Repeater1.DataBind();
```

You query the database to return the list of Manufacturers using the `ExecuteReader()` method of the Command object. `ExecuteReader()` returns a DataReader (in this case, a `SqlDataReader`), which you store in the `myReader` variable and then use as the `DataSource` of the `Repeater`.

Once the `DataSource` has been set, you must call the `DataBind()` method to perform the actual data binding. If you forget to call `DataBind()`, no data binding will occur. The data binding will not occur automatically as it would if you were using a `SqlDataSource` and specifying the `DataSourceID`.

At the point where the `DataBind()` method is called, the data binding occurs, and the `Repeater` takes the data that is supplied and outputs the different templates, as necessary, to create the output that you see in Figure 7-2.

Since the `Repeater`, unlike the `DataList` and `GridView`, does not output any layout details of its own, you need to define your own layout. You do this through a combination of `<div>` and `<hr>` tags.

The first template that the `Repeater` will output is the `HeaderTemplate`:

```
<HeaderTemplate>
  <div style="background-color:Bisque">
    <font size="+2">Manufacturers</font>
  </div>
  <hr style="color:blue"/>
</HeaderTemplate>
```

Anything between the opening and closing tags will be output once to the page (if there is no `HeaderTemplate`, nothing will be output). In this case, you're outputting a simple title contained within a `<div>` tag with a slightly odd background color followed by an `<hr>` tag colored blue. You've used a `<div>` element to contain the title, so that each part of the template is displayed on a new line; the `<div>` element is a block element that starts on a new line.

The `HeaderTemplate` contains an `<hr>` tag to separate the header from the first `ItemTemplate` that is output. Although you can use a `SeparatorTemplate` to specify what you want to appear

between the ItemTemplate and AlternatingItemTemplate templates, the separator appears only between the templates, not before or after. The <hr> tag here is used to show the separation between the HeaderTemplate and ItemTemplate more easily.

The ItemTemplate is the first template that is shown that contains data returned from the database. This output is remarkably similar to the output that you saw in Figure 6-3 in the previous chapter. You show the Name of the Manufacturer on the first line, followed by the Country, Email, and Website of the Manufacturer on separate lines, as shown in Figure 7-3.

Apple
Country: USA
Email: lackey@apple.com
Website: http://www.apple.com

Figure 7-3. *The output from the ItemTemplate of the Repeater*

Each ItemTemplate contains four Web controls separated by <br> tags to ensure that each Web control is on a separate line. It has two Label controls and two HyperLink controls. The inline binding is the same for all four Web controls. Looking at the Country and Email markup, you can see that you use the Eval method to show what you want:

```
Country:
<asp:Label ID="lblCountry" runat="server"
  Text='<%# Eval("[ManufacturerCountry]") %>'>
</asp:Label>
<br />
Email:
<asp:HyperLink ID="lnkEmail" runat="server"
  NavigateUrl='<%# Eval("[ManufacturerEmail]", "mailto:{0}") %>'
  Text='<%# Eval("[ManufacturerEmail]") %>'>
</asp:HyperLink>
<br />
```

So for lblCountry, you're setting the Text property of the label to the ManufacturerCountry column from the DataReader. You do the same for the Text property of the lnkEmail hyperlink, except that you use the ManufacturerEmail column from the DataReader.

The NavigateUrl property is slightly different. As you discovered when you looked at inline binding in the previous chapter, you can't mix text with data-binding tags. So the following will not work:

```
NavigateUrl='mailto: <%# Eval("[ManufacturerEmail]") %>'
```

Instead, you need to use a slightly different version of Eval and format the data that you're trying to display:

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

You can also use the column index rather than the column name within the Eval statement. This works fine, unless the query to retrieve the data changes and column 2 is no longer the e-mail address. For that reason, it's always better to use the column name to reference a column.

Once the ItemTemplate has been processed, the Repeater will then output the SeparatorTemplate if one is present. In this case, you output a blue horizontal line:

```
<SeparatorTemplate>
  <hr style="color:blue"/>
</SeparatorTemplate>
```

If a SeparatorTemplate isn't specified, nothing will be output, and the Repeater will immediately show the AlternatingItemTemplate.

As you can see in Figure 7-2, the AlternatingItemTemplate is shown, not surprisingly, alternating with the ItemTemplate. If an AlternatingItemTemplate isn't specified, the ItemTemplate is simply repeated.

The usual use of the AlternatingItemTemplate is to provide a quick means of showing alternating rows in a slightly different way. In this example, you've simply changed the background color, as shown in Figure 7-4.

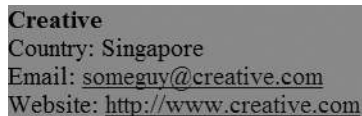


Figure 7-4. *The AlternatingItemTemplate shows rows slightly differently.*

In this case, the AlternatingItemTemplate and the ItemTemplate are the same other than the color specified for the background. The ItemTemplate has an ivory background:

```
<ItemTemplate>
  <div style="background-color:Ivory">
```

Whereas the AlternatingItemTemplate has an azure background:

```
<AlternatingItemTemplate>
  <div style="background-color:Azure">
```

Once all of the data has been shown using the ItemTemplate and AlternatingItemTemplate templates, the Repeater then shows the FooterTemplate:

```
<FooterTemplate>
  <hr style="color:blue"/>
  <div style="background-color:Bisque">
    <br />
  </div>
</FooterTemplate>
```

Again, you start with the horizontal rule, as the SeparatorTemplate isn't shown after the final *real* template. You then show an empty <div> in the odd background color that you used for the header. In this example, the footer just ties back to the header to round things off. You wouldn't lose anything if you didn't include this FooterTemplate.

The Repeater Control Events

You've seen how easy it is to use a `Repeater` to output data. With the requisite HTML and inline binding, you can create quite complex pages. However, this isn't the end of the story. What if you need to make decisions based on the data returned? Inline binding allows you to only display the data. By using some of the events that the `Repeater` exposes, you can make complex decisions based on the data that you want to show. For example, you could display an image depending on a value returned from the database.

Like all Web controls, the `Repeater` acquires some events from its inheritance hierarchy, but we're not particularly interested in the standard events (such as `Init` and `Load`). What we're interested in here are the events that are specific to the `Repeater`, as shown in Table 7-3.

Table 7-3. *The Important Events of the Repeater*

Name	Description
`ItemCommand`	Fired when a button Web control (a `Button`, `LinkButton`, or `ImageButton`) within the `Repeater` is clicked.
`ItemCreated`	Fired for each row that is output by the `Repeater`, for all five supported templates. If this event is fired in response to an `ItemTemplate` or an `AlternatingItemTemplate`, you don't have access to the data that is to be output. You need the `ItemDataBound` event for that.
`ItemDataBound`	Fired for each row that is output by the `Repeater`, for all five supported templates. If the row that is being output is an `ItemTemplate` or an `AlternatingItemTemplate`, you have access to the row from the data source that is being output.

In the next example, you'll see how to use the `ItemCommand` event to respond to the selection of a Manufacturer. The other two events, however, may cause a little confusion.

The `ItemDataBound` event is fired immediately after the `ItemCreated` event for every row that the `Repeater` is going to output. The only difference between the two events is that for the `ItemTemplate` and `AlternatingItemTemplate`, the `ItemDataBound` event has access to the row of data that is to be displayed. The `ItemCreated` event doesn't have access to this information. Because of this, you'll see the `ItemDataBound` event used far more often than the `ItemCreated` event. This is true not only for the `Repeater`, but the `DataList`, `GridView`, `DetailsView`, and `FormView` also support both of these events—although the `GridView` events are called `RowDataBound` and `RowCreated`. Again, the `ItemDataBound` (or `RowDataBound`) event will be used more frequently.

Try It Out: Using the Repeater to Display the Players

In this example, you'll build on the previous example and add a link button for each Manufacturer, allowing the user to view the Players for that Manufacturer. The list of Players will be shown on its own page, which will use a `SqlDataSource` to query the database.

1. In the code download for this chapter, you'll find an `images` folder. Copy this folder to your Chapter07 Web site.

2. If you've closed Repeater_DataReader.aspx from the previous example, reopen it and switch to the Source view.

3. Change the query in the Page_Load event to also return the ManufacturerID:

```
string CommandText = "SELECT ManufacturerID, ManufacturerName, ➥
  ManufacturerCountry, ManufacturerEmail, ManufacturerWebsite ➥
  FROM Manufacturer";
```

4. Add a LinkButton to the end of the ItemTemplate for the Repeater:

```
    <%# Eval("[ManufacturerWebsite]") %>
    </asp:HyperLink>
    <br />
    <asp:LinkButton ID="btnProducts" runat="server"
      CommandName="Players" Text="View Players"
      CommandArgument='<%# Eval("[ManufacturerID]") %>' />
  </div>
</ItemTemplate>
```

5. Add the same LinkButton to the end of the AlternatingItemTemplate for the Repeater:

```
    <%# Eval("[ManufacturerWebsite]") %>
    </asp:HyperLink>
    <br />
    <asp:LinkButton ID="btnProducts" runat="server"
      CommandName="Players" Text="View Players"
      CommandArgument='<%# Eval("[ManufacturerID]") %>' />
  </div>
</ItemTemplate>
```

6. Switch to the Design view and select the Repeater. In the Properties window, look at the events and double-click the ItemCommand event to add the event handler to the page.

7. Add the following code to the Repeater1_ItemCommand event:

```
protected void Repeater1_ItemCommand(object source,
  RepeaterCommandEventArgs e)
{
  if (e.CommandName == "Players")
  {
    Response.Redirect("./Repeater_DataSource.aspx?ManufacturerID=" +
      e.CommandArgument);
  }
}
```

8. Add a new Web Form to the Web site, called Repeater_DataSource.aspx.

9. In the Source view, find the <title> tag within the HTML and change the page title to **Binding a DataSource to a Repeater**.

10. Switch to the Design view and add a `SqlDataSource` to the page. Select Configure Data Source from the Tasks menu.

11. Select `SqlConnectionString` from the drop-down list on the Choose Your Data Connection step, and then click the Next button.

12. On the Configure the Select Statement step, select Player from the Name drop-down list and check the PlayerName, PlayerCost, and PlayerStorage columns. Click the WHERE button.

13. As shown in Figure 7-5, select PlayerManufacturerID from the Column drop-down list. Then select QueryString from the Source drop-down list and enter **ManufacturerID** in the QueryString Field text box.

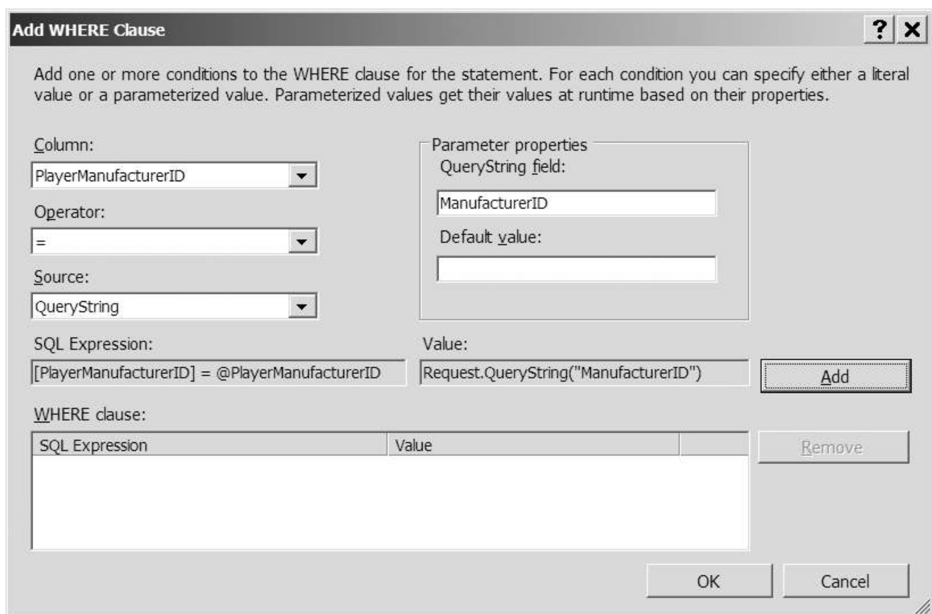


Figure 7-5. *Adding a QueryString parameter as the WHERE clause*

14. Click the Add button to add the parameter to the `WHERE` clause, and then click the OK button to close the dialog box.

15. Click the Next button. If you want to test the query, you can do so. You'll need to enter a valid ManufacturerID when prompted.

16. Once you're happy that the query returns a filtered list of Players, click the Finish button to close the Configure Data Source dialog box.

17. In the Design view, add a `Repeater` to the page. Select `SqlDataSource1` as the data source from the Tasks menu.

18. Switch to the Source view and make sure you've included the correct data provider at the top of the page:

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

19. Add the following two templates to the Repeater:

```
<asp:Repeater ID="Repeater1" runat="server">
  <HeaderTemplate>
    <div style="background-color:Bisque">
      <font size="+2">Players</font>
    </div>
    <hr style="color:blue"/>
  </HeaderTemplate>
  <ItemTemplate>
    <table>
      <tr>
        <td rowspan="2">
          <asp:Image ID="imgType" runat="server" />
        </td>
        <td>
          <b><asp:Label ID="lblName" runat="server" text="name" /></b>
        </td>
      </tr>
      <tr>
        <td>
          <asp:Label ID="lblCost" runat="server" text="cost" />
        </td>
      </tr>
    </table>
  </ItemTemplate>
</asp:Repeater>
```

20. Switch to the Design view and select the Repeater. Switch to the Events view in the Properties window and double-click the ItemDataBound event. Add the following code to the event handler that is created:

```
protected void Repeater1_ItemDataBound(object sender,
  RepeaterItemEventArgs e)
{
  if (e.Item.ItemType == ListItemType.Item
    || e.Item.ItemType == ListItemType.AlternatingItem)
  {
    // get the item that we're binding to
    DataRowView objData = (DataRowView)e.Item.DataItem;

    // set the two labels
    ((Label)e.Item.FindControl("lblName")).Text =
      objData["PlayerName"].ToString();
    ((Label)e.Item.FindControl("lblCost")).Text =
      String.Format("{0:n}", objData["PlayerCost"]);
```

```
        // set the correct image
        if (objData["PlayerStorage"].ToString() == "Hard Disk")
        {
          ((Image)e.Item.FindControl("imgType")).ImageUrl =
            "./images/disk.gif";
        }
        else
        {
          ((Image)e.Item.FindControl("imgType")).ImageUrl =
            "./images/solid.gif";
        }
      }
    }
```

21. Save both pages, and then view `Repeater_DataReader.aspx` in your browser. The list of Manufacturers from the previous example will be displayed, along with a View Players link, as shown in Figure 7-6.

Figure 7-6. *The list of Manufacturers along with the View Players link*

22. Click the View Players link for Apple, and the list of Players for Apple will be displayed, as shown in Figure 7-7.

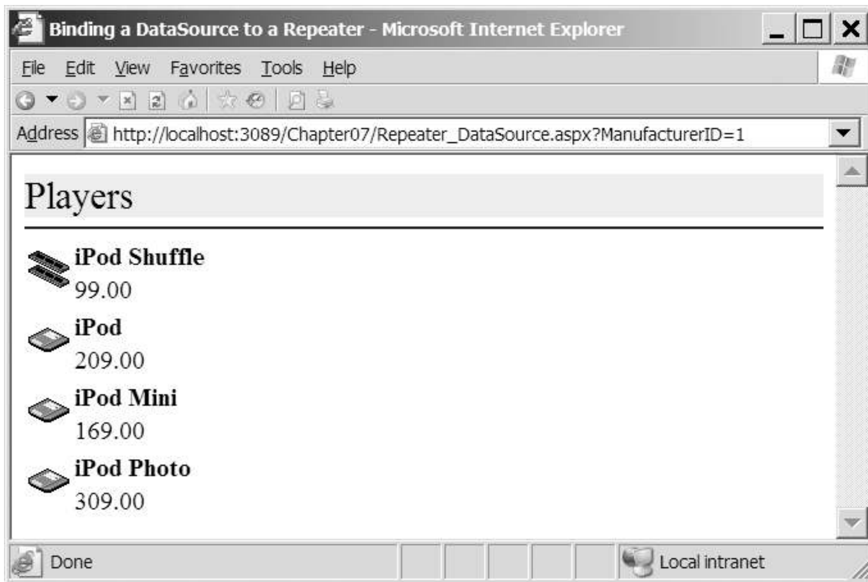


Figure 7-7. *Showing the Players made by Apple*

23. Click the back button in your browser and select any of the other Manufacturers. You'll see that the list of Players changes to show only those for the selected Manufacturer.

How It Works

You've used the `ItemCommand` and `ItemDataBound` events to create a little interactive example that starts to show some of the functionality that you can use to build your own pages. This example also uses the `SqlDataSource`, rather than accessing the database in code, to show that data binding to the `Repeater` can be done either way. You can also use the `ItemDataBound` event when using a DataReader or `DataSet` as the data source. Any time `DataBind()` is called—either in code or automatically—the `ItemCreated` and `ItemDataBound` events will be fired when a row of data needs to be displayed.

The Data Source and the Repeater

First, you modified the query to retrieve the list of Manufacturers from the database. In order to filter the list of Players by Manufacturer, you need to know which Manufacturer you're filtering; the query from the previous example didn't return us the ManufacturerID. You modified the query to add this:

```
string CommandText = "SELECT ManufacturerID, ManufacturerName, ➥
  ManufacturerCountry, ManufacturerEmail, ManufacturerWebsite ➥
  FROM Manufacturer";
```

Since you're modifying the query, you need to be careful that you don't break any data binding that is set up to use column indexes rather than column names. Thankfully we're sensible developers, and we always use column names rather than the index, so this isn't a problem here.

In order to display the list of Players, you need to link to a separate page. You could do this using a HyperLink and set the NavigateUrl property to the correct page, specifying the ManufacturerID that you want to show. While that would be the better solution in a real Web site (after all, you don't want a round trip to the server if you can avoid it), here, you're seeing how the ItemCommand event works. So, you add a LinkButton and set its CommandName and CommandArgument properties:

```
<asp:LinkButton ID="btnProducts" runat="server"
  CommandName="Players" Text="View Players"
  CommandArgument='<%# Eval("[ManufacturerID]") %>' />
```

Once the Repeater has output a row of data, that data is no longer available to the Web control. Thus, if you don't somehow pass the ManufacturerID to the ItemCommand event, there would be no way for you to know which Manufacturer was selected. The CommandArgument allows you to pass any arbitrary data that you want with the button click, and it will be available within the ItemCommand event. You need the ManufacturerID, so you use this as the value of the CommandArgument property.

The CommandName property works in a similar fashion. You're not limited to one button within an ItemTemplate or AlternatingItemTemplate, nor are you limited to using only one type of button; you can also use Button and ImageButton controls. In the case, where you want several different buttons (maybe View, Edit, and Delete buttons), you need some way of determining which button was pressed. You can pass any arbitrary text as the CommandName, and this will be available within the ItemCommand event. Although it's not required, as you have only one button in the row, you use the string Players to indicate that you want to see the list of Players. This will avoid problems in the future if you decide to add a second button.

Although you have a button that, when clicked, will post back to the server, you still haven't hooked up the ItemCommand event. You do this by adding the OnItemCommand property to the <asp:Repeater> tag. Even though you added the event using the Properties window, the event is still added to the markup for the Repeater:

```
<asp:Repeater ID="Repeater1" runat="server"
  OnItemCommand="Repeater1_ItemCommand">
```

You then need to handle the event in code. The first thing that you do is to check the CommandName for the clicked button:

```
if (e.CommandName == "Players")
```

The e argument passed to the event has CommandName and CommandArgument properties that simply pass in the settings from the button that was clicked. You check that the CommandName is the expected value, Players, and if it is, you redirect the user to Repeater_DataSource.aspx to display the correct list of Players using the CommandArgument property:

```
Response.Redirect("./Repeater_DataSource.aspx?ManufacturerID=" +
  e.CommandArgument);
```

The second page in the example, Repeater_DataSource.aspx, uses a SqlDataSource to populate the Repeater. You've seen several SqlDataSource examples already. The one thing that is different here is the parameter that you use to filter the list:

```
<SelectParameters>
  <asp:QueryStringParameter Name="PlayerManufacturerID"
    QueryStringField="ManufacturerID" Type="Int32" />
</SelectParameters>
```

In all the previous examples, you've used parameters based on the values from other Web controls using a `ControlParameter`. Here, you use another type of parameter: `QueryStringParameter`. As with all parameters, you have `Name` and `Type` properties that tie the parameter to the query that you're executing. For the `QueryStringParameter` to function, you need to tell it what value to extract from the query string, which you do by using the `QueryStringField` property.

The value of ManufacturerID in the query string is passed to the `SELECT` query as the `@PlayerManufacturerID` parameter, and the results are filtered correctly:

```
SELECT PlayerName, PlayerCost, PlayerStorage
FROM Player
WHERE (PlayerManufacturerID = @PlayerManufacturerID)
```

As you've already seen, the `SqlDataSource` is associated with a Web control using the `DataSourceID` property:

```
DataSourceID="SqlDataSource1"
```

Data Binding

Now that the data source and the `Repeater` have been combined, we can concentrate on the heart of the matter: the data binding that takes place. In the previous example, all of the binding took place inline using the `Eval` method. In this example, you do all the data binding within the `ItemDataBound` event. The event handler is attached using the `OnItemDataBound` attribute of the `<asp:Repeater>` tag:

```
<asp:Repeater ID="Repeater1" runat="server"
  DataSourceID="SqlDataSource1"
  OnItemDataBound="Repeater1_ItemDataBound">
```

When the `ItemDataBound` event is called, the first thing that you need to check is that you're actually trying to show a row of data. The `ItemDataBound` event is called for all five templates, and you need to check which type of template you're showing. You can check for any of the template types using the `ListItemType` enumerator. As you want to perform the data binding only if the template is an `ItemTemplate` or `AlternatingItemTemplate`, you use the `Item` and `AlternatingItem` values from the enumeration.

```
if (e.Item.ItemType == ListItemType.Item
  || e.Item.ItemType == ListItemType.AlternatingItem)
{
```

You check for `Item` and `AlternatingItem`, even though only an `ItemTemplate` is specified, because the `Repeater` is actually outputting `Item`, followed by `AlternatingItem`, followed by `Item`. If the alternating template isn't specified, the `Repeater` simply uses the item template—it's still classed as alternating row that is being displayed.

The data that you're trying to bind is available to the ItemDataBound event as e.Item.DataItem. This returns an Object that you need to cast to the correct type to use. The SqlDataSource, by default, accesses the database and returns a DataSet. Each row from the DataSet passed to the ItemDataBound event passes as a DataRowView, and e.Item.DataItem is cast to the correct type and stored for later use:

```
// get the item that we're binding to
DataRowView objData = (DataRowView)e.Item.DataItem;
```

This is the case whether you use a SqlDataSource to query the database or use code to construct a DataSet, and use this as the DataSource and call DataBind(). However, if you're using a DataReader (either through code or by setting the DataSourceMode of the SqlDataSource to DataReader), then you don't have a DataRowView. Instead, you have a DbDataRecord that you must cast slightly differently:

```
// get the item that we're binding to
DbDataRecord objData = (DbDataRecord)e.Item.DataItem;
```

Thankfully, both the DataRowView and the DbDataRecord have indexes that you can access using either a column name or a column index. Once you've cast e.Item.DataItem to the correct type, the way that you access the different values is the same.

In order to data-bind, you need to be able to access the Web controls that make up the row. As you're actually dealing with a template, the Web controls don't exist directly on the page, so you can't access them directly. You need to use the e.Item.FindControl() method to return the Web controls that you want to access, as you do for the two labels that make up the ItemTemplate:

```
// set the two labels
((Label)e.Item.FindControl("lblName")).Text =
  objData["PlayerName"].ToString();
((Label)e.Item.FindControl("lblCost")).Text =
  String.Format("{0:n}", objData["PlayerCost"]);
```

The FindControl() method returns a System.Web.UI.Control object that must be cast to the correct type before any of the Web control-specific properties can be accessed. In this case, you're dealing with labels, so you cast the object to a Label before setting the Text property.

Again, you can access the data either using a column name or a column index, but you should, ideally, always use the column name if you can. The index for both the DataRowView and DbDataRecord objects returns an Object, so you need to convert this to a string to set the Text property. You do this slightly differently for each of the labels, as you want to format the cost correctly.

The last piece of data binding that you perform shows why you sometimes need to use the ItemDataBound event. Simply setting Web control properties to a value from the database can be performed quite easily using inline binding. When you want to do something slightly different, then you need to use the ItemDataBound event. In this case, you want to display different images for the Players depending on whether the Player uses a hard disk or a solid-state storage format. You can find out which type the Player is by using the PlayerStorage column. This column is stored in the database as a string, so you can check what the value is by doing a simple string comparison. If the Player is a hard disk-based Player, then you want to show a disk for the

Player, so you set the ImageUrl of the Image (remember that you need to cast to the correct type) to disk.gif:

```
// set the correct image
if (objData["PlayerStorage"].ToString() == "Hard Disk")
{
  ((Image)e.Item.FindControl("imgType")).ImageUrl =
    "./images/disk.gif";
}
```

Otherwise, you want to show that it's a solid-state-based Player, so you set the ImageUrl to solid.gif:

```
else
{
  ((Image)e.Item.FindControl("imgType")).ImageUrl =
    "./images/solid.gif";
}
```

Inline Binding vs. Event-Based Binding

In the preceding couple of examples, we've looked at the two ways that you can perform table binding: with *inline binding* using the Eval() method or *event-based binding* within the ItemDataBound event.

Apart from the instances where you can't use inline binding, such as you saw with the example of changing the image depending on the type of Player, there really is no functional difference between the two types of binding. The main difference is performance. Inline binding uses reflection to evaluate the arguments that are passed in and to return the results. If you use the ItemDataBound event you're not using reflection, so it will be quicker.

Inline Binding Alternative

One of the main problems with inline binding is that fact that the Eval statement uses reflection to query for the requested column. This is perhaps the slowest way of getting at the data and, not surprisingly, there is an alternative.

Rather than using the Eval statement, you can cast the DataItem that you're displaying in much the same way as you do for event-based binding. So where you would use an Eval statement like the following:

```
<%# Eval("ManufacturerName") %>
```

you can instead cast the DataItem, accessed as with normal inline binding using Container.DataItem, to the correct type.

If you're using the DataSet to provide the data source, you need to cast to a DataRowView:

```
<%# ((DataRowView)Container.DataItem)["ManufacturerName"] %>
```

And if you're using a DataReader, you need to cast to a DbDataRecord:

```
<%# ((DbDataRecord)Container.DataItem)["ManufacturerName"] %>
```

Mixing Binding Types

It is also possible to use both inline binding and event-based binding within the DataBind() call. If you choose to use both types of binding at the same time, you need to be aware that the inline binding will occur before the event-based binding.

So, if you were to set the Text property of two Label controls using inline binding as follows:

```
<asp:Label ID="lblName" runat="server"
  Text='<%# Eval("[ManufacturerName]") %>'>
</asp:Label>
<asp:Label ID="lblCountry" runat="server"
  Text='<%# Eval("[ManufacturerCountry]") %>'>
</asp:Label>
```

And then in the ItemDataBound event change the Text property of lblCountry as follows:

```
((Label)e.Item.FindControl("lblCountry")).Text = "COUNTRY";
```

you would not get the results that you're expecting. During the inline binding, both lblName and lblCountry will be bound to the correct values from the database. As soon as the ItemDataBound event is fired, the correct value for lblCountry is replaced by the string COUNTRY.

Granted, this is a very contrived example, as you wouldn't set the Text property to a meaningless value like this, but it does serve to illustrate the point.

The DataList Web Control

The DataList is a Web control that sort of sits in no-man's land. It allows you to display data to the user, and with some coding in the background, it allows you edit data inline and propagate these changes back to the database. However, it's one of those Web controls that you don't see used very often and tends to get overlooked. We're not going to spend too much time looking at the DataList, because it follows the same principles as the Repeater.

■ **Note** If you want to use a DataList for editing data, refer to http://msdn.microsoft.com/en-us/library/9cx2f3ks.aspx for full details.

We'll dive straight into an example. You'll see that when displaying data, the DataList and Repeater have a lot in common.

Try It Out: Using the DataList to Display the Players

In this example, you'll use a SqlDataSource to query the database for a complete list of Players from the database. The list of Players will be displayed in a DataList.

1. Add a new Web Form to the Chapter07 Web site called DataList_DataSource.aspx.
2. In the Source view, find the <title> tag within the HTML and change the page title to **Binding a DataSource to a DataList**.

3. Switch to the Design view and add a `SqlDataSource` to the page. Select Configure Data Source from the Tasks menu.

4. Select `SqlConnectionString` from the drop-down list on the Choose Your Data Connection step, and then click the Next button.

5. On the Configure the Select Statement step, select Player from the Name drop-down list and check the PlayerName, PlayerCost, and PlayerStorage columns.

6. Click the ORDER BY button and select PlayerName as the Sort By column. Click OK to close the Add ORDER BY Clause dialog box.

7. Click the Next button. If you want to test the query, you can choose to do so on the next step. Once you're happy that the query returns a complete list of Players, click the Finish button to close the Configure Data Source dialog box.

8. Add a `DataList` onto the page. Select `SqlDataSource1` as the data source from the Tasks menu.

9. Select the `DataList`, and from the Properties window, set `RepeatDirection` to Horizontal and set `RepeatColumns` to 3.

10. Switch to the Source view and make sure you've included the correct namespace declaration at the top of the page:

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

11. Replace the auto-generated `ItemTemplate` with the following `ItemTemplate` and `AlternatingItemTemplate`:

```
<ItemTemplate>
  <table bgcolor="Ivory">
    <tr>
      <td rowspan="2">
        <asp:Image ID="imgType" runat="server" />
      </td>
      <td>
        <b><asp:Label ID="lblName" runat="server" text="name" /></b>
      </td>
    </tr>
    <tr>
      <td>
        <asp:Label ID="lblCost" runat="server" text="cost" />
      </td>
    </tr>
  </table>
</ItemTemplate>
<AlternatingItemTemplate>
  <table bgcolor="Azure">
    <tr>
```

```
      <td rowspan="2">
        <asp:Image ID="imgType" runat="server" />
      </td>
      <td>
        <b><asp:Label ID="lblName" runat="server" text="name" /></b>
      </td>
    </tr>
    <tr>
      <td>
        <asp:Label ID="lblCost" runat="server" text="cost" />
      </td>
    </tr>
  </table>
</AlternatingItemTemplate>
```

12. Switch to the Design view and select the `DataList`. Switch to the Events view in the Properties window and double-click the `ItemDataBound` event. Add the following code to the event handler that is created:

```
protected void DataList1_ItemDataBound(object sender,
  DataListItemEventArgs e)
{
  if (e.Item.ItemType == ListItemType.Item
     || e.Item.ItemType == ListItemType.AlternatingItem)
  {
    // get the item that we're binding to
    DataRowView objData = (DataRowView)e.Item.DataItem;

    // set the two labels
    ((Label)e.Item.FindControl("lblName")).Text =
      objData["PlayerName"].ToString();
    ((Label)e.Item.FindControl("lblCost")).Text =
      String.Format("{0:n}", objData["PlayerCost"]);

    // set the correct image
    if (objData["PlayerStorage"].ToString() == "Hard Disk")
    {
      ((Image)e.Item.FindControl("imgType")).ImageUrl =
        "./images/disk.gif";
    }
    else
    {
      ((Image)e.Item.FindControl("imgType")).ImageUrl =
        "./images/solid.gif";
    }
  }
}
```

13. Save the page, and then view it in a browser. When the page loads, you'll see that the list of Players is presented, as shown in Figure 7-8.

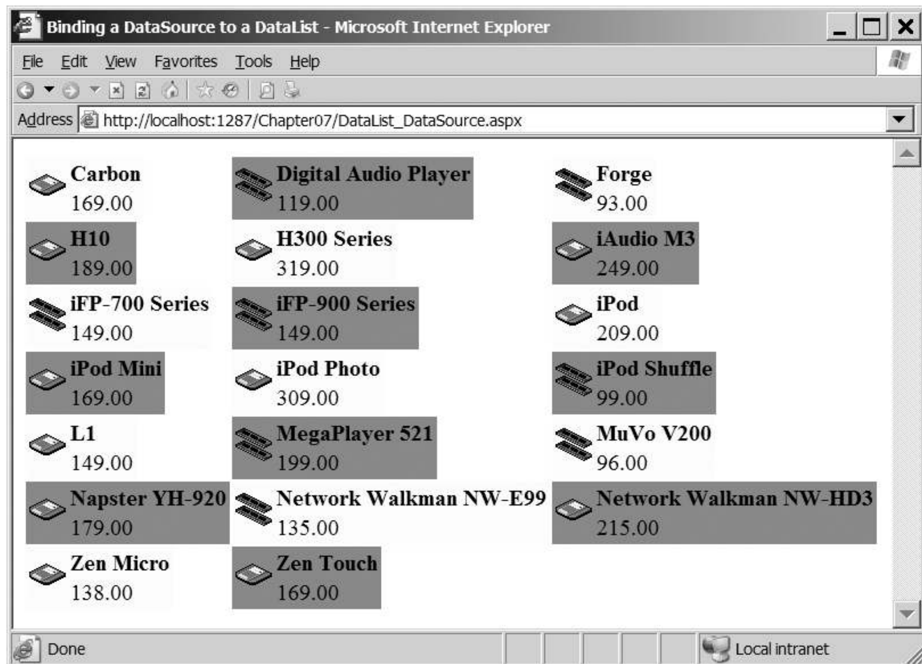


Figure 7-8. *Showing the Players using a DataList*

How It Works

The main difference here is the definition of the `DataList`. The first thing that you'll notice is that you've set two properties on the `DataList`: `RepeatDirection` to Horizontal and `RepeatColumns` to 3. As the `DataList` outputs in a table, you're effectively telling it to show three rows from the database on a line before moving to the next line. The list of Players is sorted by the name of the Player, and you can see that they are displayed horizontally before vertically.

Within the `DataList`, you've used the same `ItemTemplate` and `AlternatingItemTemplate` as you did with the `Repeater`. You've altered the color of the table that is displayed so that you can see the templates alternating, but apart from that, the templates are the same. Each template is output in its own cell within the table, and you've chosen to keep a table structure within the template, as it provides a nice enough display for your purposes.

When the `DataList` is data-bound, you've used the `ItemDataBound` event to perform some more complex processing than is available with inline binding. The code within the `ItemDataBound` event is the same as the `Repeater`; not a single line has changed.

As you can see in this very brief example, the `DataList` can provide a little more control over the output than a `Repeater`, but basically how you control what is displayed is the same.

The GridView Web Control

So far, we've looked at table binding in the context of the Repeater and, very briefly, the DataList. We've concentrated on the Repeater, as it's the "least-featured" of the table-binding Web controls, and it allowed us to cover most of what you need to know without being distracted by any functionality that other Web controls may provide.

Now, we'll look at the GridView, beginning with a brief recap of what we've looked at in previous chapters. Then you'll see how you can customize the layout and the columns for the GridView. Finally, we'll cover the sorting and paging functionality that the GridView supports.

■ **Note** As you've already seen, you can construct the data source for a Web control in four ways. You can write code that populates a DataSet or returns a DataReader that you pass to the DataSource property of the Web control, or you can use a SqlDataSource (in either DataReader or the default DataSet mode) and set the DataSourceID property of the Web control. With ASP.NET 2.0, Microsoft has tied the SqlDataSource and GridView quite closely together, and by far the best option is to use them in conjunction. Therefore, all the examples here will use a SqlDataSource as the data source to the GridView.

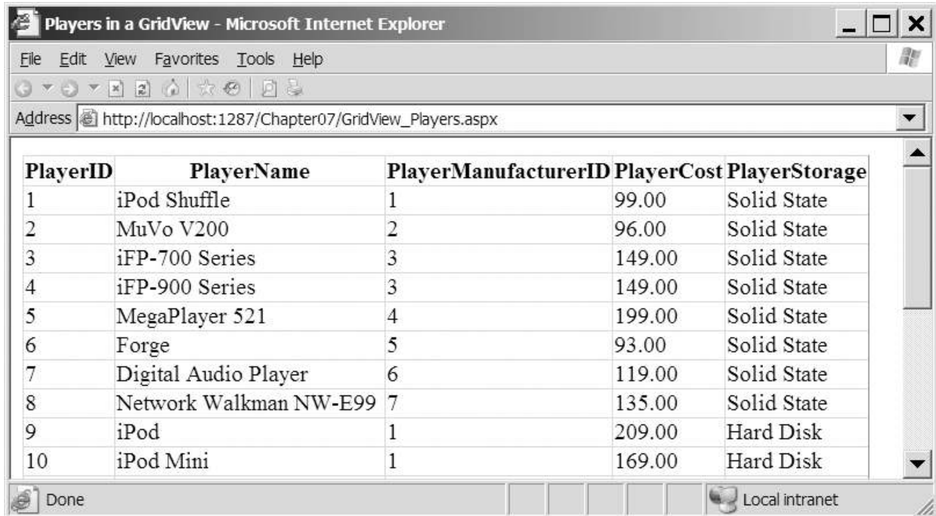
Try It Out: Showing Data in a GridView

In this example, you'll build a page that displays the list of Players and let the GridView take care of creating all the columns. This is the same page as you've seen in Chapters 3 and 4. In the following examples, you'll build on this page to see how you can modify the GridView to create much more useful pages.

1. In Visual Web Developer, add a new Web Form called GridView_Players.aspx. Make sure that the Place Code in Separate File check box is unchecked.
2. In the Source view, find the <title> tag within the HTML at the bottom of the page and change the page title to **Players in a GridView**.
3. Switch to the Design view and add a SqlDataSource to the page. Select Configure Data Source from the Tasks menu.
4. Select SqlConnectionString as the connection string to use on the first step of the wizard. Click the Next button.
5. Select the Player table. In the Columns list, select all five columns individually. Click the Next button.
6. If you want to test that the query returns the correct results, you can do so. When you're happy that the results are correct, click the Finish button.
7. Switch to the Source view. Within the body of the page, between the <form> and </form> tags, add a GridView to the page:

```
<asp:GridView ID="GridView1" runat="server"
  DataSourceID="SqlDataSource1">
</asp:GridView>
```

8. Save the page, and then view it in a browser. When the page loads, you'll see that the list of Players is presented in a tabular format, as shown in Figure 7-9.



PlayerID	PlayerName	PlayerManufacturerID	PlayerCost	PlayerStorage
1	iPod Shuffle	1	99.00	Solid State
2	MuVo V200	2	96.00	Solid State
3	iFP-700 Series	3	149.00	Solid State
4	iFP-900 Series	3	149.00	Solid State
5	MegaPlayer 521	4	199.00	Solid State
6	Forge	5	93.00	Solid State
7	Digital Audio Player	6	119.00	Solid State
8	Network Walkman NW-E99	7	135.00	Solid State
9	iPod	1	209.00	Hard Disk
10	iPod Mini	1	169.00	Hard Disk

Figure 7-9. *Showing the Players in a GridView*

How It Works

This is perhaps the simplest example that we've looked at so far! Within the markup, you've let the automatic features of the GridView take control. It's the simplest data-bound Web control definition you've seen:

```
<asp:GridView ID="GridView1" runat="server"
  DataSourceID="SqlDataSource1">
</asp:GridView>
```

You have not defined any columns or told the GridView how you want the data presented. It took its complete definition from the SQL query that you ran against the database via the SqlDataSource:

```
SELECT PlayerID, PlayerName, PlayerManufacturerID, PlayerCost, PlayerStorage
FROM Player
```

Every column you specified in the query is output by the GridView, and the name of the column in the table is the name of the column in the database.

Although you've achieved your goal of using the GridView, I think you'll agree that the data isn't presented in the best way. We'll now look at how you can make the output from the GridView more presentable, before moving on to look at how you can allow the user to page through results and to sort the results that they are viewing.

GridView Customization

In this section, you'll look at the following ways to improve the example's dreary black-and-white table:

- Changing the column headings and making the data itself more readable
- Adding some color to the table by defining styles
- Defining the columns that are displayed

You'll stick to using a `SqlDataSource` as the data source, but these techniques apply to all four data-access mechanisms.

Customizing the Data

The first task is to make the data itself more presentable. When data is bound to a `GridView`, the column name used in the database table is presented as the column header. That's fine when you're designing databases, but PlayerManufacturerID isn't suitable for user viewing. It's preferable to show the name of the Manufacturer instead.

You could customize the `GridView` by setting its `AutoGenerateColumns` property to `False` and explicitly defining the columns that you want to display. In this case, you define how the `GridView` renders each particular column with `BoundField` and `TemplateField` objects, rather than templates for each row, but the principles are the same. You'll look at this method shortly, but for now, leave `AutoGenerateColumns` as `True` (the default value). Instead, you'll see how to use the SQL query to make the change.

You saw in Chapter 3 how to use an `INNER JOIN` in a SQL query to retrieve data from one table based on values from another. You can use that technique here to return the Manufacturer's name for each Player, rather than the ManufacturerID number. Instead of the following:

```
SELECT PlayerID, PlayerName, PlayerManufacturerID, PlayerCost, PlayerStorage
FROM Player
```

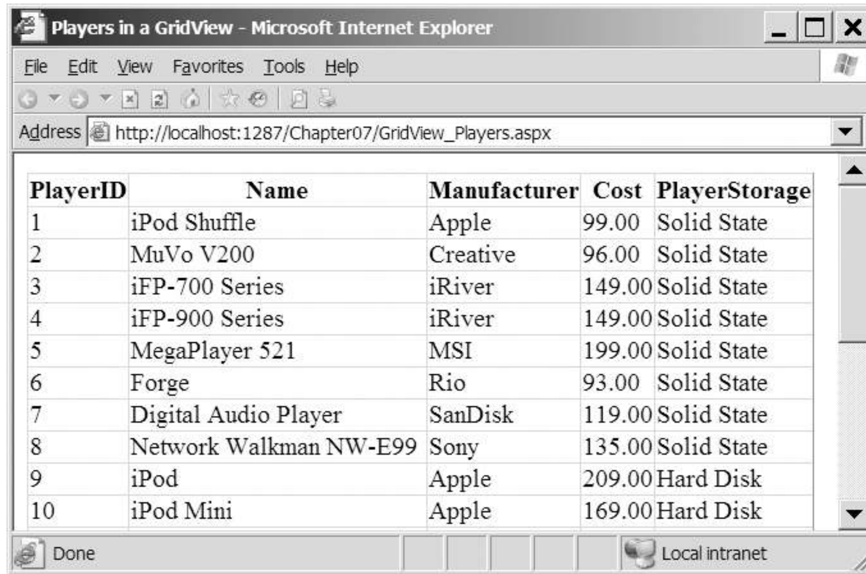
You now have this:

```
SELECT Player.PlayerID, Player.PlayerName, Manufacturer.ManufacturerName,
  Player.PlayerCost, Player.PlayerStorage
FROM Player INNER JOIN Manufacturer
  ON Player.PlayerManufacturerID = Manufacturer.ManufacturerID
```

That sorts the contents of the query, but how do you influence the column names? You alter the SQL query again and use *aliases* as needed for each column. In `SELECT` queries, you use the `AS` keyword to tell a database to return an alternate name for a column in its results for a query. If you wanted to show the PlayerName column in the Player table as just Name in the results of a query, `SELECT Player.PlayerName AS Name` would do the trick. For this example, you can add aliases to the query and complete the first of the tasks:

```
SELECT Player.PlayerID, Player.PlayerName AS Name,
  Manufacturer.ManufacturerName AS Manufacturer,
  Player.PlayerCost AS Cost, Player.PlayerStorage
FROM Player INNER JOIN Manufacturer
  ON Player.PlayerManufacturerID = Manufacturer.ManufacturerID
```

You can see the results in Figure 7-10.



PlayerID	Name	Manufacturer	Cost	PlayerStorage
1	iPod Shuffle	Apple	99.00	Solid State
2	MuVo V200	Creative	96.00	Solid State
3	iFP-700 Series	iRiver	149.00	Solid State
4	iFP-900 Series	iRiver	149.00	Solid State
5	MegaPlayer 521	MSI	199.00	Solid State
6	Forge	Rio	93.00	Solid State
7	Digital Audio Player	SanDisk	119.00	Solid State
8	Network Walkman NW-E99	Sony	135.00	Solid State
9	iPod	Apple	209.00	Hard Disk
10	iPod Mini	Apple	169.00	Hard Disk

Figure 7-10. *Column aliases appear as column headers.*

Although the column headings are now more readable, the data that is displayed may not be quite correct. This is where explicitly defining the columns that you want to display comes into its own.

Adding Styles

The contents of the `GridView` are now a bit more readable, so you'll now focus on the `GridView` itself. It's a bit drab in black and white, so you might want to add some color. To do this, you can define style templates for the rows. The whole process is similar to the `ItemTemplate` in the `Repeater`, but as the `GridView` automatically generates a grid for a row, all you need to decide is how each row will be presented: colors, fonts, text alignment, cell padding, cell width, and so on.

The whole process is actually made quite easy by Visual Web Developer as it contains several predefined color schemes to apply directly to the `GridView`. Simply select your `GridView` in the Design view, and then click the Auto Format option on the Tasks menu. A dialog box will appear, allowing you to select a color scheme on the left and preview it on the right, as shown in Figure 7-11.

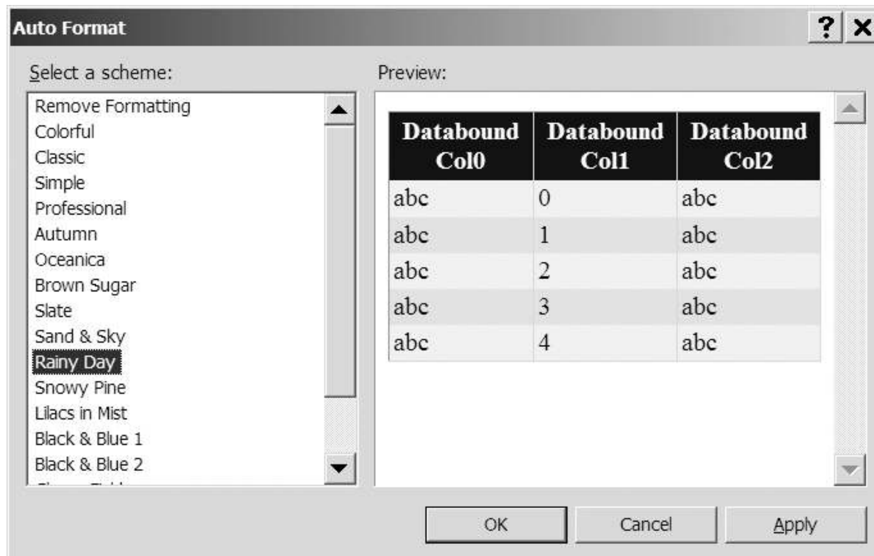


Figure 7-11. *The Auto Format dialog box lets you choose several predefined styles.*

Selecting one of the format options from the Auto Format dialog adds the required definitions for the selected style to the markup for the `GridView`. For example, you could choose the Rainy Day scheme (which uses most of the available row styles). The resulting code looks like this:

```
<asp:GridView ID="GridView1" runat="server" BackColor="White"
  BorderColor="#999999" BorderStyle="None" BorderWidth="1px"
  CellPadding="3" GridLines="Vertical">
  <FooterStyle BackColor="#CCCCCC" ForeColor="Black" />
  <RowStyle BackColor="#EEEEEE" ForeColor="Black" />
  <SelectedRowStyle BackColor="#008A8C" Font-Bold="True"
    ForeColor="White" />
  <PagerStyle BackColor="#999999" ForeColor="Black"
    HorizontalAlign="Center" />
  <HeaderStyle BackColor="#000084" Font-Bold="True"
    ForeColor="White" />
  <AlternatingRowStyle BackColor="#DCDCDC" />
</asp:GridView>
```

The style information in the `<asp:GridView>` tag itself sets the defaults for the whole grid, and the various `<`*xxx*`Style>` tags define deviations from that default, similar to how CSS works. In fact, there are eight different styles you can work with, as shown in Table 7-4.

Table 7-4. *The GridView Styles*

Name	Description
AlternatingRowStyle	Defines a style for every other row in the GridView, making it easier for the reader to follow a line across the grid
EditRowStyle	Applied to a row when it is being edited
EmptyDataRowStyle	Defines the style to be used when the GridView has no content to display
FooterStyle	Defines the style for the footer of the GridView
HeaderStyle	Defines the style for the header of the GridView
PagerStyle	Applied to the footer when it displays page-navigation links
RowStyle	Sets up the default style for a row in the GridView
SelectedRowStyle	Applied to a row when it's selected

Explicitly Defining the Columns

As you've seen, by default, the GridView automatically defines the columns it displays. However, this isn't always what you want, and with auto-generated columns, you must be very specific in the data that is used to populate the GridView. The column headings are the names of the columns from the database (or aliases, if you define them), and any changes to the query will change the layout of the GridView.

Setting the AutoGenerateColumns property to False prevents the GridView from generating the columns automatically and allows you to explicitly define the columns that you want to show. You do this by specifying a list of columns, or *Field controls* in GridView parlance. Table 7-5 lists the seven Field controls that you can define.

Table 7-5. *The GridView Field Controls*

Name	Description
BoundField	Displays data directly from the data source. The DataField property indicates which particular column of the results you want to show, and you can use the optional DataFormatString property to format the results as required.
ButtonField	Shows a button that causes a postback to the server, allowing an action to be performed on the selected row.
CheckBoxField	Shows a check box. This type of column is usually used to display columns from the database with Boolean values.
CommandField	Creates a column that contains Edit, Update, and Cancel buttons (as appropriate) to allow editing of the data within the selected row.
HyperLinkField	Shows a hyperlink.
ImageField	Shows an image.
TemplateField	Allows full control over what is displayed for the column: the header, the item displayed, and the footer. The TemplateField contains various templates controlling what is displayed and, within these templates, you can place whatever content you desire. This makes the TemplateField the most customizable of the column types.

Although the seven Field controls provide completely different functionality, they are all derived from the same base class, `System.Web.UI.WebControls.DataControlField`, and so share some very basic functionality. The most important properties are summarized in Table 7-6.

Table 7-6. *Important Properties for Field Controls*

Name	Description
`FooterStyle`	Sets the style of an individual column's footer. If the `FooterStyle` is specified for a column and any of the values clash (such as defining a background color in both), the setting here overrides the setting applied to the `GridView` as a whole.
`FooterText`	Sets the text to be displayed in the footer. The footer is shown only if the `GridView` has its `ShowFooter` property set to `True`.
`HeaderStyle`	Sets the style of an individual column's header. If the `HeaderStyle` is specified for a column and any of the values clash (such as defining a background color in both), the setting here overrides the setting applied to the `GridView` as a whole.
`HeaderText`	Sets the text to be displayed in the header. The header is shown only if the `GridView` has its `ShowHeader` property set to `True`.
`ItemStyle`	Sets the style of an individual column. If the `ItemStyle` is specified for a column and any of the values clash (such as defining a background color in both), the setting here overrides the setting applied to the `GridView` as a whole (either in the `ItemStyle` or the `AlternatingItemStyle`).

The `TemplateField` is the most versatile column supported by the `GridView`. It allows you to display and edit data using whatever Web controls you desire. It does this by allowing you to specify templates that you want to use in the different situations. The four templates that the `TemplateField` supports for displaying data are summarized in Table 7-7.

Table 7-7. *Templates Supported by the TemplateField in a GridView*

Name	Description
`AlternatingItemTemplate`	Defines the template for every other row in the `GridView`, making it easier for the reader to distinguish between the rows in the grid.
`FooterTemplate`	The HTML content of the `FooterTemplate` is output as the footer of the column being displayed. If a `FooterTemplate` is defined, any value set for the `FooterText` property of the column is ignored.
`HeaderTemplate`	The HTML content of the `HeaderTemplate` is output as the header of the column being displayed. If a `HeaderTemplate` is defined, any value set for the `HeaderText` property of the column is ignored.
`ItemTemplate`	The `ItemTemplate` is used to display the data returned from the data source in the format that you specify. As a minimum, this template must be specified in order for the `GridView` to output the returned data.

Templates in the `GridView` serve the same purpose as those in the `Repeater`. They allow you to control how the data is displayed. However, whereas the templates in the `Repeater` work on the entire row of data, the templates in the `GridView` are for a single column.

■ **Note** The `GridView` allows you to edit data inline. To support this functionality, there is also an `EditItemTemplate` that you can use to specify what is displayed when the `GridView` is in edit mode. The `GridView` shares the use of the Field controls with its sibling the `DetailsView`. When using a `TemplateField` within the `DetailsView`, you can specify another template, the `InsertItemTemplate`. We'll look at these two templates when we cover editing using the `GridView`, `DetailsView`, and `FormView` in Chapter 9.

Try It Out: Customizing the GridView

In this example, you'll customize a `GridView` by defining the exact columns that you want to show. By combining the three techniques discussed in the preceding sections, you'll extend the previous example and build a page that is much more user-friendly.

1. Open `GridView_Players.aspx` from the previous example and switch to the Source view.

2. Change the query that is executed by the `SqlDataSource` to return a more usable set of data as follows:

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
  ConnectionString="<%$ ConnectionStrings:SqlConnectionString %>"
  SelectCommand="SELECT Player.PlayerID, Player.PlayerName,
    Manufacturer.ManufacturerName, Player.PlayerCost,
    Player.PlayerStorage FROM Player INNER JOIN Manufacturer
    ON Player.PlayerManufacturerID = Manufacturer.ManufacturerID">
</asp:SqlDataSource>
```

3. Add the correct namespace declaration at the top of the page:

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

4. Switch to the Design view for the page and select Auto Format from the `GridView` Tasks menu. Choose the Rainy Day style for this example.

5. Select the `GridView`. From the Properties window, set the `AutoGenerateColumns` property to `False`.

6. Switch to the Source view for the page and add the following to the `GridView` definition, after the style elements that have been added:

```
<Columns>
  <asp:BoundField DataField="PlayerID" HeaderText="PlayerID" />
  <asp:BoundField DataField="PlayerName" HeaderText="Name" />
  <asp:BoundField DataField="ManufacturerName"
    HeaderText="Manufacturer" />
  <asp:BoundField DataField="PlayerCost" DataFormatString="{0:n}"
    HeaderText="Cost" />
  <asp:TemplateField>
    <ItemTemplate>
      <asp:Image ID="imgType" runat="server" />
    </ItemTemplate>
  </asp:TemplateField>
</Columns>
```

7. Switch to the Design view. You'll see that the columns are displayed correctly, as shown in Figure 7-12.

GridView_Players.aspx*

SqlDataSource - SqlDataSource1

PlayerID	Name	Manufacturer	Cost	
0	abc	abc	0	
1	abc	abc	0.1	
2	abc	abc	0.2	
3	abc	abc	0.3	
4	abc	abc	0.4	

Figure 7-12. *Defined columns are shown correctly in the Design view.*

8. Add a `RowDataBound` event for the `GridView` and add the following code to the event handler:

```
protected void GridView1_RowDataBound(object sender,
  GridViewRowEventArgs e)
{
  if (e.Row.RowType == DataControlRowType.DataRow)
  {
    // get the item that we're binding to
    DataRowView objData = (DataRowView)e.Row.DataItem;
```

```
        // set the correct image
        if (objData["PlayerStorage"].ToString() == "Hard Disk")
        {
          ((Image)e.Row.FindControl("imgType")).ImageUrl =
            "./images/disk.gif";
        }
        else
        {
          ((Image)e.Row.FindControl("imgType")).ImageUrl =
            "./images/solid.gif";
        }
      }
    }
```

9. Save the page, and then view it in a browser. As shown in Figure 7-13, when the page loads, you'll see that the list of Players is presented in a much more user-friendly format. You'll see different images, depending on the storage format of the Player.

Figure 7-13. *A much more presentable view of the Players*

How It Works

Now you're starting to see how powerful the `GridView` can be, but also that it's quite similar, when displaying data, to the `Repeater` and `DataList`.

The first change that you've made to the example is to return a different set of results from the database. You've added an `INNER JOIN` to the query, and rather than returning the ManufacturerID, you return the ManufacturerName:

```
SELECT Player.PlayerID, Player.PlayerName, Manufacturer.ManufacturerName,
  Player.PlayerCost, Player.PlayerStorage
FROM Player INNER JOIN Manufacturer
  ON Player.PlayerManufacturerID = Manufacturer.ManufacturerID
```

The main changes that you made in the example are to the `GridView` definition itself. You selected the Rainy Day style from the Auto Format dialog box, and this has added several different styles to the definition:

```
<asp:GridView ID="GridView1" runat="server" BackColor="White"
  BorderColor="#999999" BorderStyle="None" BorderWidth="1px"
  CellPadding="3" GridLines="Vertical">
  <FooterStyle BackColor="#CCCCCC" ForeColor="Black" />
  <RowStyle BackColor="#EEEEEE" ForeColor="Black" />
  <SelectedRowStyle BackColor="#008A8C" Font-Bold="True"
    ForeColor="White" />
  <PagerStyle BackColor="#999999" ForeColor="Black"
    HorizontalAlign="Center" />
  <HeaderStyle BackColor="#000084" Font-Bold="True"
    ForeColor="White" />
  <AlternatingRowStyle BackColor="#DCDCDC" />
</asp:GridView>
```

The style has added settings that apply to the `GridView` as a whole (the attributes on the `<asp:GridView>` element itself) and added settings for the different row types (shown as children of the `<asp:GridView>` element).

Rather than let the `GridView` show the columns automatically, you specify the columns that you want to show in the `<Columns>` element. You have five columns, specified as five Field controls, and three of these are about as simple as it gets:

```
<asp:BoundField DataField="PlayerID" HeaderText="PlayerID" />
<asp:BoundField DataField="PlayerName" HeaderText="Name" />
<asp:BoundField DataField="ManufacturerName"
  HeaderText="Manufacturer" />
```

You use a `BoundField` to show the data returned from the database as a string on the page. The `DataField` property specifies the column from the results that you want to bind to, and the `HeaderText` property sets what is displayed in the header of the column. You used the `HeaderText` property to make the results a little more user-friendly. Rather than the column names from the database, you use Name and Manufacturer.

■ **Note** The PlayerID column is pretty meaningless to the reader. Here, you've left it in the `GridView` for a later example. When we look at sorting and paging shortly, you'll see that this column offers a quick and easy way to see that the data being displayed is indeed changing. In a real page, you probably would not display it.

The fourth column is a little more complex. It's again a `BoundField`, but you specify an additional property, `DataFormatString`. One of the problems with the auto-generated columns was that it was impossible to show the data in the correct format. The `DataFormatString` property allows you to apply whatever format you require to the data that you're displaying. In this case, you want to show the price of the Player as a decimal:

```
<asp:BoundField DataField="PlayerStorePrice" DataFormatString="{0:n}"
  HeaderText="Cost" />
```

The last column that you define is a `TemplateField`. We define an `ItemTemplate` to display data, and use an `Image` to show a different image depending on the storage mechanism of the Player:

```
<asp:TemplateField>
  <ItemTemplate>
    <asp:Image ID="imgType" runat="server" />
  </ItemTemplate>
</asp:TemplateField>
```

In order to set the URL of the image that you want to display, you use the `RowDataBound` event. You could use inline binding, but as you saw with the `Repeater` earlier in the chapter, the `RowDataBound` event gives you a little more control over what you're going to display. As with the `ItemDataBound` event for the `Repeater` and `DataList`, the `RowDataBound` event is fired whenever a row of data is bound to the `GridView`.

Within the event handler, you add code that's remarkably similar to the `ItemDataBound` event handler that you saw for the `Repeater` earlier. You first check whether the row you're displaying is a `DataRow` (this is equivalent to checking for the `ItemType` being an `Item` or an `AlternatingItem` for the `Repeater` or `DataList`). If you are binding a `DataRow`, you cast the `DataItem` to the correct type:

```
if (e.Row.RowType == DataControlRowType.DataRow)
{
  // get the item that we're binding to
  DataRowView objData = (DataRowView)e.Row.DataItem;
```

You're using a `DataSet` to populate the `GridView` (as you're using the default access mode for the `SqlDataSource`), so you need to cast to a `DataRowView`. You then set the `ImageUrl` of the `Image` based on the PlayerStorage column returned from the database:

```
  // set the correct image
  if (objData["PlayerStorage"].ToString() == "Hard Disk")
  {
    ((Image)e.Item.FindControl("imgType")).ImageUrl =
      "./images/disk.gif";
  }
  else
  {
    ((Image)e.Item.FindControl("imgType")).ImageUrl =
      "./images/solid.gif";
  }
}
```

Now that we've covered the basics of the GridView, it's time to move forward and look at some of the interactive functionality that is available. We'll look at paging and sorting the results, and then move on to look at the events that are available for the GridView.

Paging and Sorting

Now that you've made the GridView more presentable and the information it contains more readable, you have one other card to play: to have the GridView present the information interactively. At the moment, it just displays all the information that results from a query in one go.

One interactive feature is *sorting*. You can set up the GridView so that clicking a column header sorts the information it contains by the entries in that column. A GridView that sorts can come in handy for users when they're checking data. On eBay, for example, it's a lot more helpful if you can sort which auctions are finishing first than having the information sorted by description. In the examples so far, it would be a lot easier to see which Manufacturer is associated with which Players if you could sort the data in the GridView by the contents of the Manufacturer column.

Another useful feature is *paging*. As you've no doubt experienced, unless you're incredibly specific in searching the Web, chances are that the search engine returns thousands of results. It would be impractical to display all those on one page, so they're divided into pages of 10 (or 25 or whatever you've set the default to) results, so you can move through them more effectively. You can apply this same technique to your GridView, setting it up so that it displays only a small number of rows per page, making it easier to read, with links to move through other pages of data until it has all been shown.

Try It Out: Sorting the GridView

Now you'll build on the previous example and implement sorting on the GridView.

1. Open GridView_Players.aspx from the previous example.

2. In the Source view, modify the column definitions for the GridView and add SortExpression attributes to the PlayerID, Name, and Manufacturer columns:

```
<asp:BoundField DataField="PlayerID" HeaderText="PlayerID"
  SortExpression="PlayerID" />
<asp:BoundField DataField="PlayerName" HeaderText="Name"
  SortExpression="PlayerName" />
<asp:BoundField DataField="ManufacturerName"
  HeaderText="Manufacturer" SortExpression="ManufacturerName" />
```

3. Switch to the Design view and select the GridView. From the Tasks menu, set the EnableSorting property to True.

4. Execute the page. You'll see that the PlayerID, Name, and Manufacturer column headings are now hyperlinks. Clicking one of the columns, such as the Manufacturer column, sorts the data on the column, as shown in Figure 7-14.

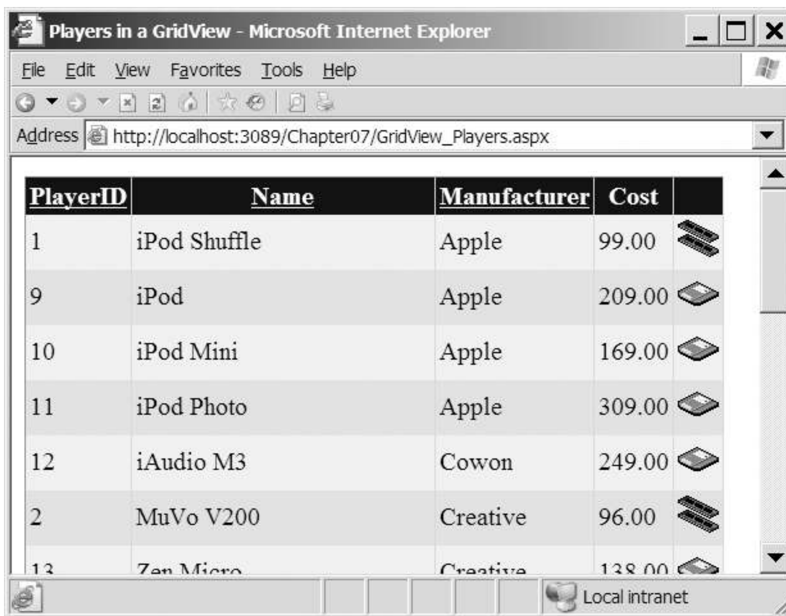


Figure 7-14. *The GridView can be sorted in ascending order.*

5. Click the same column heading again. You'll see that the results are now sorted in reverse order, as shown in Figure 7-15.

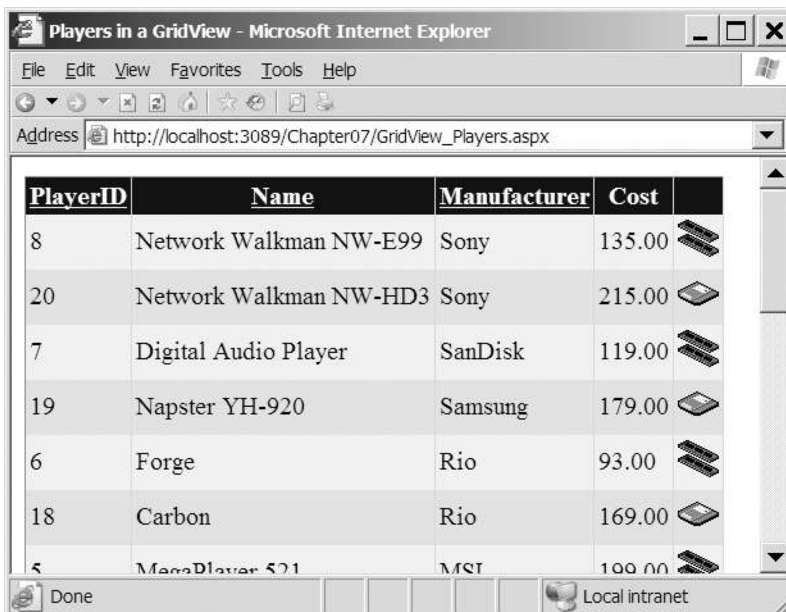


Figure 7-15. *The GridView can also be sorted in descending order.*

How It Works

Before enabling sorting on the `GridView`, you made some changes to the columns that you're displaying. In the example, you want to sort on only three of the five columns, so you add a `SortExpression` to only those columns:

```
<asp:BoundColumn DataField="PlayerID" HeaderText="PlayerID"
  SortExpression="PlayerID" />
<asp:BoundColumn DataField="PlayerName" HeaderText="Name"
  SortExpression="PlayerName" />
<asp:BoundColumn DataField="ManufacturerName"
  HeaderText="Manufacturer" SortExpression="ManufacturerName" />
```

You set the `SortExpression` to the name of the column in the database that you want to sort. If you were using column aliases (as you saw earlier), the `SortExpression` would be set to the alias rather than the actual name of the column. If you use auto-generated columns and enable sorting for the `GridView`, all of the columns that are displayed can be sorted. In this case, the `SortExpression` is added automatically as the column (or its alias) being displayed.

Clicking a column heading causes the page to post back to the server, and the results to be sorted, either in ascending or descending order. But what the code doesn't do is requery the database. As you'll recall, the `SqlDataSource`, when in `DataSet` mode, is clever enough to know that it doesn't always have to go back to the database to retrieve the results. Because the `DataSet` within the `SqlDataSource` is disconnected from the database, the sort can occur without requiring the database to be requeried.

As well as automatic sorting of the results, the `GridView` also raises two events when sorting:

- **`Sorting`**: The `Sorting` event is fired before the sort operation takes place and allows you to see which column the user wants to sort and gives you the opportunity to cancel the sort operation.
- **`Sorted`**: The `Sorted` event is fired after the sort operation has taken place.

Try It Out: Paging the GridView

In this example, you'll add paging to the displayed results.

1. Open `GridView_Players.aspx` from the previous example.
2. Switch to the Design view and select the `GridView`. From the Properties window, set the `AllowPaging` property to `True` and set the `PageSize` property to 5. As shown in Figure 7-16, the `GridView` shows that it's paging-enabled.
3. Execute the page. You'll see that the `GridView` is aware that there is more than one page and has added links for the different pages, as shown in Figure 7-17.

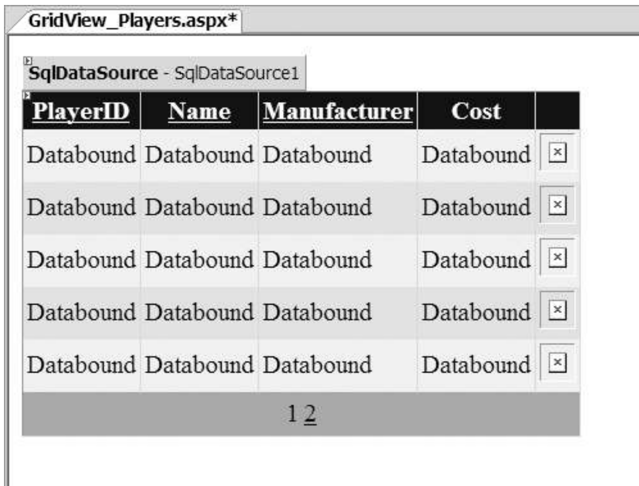


Figure 7-16. *The GridView shows that it's paging-enabled.*

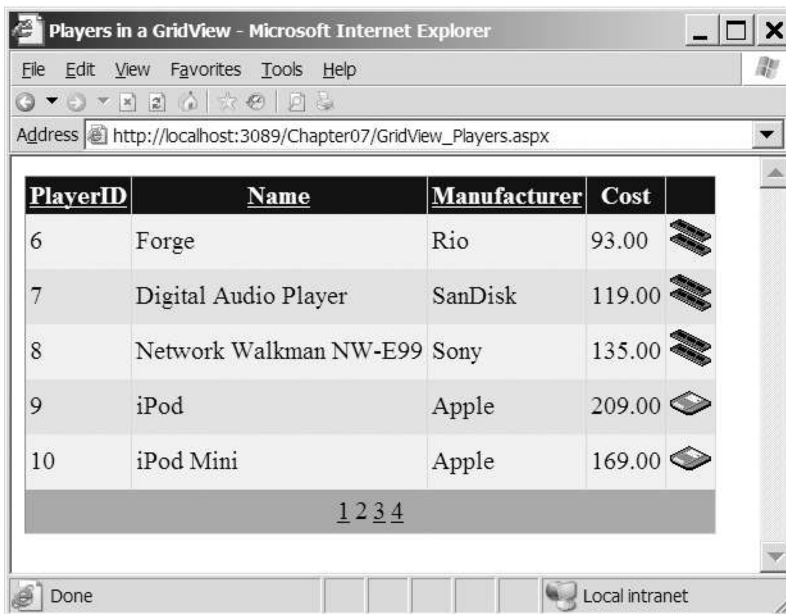


Figure 7-17. *The GridView has added links for the different pages of results.*

4. Click the link for any of the other pages, and you'll see that the correct set of results is displayed. Also try changing the sort ordering and then selecting any of the other pages to view. You'll see that the sorting and paging work in conjunction with each other.

How It Works

This example shows how easy the SqlDataSource and the GridView make it to add paging. All you need to do is set the AllowPaging property to True and the GridView will page the results automatically, showing the number of results set as the PageSize on each page.

The interesting point to note is that paging and sorting work in conjunction. If you have sorted results, then selecting a different page of results will preserve the sort order. In previous versions of ASP.NET, this functionality required quite a bit of code to make it work. Using the SqlDataSource and GridView, you get this functionality for free.

As with sorting, the GridView also raises two events when paging:

- PageIndexChanging: This event is fired before the paging operation takes place and allows you to see which page the user wants to navigate to and, if necessary, cancel the navigation.
- PageIndexChanged: This event is fired after the paging operation has taken place.

GridView Events

When we looked at the Repeater earlier in this chapter, you saw how to add buttons to the output and respond to the user clicking a button. Not surprisingly, the GridView also supports this functionality. You can respond to user events within the GridView through the following:

- ButtonField: This allows you to add a single button within a column, specify a CommandName, and then respond to the user clicking the button. You'll see how to use this type of column in the following example.
- CommandField: This is a column that you'll use only when you're allowing the user to edit the data in the GridView. The state of the GridView determines what buttons the CommandField displays. As this column is only used when editing, we'll discuss it in Chapter 9.

■ **Note** Rather than using a ButtonField or CommandField to display buttons that the user can click, you could use a Button, ImageButton, or LinkButton within a TemplateField and respond to the user clicking these buttons. The ButtonField and CommandField simply provide a shortcut way of adding the same functionality.

Try It Out: Responding to Events

In this example, you'll build a new page to show the list of Manufacturers and add a View Players button that allows the user to select a Manufacturer and view the list of Players. To filter the Players list, you'll also make a few changes to the Player list as well.

1. In Visual Web Developer, add a new Web Form called GridView_Manufacturers.aspx. Make sure that the Place Code in Separate File check box is unchecked.
2. In the Source view, find the <title> tag within the HTML at the bottom of the page and change the page title to **Manufacturers in a GridView**.

3. Switch to the Design view and add a `SqlDataSource` to the page. Select Configure Data Source from the Tasks menu.

4. Select `SqlConnectionString` as the connection string to use on the first step of the wizard. Click the Next button.

5. Select the Manufacturer table from the drop-down list. In the Columns list, select the ManufacturerID and ManufacturerName columns. Click the Next button.

6. If you want to test that the query is returning the correct results, you can do so. When you're happy that the results are correct, click the Finish button.

7. Add a `GridView` to the page. From its Tasks menu, set `SqlDataSource1` as the data source. Also open the Auto Format dialog box and select Colorful.

8. Switch to the Source view and add a `ButtonField` to the end of the `<Columns>` collection for the `GridView`:

```
<asp:ButtonField ButtonType="Link" CommandName="Players"
  Text="View Players" />
```

9. In the Design view, make sure that the `DataKeyNames` property of the `GridView` is set to ManufacturerID.

10. Add the `RowCommand` event to the `GridView`. Then add the following code to the event handler:

```
protected void GridView1_RowCommand(object source,
  GridViewCommandEventArgs e)
{
  if (e.CommandName == "Players")
  {
    int intIndex = Convert.ToInt32(e.CommandArgument);
    string strManufacuterID =
      Convert.ToString(GridView1.DataKeys[intIndex].Value);
    Response.Redirect("./GridView_Players.aspx?ManufacturerID=" +
      strManufacuterID);
  }
}
```

11. Open `GridView_Players.aspx` from the previous example.

12. Modify the `SelectCommand` property for the `SqlDataSource` as follows:

```
SELECT Player.PlayerID, Player.PlayerName,
  Manufacturer.ManufacturerName, Player.PlayerCost,
  Player.PlayerStorage FROM Player INNER JOIN Manufacturer
  ON Player.PlayerManufacturerID = Manufacturer.ManufacturerID
  WHERE Player.PlayerManufacturerID = @ManufacturerID
```

13. Add the following parameters to the SqlDataSource:

```
<SelectParameters>
  <asp:QueryStringParameter Name="ManufacturerID"
    QueryStringField="ManufacturerID" Type="Int32"  />
</SelectParameters>
```

14. Save the pages, and then view GridView_Manufacturers.aspx in your web browser. As shown in Figure 7-18, each row in the GridView now has a View Players link at the end.

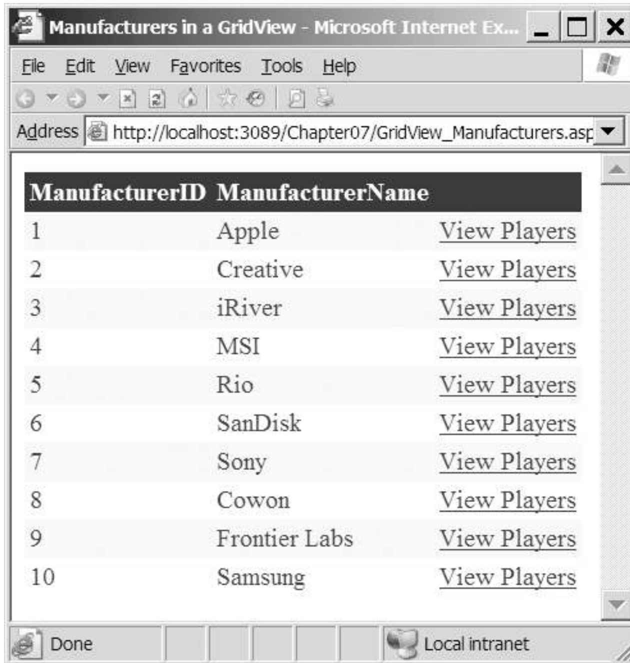


Figure 7-18. *The page now offers the ability to view the Players for a Manufacturer.*

15. Select one of the View Players links, and the filtered list of Players for the Manufacturer will be displayed. As shown in Figure 7-19, the list allows sorting of the filtered results.

Figure 7-19. *Filtered results can be sorted.*

How It Works

In this example, you've seen further evidence of how data binding the different Web controls is similar, no matter which Web controls you use. Here, you've responded to the `RowCommand` event (the `GridView` equivalent of the `ItemCommand`) to handle the user clicking a button.

To add a button to a `GridView`, you need to add a `ButtonField` to the `<Columns>` collection:

```
<asp:ButtonField ButtonType="Link" CommandName="Players"
  Text="View Players" />
```

You specify the `Text` that you want to appear on the button and a `CommandName` that you can use to determine which button was clicked. You can also specify what type of button you want to use. You've specified a `Link` in this case, but you could have used a normal button by setting the `ButtonType` property to `Button`.

The one piece of information that you're missing from this definition is a way to determine which row was clicked. If you recall from the `Repeater` example earlier, you used the `CommandArgument` property of the `LinkButton` to store the `ManufacturerID`, and you used this to filter the displayed results, but the `ButtonField` doesn't support setting the `CommandArgument` property. Instead, the row index of the selected row is automatically passed as the `CommandArgument` to the `RowSelected` event handler. You need to use that index in conjunction with the `GridView` to determine the correct ManufacturerID.

The `GridView` is capable of storing an ID value for each row by setting the `DataKeyNames` property:

```
<asp:GridView DataKeyNames="ManufacturerID">
...
</asp:GridView>
```

When the GridView is bound, all the ManufacturerID values for the rows that are displayed are stored in a collection that can be accessed via the DataKeys property of the GridView.

So, when the user clicks a button, the RowCommand event is fired. After checking that you're responding to the correct button, you retrieve the row index from the CommandArgument, and then use this to interrogate the DataKeys collection:

```
int intIndex = Convert.ToInt32(e.CommandArgument);
string strManufacturerID =
  Convert.ToString(GridView1.DataKeys[intIndex].Value);
```

The DataKeys collection is stored in the same order as the rows are displayed. You can therefore use the row's position, from the CommandArgument property, to return the correct ManufacturerID.

You cast this to a string which is then used to construct the correct URL for the Players list:

```
Response.Redirect("./GridView_Players.aspx?ManufacturerID="
  + strManufacturerID);
```

In order to filter the results, you need to make a slight change to the query that is constructed by adding a simple WHERE clause that returns only the Players where the ManufacturerID matches the value passed as part of the query string:

```
SELECT Player.PlayerID, Player.PlayerName,
  Manufacturer.ManufacturerName, Player.PlayerCost,
  Player.PlayerStorage FROM Player INNER JOIN Manufacturer
  ON Player.PlayerManufacturerID = Manufacturer.ManufacturerID
  WHERE Player.PlayerManufacturerID = @ManufacturerID
```

You pass the correct value for ManufacturerID to the SqlDataSource by using a QueryStringParameter object as part of the SelectParameters collection:

```
  <asp:QueryStringParameter Name="ManufacturerID"
    QueryStringField="ManufacturerID" Type="Int32"  />
```

Note that the paging and sorting of the Players list still works, even though you have a filtered result set. Because the DataSet inside the SqlDataSource is populated when the page first loads with the filtered results, any sorting or paging applied to the SqlDataSource by the GridView works against the filtered results.

DataSet vs. DataReader

Now, let's quickly look at how the DataSet and DataReader compare across the techniques demonstrated in this and the previous chapters. So, before you learn how to update a data source, you'll take stock and compare DataReader and DataSet. Each has its own advantages and disadvantages, so it's pretty key to make the right choice for each page. This is true whether you're using code to access the database or using a SqlDataSource—you have the choice to use a DataReader or DataSet in both cases. Table 7-8 outlines the main differences.

Table 7-8. *DataSet vs. DataReader for Displaying Data*

DataReader	DataSet
The DataReader is "connected" to the data source at all times.	The DataSet, once populated, is no longer connected to the data source. This allows the DataSet to be persisted between postbacks, reducing the usage of the data source.
The data retrieved through a DataReader is forward-only and must be cycled through by calling Read(). Once the data has been cycled through, the DataReader must be closed and re-created to reaccess the data.	You can work with data in a DataSet in any order you choose as many times as you like.
Binding expressions can use either the index number of the column in the DataReader or the column name itself. Thus, they're quite simple to write and use.	Columns in a DataSet can be referenced by name, but you must also name the DataTable and identify the row that contains the column. This means more complex binding expressions.
Data retrieved through a DataReader must be requeried for after a postback.	A DataSet may be stored during postbacks if it isn't too great a drain on resources.
A DataReader connects to only one data source.	A DataSet can be filled using Fill() from multiple data sources and isn't tied to a particular data source. You can even create a DataSet manually without ever using a data source, as you saw in Chapter 5.
A DataReader is most suitable for quickly binding data to Web controls.	A DataSet is most suitable for working with complex data and data that needs to be updated on a page.
You can bind only one list or table to a DataReader. It must be re-created for other Web controls on the same page.	You can bind as many list or table Web controls to a single DataSet as you want.

Summary

In this chapter, you learned how to bind the query results to a table-based Web control such as a GridView, DataList, or Repeater. Unless you tell it otherwise, the GridView will present the results in a preformatted grid, one column per table cell; the DataList and Repeater, on the other hand, must be given a template for each row of information to be displayed.

You learned that you can customize the GridView quite heavily, even when it auto-generates a grid to display query results. You can use SQL to make the data more readable, use styles to make it more attractive, and implement simple sorting and paging functions to improve the way that users can view the results.

In the next chapter, you'll finish your look at handling the data from a query by exposing it as read-write data. You'll also learn how to send the changes made to that data back to the database.