3

Customizing the HTML Output

IN THIS CHAPTER

- Specifying Display Properties in the DataGrid and DataList
- Customizing Data-Binding Output in Templates
- On the Web

We have used the data Web controls a number of times in code examples in the past two chapters. However, the appearance of the output has left a lot to be desired. Fortunately, making the DataGrid and DataList output more visually pleasing is quite simple, even for artistically challenged developers like myself!

As we will see in this chapter, both the DataGrid and DataList expose a number of properties that make specifying these details a breeze. Additionally, we'll look at how editors like Visual Studio .NET and the Web Matrix Project make specifying the appearance of the DataGrid and DataList as easy clicking a few buttons. (Note that the Repeater control does not contain any sort of stylistic properties; the developer is responsible for specifying any aesthetic properties directly in the HTML markup of the Repeater's templates.)

In this chapter we will also look at how to use built-in methods and custom functions to alter the output of data binding expressions in templates. For example, in Chapter 2's Listing 2.5, the `String.Format()` method is used within the data-binding expression of an ItemTemplate to have the number returned by `Container.DataItem` formatted with commas every three digits (line 17). (That is, 17711 [the 22nd Fibonacci number] is displayed as 17,711.) We'll examine how to use this approach to further customize the HTML emitted by data-bound values in a template.

By the end of this chapter, you will be able to generate professional-looking DataGrids and DataLists with just the setting of a few properties, or if you're using Visual Studio .NET or the Web Matrix Project, with simply the click of the mouse.

Specifying Display Properties in the DataGrid and DataList

Although the code examples we've studied thus far have been very useful in their structured display of data, they have been far from eye-pleasing. When viewing the screenshots of the code examples, did you ever think, "I wonder how I can change the font?" or "Wouldn't it look nicer if the cells in the DataGrid header were center-aligned?" If you did, you'll be pleased to learn that specifying such stylistic properties for the DataGrid and DataList is quite simple, as we will see in this section.

Recall that the DataGrid and DataList controls render into HTML `table` tags. When you think of a table, there are three levels to which stylistic formatting can be applied.

First, formatting can be applied to the entire table. This might include setting the font for the entire table to Verdana, or specifying the CellPadding or CellSpacing for the table.

NOTE

CellPadding and CellSpacing are two stylistic properties of an HTML table. The CellPadding specifies the number of pixels between the cell's border and the cell's textual content. The CellSpacing specifies, in pixels, the spacing between neighboring table cells.

Second, formatting can be applied at the table row level. This row-level formatting can be specified for all rows in the table; for example, you might want all the rows to have a certain background color. Of course, setting this property for all the rows is synonymous with setting the property for the entire table.

You can also specify row-level formatting for just a subset of the rows. For example, you might opt for the rows' background to alternate between two colors. (That is, each even row might have a white background, and each odd row might have a light gray background.) Additionally, you can use row-level formatting to specify formatting for a single row. With the DataGrid and DataList, you can select a row and edit its data. You might want to use an alternative font for the row that is being edited. (We'll look at how to edit data in Chapter 9, "Editing the DataGrid Web Control.") In a similar vein, you might want to have a different style for the header and footer rows of the table.

The third level of stylistic formatting can be applied to the column level. Recall that the DataGrid has a `Columns` tag in which you can explicitly specify what columns should appear in the DataGrid control. Not surprisingly, you can apply stylistic settings to these columns. For example, you might want to have certain columns center- or right-aligned.

In this section, we will look at each of these levels of formatting separately and how to apply them programmatically. After this, we will look at specifying this formatting

information using Visual Studio .NET and the Web Matrix Project. As we will see, specifying this property information with these tools is quite simple, requiring just a few clicks of the mouse.

Specifying Table-Level Display Properties

The DataGrid and DataList contain a number of table-level properties. Table 3.1 lists some of the more useful ones, along with a short description of each.

TABLE 3.1 Common Table-Level Properties for DataGrid and DataList

Property	Description
`BackColor`	The background color of the table.
`BorderColor`	The color of the table border.
`BorderStyle`	The border style. Must be set to a member of the `BorderStyle` enumeration.
`BorderWidth`	The width of the border.
`CellPadding`	The `cellpadding` attribute of the `table` tag.
`CellSpacing`	The `cellspacing` attribute of the `table` tag.
`CssClass`	A cascading stylesheet class for the table that specifies style information.
`Font`	Font information for the table. Specifies the font's name, size, and style options (bolded, italicized, underlined, and so on).
`ForeColor`	The foreground color of the table.
`Height`	The height of the table, in either pixels or percentages.
`HorizontalAlign`	The horizontal alignment of the table (`Left`, `Right`, or `Center`).
`Width`	The width of the table, in either pixels or percentages.

These stylistic properties can be assigned programmatically, in the ASP.NET Web page's server-side script block or code-behind page, or declaratively, in the control's definition in the HTML section of the ASP.NET Web page.

Listing 3.1 illustrates the setting of a DataGrid's stylistic properties both programmatically and declaratively. Note that the majority of the contents of Listing 3.1 are identical to the code in Listing 2.1. The only differences are the DataGrid declaration on lines 30 and 31 and lines 23–25, where two display properties of the DataGrid are set programmatically.

LISTING 3.1 A DataGrid's Display Properties Are Set Declaratively and Programmatically

```
1: <%@ import Namespace="System.Data" %>
2: <%@ import Namespace="System.Data.SqlClient" %>
3: <script runat="server" language="VB">
4:
5:   Sub Page_Load(sender as Object, e as EventArgs)
6:    '1. Create a connection
```

LISTING 3.1 Continued

```
 7:    Const strConnString as String = "server=localhost;uid=sa;pwd=;
➥database=pubs"
 8:    Dim objConn as New SqlConnection(strConnString)
 9:
10:    '2. Create a command object for the query
11:    Const strSQL as String = "SELECT * FROM authors"
12:    Dim objCmd as New SqlCommand(strSQL, objConn)
13:
14:    objConn.Open()  'Open the connection
15:
16:    'Finally, specify the DataSource and call DataBind()
17:    dgAuthors.DataSource = objCmd.ExecuteReader(CommandBehavior.
➥CloseConnection)
18:    dgAuthors.DataBind()
19:
20:    objConn.Close()  'Close the connection
21:
22:
23:    ' Set some DataGrid display properties programmatically
24:    dgAuthors.HorizontalAlign = HorizontalAlign.Center
25:    dgAuthors.Font.Bold = True
26:   End Sub
27:
28: </script>
29:
30: <asp:datagrid id="dgAuthors" runat="server" Font-Name="Verdana"
31:   Width="50%" />
```

Note that the DataGrid in Listing 3.1 has its display properties specified both programmatically (lines 24 and 25) and declaratively (lines 30 and 31). In the declarative section, we specify that the Verdana font should be used, and that the table should have a width of 50%. And in the `Page_Load` event handler, we specify that the table should be center-aligned and have a bold font.

One thing to note is the syntax used when specifying the font name. The `Font` property of the DataGrid (and DataList) is an instance of the `System.Web.UI.WebControls.FontInfo` class, which contains a number of properties, such as `Name`, `Bold`, `Italic`, `Size`, and so on. Therefore, we don't want to assign `"Verdana"` to the `Font` property, but instead to the `Font.Name` property. However, this dot notation, which we use when specifying a property programmatically (see line 25), does not work when setting a property declaratively. Rather, we have to replace all dots with

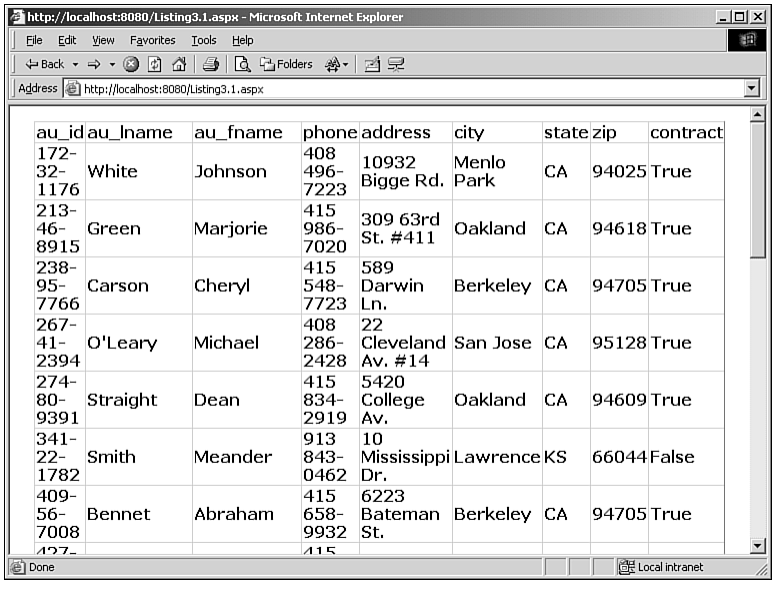
dashes. Hence, instead of saying `Font.Name = "Verdana"` we do `Font-Name="Verdana"` (line 30).

On lines 24 and 25 we set two properties declaratively. The first is the `HorizontalAlign` property, which specifies that the table should be center-aligned. Note that the `HorizontalAlign` property needs to be assigned a value from the `HorizontalAlign` enumeration, so we assign it to `HorizontalAlign.Center` (line 24). If we were setting this property declaratively, however, we could omit the enumeration name, and just use `HorizontalAlign="Center"`. On line 25 we specify that the contents of the table should be displayed in bold.

NOTE

Had we used a DataList instead of a DataGrid in Listing 3.1, neither the programmatic nor declarative property settings would need to be changed in any way.

Figure 3.1 depicts a screenshot of Listing 3.1 when viewed through a browser.



au_id	au_lname	au_fname	phone	address	city	state	zip	contract
172-32-1176	White	Johnson	408 496-7223	10932 Bigge Rd.	Menlo Park	CA	94025	True
213-46-8915	Green	Marjorie	415 986-7020	309 63rd St. #411	Oakland	CA	94618	True
238-95-7766	Carson	Cheryl	415 548-7723	589 Darwin Ln.	Berkeley	CA	94705	True
267-41-2394	O'Leary	Michael	408 286-2428	22 Cleveland Av. #14	San Jose	CA	95128	True
274-80-9391	Straight	Dean	415 834-2919	5420 College Av.	Oakland	CA	94609	True
341-22-1782	Smith	Meander	913 843-0462	10 Mississippi Dr.	Lawrence	KS	66044	False
409-56-7008	Bennet	Abraham	415 658-9932	6223 Bateman St.	Berkeley	CA	94705	True

FIGURE 3.1 A centered table displayed in bold with a width of 50% is rendered.

Specifying Row-Level Display Properties

In the previous section, we noted a number of display properties that apply to the entire table rendered by the DataGrid or DataList. Most of these properties can also be applied at the row level, as we will see shortly.

To enable the developer to set row-level properties, the DataList and DataGrid provide a number of "ItemStyle" properties. These properties are called "ItemStyle" properties because they specify style information for the `DataGridItem` and `DataListItem` controls, which, as you'll recall from the previous chapter, are rendered as rows in the resulting table. The "ItemStyle" properties include the following:

- `ItemStyle`
- `AlternatingItemStyle`
- `EditItemStyle`
- `SelectedItemStyle`
- `HeaderStyle`
- `FooterStyle`

The last four "ItemStyle" properties—`EditItemStyle`, `SelectedItemStyle`, `HeaderItemStyle`, and `FooterItemStyle`—specify row-level styles for exactly one row. For example, the `HeaderItemStyle` specifies style information for the header row.

On the other hand, the `ItemStyle` and `AlternatingItemStyle` properties specify style settings for multiple rows. If the `AlternatingItemStyle` style is specified, it is applied to each alternating row in the table. If it is not specified, the `ItemStyle` is applied to all rows in the table.

Each of these "ItemStyle" properties is defined as a `TableItemStyle` instance. Table 3.2 shows the germane display properties of the `TableItemStyle` class.

TABLE 3.2 Common `TableItemStyle` Display Properties, Which Can Be Used to Specify Row-Level Display Properties

Property	Description
`BackColor`	The background color of the table row.
`BorderColor`	The color of the table row's border.
`BorderStyle`	The border style. Must be set to a member of the `BorderStyle` enumeration.
`BorderWidth`	The width of the border.
`CssClass`	A cascading stylesheet class for the table that specifies style information.
`Font`	Font information for the table row. Specifies the font's name, size, and style options (bolded, italicized, underlined, and so on).
`ForeColor`	The foreground color of the table row.
`Height`	The height of the table row, in either pixels or percentages.
`HorizontalAlign`	The horizontal alignment of the table row (`Left`, `Right`, or `Center`).
`VerticalAlign`	The vertical alignment of the table row (`Top`, `Middle`, or `Bottom`).
`Width`	The width of the table row, in either pixels or percentages.
`Wrap`	Specifies whether the contents in the cell should wrap—defaults to `True`.

Comparing Table 3.2 to Table 3.1, you can see that Table 3.2 does not include properties like `CellPadding` and `CellSpacing`, which are clearly table-level display settings. Furthermore, Table 3.1 does not contain the obvious row-level display settings, such as `VerticalAlign` and `Wrap`. The overlapping display properties, though, are common to both Tables 3.1 and 3.2.

As with the table-level display properties, the row-level display properties can be set both declaratively and programmatically. In addition to being set declaratively, the "ItemStyle" properties can also be set using embedded tags inside the DataGrid or DataList tag, as shown in Listing 3.2 on lines 9 and 10. The remaining source code has been omitted from Listing 3.2; essentially it is identical to the source code in Listing 3.1, with lines 22–25 from Listing 3.1 omitted. Figure 3.2 shows a screenshot of Listing 3.2 when viewed through a browser.

LISTING 3.2 A DataList Is Used to Display Mailing Labels for the Various Authors

```
 1: <asp:datalist id="dlAuthorMailLabels" runat="server" Font-Name="Verdana"
 2:   HorizontalAlign="Center"
 3:   CellSpacing="10"
 4:
 5:   ItemStyle-Font-Size="8"
 6:
 7:   AlternatingItemStyle-BackColor="#dddddd">
 8:
 9:   <HeaderStyle HorizontalAlign="Center" Font-Bold="True"
10:          BackColor="Blue" ForeColor="White" />
11:
12:   <HeaderTemplate>
13:    Mailing Labels
14:   </HeaderTemplate>
15:
16:   <ItemTemplate>
17:    <%# DataBinder.Eval(Container.DataItem, "au_fname") %>
18:    <%# DataBinder.Eval(Container.DataItem, "au_lname") %>
19:    <br />
20:    <%# DataBinder.Eval(Container.DataItem, "address") %>
21:    <br />
22:    <%# DataBinder.Eval(Container.DataItem, "city") %>,
23:    <%# DataBinder.Eval(Container.DataItem, "state") %>
24:         
25:    <%# DataBinder.Eval(Container.DataItem, "zip") %>
26:   </ItemTemplate>
27: </asp:datalist>
```

FIGURE 3.2 The authors' contact information is presented as a list of mailing labels.

As mentioned earlier, one of the "ItemStyles" is the `HeaderStyle`, which specifies the display properties for the header. With the DataGrid, the header is automatically generated, containing the name of the columns. With the DataList, we have to supply our own header using the HeaderTemplate. In Listing 3.2, we simply give a title to our table in the HeaderTemplate (lines 12–14).

NOTE

The *header* is the first row displayed in a data Web control, and typically contains a title for each column of data.

On lines 9 and 10, the `HeaderStyle` properties are set. Note that with the "ItemStyle" properties, we can use an alternative form of declaratively defining the properties. Simply put, we use a tag within the DataList (or DataGrid) tag. The tag name is the name of the "ItemStyle" property that we want to work with. We can then set the properties of that "ItemStyle" property by specifying them as attributes in the tag. For example, on line 10, we set the `BackColor` and `ForeColor` of the `HeaderStyle` to `Blue` and `White`, respectively.

On line 5 we set the `ItemStyle`'s `Font` property's `Size` object to 8 point. On line 7 we set the `AlternatingItemStyle`'s `BackColor` property to `#dddddd`, which is a light gray.

This has the effect of alternating the background color between white and light gray for each alternating row in the DataList.

The ItemTemplate in Listing 3.2 (lines 17–25) is fairly straightforward, and is intended to display a list that can be used for mailing labels. First, the author's first and last names are shown, then a break (`<br />`), then the author's address, then another break, and finally the author's city, state, and zip code. Note that the ` ` is HTML markup to display nonbreaking whitespace. The five instances of ` ` (line 24) set the zip code five spaces from the end of the state in the city, state, and zip code line.

The code in Listing 3.2 does not demonstrate how to set "ItemStyle" properties programmatically. However, it is likely that your intuition can tell you the syntax. For example, to set the `HeaderStyle`'s `HorizontalAlign` property to center, we could use

```
dlAuthorMailLabels.HeaderStyle.HorizontalAlign = HorizontalAlign.Center
```

To set the `ItemStyle`'s `Font`'s `Size` to 8 point, we could use

```
dlAuthorMailLabels.ItemStyle.Font.Size = FontUnit.Point(8)
```

A bit more care must be taken when setting the property of a data Web control programmatically versus setting it declaratively. When setting the font size declaratively, we simply use

```
<asp:DataList Font-Size="8pt" ... />
```

Notice that we essentially set the DataList's `Font.Size` property to the string `"8pt"`. When setting the property programmatically, however, we just assign the `Font.Size` property to a string. That is, if we tried to set the DataList's `Font.Size` property to `"8pt"` using

```
dlAuthorMailLabels.ItemStyle.Font.Size = "8pt"
```

we'd get a compile-time error because of the mismatched types. That is, the `Size` property of the DataList's `Font` property expects a value of type `System.Web.UI.WebControls.FontUnit`, not of type string. That is why, in our earlier example, we set the DataList's `Font.Size` property to the `FontUnit` instance returned by the `FontUnit.Point` method:

```
dlAuthorMailLabels.ItemStyle.Font.Size = FontUnit.Point(8)
```

You might be wondering how I knew that the DataList's `Font` property's `Size` property was expecting a value of type `FontUnit` and not string. If you are given an expression like

```
dlAuthorMailLabels.ItemStyle.Font.Size
```

and you need to determine the type of that expression, simply start with the leftmost object and work your way to the right end. That is, the leftmost object, `dlAuthorMailLabels`, is a DataList object. It has an `ItemStyle` property, which is the second leftmost object. This `ItemStyle` property is of type `TableItemStyle`, as we discussed eariler. The `TableItemStyle` class has a property `Font`, which is of type `FontInfo`. The `FontInfo` class has a property `Size`, which is of type `FontUnit`. Hence, the type of the entire expression

```
dlAuthorMailLabels.ItemStyle.Font.Size
```

is `FontUnit`, meaning we must assign a value of type `FontUnit` to this expression.

NOTE

Given this information, you might now be wondering why the `Font.Size` property can be assigned a string in the declarative form. That is, given what we just discussed, why doesn't

```
<asp:DataList Font-Size="8pt" ... />
```

generate a type mismatch compile-time error? The reason is because, behind the scenes, the string `"8pt."` is being automatically converted into an appropriate `FontUnit` instance.

Specifying Column-Level Display Properties

As we've seen in previous chapters, the DataGrid control, by default, creates an HTML table with as many rows and columns as there are in the `DataSource`. However, by setting the `AutoGenerateColumns` property to `False`, you can explicitly specify what fields from the `DataSource` should be included in the DataGrid. Recall that to add a column to the DataGrid you must add an appropriate control in the DataGrid's `Columns` tag. That is, using our authors example, a DataGrid that had columns for the author's first and last name might look like this:

```
<asp:DataGrid runat="server" id="dgAuthorsNames">
 <Columns>
  <asp:BoundColumn DataField="au_fname" HeaderText="First Name" />
  <asp:BoundColumn DataField="au_lname" HeaderText="Last Name" />
 </Columns>
</asp:DataGrid>
```

Recall that the following controls can be used within the `Columns` tag:

- BoundColumn
- TemplateColumn
- EditColumn

- ButtonColumn
- HyperLinkColumn

Not surprisingly, you can specify various display properties for each of these column controls. In this chapter, we'll look at customizing the stylistic properties for the BoundColumn and TemplateColumn controls. For now we'll omit examination of EditColumn, ButtonColumn, and HyperLinkColumn, because we'll be seeing much more of these controls in later chapters.

Specifying Display Properties for the BoundColumn Control

The BoundColumn control contains three "ItemStyle" properties: `ItemStyle`, `HeaderStyle`, and `FooterStyle`. These "ItemStyle" properties indicate style information for the particular column. They contain the same set of display properties as the "ItemStyle" properties examined in the previous section.

In addition to these "ItemStyle" properties, the BoundColumn control contains a few other display properties. One we've seen in past examples is the `HeaderText` property, which specifies the text that should appear in the header of the column. This is useful if your `DataSource` has obscure field names (such as `au_lname`). A `FooterText` property is also included. The BoundColumn control also contains a `HeaderImageUrl` property, which, if specified, will place an `img` tag in the header.

The final display property for the BoundColumn control is the `DataFormatString` property. This string property specifies how the data in the column should be formatted. The syntax for the format string is a bit strange at first glance: `{0:`*`formatString`*`}`. This is the same syntax used for the `String.Format()` method, which we saw in a code example at the end of Chapter 2.

The `String.Format()` method accepts a string as its first parameter, and then accepts a variable number of parameters of type `Object`. The string parameter is a format string, which can contain a mix of normal text and format placeholders. The objects are the variables whose values we want to have plugged into the formatting placeholders.

For example, imagine that we have two variables of type `float` in C#, or of type `Single` in Visual Basic .NET, called `basePrice` and `salesTax`. The `basePrice` variable contains the sum of the prices for the products ordered by our customer, whereas the `salesTax` price is the sales tax on the customer's purchase; hence the total price due is the sum of these two variables.

Now, it would be nice to display a message indicating the customer's total, and how it was derived. We might want to have an output like: "Your total comes to $6.50. The base price was $5.75 and the sales tax was $0.75." To accomplish this, we could use the `String.Format` method like so:

```
String s = String.Empty;
String strFormat = "Your total comes to {0:c}. The base price was {1:c} " +
          "and the sales tax was {2:c}.";
s = String.Format(strFormat, basePrice+salesTax, basePrice, salesTax)
```

Note that the formatting placeholders (`{0:c}`, `{1:c}`, and `{2:c}`) indicate two things: the ordinal ranking of the variable whose value should be used when applying the formatting information, and the formatting to use. That is, `{0:c}` will apply a currency formatting to the 0th parameter (`basePrice+salesTax`)—note that `c` represents a currency formatting. `{1:c}` will apply a currency formatting to the 1st parameter, `basePrice` (really the 2nd one, because we start counting at 0). Finally, `{2:c}` will apply a currency formatting to the 2nd parameter, `salesTax`.

The `DataStringFormat` property of the BoundColumn control works in a similar fashion: Here you must specify the formatting placeholder. Keep in mind that the value that is inserted into the cell from the `DataSource` is the 0th parameter. Hence, if you have a field that stores, say, the price of each product, you can have it displayed as a currency by setting the BoundColumn control's `DataFormatString` to `{0:c}`. That is, the DataGrid might look like this:

```
<asp:DataGrid runat="server" id="dgAuthorsNames">
 <Columns>
  <asp:BoundColumn DataField="Name" HeaderText="Product Name" />
  <asp:BoundColumn DataField="Price" HeaderText="Price"
           DataFormatString="{0:c}" />
 </Columns>
</asp:DataGrid>
```

Clearly the `c` format string formats the specified variable value as a currency, but what other format strings are available? Table 3.3 lists a few of the more common ones, but there are far too many possibilities to list here. Instead, check out Microsoft's Formatting Overview page at `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconformattingoverview.asp`.

TABLE 3.3 Common Formatting Strings

Formatting String	Effect
c	Displays numeric values in currency format
d	Displays numeric values in decimal format and date/time variables in a short date pattern
x	Displays numeric values in hexadecimal format

Listing 3.3 illustrates setting a number of display properties for the BoundColumn controls. The source code is omitted from the listing, but it is the same as the source code from Listing 3.1, except that the SQL query on line 11 has been changed from `SELECT * FROM authors` to `SELECT * FROM titles`. Hence, the code in Listing 3.3 presents information about the various books in the database.

LISTING 3.3 Column-Level Display Properties Can Be Set via the BoundColumn Control

```
 1: <asp:datagrid id="dgTitles" runat="server"
 2:    AutoGenerateColumns="False"
 3:    Font-Name="Verdana" Width="50%"
 4:    HorizontalAlign="Center" ItemStyle-Font-Size="9">
 5:
 6:  <HeaderStyle BackColor="Navy" ForeColor="White"
 7:     HorizontalAlign="Center" Font-Bold="True" />
 8:
 9:  <AlternatingItemStyle BackColor="#dddddd" />
10:
11:  <Columns>
12:   <asp:BoundColumn DataField="title" HeaderText="Title"
13:            ItemStyle-Width="70%" />
14:
15:   <asp:BoundColumn DataField="price" HeaderText="Price"
16:            DataFormatString="{0:c}"
17:            ItemStyle-HorizontalAlign="Right"
18:            ItemStyle-Width="15%" />
19:
20:   <asp:BoundColumn DataField="ytd_sales" HeaderText="Sales"
21:            DataFormatString="{0:#,###}"
22:            ItemStyle-HorizontalAlign="Right"
23:            ItemStyle-Width="15%" />
24:  </Columns>
25: </asp:datagrid>
```

In Listing 3.3, there are three BoundColumns presenting key information from the `titles` table in the `pubs` database. The first BoundColumn (lines 12 and 13) simply shows the book's title. On line 13, the `Width` of the column is set to 70% via the `ItemStyle` property. Because we are working with column-level display properties, this width setting is relative to the table width, meaning the first column will be 70% of the total width of the table. Note that in line 3 we specified that the table should have a `Width` of 50% (this is relative to the Web page).

The second BoundColumn (lines 15–18) displays the price of the book. Line 16 sets the `DataFormatString` so that the value will be formatted as a currency. On line 17, the column values are specified to be right-aligned, and on line 18, the `Width` of the column is set to 15% (of the total width of the table).

The last BoundColumn (lines 20–23) displays the year-to-date sales. The `DataFormatString` on line 21 indicates that there should be a comma separating each of the three digits. The `HorizontalAlign` and `Width` properties on lines 22 and 23 mirror those on lines 17 and 18.

Figure 3.3 shows a screenshot of Listing 3.3 when viewed through a browser.

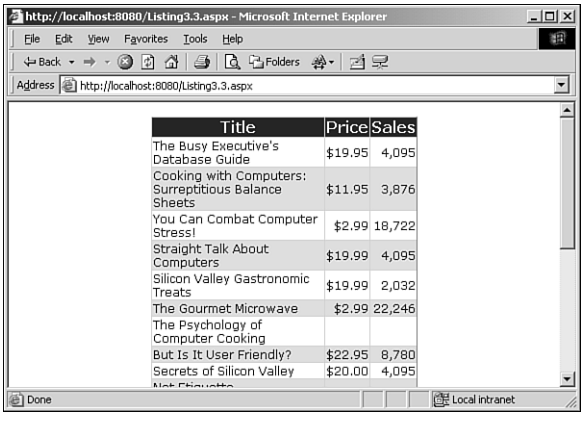


FIGURE 3.3 Each column has specific stylistic properties set, such as formatting and alignment.

Specifying Display Properties for the TemplateColumn

Recall that the DataGrid can employ templates through the use of the TemplateColumn control. This control applies a template to a specific column in the DataGrid, as shown in the following code snippet:

```
<asp:DataGrid runat="server" id="dgTemplateColumnExample">
 <Columns>
  <asp:TemplateColumn>
   <ItemTemplate>
    <%# DataBinder.Eval(Container.DataItem, "SomeColumnName") %>
   </ItemTemplate>
  </asp:TemplateColumn>
 </Columns>
<asp:DataGrid>
```

The display properties of the TemplateColumn are a subset of those of the BoundColumn. Specifically, the TemplateColumn's display properties include the following:

- `HeaderText`
- `HeaderImageUrl`
- `FooterText`
- `ItemStyle`
- `HeaderStyle`
- `FooterStyle`

With the TemplateColumn, you have more control over the display in the header and footer through the use of the HeaderTemplate and FooterTemplate.

Let's look at some code. Listing 3.4 contains a DataGrid that uses the same source code as Listing 3.3—it populates a `SqlDataReader` with the query `SELECT * FROM titles` and binds it to the DataGrid `dgTitles`. As with Listing 3.3, the source code has been removed from Listing 3.4 for brevity.

In Listing 3.4, a DataGrid with a BoundColumn control and a TemplateColumn control is created. The BoundColumn control displays the title of the book, and the TemplateColumn control displays information on the book's sales performance—essentially a simple sentence noting how many copies have been sold at what price. Three of the TemplateColumn control's display properties—`HeaderText` (line 15), and two `ItemStyle` properties (`HorizontalAlign` and `Wrap` on lines 16 and 17, respectively)—have been set declaratively.

LISTING 3.4 The TemplateColumn Control Contains Display Properties Nearly Identical to That of the BoundColumn Control

```
 1: <asp:datagrid id="dgTitles" runat="server"
 2:    AutoGenerateColumns="False"
 3:    Font-Name="Verdana" Width="50%"
 4:    HorizontalAlign="Center" ItemStyle-Font-Size="9">
 5:
 6:  <HeaderStyle BackColor="Navy" ForeColor="White"
 7:     HorizontalAlign="Center" Font-Bold="True" />
 8:
 9:  <AlternatingItemStyle BackColor="#dddddd" />
10:
11:  <Columns>
12:   <asp:BoundColumn DataField="title" HeaderText="Title"
```

LISTING 3.4 Continued

```
13:            ItemStyle-Width="70%" />
14:
15:   <asp:TemplateColumn HeaderText="Sales Performance"
16:      ItemStyle-HorizontalAlign="Right"
17:      ItemStyle-Wrap="False">
18:    <ItemTemplate>
19:      <b><%# DataBinder.Eval(Container.DataItem, "ytd_sales", "{0:#,###}")
➥%></b>
20:      copies sold at
21:      <b><%# DataBinder.Eval(Container.DataItem, "price", "{0:c}") %></b>
22:    </ItemTemplate>
23:   </asp:TemplateColumn>
24:  </Columns>
25: </asp:datagrid>
```

Notice that we set the `ItemStyle.Wrap` property to `False` on line 17. This means that the text in this column will not wrap. Because the overall table width is defined to be 50% of the page (see the `Width` property on line 3), if we shrink the browser or lower the monitor resolution, the second column's text will remain all on one line, and the Title column's text will start wrapping.

Also notice that the `DataBinder.Eval()` method call on lines 19 and 21 uses *three* parameters instead of the two we've seen thus far. The third parameter specifies a format string (which follows the same guidelines and rules as the format string in the `String.Format()` method or the BoundColumn control's `DataFormatString` property. Hence, on lines 19 and 21 the value of the `ytd_sales` and `price` fields are formatted as currency (recall that the `c` applies a currency format).

Figure 3.4 shows a screenshot of the code in Listing 3.4 when viewed through a browser. Note that the rows that have `NULL`s for `price` and `ytd_sales` (*The Psychology of Computer Cooking* and *Net Etiquette*) are missing the numeric values for the sales number and sales price, but still have the string "copies sold at" present. Ideally, we don't want this string to display fields that have `NULL` values for these two database fields; perhaps we'd like to have a message like "Currently no copies of this book have been sold" appear instead. We'll see how to accomplish this later in this chapter, in the section "Customizing Data-Binding Output in Templates."

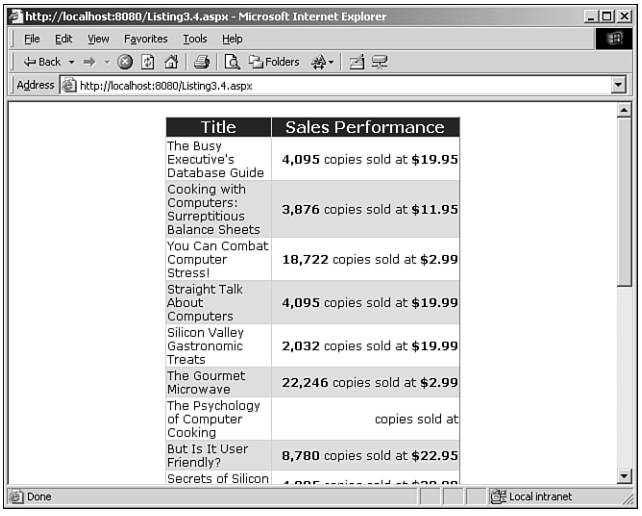


FIGURE 3.4 A TemplateColumn is used to display the sales and price for each book.

Using Developer Tools to Specify Display Properties

Hopefully you've found that specifying display properties for the DataGrid and DataList controls is relatively easy. Unfortunately, for people like myself who are anything but artistic, getting an aesthetically pleasing DataGrid or DataList can still pose a bit of a challenge. I find that while setting the display properties might be easy enough, knowing what colors go well together and what fonts and font settings to use to create eye-pleasing output can be quite a challenge.

If your artistic skills are at all like mine, you'll find developer tools like Visual Studio .NET and the Web Matrix Project to be indispensable.

When creating ASP.NET Web pages with either of these tools, you can enter what is called the Design view, which presents a WYSIWYG view of the ASP.NET Web page called the Designer. From this Design view, you'll be able to drag and drop Web controls that are shown in the toolbox on the left.

After you drag a control onto the Designer you can click on the control, and a list of its properties should appear in the bottom right-hand corner. Figure 3.5 shows a screenshot of an ASP.NET Web page in the Web Matrix Project after a DataGrid has been dragged onto the Designer. You can specify various display properties from the properties window in the bottom right-hand corner.

NOTE

Although the following screenshots show the Web Matrix Project in use, Visual Studio .NET provides an identical user experience when it comes to working with the Designer.

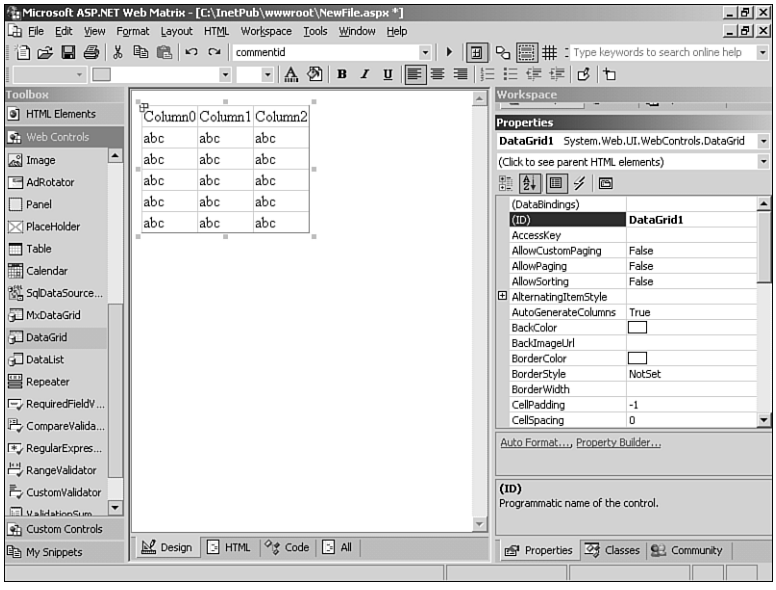


FIGURE 3.5 You can set the DataGrid's display properties through the Properties window.

Although the Properties window is a nice way to quickly set display properties, a real gem for artistically challenged individuals is hidden in the Auto Format link at the bottom of the Properties window. After clicking on the Auto Format link, a dialog box appears with a listing of various style schemes and a preview of the style scheme (see Figure 3.6).

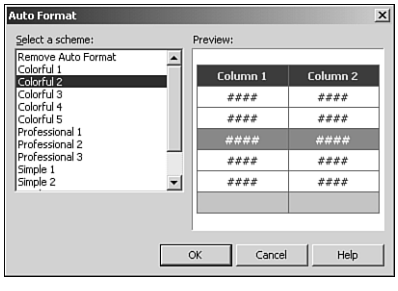


FIGURE 3.6 The Auto Format dialog box allows you to quickly choose a style scheme for your DataGrid or DataList control.

After you have selected a style scheme and clicked the OK button, the Designer is updated to show the new look and feel of the style scheme you have chosen (see Figure 3.7). This also has the effect of updating the DataGrid (or DataList) control, automatically specifying the style properties. For example, Listing 3.5 contains the control markup automatically added to the ASP.NET Web page by simply dragging a DataGrid onto the Designer and opting for the Colorful 5 style scheme.

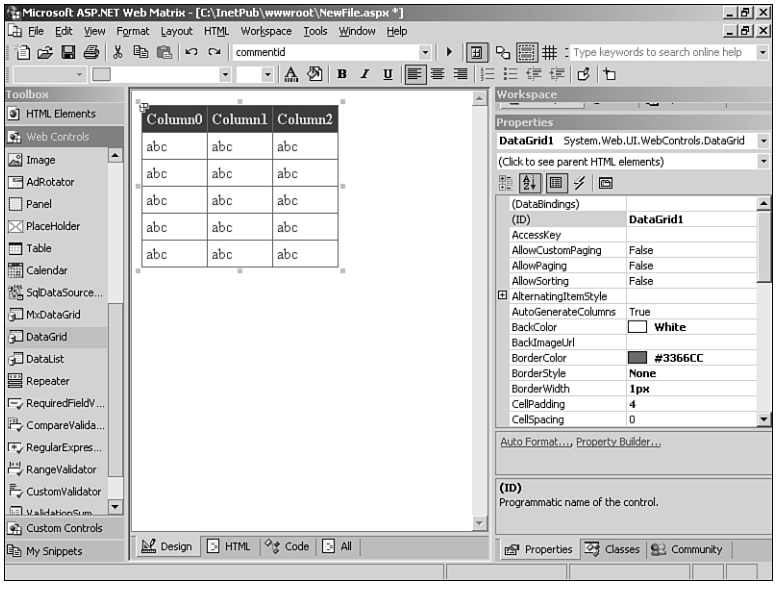


FIGURE 3.7 The DataGrid's style has been updated with the format chosen in Figure 3.6.

LISTING 3.5 The Auto Format Dialog Box Can Automatically Set Display Properties for Your DataGrid

```
1: <asp:DataGrid id="DataGrid1" runat="server" GridLines="None" BorderWidth="1px"
➥BorderColor="Tan" BackColor="LightGoldenrodYellow" CellPadding="2"
➥ForeColor="Black">
2:   <FooterStyle backcolor="Tan"></FooterStyle>
3:   <HeaderStyle font-bold="True" backcolor="Tan"></HeaderStyle>
4:   <PagerStyle horizontalalign="Center" forecolor="DarkSlateBlue"
➥backcolor="PaleGoldenrod"></PagerStyle>
5:   <SelectedItemStyle forecolor="GhostWhite" backcolor="DarkSlateBlue">
➥</SelectedItemStyle>
6:   <AlternatingItemStyle backcolor="PaleGoldenrod"></AlternatingItemStyle>
7: </asp:DataGrid>
```

NOTE

Line 4 in Listing 3.5 contains a `PagerStyle` tag, which sets properties for the `PagerStyle` property of the DataGrid. The `PagerStyle` property, which we've yet to discuss, is an "ItemStyle" property of the DataGrid. It is useful for specifying paging styles for DataGrids that support pagination. We'll be examining this property in more detail in Chapter 8, "Providing DataGrid Pagination."

A Note on How the Display Properties Are Rendered in the Browser

One of the nice things about ASP.NET Web controls is that they are adaptive to the user's browser. By that I mean the HTML markup emitted by an ASP.NET Web page's Web controls is dependent upon the browser being used to view the page. For example, the ASP.NET validation Web controls will emit client-side validation code if the user's browser is deemed to be an *uplevel* browser; if the browser is deemed to be a *downlevel* browser, then no such client-side code is emitted.

An uplevel browser is one that can support client-side JavaScript, HTML version 4.0, the Microsoft Document Object Model, and cascading stylesheets (CSS). You can define what constitutes an uplevel and a downlevel browser in the `machine.config` file; by default, Microsoft Internet Explorer 4.0 and later are considered uplevel, whereas all other browsers are considered downlevel.

NOTE

Uplevel and downlevel status is only based upon the browser being used. That is, a user with Internet Explorer 6.0 would be detected as having an uplevel browser, even if the user has disabled JavaScript support in his browser.

The display properties for the DataGrid and DataList are rendered differently depending upon whether the user visiting the page is using an uplevel or a downlevel browser. If an uplevel browser is being used, the display properties are emitted as CSS. For example, in Listing 3.4, the header's style properties are set on lines 6 and 7. Specifically, the `BackGround` is set to Navy, and the `ForeGround` to White; the text is centered and made bold. When visiting with an uplevel browser, the header for the HTML `table` is rendered as the following:

```
<tr align="Center" style="color:White;background-color:Navy;font-weight:bold;">
    <td>Title</td><td>Sales Performance</td>
</tr>
```

Note that the properties are all set in a `style` attribute of the `tr` tag.

However, if we visit this page with a downlevel browser (*any* browser other than Internet Explorer 4.0 or later), the following HTML is emitted:

```
<tr align="Center" bgcolor="Navy">
    <td><font face="Verdana" color="White"><b>Title</b></font></td>
        <td><font face="Verdana" color="White"><b>Sales Performance</b></font></td>
</tr>
```

Here the display properties are set using HTML instead of CSS.

If you are curious about the rendering differences between an uplevel and downlevel browser, you can do one of two things: view the same page with an uplevel browser (such as Internet Explorer 6.0) and a downlevel browser (such as Opera), comparing the HTML output that each browser receives; or programmatically specify how the controls on the page should be rendered by using the `ClientTarget` attribute in the `@Page` directive.

Using the `ClientTarget` attribute, you can have the controls on an ASP.NET Web page render as if the page were being visited by an uplevel or downlevel browser, independent of the actual browser being used. To specify that the controls should render as if being visited by an uplevel browser, use the following:

```
<%@ Page ClientTarget="uplevel" %>
```

Similarly, to force the controls to render as if they were being visited by a downlevel browser, use this:

```
<%@ Page ClientTarget="downlevel" %>
```

What About the Repeater Control?

The Repeater control does not have any display properties. Rather, to customize the HTML output of the Repeater, you must explicitly specify the HTML markup to use in the Repeater's templates. That is, if you want to have the Repeater output bold text, you have to specifically surround the text with the `bold` tag, or surround the text with a `span` tag whose `style` attribute contains a CSS rule to set the `font-weight` accordingly. (Realize that the DataList and DataGrid's templates can also be customized in this fashion.)

Of course, any time you encapsulate the appearance in HTML markup in a template, you are mixing code and content. Ideally, these two should be kept separate. By setting the appearance of the DataGrid and DataList via the display properties, it is much easier to make changes to the control or template markup later.

For example, imagine that you are using a DataGrid with a couple of TemplateColumns and you want the text in the TemplateColumn to be bold. Although you could simply add

```
<span style="font-weight:bold;">
 Template Markup
</span>
```

it would be cleaner to set the TemplateColumn's `ItemStyle-Font-Bold` property to `True`. If you later needed to change the TemplateColumn's output to italic instead of bold, changing the display property would be simpler than picking through the template markup. (Although it might not be much simpler in this trivial example, imagine that you have a large number of display properties set, and you want to change just one of them. Clearly, making such a change would be simpler using the display properties than if you use a large number of HTML tags to specify the style settings.)

Another advantage of using the display properties is that the HTML emitted is adaptive to the user's browser. If you had used the CSS markup shown previously to make the template's contents bold, a visitor using an older browser that does not support CSS would miss out on the intended visual experience. By using display properties, the correct markup is automatically chosen and sent to the user based upon her browser.

Customizing Data-Binding Output in Templates

Until now, the majority of the examples we've examined that have used data-binding syntax within templates have simply output a particular value from the current item from the `DataSource`. The syntax for this has looked like either

```
<%# Container.DataItem %>
```

or

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

depending on the `DataSource`. The former example would be used if the `DataSource` is a string array, and the latter example might be used if the `DataSource` is a DataSet, DataReader, or some other complex object.

If we limit ourselves to such simple data binding, templates are only useful for customizing the look and feel of the data Web control's output, and little else. That is, because data-binding syntax is used only within templates, if the data binding is limited only to re-emitting the `DataSource` values as is, then why would one need to use a template, other than to specify the placement of HTML markup and data?

In this section, we will examine how to customize the values output by data-binding expressions. Before we look at ways to accomplish this, though, it is important that you have a thorough understanding of the type and value of data-binding expressions.

The Type and Value of a Data-Binding Expression

Every expression in any programming language has a type and a value. For example, the expression 4+5 is of type integer and has a value of 9. One of the things a compiler does is determine the type of expressions to ensure type safety. For example, a function that accepts an integer parameter could be called as

```
MyFunction(4+5)
```

because the expression 4+5 has the type integer.

Not surprisingly, data-binding expressions have a type and a value associated with them as well. In Listing 3.2, line 17, the data-binding expression

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

has a type of string, because the field `au_fname` in the `authors` table is a `varchar(20)`. The value of the data binding expression depends on the value of `Container.DataItem`, which is constantly changing as the `DataSource` is being enumerated over. Hence, for the first row produced, the data-binding expression has a value of `Johnson`; the next row has a value of `Marjorie`; and so on. (See Figure 3.2 for a screenshot of Listing 3.2.)

When the Type of an Expression Is Known: Compile-Time Versus Runtime

The type of an expression is most commonly known at compile-time. If you have a type error, such as trying to assign a string to an integer, you will get a compile-time error. Being able to detect errors at compile-time has its advantages when it comes to debugging—it's much easier to fix a problem you find when trying to compile the program than it is to fix a problem you can't find until you actually run the program.

Data binding expressions, however, have their type determined at runtime. This is because it would be impossible to determine the type at compile-time, because there's no way the compiler can determine the type structure of the `DataSource`, let alone the type that the `DataBinder.Eval()` method will return.

What Type Should Be Returned by a Data-Binding Expression?

Just like other expressions, the type that should be returned by the data-binding syntax depends upon where the data-binding expression is used. If it is used in a template to emit HTML, then a string type should be returned. Because all types

support the `ToString()` method, literally any type can be accepted by a data-binding syntax that is to emit HTML in a template.

NOTE

There are data-binding situations that we've yet to examine in which the data-binding expression needs to be a complex data type.

Determining the Type of a Data-Binding Expression

Determining the type of a data-binding expression is fairly straightforward. Recall that the `Container.DataItem` has the type of whatever object the `DataSource` is enumerating over. Hence, if the `DataSource` is an integer array, the integers are being enumerated over, so the type of `Container.DataItem` is integer.

For complex objects like the DataSet and DataReader, determining what type of object being enumerated over can be a bit more work. The simplest way, in my opinion, is to just create a simple ASP.NET Web page that uses a DataList with an ItemTemplate. Inside the ItemTemplate, simply place `<%# Container.DataItem %>` (see Listing 3.6, line 27). When emitted as HTML, the `Container.DataItem`'s `ToString()` method will be called, which will have the effect of displaying the `Container.DataItem`'s namespace and class name. Listing 3.6 shows a simple DataGrid example that illustrates this point.

LISTING 3.6 To Determine the Type of `Container.DataItem`, Simply Add `<%# Container.DataItem %>` to the ItemTemplate

```
 1: <%@ import Namespace="System.Data" %>
 2: <%@ import Namespace="System.Data.SqlClient" %>
 3: <script runat="server" language="VB">
 4:
 5:   Sub Page_Load(sender as Object, e as EventArgs)
 6:     '1. Create a connection
 7:      Const strConnString as String = "server=localhost;uid=sa;pwd=;
➥database=pubs"
 8:      Dim objConn as New SqlConnection(strConnString)
 9:
10:     '2. Create a command object for the query
11:     Const strSQL as String = "SELECT * FROM authors"
12:     Dim objCmd as New SqlCommand(strSQL, objConn)
13:
14:     objConn.Open()  'Open the connection
15:
16:     'Finally, specify the DataSource and call DataBind()
```

LISTING 3.6 Continued

```
17:     dlDataItemTypeTest.DataSource = objCmd.ExecuteReader(CommandBehavior.
➥CloseConnection)
18:     dlDataItemTypeTest.DataBind()
19:
20:     objConn.Close()  'Close the connection
21:   End Sub
22:
23: </script>
24:
25: <asp:datalist id="dlDataItemTypeTest" runat="server">
26:  <ItemTemplate>
27:   <%# Container.DataItem %>
28:  </ItemTemplate>
29: </asp:datalist>
```

Figure 3.8 shows the output of Listing 3.6 when viewed through a browser. Note that the type is `System.Data.Common.DbDataRecord`.

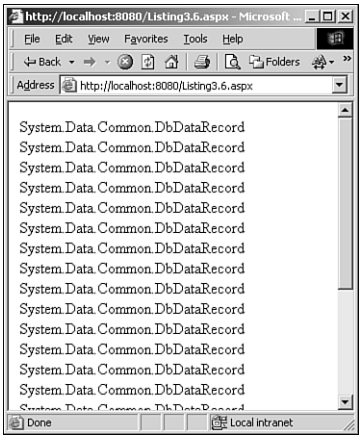


FIGURE 3.8 The type of `Container.DataItem` is displayed on the page.

Building on the type of `Container.DataItem`, we can determine the type returned by `DataBinder.Eval()`. For example, if line 27 in Listing 3.6 was changed to

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

the type returned by the data binding expression would be the type of the `au_fname` field represented by the current `DbDataRecord` object. Because `au_fname` is of type

`varchar(20)`, the type of `<%# DataBinder.Eval(Container.DataItem, "au_fname") %>` is string.

Using Built-In Methods and Custom Functions in Data-Binding Syntax

At this point, we've examined the type issues involving data-binding syntax, and how to determine the type of a data-binding expression. Furthermore, we discussed what the type of a data-binding expression should be when it is in a template for the purpose of emitting HTML (type string).

Now that you have a solid understanding of typing and the data-binding syntax, we can look at using both built-in methods and custom functions to further enhance the output generated by data-binding expressions.

NOTE

By *built-in methods* I am referring to methods available in the .NET Framework classes, such as `String.Format()` or `Regex.Replace()`; recall that in Chapter 2's Listing 2.5 we saw an example of using the `String.Format()` method call in a data-binding syntax setting (line 17). By *custom functions*, I am referring to functions that you write and include in your ASP.NET page's server-side script block or code-behind class.

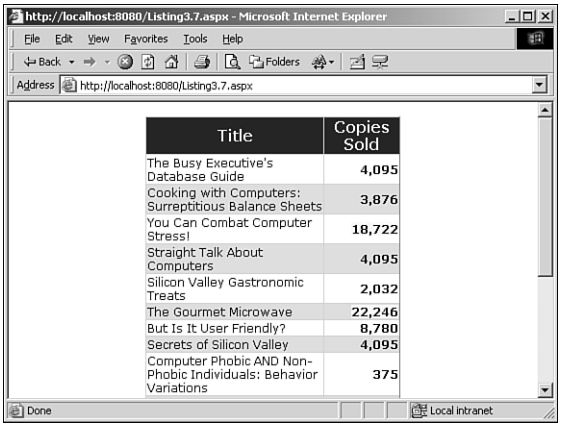
A built-in method or custom function can be used in a data-binding expression like so:

```
<%# FunctionName(argument1, argument2, ..., argumentN) %>
```

More likely than not, the argument(s) to the function will involve the `Container.DataItem` object, most likely in a `DataBinder.Eval()` method call. The function *`FunctionName`* should return a string (the HTML to be emitted) and accept parameters of the appropriate type.

To help clarify things, let's look at a simple example. Imagine that we are displaying sales information about the various books in the `pubs` database, as we did in Listing 3.4. As you'll recall, Listing 3.4 displayed a list of each book along with the book's year-to-date sales and price. Perhaps we'd like to customize our output so that books that have sold more than 10,000 copies have their sales data highlighted. Listing 3.7 contains the code for an ASP.NET Web page that provides such functionality.

Before you delve into the code in any great detail, first take a look at the screenshot of Listing 3.7, shown in Figure 3.9. Note that those books that have sold more than 10,000 copies have their sales figure highlighted.



Title	Copies Sold
The Busy Executive's Database Guide	4,095
Cooking with Computers: Surreptitious Balance Sheets	3,876
You Can Combat Computer Stress!	18,722
Straight Talk About Computers	4,095
Silicon Valley Gastronomic Treats	2,032
The Gourmet Microwave	22,246
But Is It User Friendly?	8,780
Secrets of Silicon Valley	4,095
Computer Phobic AND Non-Phobic Individuals: Behavior Variations	375

FIGURE 3.9 Books with more than 10,000 sales have their sales figure highlighted.

LISTING 3.7 Books with Sales Greater Than 10,000 Copies Are Highlighted

```
 1: <%@ import Namespace="System.Data" %>
 2: <%@ import Namespace="System.Data.SqlClient" %>
 3: <script runat="server" language="C#">
 4:
 5:   void Page_Load(Object sender, EventArgs e)
 6:   {
 7:    // 1. Create a connection
 8:    const string strConnString = "server=localhost;uid=sa;pwd=;database=pubs";
 9:    SqlConnection objConn = new SqlConnection(strConnString);
10:
11:    // 2. Create a command object for the query
12:    const string strSQL = "SELECT * FROM titles WHERE NOT ytd_sales IS NULL";
13:    SqlCommand objCmd = new SqlCommand(strSQL, objConn);
14:
15:    objConn.Open();  // Open the connection
16:
17:    //F inally, specify the DataSource and call DataBind()
18:    dgTitles.DataSource = objCmd.ExecuteReader(CommandBehavior.
➥CloseConnection);
19:    dgTitles.DataBind();
20:
21:    objConn.Close();  // Close the connection
22:   }
23:
```

LISTING 3.7 Continued

```
24:   string Highlight(int ytdSales)
25:   {
26:    if (ytdSales > 10000)
27:    {
28:     return "<span style=\"background-color:yellow;\">" +
29:          String.Format("{0:#,###}", ytdSales) + "</span>";
30:    }
31:    else
32:     return String.Format("{0:#,###}", ytdSales);
33:   }
34:
35: </script>
36: <asp:datagrid id="dgTitles" runat="server"
37:    AutoGenerateColumns="False"
38:    Font-Name="Verdana" Width="50%"
39:    HorizontalAlign="Center" ItemStyle-Font-Size="9">
40:
41:  <HeaderStyle BackColor="Navy" ForeColor="White"
42:     HorizontalAlign="Center" Font-Bold="True" />
43:
44:  <AlternatingItemStyle BackColor="#dddddd" />
45:
46:  <Columns>
47:   <asp:BoundColumn DataField="title" HeaderText="Title"
48:            ItemStyle-Width="70%" />
49:
50:   <asp:TemplateColumn HeaderText="Copies Sold"
51:      ItemStyle-HorizontalAlign="Right"
52:      ItemStyle-Wrap="False">
53:    <ItemTemplate>
54:      <b><%# Highlight((int) DataBinder.Eval(Container.DataItem,
➥"ytd_sales")) %></b>
55:    </ItemTemplate>
56:   </asp:TemplateColumn>
57:  </Columns>
58: </asp:datagrid>
```

In examining the code in Listing 3.7, first note that the SQL string (line 12) has a `WHERE` clause that retrieves only books whose `ytd_sales` field is not `NULL`. The `titles` database table is set up so that the `ytd_sales` field allows `NULL`, representing a book

with no sales. Because accepting NULLs slightly complicates our custom function, I've decided to omit books whose ytd_sales field contains NULLs—we'll look at how to handle NULLs shortly.

Next, take a look at the Highlight() function (lines 24—33). The function is rather simple, taking in a single parameter (an int) and returning a string. The function simply checks whether the input parameter, ytdSales, is greater than 10,000—if it is, a string is returned that contains a span HTML tag with its background-color style attribute set to yellow; the value of ytdSales, formatted using String.Format() to include commas every three digits; and a closing span tag (lines 28 and 29). If ytdSales is not greater than 10,000, the value of ytdSales is returned, formatted to contain commas every three digits (line 22).

Finally, check out the ItemTemplate (line 54) in the TemplateColumn control. Here we call the Highlight() function, passing in a single argument: DataBinder.Eval(Container.DataItem, "ytd_sales"). At first glance, you might think that the type returned by DataBinder.Eval() here is type integer, because ytd_sales is an integer field in the pubs database. This is only half-correct. While the DataBinder.Eval() call is returning an integer, the compiler is expecting a return value of type Object. (Realize that all types are derived from the base Object type.) Because C# is type-strict, we must cast the return value of the DataBinder.Eval() method to an int, which is the data type expected by the Highlight() function. (Note that if you are using Visual Basic .NET, you do not need to provide such casting unless you specify strict casting via the Strict="True" attribute in the @Page directive; for more information on type strictness, see the resources in the "On the Web" section.)

Handling NULL Database Values

The custom function in Listing 3.7 accepts an integer as its input parameter. If the database field being passed into the custom function can return a NULL (type DBNull), then a runtime error can result when a NULL value is attempted to be cast into an int.

NOTE

NULL values in database systems represent unknown data. For example, imagine that you had a database table of employees, and one of the table fields was BirthDate, which stored the employee's date of birth. If for some reason you did not have the date of birth information for a particular employee, you could store the value NULL for that employee's BirthDate field. It is important to note that NULL values should not be used to represent zero, since any expression involving a NULL field results in a NULL value.

To handle NULLs, we'll adjust the Highlight() function to accept inputs of type Object, thus allowing *any* class to be passed in (because all classes are derived, directly or indirectly, from the Object class). Then, in the Highlight() function we'll

check to see whether the passed-in value is indeed a NULL. If it is, we can output some message like, "No sales yet reported." If it is not NULL, we'll simply cast it to an int and use the code we already have to determine whether it should be highlighted. Listing 3.8 contains the changes needed for Listing 3.7 to accept NULLs:

LISTING 3.8 The Highlight Function Has Been Adjusted to Accept NULLs

```
<%@ Page Language="c#" %>
<%@ import Namespace="System.Data" %>
<%@ import Namespace="System.Data.SqlClient" %>
<script runat="server">

  void Page_Load(Object sender, EventArgs e)
  {
   // 1. Create a connection
   const string strConnString = "server=localhost;uid=sa;pwd=;database=pubs";
   SqlConnection objConn = new SqlConnection(strConnString);

   // 2. Create a command object for the query
   const string strSQL = "SELECT * FROM titles";
   SqlCommand objCmd = new SqlCommand(strSQL, objConn);

   objConn.Open();  // Open the connection

   //F inally, specify the DataSource and call DataBind()
   dgTitles.DataSource = objCmd.ExecuteReader(CommandBehavior.
➥CloseConnection);
   dgTitles.DataBind();

   objConn.Close();  // Close the connection
  }

  string Highlight(object ytdSales)
  {
   if (ytdSales.Equals(DBNull.Value))
   {
    return "<i>No sales yet reported...</i>";
   }
   else
   {
    if (((int) ytdSales) > 10000)
    {
     return "<span style=\"background-color:yellow;\">" +
```

LISTING 3.8 Continued

```
36:          String.Format("{0:#,###}", ytdSales) + "</span>";
37:     }
38:     else
39:     {
40:      return String.Format("{0:#,###}", ytdSales);
41:     }
42:    }
43:   }
44:
45: </script>
46:
47: <asp:datagrid id="dgTitles" runat="server"
48:
49:  ... some DataGrid markup removed for brevity ...
50:
51:   <asp:TemplateColumn HeaderText="Copies Sold"
52:      ItemStyle-HorizontalAlign="Right"
53:      ItemStyle-Wrap="False">
54:    <ItemTemplate>
55:      <b><%# Highlight(DataBinder.Eval(Container.DataItem, "ytd_sales")) %></b>
56:    </ItemTemplate>
57:   </asp:TemplateColumn>
58:  </Columns>
59: </asp:datagrid>
```

Listing 3.8 contains a few changes. The two subtle changes occur on lines 13 and 55. On line 13, the SQL query is altered so that *all* rows from the `titles` table are returned, not just those that are not `NULL`. On line 55, the cast to `int` was removed, because the `DataBinder.Eval()` function might return `NULL`s as well as integers.

The more dramatic change occurs in the `Highlight` function (lines 25–43). In Listing 3.8, `Highlight()` accepts an input parameter of type `Object`; this allows for any type to be passed into the function (of course, we only expect types of integers or `DBNull` to be passed in). On line 27, a check is made to determine whether the `ytdSales` parameter is of type `DBNull` (the `DBNull.Value` property is a static property of the `DBNull` class that returns an instance of the `DBNull` class—the `Equals` method determines whether two objects are equal). If `ytdSales` represents a `NULL` database value, the string `"No sales yet reported"` is returned on line 29. Otherwise, `ytdSales` is cast to an `int` and checked to see whether it is greater than 10,000. The remaining lines of `Highlight()` are identical to the code from `Highlight` in Listing 3.7.

Figure 3.10 shows a screenshot from the code in Listing 3.8. Note that those rows that have `NULL` values for their `ytd_sales` field have `No sales yet reported...` listed under the Copies Sold column.

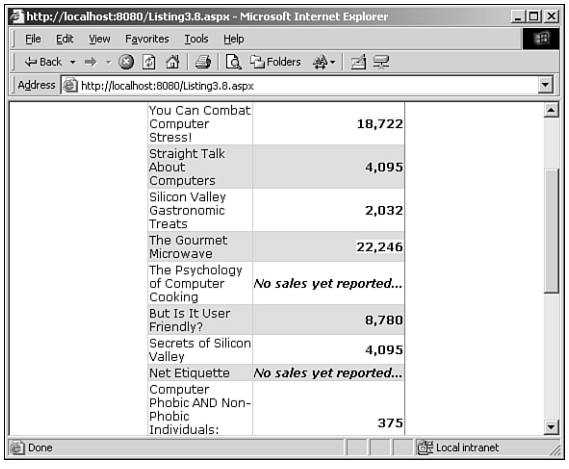


FIGURE 3.10 Database columns that have a `NULL` value have a helpful message displayed.

A Closing Word on Using Custom Functions in Templates

As we have seen, using customized functions in a data-binding setting allows for the data-bound output to be altered based upon the value of the data itself. For example, if you were displaying profit and loss information in a data Web control, you might opt to use such custom functions to have negative profits appear in red.

The last two examples showed the use of simple custom functions that accept one parameter. Of course, there is no reason why you couldn't pass in more than one parameter. Perhaps you want to pass in two database field values and then output something based on the two values.

Another neat thing you can do with these custom functions is add information to your data Web controls that might not exist in the database. For example, from the `titles` table, we have both a `ytd_sales` field and a `price` field. Imagine that we want to display the book's total revenues—this would be simply the sales multiplied by the price.

Listing 3.9 shows a quick example of how to accomplish this. Essentially, all we have to do is pass both the `ytd_sales` and `price` fields to our custom function, multiply them, and then return the value.

LISTING 3.9 Custom Functions Can Include Multiple Input Parameters

```
 1: <%@ import Namespace="System.Data" %>
 2: <%@ import Namespace="System.Data.SqlClient" %>
 3: <script runat="server" language="VB">
 4:
 5:   Sub Page_Load(sender as Object, e as EventArgs)
 6:    '1. Create a connection
 7:    Const strConnString as String = "server=localhost;uid=sa;pwd=;
➥database=pubs"
 8:    Dim objConn as New SqlConnection(strConnString)
 9:
10:    '2. Create a command object for the query
11:    Const strSQL as String = "SELECT * FROM titles WHERE NOT ytd_sales IS NULL"
12:    Dim objCmd as New SqlCommand(strSQL, objConn)
13:
14:    objConn.Open()  'Open the connection
15:
16:    'Finally, specify the DataSource and call DataBind()
17:    dgTitles.DataSource = objCmd.ExecuteReader(CommandBehavior.CloseConnection)
18:    dgTitles.DataBind()
19:
20:    objConn.Close()  'Close the connection
21:   End Sub
22:
23:
24:   Function CalcRevenues(sales as Integer, price as Single)
25:    Return String.Format("{0:c}", sales * price)
26:   End Function
27:
28: </script>
29: <asp:datagrid id="dgTitles" runat="server"
30:
31:  ... some DataGrid markup removed for brevity ...
32:
33:   <asp:TemplateColumn HeaderText="Revenues"
34:      ItemStyle-HorizontalAlign="Right"
35:      ItemStyle-Wrap="False">
36:    <ItemTemplate>
37:      <b><%# CalcRevenues(DataBinder.Eval(Container.DataItem, "ytd_sales"),
➥DataBinder.Eval(Container.DataItem, "price")) %></b>
38:    </ItemTemplate>
39:   </asp:TemplateColumn>
40:  </Columns>
41: </asp:datagrid>
```

Note that here, as with Listing 3.7, our SQL query only grabs those rows whose `ytd_sales` field is not `NULL` (line 11). My motivation behind this was to keep the `CalcRevenues()` custom function as clean as possible; you are encouraged to adjust the code in Listing 3.9 to allow for `NULL`s.

The custom function, `CalcRevenues()`, calculates the total revenues for each book. It accepts two input parameters (`sales` and `price`, an integer and single, respectively), and then multiplies them, using the `String.Format()` method to format the result into a currency. The string returned by the `String.Format()` method is then returned as the data-binding expression's value and is emitted as HTML.

Line 37 contains the call to `CalcRevenues()` in the data-binding syntax. Note that two database values are passed into `CalcRevenues()`—the `ytd_sales` field value and the `price` field value.

Figure 3.11 contains a screenshot of the code in Listing 3.9 when viewed through a browser.

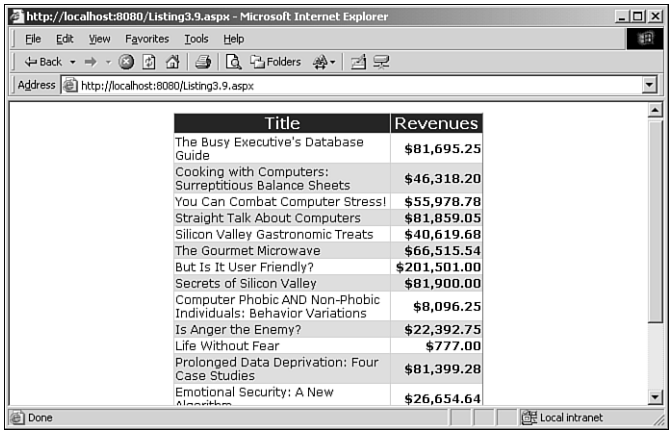


FIGURE 3.11 The total revenue for each book is shown.

In closing, realize that the amount of customization you can do using the data-binding syntax is virtually limitless.

Summary

In this chapter, we examined how to improve the appearance of the DataGrid and DataList Web controls though the use of display properties. These properties can be applied at the table level, the row level, and the column level, allowing developers to create aesthetically pleasing DataGrids and DataLists without requiring extensive HTML and CSS markup.

Additionally, we saw how tools like Visual Studio .NET and the Web Matrix Project can be used to automatically provide style schemes. For those of us who are artistically challenged, the Auto Formatting features are a godsend.

This chapter concluded with an examination of customizing the data-binding output in templates. We saw how to supply dynamic data as parameters to built-in and custom functions, which could then tweak the data and produce customized output. The capability to provide custom functions to alter the data-bound output at runtime is a powerful technique that we will take advantage of many times throughout the remainder of this book.

This chapter concludes Part I. By this point, you should be comfortable using the data Web controls, binding a variety of types of data to the controls, and making the controls visually appealing. In Part II, we'll examine how to associate actions with the data Web controls!

On the Web

For more information on the topics discussed in this chapter, consider perusing the following online resources:

- Formatting Overview—`http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconformattingoverview.asp`
- The `<clientTarget>` Element (used for determining uplevel and downlevel browsers)—`http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/gngrfclienttargetelement.asp`
- "Formatting the DataGrid with the ASP.NET Web Matrix Project"—`http://www.asp.net/webmatrix/tour/section4/formatdatagrid.aspx`
- "Highlighting Search Keywords in a DataGrid"—`http://aspnet.4guysfromrolla.com/articles/072402-1.aspx`
- "Strict Typing in VB.NET"—`http://msdn.microsoft.com/vstudio/techinfo/articles/developerproductivity/language.asp#type`