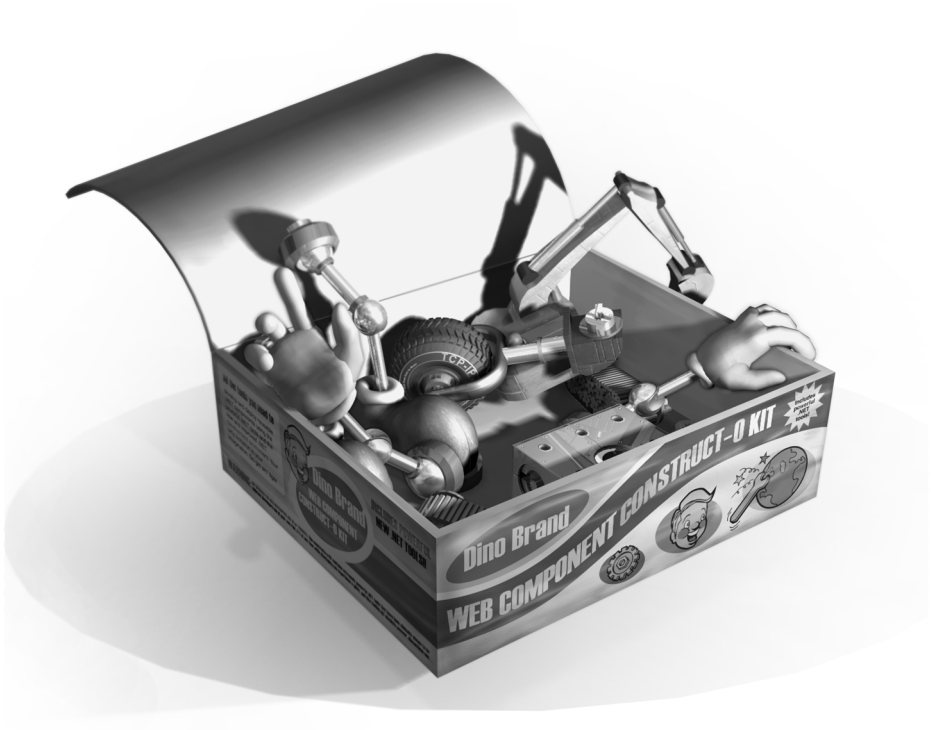
Part I

Data Access and Reporting



1

Data Bound .NET Controls

Most software applications involve, in one way or another, data access and reporting. Web applications in particular must perform these two functions, so Web programmers are looking for any automated (or even semi-automated) method of associating rows of data with graphical HTML elements such as drop-down lists or tables. The answer to this growing demand for automatic binding between data sources and graphical elements is .NET data bound controls.

Data bound controls play a key role in the development of .NET applications because they allow you to associate controls and their individual properties with one or more fields in any .NET-compliant data source. The power of .NET is the versatility of these data bound controls. You use them in the same way regardless of the programming model—Microsoft Windows Forms, Microsoft Web Forms, or Microsoft .NET Web services.

In this chapter, we'll discuss the two levels of data binding, simple and complex, as well as the most important Web controls that can be data bound. Data binding and data bound Web controls are the focus of any Active Server Pages (ASP) .NET application, so we'll explore them at length in the next few chapters. (Web controls are also referred to as ASP.NET controls, server controls, and .NET Web controls.) Getting a handle on the key programming techniques is vital for building effective Web solutions.

ASP.NET Data Binding

Data binding is the process of retrieving data from a source and dynamically associating it to a property of a visual element. Depending on the context in which the element will be displayed, you can map the element to either an HTML tag or a .NET Web control.

In ASP.NET, the atomic elements that work together to generate page output are the Web controls. In ASP.NET data binding, the application retrieves data from any .NET-compliant data source and then allows you to selectively manipulate and assign this data to properties of controls that have been specially designed to support data binding. These controls are known as data bound controls. ASP.NET Web server controls can be bound to a data source by using a heterogeneous bunch of properties, including *Text*, *DataSource*, and *DataTextField*. Later in the chapter, you'll see how to handle binding by using a combination of these and other control-specific properties.

The *DataSource* property in particular lets you specify the data source object to which the control is linked. Notice that this link is logical and does not result in any underlying operation until you explicitly call another method. Not all properties in a Web control programming interface can be data bound. You activate data binding on a control by calling the method *DataBind*. When this method executes, the control loads data from the associated data source, evaluates the data bound properties, and redraws its user interface to reflect changes.

Feasible Data-Binding Sources

Many .NET classes can be used as data sources—not just those dealing with database contents. In general, any object that exposes the *ICollection* interface is a potential source of data binding. The *ICollection* interface defines size, enumerators, and synchronization methods for all .NET collections. As a result, you can bind a Web control to any of the following data structures:

- In-memory .NET collection classes including arrays, dictionaries, sorted and linked lists, hash tables, stacks, and queues.
- User-defined data structures, as long as the structure exposes *ICollection* or one of the interfaces based on it (such as *IList*).
- Database-oriented classes such as *DataTable* and *DataSet*. Bear in mind that in general a *DataSet* object can contain multiple tables. In this case, you would assign the *DataSet* object to the *DataSource* property and assign the name of the selected *DataTable* object to the control's *DataMember* property.
- Filtered views built on top of *DataTable* objects that show only a subset of the contained rows. Views are represented by a *DataView* class or any class that you have built from a *DataTable* object.

> **Note** You cannot directly bind XML documents unless you load their contents in one of the classes mentioned in the preceding bulleted list. You can bind XML documents in two ways. You can load the XML document into a *DataSet* object and then bind the *DataSet* object. Alternatively, you can parse the XML document with an XML reader class and bind the collection of nodes you obtain.

Simple Data Binding

In ASP.NET, simple data binding is a connection between one piece of data and a server control property. This connection is established through a special expression that is evaluated when the code in the page being processed calls the *DataBind* method either on the *Page* object or on the control.

A data-binding expression is any text enclosed in angle brackets and percent signs (<%…%>) and prefixed by the number symbol (#). You can include a data-binding expression as the value of an attribute in the opening tag of a server control or anywhere else on the page. You cannot programmatically assign a data-binding expression to a property of a Web control. If the data-binding expression contains quotation marks, use a single quotation mark (') to wrap the whole expression, as shown here:

```
theLabel.Text='<%# "Hello, world" %>';
```

Within the delimiters, you can invoke user-defined page methods, static methods and properties, and methods of any other page components. The following code shows a label control whose *Text* property is bound to the name of the currently selected element in a drop-down list control:

```
<asp:label runat="server" Text='<%# dropdown.SelectedItem.Text %>' />
```

The data-binding expression can accept a minimal set of operators, mostly for concatenating subexpressions. When you need more advanced processing and are using external arguments, resort to a user-defined method. The only requirement for a user-defined method is to be callable within the page. When you plan to use code that is not resident in the .ASPX page (for example, if you use code-behind techniques), make sure the method is declared public or protected.

Let's walk through an example that illustrates how to arrange a form-based record viewer by using simple data binding with text boxes. (The full source code for the FormViewer.aspx application is available on the companion CD.) The user interface supplies three text boxes, one for each of the retrieved fields. The data source is the *Employees* table of the Northwind database in SQL Server 2000. The query selects three fields: *employeeid*, *firstname*, and *lastname*. The binding takes place between the text boxes and the fields. Only one record at a time is displayed; the user clicks the Previous and Next buttons to move through the data. You can see what the application looks like in Figure 1-1.

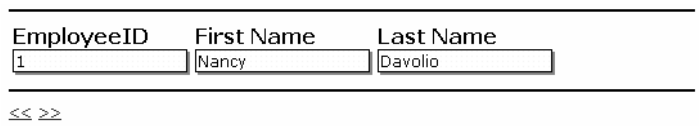


Figure 1-1 The text boxes are bound to fields in the *Employees* table from the SQL Server 2000 Northwind sample database.

The binding expression employs a user-defined method named *GetBoundData*.

```
<asp:textbox runat="server" id="txtID" cssclass="StdTextBox"
    Text = '<%# GetBoundData("employeeid") %>' />
```

GetBoundData takes the name of the field to display and retrieves the position of the current record from the ASP.NET *Session* object. It then retrieves the value and passes it to the ASP.NET run time for actual rendering.

```
public String GetBoundData(String fieldName)
{
    DataSet ds = (DataSet) Session["MyData"];
    DataTable dt = ds.Tables["EmpTable"];
    int nRowPos = (int) Session["CurrentRecord"];

    String buf = dt.Rows[nRowPos][fieldName].ToString();
    return buf;
}
```

Complex Data Binding

Complex data binding occurs when you bind a list control or an iterative control to one or more columns of data. List controls comprise *DropDownList*, *CheckBoxList*, *RadioButtonList*, and *ListBox*. Iterative controls are *Repeater*, *DataList*, and *DataGrid*. The characteristics of ASP.NET iterative controls are briefly summarized in Table 1-1.

Table 1-1 ASP.NET Iterative Controls

Control	Description
Repeater	Creates a custom layout for displaying the control's contents by repeating a specified template for each item in the bound list. You define an ASP.NET template for various categories of items (header, footer, item, separator, and so forth), and the control applies it repeatedly in the page. The *Repeater* control has no built-in layout, so you must explicitly declare all ASP.NET formatting and style tags.
DataList	Displays the contents of a data bound list through ASP.NET templates. The *DataList* control supports selecting and editing. The appearance of the *DataList* control may be customized to some extent by setting style properties for different parts of the control. Compared to the *Repeater* control, *DataList* shows off predefined layouts, more advanced formatting capabilities, plus support for selecting and editing.
DataGrid	The *DataGrid* control renders a multi-column, data bound grid of data. It allows you to define various types of columns and control the layout of the resulting cells through predefined layouts and ASP.NET templates. You can also add specific functionality such as edit button columns and link columns. The *DataGrid* control also supports a variety of options for paging through data.

List controls have a more complex and versatile user interface than labels and text boxes. Because a list control handles (or at least has in memory) more items simultaneously than a label or a text box, you should associate the list control explicitly with a collection of data—that is, the data source. Depending on its expected behavior, a list control will select the needed items from memory and properly format and display them.

Iterative controls take a data source, loop through its items, and apply HTML templates to its rows. This basic behavior is common to all three ASP.NET iterative controls. The controls differ in their layout capabilities and the functionality they support.

Let's see how complex forms of data binding apply to key Web controls.

The *DropDownList* Web Control

The *DropDownList* Web control enables users to select from a single-selection drop-down list (for example, a combo box). You can control the look of the *DropDownList* control by setting its height and width in pixels, but you cannot control the number of items displayed when the list drops down. The *SelectedIndex* and *SelectedItem* properties provide details about the currently selected element. The *DropDownList* control supports data binding by using the following five properties: *DataSource*, *DataMember*, *DataTextField*, *DataValueField*,

and *DataTextFormatString*. As shown in the following code, of these five properties, only *DataSource*, *DataTextField*, and *DataValueField* have a corresponding ASP.NET attribute you can declare:

```
<asp:DropDownList id="DropDownList1" runat="server"
    DataSource="<%# databindingexpression %>"
    DataTextField="DataSourceField"
    DataValueField="DataSourceField">
```

You can also use a data-binding expression to set the *DataSource* property. The expression you use must evaluate to a .NET object that exposes the *ICollection* interface. You cannot use expressions to set the other properties. The values for *DataTextField* and *DataValueField* each must match the name of one field in the data source, so you cannot assign *DataTextField* by combining two or more fields in the data source. Let's see how to work around this limitation.

Suppose you want a drop-down list to display both the first name and the last name of each employee in the company. You could ask SQL Server to return a calculated column that has the required format. In this case, you would use a query string, as shown in the following code, and set *DataTextField* to the name of the precalculated field:

```
SELECT lastname + ', ' + firstname AS 'EmployeeName'
    FROM Employees
```

You can obtain the same result, however, without the involvement of SQL Server. After you hold a *DataTable* object that contains the result of the query, you can add a new column on the fly. The content of the column is determined by the expression you use. The following code uses this approach to generate the drop-down list shown in Figure 1-2:

```
String strConn, strCmd;
strConn = "DATABASE=Northwind;SERVER=localhost;UID=sa;PWD=;";
strCmd = "SELECT employeeid, firstName, lastname FROM Employees";
SqlDataAdapter oCMD = new SqlDataAdapter(strCmd, strConn);
DataSet oDS = new DataSet();
oCMD.Fill(oDS, "EmployeesList");
DataTable dt = oDS.Tables["EmployeesList"];
dt.Columns.Add("EmployeeName",
    typeof(String),
    "lastname + ', ' + firstname");
```

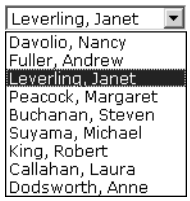


Figure 1-2 You can create a drop-down list box such as this as you work by using the *DataTable* object.

The next code example shows how to bind the drop-down list control named *EmpList* to the dynamically created *EmployeeName* field:

```
EmpList.DataSource = oDS.Tables["EmployeesList"].DefaultView;
EmpList.DataTextField = "EmployeeName";
EmpList.DataValueField = "employeeid";
```

DataTextField links the *Text* property of any individual item in the list with the *EmployeeName* field. *DataValueField* ensures that the *Value* property of each item is set with the value stored in the corresponding record of the *employeeid* field. The full source code for the DropDown.aspx application is available on the companion CD.

An expression-based column does not have to be filled explicitly. Whenever the program needs to read the value of one of its rows, it just evaluates the expression as it processes data and then takes the result. Note that expressions can reference other expression columns. Circular references, though, are not allowed. The ADO.NET run time promptly detects them and raises an exception.

Which of the two approaches is preferable? Should you ask SQL Server to return a made-to-measure column, or should you create the extra column you need when you need it? The final result of both approaches is identical, but the cost of using each is not. SQL Server is not as efficient. It returns a new column of *potentially* duplicated data. Processing data to produce a column creates overhead, especially when the expression is complex. Unless the precalculated column is needed for further processing, you should avoid using SQL Server for this type of operation.

The *CheckBoxList* Web Control

The *CheckBoxList* control is a single control that acts as a parent control for a collection of checkable list items, each of which is rendered through an individual *CheckBox* control. You insert a check box list as follows:

```
<asp:CheckBoxList runat="server" id="cbEmployees">
```

The *CheckBoxList* control supports data binding by using five properties: *DataSource*, *DataMember*, *DataTextField*, *DataValueField*, and *DataTextFormatString*. You bind the *CheckBoxList* control to a data source by using the following code:

```
cbEmployees.DataSource = oDS.Tables["EmployeesList"];
cbEmployees.DataTextField = "lastname";
cbEmployees.DataValueField = "employeeid";
```

The properties of the *CheckBoxList* control play the same role as the properties for the *DropDownList* control, which we discussed earlier in the chapter. Unlike the *DropDownList* control, however, the *CheckBoxList* control

does not supply properties that reveal information about the selected items. As you know, this information is vital for any Web application that uses checkable elements.

At any time, *CheckBoxList* can have one or more items selected. How can you retrieve these items? All list controls have an *Items* property that contains the collection of the child items. The *Items* property is implemented through the *ListItemCollection* class, and each list item can be accessed via a *ListItem* object. The following code loops through the items stored in a *CheckBoxList* control and checks the *Selected* property of each. (*Selected* is a Boolean property that indicates whether the item is selected.) The code then uses the *StringBuilder* object to concatenate the text of each selected item into a single comma-separated string.

```
StringBuilder buf = new StringBuilder("");
foreach(ListItem item in chkList.Items)
{
    if (item.Selected)
    {
        buf.Append(item.Text);
        buf.Append(", ");
    }
}
```

> **Caution** In .NET, the *String* object is immutable, so each time you modify a string, a brand new *String* object is created. This might have an unpleasant impact on performance, especially when repeated modifications to a string are required. Repeated modifications typically occur when you loop through potentially lengthy data sources to build up a summary string. The *StringBuilder* class solves the problem by letting you modify a string without creating a new object.

Notice in the following code that the ASP.NET *CheckBox* control is a bit more powerful than the ASP.NET counterpart of the well-known HTML <*input*> tag. The ASP.NET *CheckBox* control also supplies the *Text* property that allows you to automatically associate the check box with descriptive text.

```
<asp:checkbox runat="server" id="theCheckBox"
    Text="Click me..." />
```

When the preceding code is processed by the ASP.NET run time, the ASP.NET *CheckBox* control generates the following HTML code:

```
<input id="theCheckBox" type="checkbox" name="theCheckBox" />
<label for="theCheckBox">Click me...</label>
```

This code displays the check box shown here:

A slick trick used by ASP developers to quickly obtain the selected items of a logically related group of check boxes does not work in ASP.NET. When you have several check boxes with identical names in an ASP application, you can use a single line of code to obtain the corresponding values of check boxes that are selected when a form is posted.

```
<input type="checkbox" name="foo" value="...">
```

After the form values in the above code are posted, the following line of code written in Microsoft Visual Basic, Scripting Edition (VBScript) sets an array with all the checked values:

```
<% arrSelected = split(Request.Form("foo"), ",") %>
```

Request.Form("foo") returns a comma-separated string formed by the value strings of all checked items. You then pass this string to VBScript's *Split* function, which transforms the string into an array.

This code won't work in ASP.NET if you use the *CheckBox* server control to render the check box component, but it will work if you use the *<input>* tag. However, using the *CheckBoxList* control is the preferred ASP.NET way to handle lists of check boxes.

> **Note** The HTML tag *<input>* and all other HTML tags are rendered through an ASP.NET control taken from the *System.Web.UI.HtmlControls* namespace. The HTML check box element is rendered through the ASP.NET *HtmlInputCheckBox* class.

The *RadioButtonList* Web Control

The *RadioButtonList* control acts as a parent control for a collection of radio button list items. All child items of *RadioButtonList* are rendered by using a *RadioButton* control. A *RadioButtonList* control can have either no items or one item selected. The *SelectedItem* property returns the selected element as a *ListItem* object. However, you have no guarantee that only one item is selected at any given time. For this reason, be extremely careful when you access the *SelectedItem* property of a *RadioButtonList* control. Depending on the structure and flow of your code, the item returned could be a null object. A workaround that always shields you from unpleasant surprises is wrapping any call to

SelectedItem in a *try/catch* block, but don't abuse this technique because a *try/catch* block is always costly. An even better solution is to design your code to eliminate the risk of returning an unselected item.

The contents of a *RadioButtonList* control can be obtained from a data source, as shown here:

```
<asp:RadioButtonList id="rbEmployees" runat="server"
    DataValueField="employeeid"
    DataTextField="employeename" />
```

You also need to programmatically set the *DataSource* property as follows:

```
rbEmployees.DataSource = myDataTable;
```

The *RadioButtonList* control supports data binding by using five properties: *DataSource*, *DataMember*, *DataTextField*, *DataValueField*, and *DataTextFormatString*. These properties behave in the same way as the properties of controls discussed earlier in the chapter.

Programmers and designers who work with data bound controls in lists are always greatly concerned about content presentation. A list of items can flow horizontally or vertically, can be expressed in a fixed number of columns or rows, and so on.

Some Web controls, such as *RadioButtonList*, accept layout directives. For example, you can control how a list is rendered by using the *RepeatLayout* and/or *RepeatDirection* properties of *RadioButtonList*. *RepeatLayout* governs the default layout in which the list items are rendered within a table, ensuring the vertical alignment of the companion text. The alternative is displaying the items as free HTML text using blanks and breaks to guarantee a minimum of structure. With or without a tabular structure, *RepeatDirection* controls how the items flow. Feasible values for *RepeatDirection* are *Vertical* (the default) and *Horizontal*. The *RepeatColumns* property determines how many columns the list should have. The default value of 0 indicates that all items are displayed in a single row. The direction of that row—vertical or horizontal—depends on the value of *RepeatDirection*.

The *ListBox* Web Control

The *ListBox* control represents a vertical sequence of items displayed in a scrollable window. As shown in the following code, *ListBox* allows single or multiple item selection and exposes its contents by using the familiar *Items* collection:

```
<asp:listbox runat="server" id="theListBox"
    rows="5" selectionmode="Multiple" />
```

You can determine the height of the control by using the *Rows* property. As you might have guessed, the height is measured in numbers of rows rather than in pixels or percentages.

When it comes to data binding, the *ListBox* control behaves like the other controls discussed earlier in the chapter—that is, it supports properties such as *DataSource*, *DataMember*, *DataTextField*, *DataValueField*, and *DataTextFormatString*, and it can be bound to a data source and show its contents, as follows:

```
lstEmp.DataSource = ds.Tables["EmpTable"];
lstEmp.DataTextField = "lastname";
lstEmp.DataValueField = "employeeid";
lstEmp.DataBind();
```

The next code snippet illustrates how to write a comma-separated string with the values of the selected items. (This code is nearly identical to the code you would write to accomplish the same operation for a *CheckBoxList* control.)

```
public void ShowSelectedItems(Object sender, EventArgs e)
{
    StringBuilder sb = new StringBuilder("");
    for (int i=0; i < lstEmp.Items.Count; i++)
    {
        if (lstEmp.Items[i].Selected)
        {
            sb.Append(lstEmp.Items[i].Text);
            sb.Append(", ");
        }
    }
    lblResults.Text = sb.ToString();
}
```

Figure 1-3 shows the results of the code. The full source for the ListBox.aspx application is available on the companion CD.

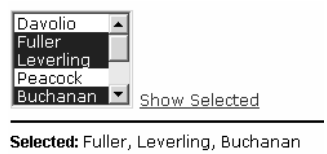


Figure 1-3 This sample application illustrates the use of a data bound, multi-selection list box.

The programming interface of the list box also contains a *SelectedItem* property, which at first appears to make little sense because you are working with a multi-selection control. In this case, however, the *SelectedItem* property returns the selected item with the lowest index.

The *Repeater* Control

The *Repeater* control is a simple container control that binds to a list of items. It walks through the bound items and produces graphical elements according to a basic rendering algorithm and the HTML templates you supply. The *Repeater* control supports from one through five templates. These templates, which form a tabular structure, are described in Table 1-2.

Table 1-2 Templates Supported by the *Repeater* Control

Template	Description
HeaderTemplate	Determines the heading of the final output. It is rendered only once and prior to any row. It cannot contain data bound information.
ItemTemplate	Determines the output format of each row in the data source. This template is called for each item in the list and can contain data binding expressions.
AlternatingItemTemplate	Functions similarly to *ItemTemplate*. This template applies only to rows with an odd ordinal position.
SeparatorTemplate	Determines the HTML content that goes between each row. It cannot contain data bound information.
FooterTemplate	Rendered only once when all items have been rendered. It cannot contain data bound information.

When you call *DataBind* on the *Repeater* control, the control first attempts to locate the *HeaderTemplate* and, if found, applies it. The control then loops through the list items and applies the *ItemTemplate* to each one. When the *AlternatingItemTemplate* is defined, odd rows take it. If the *SeparatorTemplate* is found, the template is applied in between two consecutive items, regardless of whether the template item is normal or an alternate. When the *FooterTemplate* is specified, it is applied at the end of the loop.

The *Repeater* control has no built-in layout facility or predefined style that you can apply by using a declaration, nor does it support selection and editing. You could manually code selection and editing, but for those tasks, you are better off dropping the *Repeater* control in favor of the more powerful *DataList* and *DataGrid* controls. You'll learn more about these controls later in the chapter.

Accessing Data Bound Information

The *Items* property of the *Repeater* control is a collection of *RepeaterItem* objects, each representing an individual row in the list. To define the template structure for normal and alternating items, you need a way to access the data source

fields for the particular row being processed. This information is exactly what the expression *Container.DataItem* returns when you use *Container.DataItem* in a data-binding expression for the normal or alternate templates.

Container.DataItem represents the *n*th object bound to the *RepeaterItem* objects in a list. It evaluates to the same type of element as the caller placed in the *Repeater* control's data source. Let's look at some examples. If you place a *DataTable* object in the *DataSource* property of the *Repeater* control, then *Container.DataItem* evaluates to a *DataRow* object. It evaluates to *DataRow* because in .NET, a *DataTable* is perceived as a collection of rows and each row is implemented through a *DataRow* object. If you use a *DataView* object, the individual item is evaluated to a *DataRowView* object. If you use, say, an array of strings, *Container.DataItem* evaluates to a simple text string. Table 1-3 shows how to obtain the value of a field for the current row.

Table 1-3 Ways to Retrieve Data Item Values in Iterative Controls

Data Source	Data Item
DataTable	`<%# ((DataRow) Container.DataItem)["Field"] %>`
DataView	`<%# ((DataRowView) Container.DataItem)["Field"] %>`
Array of strings	`<%# ((String) Container.DataItem) %>`
Dictionary	`<%# ((Dictionary) Container.DataItem)["entry"] %>`

Because .NET objects are strongly typed, coercing types is an absolute necessity. If you find the syntax in Table 1-3 somewhat quirky, you can alternatively resort to the *DataBinder* class. This class features a static method named *Eval* that has an easier syntax to remember than that of standard data-binding expressions. When you use *DataBinder.Eval* to get the current data item in the expressions in Table 1-3, the code looks like this:

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

As you can see, *DataBinder.Eval* blurs the distinction between the element types in the control's data source. However, because of automatic type inference and conversion, the housekeeping code for *DataBinder.Eval* can result in a slightly slower server response.

The *DataBinder.Eval* method has two prototypes:

```
public static Object Eval(Object container, String expr);
public static String Eval(Object container, String expr, String format);
```

The first argument identifies the data provider to which the next expression must be applied. The third argument, which is optional, specifies a format string that converts the results of evaluating the data-binding expression to

String. When you need to apply formatting, *DataBinder.Eval* helps considerably to build a more readable statement. The following two instructions obtain the same result—formatting a numeric column as a currency value—but the first is easier to read and understand, albeit slightly slower:

```
<%# DataBinder.Eval(Container.DataItem, "Price", "{0:c}") %>
<%# String.Format("{0:c}", ((DataRowView) Container.DataItem)["Price"]) %>
```

> **Caution** Using *DataBinder.Eval* does not give you more programming power. It just makes programming easier because it shields you from many of the casting and formatting details. In return for this, it adds a little more overhead to your code, so try to use it only when necessary, such as when you need formatting.

Repeater Control Events

In addition to firing the standard events supported by all .NET controls (such as *Init*, *Load*, *PreRender*, and *DataBinding*), the *Repeater* control can fire three of its own events, which are described in Table 1-4.

Table 1-4 Events Supported by the *Repeater* Control

Event	Description

ItemCreated	Fires whenever a new item is created, regardless of its type (header, footer, item, or separator) or its ordinal position.
	`ItemCreated(Object s,`
	`  RepeaterItemEventArgs e)`
	The event structure makes available to your code the current item (the *Item* property), the index (the *ItemIndex* property), the type (the *ItemType* property), and the *DataItem* associated with it (the *DataItem* property).

ItemCommand	Fires whenever a user clicks a link button or a push button embedded in the control's template. `ItemCommand(Object s, RepeaterItemEventArgs e)` The event structure provides the *CommandSource* property to let you know about the button that triggered the event.
ItemDataBound	Fires when an item is data bound, always before it is rendered. `ItemDataBound(Object s, RepeaterItemEventArgs e)`

The *ItemCommand* event interface deserves a little more explanation because it is a peculiarity of iterative control and, like the *Repeater* control, is widely used with the *DataList* and *DataGrid* controls. *ItemCommand* fires whenever the user clicks any button in the *Repeater* control. You retrieve the current instance of the clicked object by using the *CommandSource* property. You can give each button control a name that identifies its action. You retrieve this name by using the *CommandName* property of *RepeaterItemEventArgs*.

Figure 1-4 shows an HTML table that was generated by using the *Repeater* control. This figure displays a few fields selected from the Northwind *Employees* database. (The full source code for the EmployeeList.aspx application is available on the companion CD.)

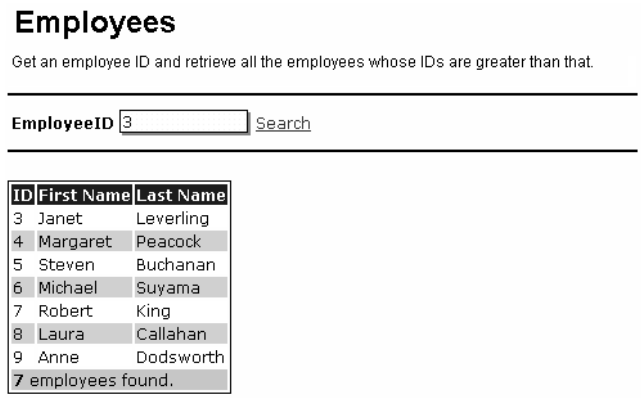


Figure 1-4 A data bound HTML table created using the *Repeater* control.

When the user clicks the Search button on the screen in Figure 1-4, the page runs a query against SQL Server and stores the default view of the retrieved table as the data source of the *Repeater* control.

Repeater has the following structure:

```
<asp:repeater runat="server" id="Repeater1">
    <HeaderTemplate> ... </HeaderTemplate>
    <ItemTemplate> ... </ItemTemplate>
    <AlternatingItemTemplate> ... </AlternatingItemTemplate>
    <SeparatorTemplate> ... </SeparatorTemplate>
    <FooterTemplate> ... </FooterTemplate>
</asp:repeater>
```

For each template, you provide the ASP.NET code that will be evaluated dynamically during the execution of the *DataBind* method. You don't need to specify all the templates. However, for a reasonably complex control, you need to specify at least the header and the item. Let's take a look at each of the templates for the example in Figure 1-4.

The *Header* template is processed only once and is responsible for opening the table and defining the caption.

```
<table style="border:1px solid black;" class="stdtext">
    <thead bgcolor="blue" style="color:white">
        <td><b>ID</b></td>
        <td><b>First Name</b></td>
        <td><b>Last Name</b></td>
    </thead>
```

If you don't define the same number of headers as expected columns, the graphical effect will be poor but you won't get run-time errors.

Items and alternating items are element types that the control can represent with different graphical settings. Suppose your goal is to build an HTML table. As the following code illustrates, the item templates can add only the necessary number of rows and cells for the overall buffer.

```
<ItemTemplate>
    <tr>
        <td bgcolor="white"> <%# DataBinder.Eval(Container.DataItem,
            "EmployeeID") %> </td>
        <td bgcolor="white"> <%# DataBinder.Eval(Container.DataItem,
            "FirstName") %> </td>
        <td bgcolor="white"> <%# DataBinder.Eval(Container.DataItem,
            "LastName") %> </td>
    </tr>
</ItemTemplate>

<AlternatingItemTemplate>
    <tr>
        <td bgcolor="lightblue"> <%# DataBinder.Eval(Container.DataItem,
            "EmployeeID") %> </td>
        <td bgcolor="lightblue"> <%# DataBinder.Eval(Container.DataItem,
            "FirstName") %> </td>
        <td bgcolor="lightblue"> <%# DataBinder.Eval(Container.DataItem,
            "LastName") %> </td>
    </tr>
</AlternatingItemTemplate>
```

For any of the fields, an alternative approach would be to use more direct syntax and avoid *DataBinder.Eval*, as shown here:

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

Like the *Header* template, the *Footer* template is drawn only once and only when all items are rendered.

```
<FooterTemplate>
    <tfoot>
        <td bgcolor="silver" colspan="3">
```

```
            <%# "<b>" + ((DataView) Repeater1.DataSource).Count +
                "</b> employees found." %>
        </td>
    </tfoot>
</table>
</FooterTemplate>
```

The *Footer* template defines the final row by using a <*tfoot*> HTML tag and makes sure that the final row spans all columns. Typically, you use the footer to display summary information. To obtain the total number of employees displayed, the code resorts to the *Count* property of the *DataView* class. To get the same information with a *DataTable* set as the data source, you would use the following expression:

```
<%# ((DataTable) Repeater1.DataSource).Rows.Count +
    " employees found." %>
```

> **Note** If *DataSource* is set but no data is returned, the *Repeater* control attempts to render only the header and footer templates, if either is specified. No items and separators are rendered. If the *DataSource* property is not set, the *Repeater* control is not rendered.

The *DataList* Control

If in the previous example you absentmindedly replaced *Repeater* with *DataList*, your code would run perfectly and function in the same way. However, the two controls are not identical. *DataList*, which is a data bound control, offers you a full bag of new features, mostly in the area of graphical layout—in fact, its set of features begins where the set of features supported by the *Repeater* control ends. For example, it supports directional rendering in which items can flow horizontally or vertically, depending on the specified number of columns. *DataList* provides facilities to retrieve a key value associated with a current row in the data source and has built-in support for selection and editing. *DataList* also supports more templates and can fire more events than *Repeater*.

Basic data-binding operations that use *DataList* are similar to those that use *Repeater*. You use the *DataSource* property to bind the control to data and the *DataBind* method to refresh the user interface.

```
DataList1.DataSource = ds.Tables["EmpTable"];
DataList1.DataBind();
```

In template code, you can access data bound information using both the *DataBinder* object and the *Container.DataItem* expression.

Templates Specific to *DataList*

The *DataList* control supports two templates that we haven't yet discussed: *SelectedItemTemplate* and *EditItemTemplate*. *SelectedItemTemplate* controls how the selected item is displayed. You should use it only when you need to display other controls or apply a particular logic. When you want to change only the appearance of the selected item, resort to a simpler cascading style sheet (CSS) style set via the *SelectedItemStyle* property.

EditItemTemplate specifies the graphical template for the row currently being edited. If you don't specify your own template, the standard template is used, which renders any data bound element by using text boxes. Note that the template defines the layout of the control only when in edit mode. Managing the user interaction, saving the edit, or canceling the edit operation must be handled separately. For each template, you can use special style tags to set all the CSS attributes that you need via a declaration. These style tags are *<AlternatingItemStyle>*, *<ItemStyle>*, *<HeaderStyle>*, *<FooterStyle>*, *<SeparatorStyle>*, *<SelectedItemStyle>*, and *<EditItemStyle>*. The following code assigns a white background to the items and a thick black border:

```
<ItemStyle BorderColor="black" BorderWidth=3 BackColor="white" />
```

You can also use style properties that have the same name as the style tags, as shown in this code:

```
DataList1.ItemStyle.BackColor = Color.White;
```

> **Note** In ASP.NET pages, you indicate colors by using strings. The ASP.NET run time silently converts the color name to a valid .NET type. To assign a color programmatically, you must use the predefined values of the *Color* enumeration in the *System.Drawing* namespace. If you don't know the .NET name of a given color, use any of the following expression formats:
>
> ```
> Color.FromName("white");
> ```
>
> ```
> Color.FromName("#00ee00");
> ```

Events Specific to *DataList*

All iterative controls fire a generic *ItemCommand* event when a user clicks a link button or a push button, and this button is associated with a command name that is specified by the programmer. You use the command name within the *ItemCommand* event to recognize which button has actually been clicked. If these links and buttons are automatically generated during the execution of *DataBind*, it is especially difficult for you to assign a unique value to the *OnClick* event for each control. The next code example clarifies my point. The *DataList* control generates buttons based on the content of an array of strings.

```
<asp:DataList runat="server" id="DataList1"
    RepeatDirection="Horizontal"
    OnItemCommand="HandleItemCommand">

<ItemTemplate>
    <asp:linkbutton runat="server" CommandName="<%# Container.DataItem %>"
        Text="<%# Container.DataItem %>" />    
</ItemTemplate>

</asp:DataList>
```

The data source returned by the *CreateDataSource* function generates the output shown in Figure 1-5.

```
public ArrayList CreateDataSource()
{
    ArrayList a = new ArrayList();
    a.Add("Search");
    a.Add("Execute");
    a.Add("Load");
    return a;
}
```

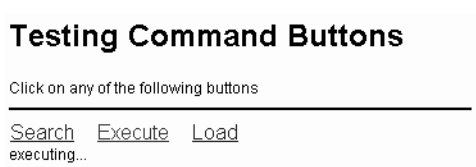


Figure 1-5 A button bar generated using *DataList*. The *ItemCommand* event is the only way you can handle the click event for buttons.

Within the body of the *HandleItemCommand* function, you can analyze the command name and decide the proper action to take.

```
void HandleItemCommand(Object sender, DataListCommandEventArgs e)
{
    switch (e.CommandName)
    {
        case "Search":
            statusbar.Text = "searching...";
            return;
        case "Execute":
            statusbar.Text = "executing...";
            return;
        case "Load":
            statusbar.Text = "loading...";
            return;
    }
}
```

The full source for the CommandButtons.aspx application is available on the companion CD.

Special Command Names

The *DataList* control provides special support for five predefined command names: *edit*, *update*, *delete*, *cancel*, and *select*. Any link or button in a *DataList* control whose command name matches one of these five keywords receives special treatment from the control. The predefined events that fire when the user clicks buttons associated with these strings are listed in Table 1-5.

Table 1-5 Predefined Events of the *DataList* Control

Event	Description
CancelCommand	Fires when a button named Cancel is clicked.
DeleteCommand	Fires when a button named Delete is clicked.
EditCommand	Fires when a button named Edit is clicked. When a user clicks the Edit button, the item automatically enters edit mode. Edit mode implies use of the *Edit-Item* template.
UpdateCommand	Fires when a button named Update is clicked.
SelectedIndexChanged	Fires when a button named Select is clicked. Clicking a Select button automatically deselects the current item and selects the new item.

Edit mode and select mode are graphically rendered through their specific templates and CSS style.

> **Note** Clicking a button named with a predefined keyword generates two events: first the generic *ItemCommand* event with the *CommandName* property set to the keyword and then the button-specific event (*UpdateCommand*).

Relating Graphical and Data Elements

To enable item selection and in-place editing, the *DataList* control has an internal mechanism that associates graphical items with corresponding elements in the data source. To retrieve this association, you set the *DataKeyField* property to the name of one data source field. The selected field must have content that allows you to uniquely identify the data item (a primary key). For example, if your data source is a table with information about a company's employees, you could assign the *employeeid* field to *DataKeyField*, as shown here:

```
<asp:DataList runat="server" DataKeyField="employeeid">
```

After *DataKeyField* is set, the control ensures that the *DataKeys* collection contains all the field's values for each item the control is currently displaying. You retrieve the key value based on the ordinal position of the item. For example, to obtain the key value of the currently selected item, you would do this:

```
int nEmployeeID = DataKeys[DataList1.SelectedIndex];
```

> **Caution** The number of displayed items does not necessarily match the number of items in the data source. In some situations, to handle a special application's requirements, you might need to programmatically hide some of the items. For example, you have to do this with *DataGrid* controls when pagination is enabled. (We'll talk more about *DataGrid* controls later in this chapter.)

The following code example illustrates how to take advantage of the *DataList* control to create a "smart" list box, and Figure 1-6 shows the results. (The full source code for the SmartList.aspx application is available on the companion CD.) All items in the data source are rendered as a link button with the

command name *select*. As you know, the *DataList* control provides special support for *select*. When the user clicks any of the selectable links, ASP.NET fires a *SelectedIndexChanged* event and updates the style of the clicked item according to the attributes set through the *SelectedItemStyle* tag. The appearance of the button is refreshed to reflect the new state. In response to the selection event, additional information about the employee is retrieved and displayed. A new button appears at the bottom of the page to let the user deselect any selected item.

When the page first loads, it retrieves only the fields needed to prepare the list of employees—that is, *employeeid*, *firstname*, and *lastname*. When the user clicks a link button, the *DataList* control is re-bound and an informative label appears with a message.

```
public void OnSearch(Object sender, EventArgs e)
{
    DataSet ds = new DataSet();

    String sConn = "server=localhost;uid=sa;Initial Catalog=Northwind;";
    String sText = "SELECT employeeid, firstname, lastname FROM Employees";
    sText += " WHERE employeeid >=" + tbEmpID.Text;
    SqlDataAdapter cmd = new SqlDataAdapter(sText, sConn);
    cmd.Fill(ds, "EmpTable");

    DataList1.DataSource = ds.Tables["EmpTable"].DefaultView;
    DataList1.DataBind();

    theLabel.Visible = true;
    theLabel.Text = "Click to read more.";
}
```

The *DataList* control has the following structure:

```
<asp:DataList runat="server" id="DataList1"
    DataKeyField="employeeid"
    OnSelectedIndexChanged="HandleSelection">

<SelectedItemStyle BackColor="lightblue" />

<HeaderTemplate>
    <h3>Northwind's Employees</h3>
</HeaderTemplate>

<ItemTemplate>
    <asp:linkbutton runat="server" commandname="select"
        Text='<%# ((DataRowView)Container.DataItem)["employeeid"] + " - " +
             ((DataRowView)Container.DataItem)["lastname"]  + ", " +
             ((DataRowView)Container.DataItem)["firstname"] %> ' />
</ItemTemplate>

<FooterTemplate> <hr> </FooterTemplate>
</asp:DataList>
```

To retrieve additional information about the clicked item, you need to issue another query based on a key value. You retrieve the key value associated with the selected item by using the *DataKeys* array.

```
void HandleSelection(Object sender, EventArgs e)
{
    int nEmpID = (int) DataList1.DataKeys[DataList1.SelectedIndex];

    String sConn = "server=localhost;uid=sa;Initial Catalog=Northwind;";
    SqlConnection cn = new SqlConnection(sConn);

    String sText = "SELECT title, hiredate, country, notes FROM Employees";
    sText += " WHERE employeeid = " + nEmpID.ToString();
    SqlCommand cmd = new SqlCommand(sText, cn);
    cn.Open();

    SqlDataReader dr = cmd.ExecuteReader();
    dr.Read();

    theLabel.Text =  "<b>" + dr["title"] + "</b><br>";
    DateTime dtime = Convert.ToDateTime(dr["hiredate"]);
    theLabel.Text += "Hired on " + dtime.ToShortDateString() + " from " +
                     dr["country"] + "<hr>";
    theLabel.Text += "<i>" + dr["Notes"] + "</i>";

    btnUnselect.Visible = true;
    dr.Close();
    cn.Close();
}
```

Using the *SqlDataReader* control instead of the *DataSet* control to retrieve data is more efficient when you have just one row to fetch. After the data has been successfully read, it gets properly formatted and the Unselect button is displayed.

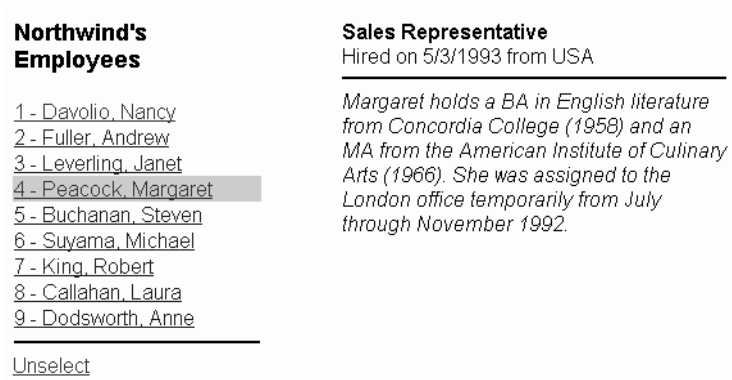


Figure 1-6 The user interface of the application changes significantly when an employee name is selected. Additional information about the employee is retrieved and displayed, and a new button appears to let you deselect the currently selected item.

To enable the user to deselect an item, set the *SelectedIndex* property to –1.

```
void RemoveSelection(Object sender, EventArgs e)
{
    DataList1.SelectedIndex = -1;
    theLabel.Text = "Click to read more.";
    btnUnselect.Visible = false;
}
```

The *DataList* control and the panel that displays additional employee information are two cells of the same all-encompassing table. By using different settings for the layout properties of *DataList,* you can obtain a significantly altered design for the links without affecting the core of the code.

As the preceding code example illustrates, the *DataList* control is much more powerful than the *Repeater* control, but it is far from being perfect. It still lacks pagination, sorting, and powerful column-based rendering capabilities.

The *DataGrid* Control

The *DataGrid* control renders a multi-column, fully templated grid and is by far the most versatile of all data bound controls. The user interface it provides closely resembles a Microsoft Excel worksheet. Despite its rather advanced programming interface and full set of attributes, *DataGrid* simply generates an HTML table with interspersed links to provide interactive functionality such as sorting and pagination commands.

With the *DataGrid* control, you can create simple data bound columns that show data retrieved from a data source, template columns that let you design the layout of cell contents, and last but not least, command-based columns that allow you to add specific functionality to a grid. In the next chapters, I'll delve more deeply into the implementation and the customizable features of the *DataGrid* control. But I think a discussion of the features that differentiate *DataGrid* from *DataList* is useful here.

The *DataGrid* control uses templates extensively but differently from the *DataList* and *Repeater* controls. *DataGrid* renders tables of data organized in columns, so the templates don't apply to the control as a whole but rather to specific columns. *DataList* and *Repeater* work on a per-item basis and make you responsible for controlling the final layout of the data. *DataGrid* has just one possible layout—a sequence of columns—and supports the same events as the *DataList* control plus two more: *PageIndexChanged* and *SortCommand*.

Now let's take a look at the *DataGrid* control in action. As you've learned, the *DataGrid* control works by displaying columns of data. In most cases, you need to specify which columns you want and how you want them displayed, but when the data to be rendered is uncomplicated, you can leave the control free to automatically generate the columns based on the structure of the data

source. When you leave the control in charge, however, you cannot control the heading of each column, which defaults to the column name. The following code arranges a *DataGrid* control that uses alternating rows and automatically generates the columns. The results are shown in Figure 1-7. The full source code for the DataGrid.aspx application is available on the companion CD.

```
<asp:DataGrid id="grid" runat="server"
    AutoGenerateColumns="true"
    CellPadding="2" CellSpacing="2" GridLines="none"
    BorderStyle="solid" BorderColor="black" BorderWidth="1"
    font-size="x-small" font-names="verdana">

    <AlternatingItemStyle BackColor="palegoldenrod" />
    <ItemStyle BackColor="beige" />
    <HeaderStyle ForeColor="white" BackColor="brown" Font-Bold="true" />
</asp:DataGrid>
```

Notice that you can use the style properties to declare CSS styles.

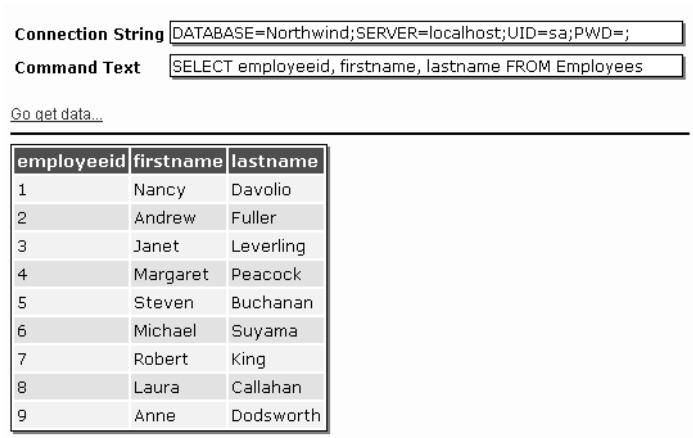


Figure 1-7 The columns displayed in this figure are created using the *DataGrid* control.

The next code example illustrates how to fill the grid and refresh the user interface.

```
void OnLoadData(Object sender, EventArgs e)
{
    SqlConnection conn = new SqlConnection(txtConn.Text);
    SqlDataAdapter da = new SqlDataAdapter(txtCommand.Text, conn);
    DataSet ds = new DataSet();
    da.Fill(ds, "MyTable");

    grid.DataSource = ds.Tables["MyTable"];
    grid.DataBind();
}
```

Conclusion

In this chapter, you learned about the important controls and techniques you can use to bind data and .NET controls. We discussed simple data-binding expressions and then moved on to grid controls, touching also on list and iterative controls. Data binding coverage doesn't end here, though, as it is too important a technology for building Web solutions with ASP.NET and ADO.NET. In the next chapter, you'll learn about how to populate a *DataGrid* control by using data that takes advantage of sophisticated features such as pagination and sorting.

2

Pageable Data Grids

Data reporting is key to Web applications, and the *DataGrid* control is the ASP.NET control of choice for data reporting. The *DataGrid*, which dynamically expands to a table, includes a number of features that customize the look and feel of the application and enhance its overall functionality. You set the content of *DataGrid* by using its *DataSource* property. As explained in Chapter 1, this content consists of any homogeneous collection of data. You can programmatically access this collection by using the *Items* property. Any item in the collection is represented by a *DataGridItem* object.

Constituent Items of *DataGrid*

The user interface of a *DataGrid* control comprises several types of items. These item types are grouped in the *ListItemType* enumeration. Each item plays a clear role and has a precise location in the hierarchy of the control's user interface, as Figure 2-1 shows.

A *DataGrid* control is formed by using any combination of eight different items. Each item maps to a template style property of the *DataGrid* control, but remember from Chapter 1 that templates for *DataGrid* differ from those of *DataList* because they apply to the columns rather than to the control. Table 2-1 describes the graphical items that form an ASP.NET *DataGrid* control.

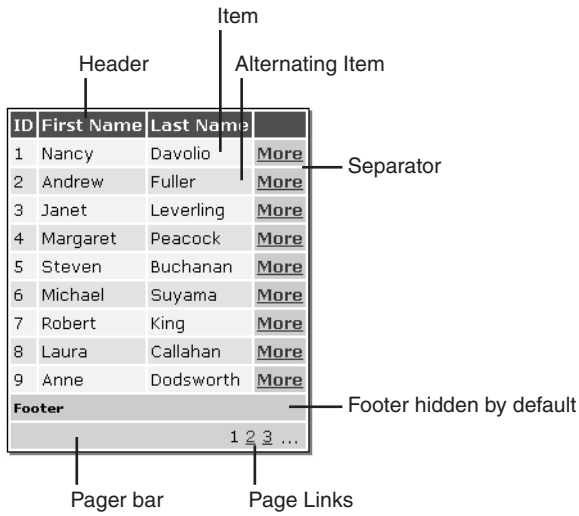


Figure 2-1 The layout of a *DataGrid* control.

Table 2-1 Graphical Items of the *DataGrid* Control

Item type	Description
Header	The grid's header. This item cannot be data bound. The *HeaderStyle* property lets you control the look and feel of the header.
Item	A normal data bound row displayed in the grid. The *ItemStyle* property lets you control the look and feel of the row.
Separator	An item used to separate rows. This item type is not data bound.
AlternatingItem	A data bound row in an odd-numbered position. This item type is useful when you want to use different styles for alternating rows. The *AlternatingItemStyle* property lets you control the look and feel of the row.
Footer	The grid's footer. This item type cannot be bound to a data source. The *FooterStyle* property lets you control the look and feel of the footer.
EditItem	The grid item currently in edit mode. This item type can be data bound. The *EditItemStyle* property lets you control the look and feel of the edited item.
Pager	The pager bar item you can place at the bottom of the grid to enable scrolling between pages. This item type cannot be data bound. The *PagerStyle* property lets you control the look and feel of the pager bar.
SelectedItem	The grid item currently selected. This item type can be data bound. The *SelectedItemStyle* property lets you control the look and feel of the selected item.

Each time an item in Table 2-1 is about to be created, an *ItemCreated* event is fired. By handling this event, you can customize the way in which the item—for example, a column caption or the pager bar—is rendered. Note that the *ItemCreated* event is not raised for each column but only once when the *DataGrid* control has finished creating the whole item. For example, the footer and header events are raised only when the header and the footer for each column have been created. Later in this chapter, you'll learn to use the *ItemCreated* event.

Column Types

A *DataGrid* control is formed by data bound columns. The control has the ability to automatically generate columns that are based on the structure of the data source. Auto-generation is the default behavior, but you can manipulate that behavior by using a Boolean property named *AutoGenerateColumns*. Set the property to *false* when you want the control to display only the columns you explicitly add to the *Columns* collection. Set it to *true* (the default) when you want the control to add as many columns as is required by the data source. The default algorithm for column generation creates simple data bound columns that use literal controls to display the contents of the corresponding data source fields. The *DataGrid* control supports additional column types that can render the column data so that it performs an action when clicked. The column types are explained in Table 2-2. Each is derived from the *DataGridColumn* class.

Table 2-2 Column Types Supported by the *DataGrid* Control

Column Type	Description
BoundColumn	Displays a column that is bound to a field in a data source. It displays each item in the field as plain text.
ButtonColumn	Displays a command button for each item in the column. The text of the button can be data bound. The command name of the button must be common to all items in the column. When the value for the command name is *Select*, clicking the column button automatically selects the row.
EditCommandColumn	Displays a button column that is automatically associated with the *Edit* command. This class receives special support from the *DataGrid* control, which is manifested as a redrawing of the clicked row using a template. (I'll talk more about this in Chapter 4.)

(continued)

Table 2-2 Column Types Supported by the *DataGrid* Control *(continued)*

Column Type	Description
HyperLinkColumn	Displays the contents of each item in the column as a link. The link text can be bound to a field in the data source, or it can be static text. Also, the target URL can be data bound. Clicking a link column causes the browser to jump to the specified URL. This column class supports target frames.
TemplateColumn	Displays each item in the column according to a specified template, allowing you to provide custom controls in the column. (I'll talk more about this in Chapter 3.)

Binding Columns

In real-world scenarios, when you create columns, you need more options than auto-generation offers. Auto-generation does not let you specify the header text, nor does it provide text formatting. Perhaps the strongest reason to opt for manual binding is that auto-generation always displays *all* columns in the data source in the order in which they are returned.

> **Caution** If you plan to manually bind columns to your *DataGrid* control, make sure you explicitly set *AutoGenerateColumns* to *false*. You can do that either declaratively or programmatically. Otherwise, the *DataGrid* control will place extra columns—all the automatically generated columns—after the last column you manually create.

You typically bind columns using the <*columns*> tag in the body of the <*asp:datagrid*> server control.

```
<asp:datagrid runat="server" id="grid" ... >
    ⋮
    <columns>
        <asp:BoundColumn runat="server" DataField="employeeid"
            HeaderText="ID" />
        <asp:BoundColumn runat="server" DataField="firstname"
            HeaderText="First Name" />
        <asp:BoundColumn runat="server" DataField="lastname"
            HeaderText="Last Name" />
    </columns>
</asp:datagrid>
```

Alternatively, you can create a new column of the desired class, fill its members, and then add the class instance to the *DataGrid* control's *Columns* collection by using the *Add* method. Here is some code that adds a *BoundColumn* class to a grid:

```
BoundColumn bc = new BoundColumn();
bc.DataField = "firstname";
bc.HeaderText = "First Name";
grid.Columns.Add(bc);
```

The order of columns in the collection determines the order in which the columns are displayed in the *DataGrid* control.

Bound Columns

The *BoundColumn* class displays in a single column the contents of a field in the data source. The structure of *BoundColumn* is characterized by the properties in Table 2-3.

Table 2-3 Structural Properties of the *BoundColumn* Class

Property	Description
DataField	The field name from the data source to which the column is bound.
DataFormatString	A string that specifies the display format for items in the column.
FooterText	The text displayed in the footer section of the column.
HeaderImageUrl	The URL of an image that will appear in the header section of the column.
HeaderText	The text displayed in the header section of the column.
ReadOnly	Indicates whether the items in the *BoundColumn* can be edited. If set to *true*, the column won't be affected by the edit mode.
SortExpression	The sort expression used when the column is selected for sorting.

All the properties listed in Table 2-3 can be declared in the ASP.NET page layout or set programmatically. The names of the class members match the <*asp:BoundColumn*> tag attributes.

In addition to the properties listed in Table 2-3, the *BoundColumn* class supports the following style properties:

- **FooterStyle** Represents the style properties for the footer
- **ItemStyle** Represents the style properties for all column cells
- **HeaderStyle** Represents the style properties for the header

The graphical style of a column is primarily influenced by these properties. If you don't set any of them, the column will inherit the styles set at the level of the *DataGrid* control.

Button Columns

A button column contains a user-defined command button that corresponds to each row in the column. You specify the button caption by setting the *Text* property of the *ButtonColumn* class. Alternatively, you can bind the caption displayed in a command button to a single field in the data source by setting the *DataTextField* property. Clicking a command button in a column raises the *ItemCommand* event, which you can programmatically respond to by providing an event handler. Table 2-4 shows the key properties of the *ButtonColumn* class.

Table 2-4 Key Properties of the *ButtonColumn* Class

Property	Description
ButtonType	The type of button that will be displayed in the column. Values are defined in the *ButtonColumnType* enumeration: *LinkButton* (the default) and *PushButton*.
CommandName	The name of the command invoked when a user clicks a button in the column.
DataTextField	The field name from the data source to which the column is bound. If you set this property, the caption of each button will reflect the contents of the corresponding data source row.
DataTextFormatString	The display format for the caption in each command button.
FooterText	The text displayed in the footer section of the column.
HeaderImageUrl	The URL of an image that will appear in the header of the column.
HeaderText	The text displayed in the header section of the column.
SortExpression	The sort expression to use when the column is selected for sorting.
Text	The text displayed as the button caption. If you set this property, all buttons have the same text.

A button column allows you to associate an action with the items displayed in a column, but in most cases, you will want to use static text to identify the action (for example, Add, More, Remove). You can also define the button's caption to depend on the underlying data source by setting *DataTextField*.

The following code is a simple but realistic example of how to create and use a button column in a *DataGrid* control. Figure 2-2 shows the corresponding output.

```
<asp:DataGrid id="grid" runat="server"
    AutoGenerateColumns="false"
    CellPadding="2" CellSpacing="2" GridLines="none"
    BorderStyle="solid" BorderColor="black" BorderWidth="1"
    font-size="x-small" font-names="verdana"
    DataKeyField="employeeid"
    OnItemCommand="HandleCommands">

    <AlternatingItemStyle BackColor="palegoldenrod" />
    <ItemStyle BackColor="beige" />

    <columns>
        <asp:BoundColumn runat="server" DataField="employeeid"
            HeaderText="ID" />
        <asp:BoundColumn runat="server" DataField="firstname"
            HeaderText="First Name" />
        <asp:BoundColumn runat="server" DataField="lastname"
            HeaderText="Last Name" />
        <asp:ButtonColumn runat="server" Text="More"
            CommandName="moreinfo">
            <itemstyle backcolor="lightblue" font-bold="true" />
        </asp:ButtonColumn>
    </columns>
</asp:DataGrid>
```

ID	First Name	Last Name	
1	Nancy	Davolio	More
2	Andrew	Fuller	More
3	Janet	Leverling	More
4	Margaret	Peacock	More
5	Steven	Buchanan	More
6	Michael	Suyama	More
7	Robert	King	More
8	Laura	Callahan	More
9	Anne	Dodsworth	More

Figure 2-2 A *DataGrid* control with a button column that can display additional information for the clicked data item.

When a user clicks a button column, to execute code in response, you must write an *ItemCommand* handler and link it to the *DataGrid* control by using the *OnItemCommand* attribute.

```
<asp:datagrid runat="server" ...
    OnItemCommand="ItemCommandHandler">
```

An *ItemCommand* handler requires the following prototype:

```
void ItemCommandHandler(Object sender, DataGridCommandEventArgs e)
```

The *DataGridCommandEventArgs* class makes available three key properties:

- **Item** Represents the *DataGridItem* element that was clicked
- **CommandName** Represents the command name associated with the clicked button
- **CommandSource** Represents the button object that raised the click event

The following code shows how to retrieve and display more information about the clicked data item. The full source code for the CmdColumns.aspx application is available on the companion CD.

```
void HandleCommands(Object sender, DataGridCommandEventArgs e)
{
    if (e.CommandName == "moreinfo")
    {
        int nEmpID = (int) grid.DataKeys[e.Item.ItemIndex];

        SqlConnection conn = new SqlConnection(txtConn.Text);
        String strCmd = "SELECT * FROM Employees " +
            "WHERE employeeid = " + nEmpID.ToString();
        SqlCommand cmd = new SqlCommand(strCmd, conn);
        conn.Open();
        SqlDataReader dr = cmd.ExecuteReader();
        dr.Read();
        MoreInfo.Text = BuildMoreInfoText(dr);

        dr.Close();
        conn.Close();
    }
}
```

The expression *e.Item.ItemIndex* evaluates to the ordinal position of the clicked item. As explained in Chapter 1 when we discussed *DataList* controls, the *DataKeys* array can contain a primary key value for each displayed item in the column. As the preceding code illustrates, the event handler retrieves this value and arranges a new query that requests more information for that particular row. This information is then organized into a readable format, as shown in Figure 2-3.

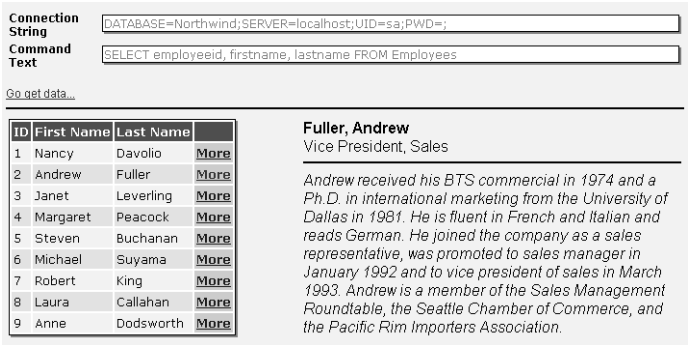


Figure 2-3 Clicking the More column button in this *DataGrid* control displays additional information about the item.

Link Columns

The *HyperLinkColumn* class works in much the same way as the *ButtonColumn* class, but *HyperLinkColumn* allows you to jump to a different URL within a specified frame. You set the *Text* property to specify the caption for the link. To specify the actual URL the user will navigate to when the link is clicked, you set the *NavigateUrl* property—if you set *Text*, all links in the column will share the same caption. Alternatively, you can bind the caption and URL in the column to a field in a data source by using the property pair *DataTextField* and *DataNavigateUrlField*. Table 2-5 shows the key properties of the *HyperLinkColumn* class.

> **Note** The *DataTextField* and *Text* properties are assumed to be mutually exclusive. The *HyperLinkColumn* class assumes that you never have both set at the same time. If you do, *DataTextField* takes precedence and *Text* will be ignored.

Table 2-5 Key Properties of the *HyperLinkColumn* Class

Property	Description
DataNavigateUrlField	The field name from the data source that determines the URL to jump to when a user clicks a link.
DataNavigateUrlFormatString	Specifies the display format for the URL of the links, letting you create parametric URLs.

(continued)

Table 2-5 Key Properties of the *HyperLinkColumn* Class *(continued)*

Property	Description
DataTextField	The field name from the data source to which the column is bound. If you set this property, the caption of each button will reflect the contents of the corresponding data source row.
DataTextFormatString	The string that specifies the display format for the caption in each command button.
FooterText	The text displayed in the footer section of the column.
HeaderImageUrl	The URL of an image to appear in the header section of the column.
HeaderText	The text displayed in the header section of the column.
NavigateUrl	The URL to jump to when the user clicks a link in the column.
SortExpression	The string that specifies the sort expression to use when the user selects the column for sorting.
Target	The target window or frame where the linked Web page should be displayed.
Text	The text displayed as the button caption. If you set this property, all the buttons have the same text.

For link columns, the formatting capabilities provided by the *DataGrid* control are extremely helpful because they allow you to arrange parametric URLs. Let's alter our previous code example and point users that want to know more about a certain employee to a Web page. Instead of a button column, use the following link column:

```
<asp:HyperLinkColumn runat="server"
    DataNavigateUrlField="employeeid"
    DataNavigateUrlFormatString="moreinfo.aspx?id={0}"
    DataTextField="lastname"
    DataTextFormatString="More on {0}"
    Target="frInfo" />
```

In .NET, the expression *{n}* is used to indicate the *n*th argument of a series in much the same way as the old acquaintance *%n* is used with the function *printf* in plain old C programs. When used with the *DataGrid* control, however, the *{n}* expression is limited to just one argument. Thus the following assignments have only one placeholder that is silently determined by the *DataGrid* control:

```
DataNavigateUrlFormatString="moreinfo.aspx?id={0}"
DataTextFormatString="More on {0}"
```

DataNavigateUrlFormatString replaces its {0} placeholder with the content of the field pointed to by *DataNavigateUrlField*. Similarly, *DataTextFormatString* relies on the services of *DataTextField*. As a result, when the user clicks the cell of the employee whose ID is 5 (this field is *employeeid*), the URL that is linked to this is:

```
moreinfo.aspx?id=5
```

Figure 2-4 shows an application that uses a Web page (MoreInfo.aspx) and an embedded frame to display more information about a given employee. The full source code for the Hyperlinks.aspx application is available on the companion CD.

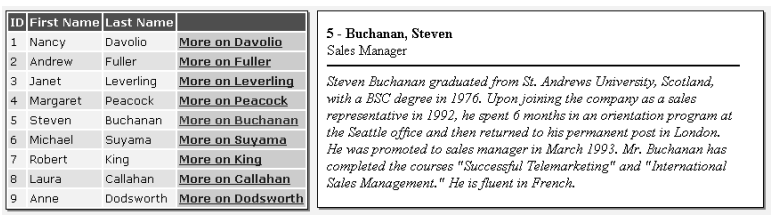


Figure 2-4 Additional information about employees can be displayed using an *<iframe>* tag and link columns.

To redirect the output of the linked URL, you can use the *Target* property to point to a particular window, frame, or floating frame. Floating frames are a browser-specific feature. Microsoft Internet Explorer renders them by using the *<iframe>* tag. Netscape browsers instead rely on the *<layer>* tag. The preceding code uses the following tag declaration:

```
<iframe frameborder="no" name="frInfo"></iframe>
```

Notice that for the code to work, you must close the *<iframe>* tag in the conventional way—that is, by using the closing tag *</iframe>*. Are you having difficulty seeing the shaded frame behind the white box in Figure 2-4? The rendering of the *<iframe>* tag is heavily affected by the *frameborder* attribute, which removes its ugly 3-D frame border, and the following CSS style:

```
<style>
    iframe {
        border:solid 1px black;
        filter:progid:DXImageTransform.Microsoft.dropshadow(OffX=2,
            OffY=2, Color='gray', Positive='true');
    }
</style>
```

The *filter* attribute can be used to give a shadow to almost all HTML tags. It works only with Internet Explorer 5 and later versions and is happily ignored on other browsers, so using it is harmless.

> **Note** For a page using child frames, the functionality of button and link columns is similar except when reviewing previously visited pages. If the user clicks a button column, the subsequent action is taken within the same page. If the user clicks a link column, the browser navigates to a new URL.

Programmatic Binding

The columns of a *DataGrid* control are stored in the *Columns* collection. The collection is filled out during the page initialization phase. When the *OnLoad* event for the page occurs, the *Columns* collection is ready for use. At this time, though, the only columns described in the collection are the columns declaratively linked through the <*columns*> ASP.NET tag. Can you bind columns programmatically? Can you show and hide columns on demand? The answer to both questions is yes.

Just as any other ASP.NET element, *DataGrid*'s columns are also rendered through objects. To add a new column programmatically, first create a new column using one of the column classes as appropriate. For example, you can create a data bound column using this code:

```
BoundColumn bc = new BoundColumn();
bc.DataField = strField;
bc.HeaderText = strHeader;
grid.Columns.Add(bc);
```

Adding link or button columns is not very different from creating a data bound column except that you have to use a different column class and might need to address a different set of properties.

With programmatic binding, the rub is that the *DataGrid* control does not persist the information about the new, programmatically bound columns. In fact, only the columns statically linked through the <*columns*> tag are automatically added to the *Columns* collection of the *DataGrid* object when the page posts back to the server. To work around this issue, make sure that you programmatically add your own extra columns in the *Page_Load* event handler:

```
public void Page_Load(Object sender, EventArgs e)
{
    if (!Page.IsPostBack)
    {
        // Execute only the first time...
        ⋮
    }
    else
        BindColumnsProgrammatically();
}
```

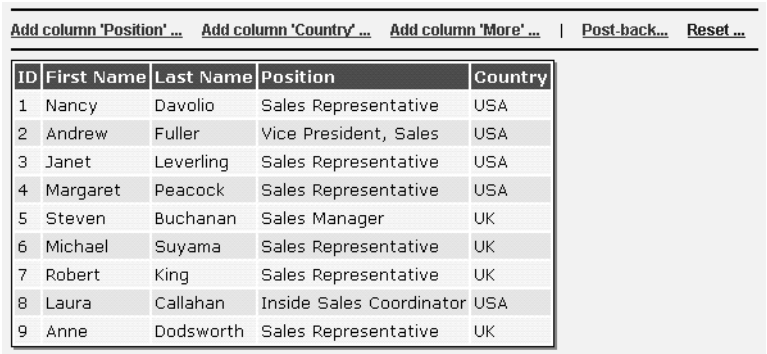
In this way, extra columns are added each time the page is requested, ensuring the *DataGrid* consistency. This solution, though, does not solve the second question raised above. What happens if columns can be added on demand by clicking a button or as the result of a certain operation? In these cases, you don't have a fixed set of columns to add through a user-defined method as in the previous code snippet.

Dynamic Column Binding

When the number of extra columns cannot be predetermined, to be able to persist them across multiple page requests, you can only track the added column and where in the grid it is added. Whenever the action that adds the column occurs, you must record the action on a persistent device and have enough information to allow you to repeat the column creation the next time the page is refreshed. You could persist column information in the page global *ViewState* property. However, since this information is eventually serialized into the page's hidden *__VIEWSTATE* field, which is sent back and forth between the server page and the client browser, you might want to keep the amount of information as lean as possible to reduce the page payload. Instead of serializing each column, you could store only the information that you need to re-create the column. In the sample application, ProgBinding.aspx, I persist a string made of comma-separated numbers, each of which identifies a particular column. To restore columns, the application retrieves all the column codes from the *ViewState* and runs a statement like the one shown here:

```
switch(colCode)
{
    case "1":
        AddBoundColumn("title", "Position");
        break;
    case "2":
        AddBoundColumn("country", "Country");
        break;
    case "3":
        AddButtonColumn("More", "moreinfo");
        break;
}
```

The full source code for the ProgBinding.aspx application is available on the companion CD. Figure 2-5 shows the dynamically displayed columns.



ID	First Name	Last Name	Position	Country
1	Nancy	Davolio	Sales Representative	USA
2	Andrew	Fuller	Vice President, Sales	USA
3	Janet	Leverling	Sales Representative	USA
4	Margaret	Peacock	Sales Representative	USA
5	Steven	Buchanan	Sales Manager	UK
6	Michael	Suyama	Sales Representative	UK
7	Robert	King	Sales Representative	UK
8	Laura	Callahan	Inside Sales Coordinator	USA
9	Anne	Dodsworth	Sales Representative	UK

Figure 2-5 Columns bound programmatically are persisted across multiple page requests.

ViewState vs. *Attributes*

ViewState is a global, page-scoped repository of persistent information. Each ASP.NET control manages internally its own *ViewState* repository, which is then poured out into the global *ViewState* object for the page.

The control-specific *ViewState* property is a protected member and cannot be accessed from ASP.NET code. As a result, applications cannot store persistent information within the state of a particular control using *ViewState*. To persist information on a per-control basis, applications could use the control's *Attributes* collection. *Attributes* is one of the properties that controls automatically store in their own *ViewState* repository, so from the applications' viewpoint the mechanism of persistence is exactly the same as with *ViewState*.

Although in practice you can interchangeably use *ViewState* and *Attributes* to persist information, they definitely have different design centers and have been introduced to serve clearly different purposes. *ViewState* is a collection of state values from the page and its controls. *Attributes* is meant to be the collection of HTML attributes for which ASP.NET controls do not provide an equivalent property. For example, you use *Attributes* to set the *src* attribute of an <*iframe*> tag or to assign a client-side JavaScript function to the *onclick* attribute of a button. All controls have (and persist) their own *Attributes* collection. Given its goal, the *Attributes* collection only accepts strings whereas *ViewState* can accept any valid .NET type. In addition, bear in mind that all attributes set through the *Attributes* collection are also written to the HTML page as custom attributes of the corresponding tags. (This is not necessarily a bad thing; just be aware of it.)

Data Pagination

Unlike the *DataList* control, the *DataGrid* control supports data pagination, that is, the ability to divide the displayed data source rows into pages. In real-world scenarios, the size of a data source easily exceeds the page real estate. So to preserve scalability on the server and to provide a more accessible page to the user, you display only a few rows at a time.

Pagination, however, requires a certain amount of work behind the scenes. The control must know how many items per page you want to display and then set up a pagination toolbar to let the user move between pages. In addition, the control must be able to download and process only the records that belong in the current page. Finally, the control must track and expose the index of the current page.

Setting up pagination for a *DataGrid* control is complicated because the control instance is on the Web server, not the client browser. The ASP.NET *DataGrid* control detects user events on the client, such as clicking a button to move to a new page. These events are automatically posted back to the Web server and processed. As a result, the browser receives a new version of the same page in which a new set of rows is shown.

What does this mean for your code? The Web server needs to know about the data source every time a postback event associated with the grid is generated. This is a key point that I'll deal with later in the chapter. For now, suffice to say that you have to figure out the best way to track the data source on the server any time processing is required, *and* you must provide the logic to carry out the desired tasks—paging, sorting, filtering, and so on. Although ASP.NET and ADO.NET classes help you implement a lot of these tasks, they don't automatically handle them all. For this reason, I like to refer to the pagination feature of *DataGrid* controls as *semi-automatic* built-in features.

Semi-Automatic Pagination

Before I discuss any further the programming details of pagination, let me spend some time explaining the overall programming interface of the *DataGrid* control so that you can understand the rationale behind some apparently odd design choices.

Remember that the *DataGrid* control—just like any other ASP.NET control—was designed to function on the Web as a bridge between the browser and Web server via postback events. To be effective, postback events need to coordinate state management with the client-side environment. The ASP.NET run time manages to restore the state of the page before actually invoking your initialization code (by using the *Page_Load* event and setting the *IsPostBack* property), and then it processes the event. To preserve any significant state

information, the run time needs to pack the information, send it to the client, and then retrieve it when the page is posted back for further processing. The control's view state (set in the *ViewState* property) is the architectural element responsible for preserving and retrieving state information.

For performance reasons, a *DataGrid* control does not store the data source within the view state. As a result, you are responsible for retrieving and making available on the server the *DataGrid* control's data source whenever it is needed. Possible options for doing this include reloading the data source from the database server each time the data source is needed or storing the data source in global objects such as *Session*, *Cache*, *Application*, or (and why not?) *ViewState* or in files. What you really need to know at this point in our discussion is that the code of the *DataGrid* control cannot make assumptions about how the data source is stored. When a postback event occurs, the *DataGrid* control's *DataSource* property is undefined, so you are responsible for filling (or refilling) it properly and then restoring data from the storage medium appropriate for the scalability of the application.

> **Caution** Do not access the *DataGrid* control's *DataSource* property in ASP.NET code without first taking precautionary measures, because the *DataGrid* control will not necessarily evaluate to a non-null object. Unless you take the appropriate counter-measures discussed later in this chapter, the *DataSource* property will be undefined when postback events are raised even though you correctly set the property when the page was first accessed. The *DataSource* property is not persisted.

So as you can see, because the *DataGrid* control by design cannot make assumptions about how its data source has been persisted on the server (the data source does not travel with the rest of the *DataGrid* control's programming interface on the Web), the control cannot implement a totally automatic engine for pagination or sorting.

To enable pagination of the *DataGrid* control, you need to tell the control about it. You do this through the *AllowPaging* property. When pagination is enabled, the rendering algorithm of the *DataGrid* control changes slightly. By default (the default is pagination turned off), the grid renders all the items in the data source in the order of the first item to the last. When pagination is on, the *DataGrid* control renders only the rows that fall in the range of the current page. The control keeps track internally of the requested page size, the subsequent

page count, and the current page index. Basically, it has the built-in capability to page through records for a given page size and data source, but it requires you to correctly set up the data source each time a new page is requested.

Properties for Pagination

Table 2-6 shows the *DataGrid* control properties that support pagination.

Table 2-6 Pagination Properties for the *DataGrid* Control

Property	Description
AllowPaging	Enables or disables pagination. This property is *false* by default.
CurrentPageIndex	Gets or sets the current page index.
PageSize	Gets or sets the current page size.
PageCount	Gets the number of pages according to the current page size.
AllowCustomPaging	Enables or disables custom pagination. This property is *false* by default.
VirtualItemCount	Gets or sets the total number of items you plan to display when using custom pagination. By contrast, the item count for default pagination is obtained from the data source.

The *DataGrid* control supports two flavors of pagination: default and custom. With default pagination, you simply provide the whole data source; the grid handles pagination internally. With custom pagination, you are expected to store in the *DataSource* property only those rows to be displayed on the current page; the *DataGrid* control renders all the rows it finds in the *Items* collection and sorts them from first item to last. So you must ensure that the collection corresponds to the current page. Furthermore, the index of the current page must be explicitly set, and the *DataGrid* control determines the value of the *PageCount* property by using *VirtualItemCount*. I'll cover this in more detail later on.

The Pager Bar

The pager bar is an interesting and complimentary feature offered by the *DataGrid* control to let users easily move from page to page. The pager bar is a row displayed at the bottom of the *DataGrid* control that contains links to available pages. When you click any of these links, the control automatically fires the *PageIndexChanged* event and updates the page index accordingly.

The pager bar offers two display options: it can show a generic pair of Next and Previous buttons or a detailed series of numeric buttons, each of which points to a particular page. (See Figure 2-6.) The next code fragment illustrates these options. You can also thoroughly customize the pager bar—determining its font, size, background and foreground colors, and alignment settings—or even hide the standard bar and roll your own. You control the pager bar by using the *PagerStyle* property's *Mode* attribute. Values for the *Mode* attribute come from the *PagerMode* enumeration.

```
if (UseNumericPages.Checked)
    grid.PagerStyle.Mode = PagerMode.NumericPages;
else
    grid.PagerStyle.Mode = PagerMode.NextPrev;
```

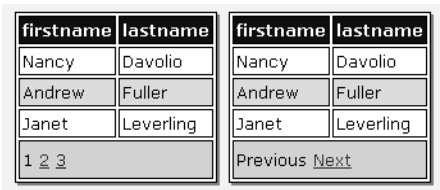


Figure 2-6 The pager bar's two default button options.

Page Button Count

When the pager mode is set to *NumericPages*, you can control the number of buttons displayed. You normally do this to avoid an endless (and thus unhelpful) list of numbers in the case of a very lengthy data source. By setting the *PageButtonCount* property, you define the maximum number of buttons you want the pager bar to host. The default value for *PageButtonCount* is 10.

```
grid.PagerStyle.PageButtonCount = 3;
```

If the *DataGrid* control has more pages than are specified in *PageButtonCount*, ellipsis buttons are displayed in the pager bar. When clicked, the ellipsis button displays the next or previous set of numeric buttons.

Page Button Text

Next and Previous are only the default labels for the pager bar. When the pager bar mode is set to *NextPrev*, you can set the text you want displayed for both buttons. To do so, you use any valid HTML expressions, including images.

```
grid.PagerStyle.PrevPageText = "<img src=prev.gif>";
grid.PagerStyle.NextPageText = "<b>Next...</b>";
```

Changing the labels for numeric buttons is a bit more complicated because you don't have predefined properties to set, but you can still change them. I'll discuss how later in this chapter.

Pagination in Action

To enable pagination in *DataGrid* controls, you must perform several tasks. First you have to enable paging at the *DataGrid* control level by setting the *AllowPaging* property to *true*. You must assign the desired value to *PageSize* to let the control know how many rows you want displayed for each page. (The default for *PageSize* is 10.)

Second you must make sure that the data source is properly refreshed whenever the user scrolls back and forth between pages. The *DataGrid* control raises the *PageIndexChanged* event each time a user clicks the pager bar to jump to a certain page. You must write a handler for *PageIndexChanged* and make it accomplish a couple of tasks.

```
<asp:datagrid runat="server" id="grid"
    ⋮
    AllowPaging="true"
    PageSize = "4"
    OnPageIndexChanged="PageIndexChanged">

    <PagerStyle PageButtonCount="3" Mode="NumericPages" />
    ⋮
</asp:datagrid>
```

The *PageIndexChanged* event handler has the following prototype:

```
void PageIndexChanged(Object sender, DataGridPageChangedEventArgs e)
{
    grid.CurrentPageIndex = e.NewPageIndex;
    // TO DO: Refresh the data source and re-bind
}
```

In the body of the handler, you must set *CurrentPageIndex* to the page the user clicked. You get the index of the newly requested page by using the *NewPageIndex* property of the event. If needed, you can also retrieve a current instance of the button that caused the event by using the *CommandSource* property of the *DataGridPageChangedEventArgs* class.

> **Caution** The *DataGrid* control does not automatically set the new page index. It only notifies the event handler of the index requested by the page. You are responsible for assigning the value to the *CurrentPageIndex* property. If you omit this step, you will not experience a run-time error but the *DataGrid* control won't page.

In addition to updating the page index, the *PageIndexChanged* event handler must refresh the *DataGrid* control. The way in which you accomplish this

depends completely on your application and how you obtain the data source. Typically, you must reassign the *DataGrid* control's *DataSource* property and invoke the *DataBind* method to initiate HTML rendering.

```
grid.DataSource = CreateDataSource();
grid.DataBind();
```

Figure 2-7 shows a pageable *DataGrid* control in action.

employeeid	firstname	lastname
1	Nancy	Davolio
2	Andrew	Fuller
3	Janet	Leverling
1 2 3		

employeeid	firstname	lastname
4	Margaret	Peacock
5	Steven	Buchanan
6	Michael	Suyama
1 2 3		

employeeid	firstname	lastname
7	Robert	King
8	Laura	Callahan
9	Anne	Dodsworth
1 2 3		

Figure 2-7 A pageable *DataGrid* control

> **Tip** Before being assigned the new page index in the body of the *PageIndexChanged* event handler, the *CurrentPageIndex* property contains the value of the previously displayed page. You can use this information for implementing special tracking features in your *DataGrid* control.

You use the *CurrentPageIndex* property to determine the currently displayed page in the *DataGrid* control when paging is enabled. You can also use this property to programmatically control which page is displayed. *CurrentPageIndex*, in fact, is a read-write property, so if you need to display a given page in the *DataGrid* control without any user intervention, set *CurrentPageIndex* to the page index you need and then rebind the data to the control. Bear in mind that the *CurrentPageIndex* property is zero-based.

Customizing the Pager Bar

You can shape the appearance of the pager bar by using the *PagerStyle* property, which evaluates to a *DataGridPagerStyle* object. *DataGridPagerStyle* exposes properties such as *BackColor*, *Font*, and *BorderStyle*, as well as a property for setting CSS styles. Many of these attributes apply to the pager bar as a whole, so what do you do when you want to change the style of the pager bar's individual buttons? You need to hook into the *ItemCreated* event.

```
<asp:datagrid runat="server" id="grid"
    ⋮
    AllowPaging="true"
    OnItemCreated="ItemCreated"
    OnPageIndexChanged="PageIndexChanged">
```

```
    <PagerStyle PageButtonCount="3" Mode="NumericPages" />
   ⋮
</asp:datagrid>
```

The *ItemCreated* event occurs on the server whenever a constituent element of the *DataGrid* control is created. By hooking into this event, you can modify on the fly both the HTML structure and the style of the element. Since this event fires for all the elements, the first task you must perform in the handler is to determine whether the item being created is exactly what you were waiting for. You check the type of the element by using the *ItemType* property. A reference to the item being created is returned by *e.Item*. The *ListItemType* enumeration lists all the feasible elements supported by the *DataGrid* control.

```
void ItemCreated(Object sender, DataGridItemEventArgs e)
{
    // Get the type of item being created
    ListItemType elemType = e.Item.ItemType;

    // Make sure it is the pager bar
    if (elemType == ListItemType.Pager)
    {
        // The pager bar as a whole has the following layout:
        //
        // <TR><TD colspan=X> ... links ... </TD></TR>
        //
        // Item points to <TR>. The code below moves to <TD>.
        TableCell pager = (TableCell) e.Item.Controls[0];

        // More code goes here
    }
}
```

In the overall layout of the *DataGrid* control, the pager bar is rendered by using a *TableRow* object. Within this *TableRow* object, you find *exactly one TableCell* control that spans all columns. The link buttons the user sees and perceives as the full pager bar are the contents of the *TableCell* control. Let's review the HTML code that is generated for the first pager bar of Figure 2-8. You can check this code at any time by snooping into the HTML source code of a page that uses a *DataGrid* control.

```
<tr style="background-color:PaleGreen;font-weight:bold;">
    <td colspan="2">
        <span>1</span> 
        <a href="javascript:__doPostBack(...)">2</a> 
        <a href="javascript:__doPostBack(...)">3</a>
     </td>
</tr>
```

The settings you enter by using *PagerStyle* apply only to the *TableRow* control—the ASP.NET counterpart of a *<tr>* tag. If you need to modify controls placed inside *TableRow*, you can use only *ItemCreated*.

```
TableCell pager = (TableCell) e.Item.Controls[0];
```

The preceding expression retrieves the *TableCell* control that represents the *<td>* tag, inside which are the pager buttons. To access the pager buttons, you need to loop through the *Controls* collection of the *TableCell* control. Note that anything generated through ASP.NET is a control, including literal content. The blanks that separate the button captions are controls as well. To skip over them, you can use, for example, a *for* statement with a step value of 2.

```
for (int i=0; i<pager.Controls.Count; i+=2)
{
    // i.th control is pager.Controls[i]
}
```

Now let's see how to associate a CSS stylesheet with each pager bar button and how to modify the text displayed for each element. To make the pager bar more readable, let's enclose the previous and next page numbers in square brackets and prefix the text *Page* to the label of the current page. Figure 2-8 shows the output. The full source code for the CustomPager.aspx application is available on the companion CD.

employeeid	firstname	lastname
1	Nancy	Davolio
2	Andrew	Fuller
3	Janet	Leverling
Page 1 [2] [3]		

employeeid	firstname	lastname
4	Margaret	Peacock
5	Steven	Buchanan
6	Michael	Suyama
[1] Page 2 [3]		

employeeid	firstname	lastname
7	Robert	King
8	Laura	Callahan
9	Anne	Dodsworth
[1] [2] Page 3		

Figure 2-8 A customized pager bar enhances accessibility.

The content of the pager bar cell is heterogeneous. It contains *n* elements of which *n-1* are clickable link buttons and one is a static label. Of course, this label refers to the current page. You need to distinguish among the control types to apply different settings to each. To check whether a given object is of a particular type, you can use the *is* operator and then cast the generic object as the needed type.

```
for (int i = 0; i < pager.Controls.Count; i += 2)
{
    Object o = pager.Controls[i];
    if (o is LinkButton)
    {
        LinkButton h = (LinkButton) o;
        h.Text = "[ " + h.Text + " ]";
    }
```

```
    else    // Can only be a Label
    {
        Label l = (Label) o;
        l.Text = "Page " + l.Text;
    }
}
```

> **Note** The C# *is* type operator checks whether an object expression is of a particular type. In Microsoft Visual Basic .NET, the *Is* operator has a different goal. You use it to determine whether two object variables refer to the same instance of an object. In Visual Basic .NET, to obtain the same behavior as the C# *is* operator, you have to resort to the *TypeOf...Is* operator.

Once you hold a valid instance of the link button or label, you can set any other property each provides. For example, you can assign each a new CSS style by using the *CssClass* property.

> **Tip** When defining a new CSS style for a control, don't be afraid to use DHTML behaviors. Although they are supported only by Internet Explorer version 5 and later versions, all other browsers ignore them, so you'll experience no ill effects.

Replacing the Standard Pager Bar with Your Own

If for any reason you don't want your pages to display the standard pager bar, you can replace it with your own. You hide the pager bar by turning off its *Visible* attribute.

```
<PagerStyle Visible="false" />
```

The *DataGrid* control does not provide you with a built-in mechanism for plugging in a custom pager bar. But changing the structure of the page slightly allows you to add your own. In this example, the *DataGrid* control and your personal pager bar are two rows of the same table.

```
<table class="PersonalPager">
    <tr><td>
        <asp:datagrid ...> ... </asp:datagrid>
    </td></tr>
```

(continued)

```
    <tr><td>
        personal pager here
    </td></tr>
</table>
```

Depending on the look you desire for the grid, some of the settings you applied at the *DataGrid* control level have to be moved to the outer table level, for example, background color and shadow settings. The outside table also needs to have a border.

The following code shows how to transform the pager bar into a numeric text box with a button enabling the user to jump to a specified page. The user enters the page number in the text box and then clicks the Go button to jump to that page. The full source code for the PersonalPager.aspx application is available on the companion CD.

```
<asp:panel runat="server">
    <asp:label runat="server" cssclass="stdtext" text="Page" />
    <asp:textbox runat="server" id="GotoPage" cssclass="stdtextbox"
        width="30px" text="1" />
    <asp:button runat="server" cssclass="FlatButton"
        Text="Go" OnClick="OnGotoPage" />
</asp:panel>
```

> **Note** If you wanted to, you could display both the ordinary pager bar and a personal pager bar. Really, the personal pager bar is any custom toolbar appended to the bottom of the *DataGrid* control.

Figure 2-9 shows what the personal pager bar looks like on a Web page. Notice in the figure which graphical element has so far been neglected in our discussion.

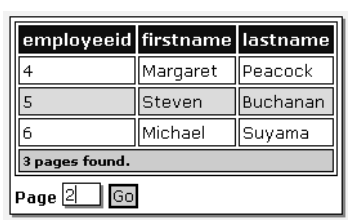


Figure 2-9 The footer and personal pager bar in a *DataGrid* control.

Although the *DataGrid* control and the personal pager bar together form a unique element, each are distinct HTML elements and can be shown individ-

ually. To avoid showing the pager bar when the *DataGrid* control is not shown, you might want to consider displaying both elements at the outer table level. For example, when the *DataGrid* control is shown only as the result of a postback event, the personal pager bar might be not be hidden and displayed. You can make sure the elements are displayed properly by handling the *Visible* attribute on the outer table.

```
<table runat="server" id="theTable" visible="false" class="PersonalPager">
```

Note, though, that *Visible* is an ASP.NET attribute, so a *runat="server"* attribute is also needed.

Using the Footer

In a *DataGrid* control, the footer is hidden by default. To turn it on, you must set the *ShowFooter* property to *true*. Of course, you can also do that declaratively by using the *ShowFooter* attribute.

```
<asp:datagrid runat="server"
    ⋮
    ShowFooter="True">
```

The footer style is subject to full modification via the *FooterStyle* property, so you can change its font, colors, and borders. One aspect you cannot change declaratively—and indeed often need to change—is the structure of the footer. The footer is rendered via an empty row, so it maintains the same structure as other rows. If this footer structure suits you, use the column's *FooterText* property to assign it HTML contents. If you want the footer to have a different number of columns or columns of different widths, you must intervene at the time the footer is created, namely in the *ItemCreated* event handler.

```
void ItemCreated(Object sender, DataGridItemEventArgs e)
{
    ListItemType elemType = e.Item.ItemType;

    // Handle other types of items here

    if (elemType == ListItemType.Footer)
    {
        // Footer is just an empty row. Remove all
        // the cells but one and span over all.
        TableCellCollection tcc = e.Item.Cells;
        int nTotalCols = tcc.Count;

        // Use nTotalCols as tcc.Count changes
        // dynamically as you remove cells
        for (int i=0; i<nTotalCols-1; i++)
            e.Item.Cells.RemoveAt(1);
```

(continued)

```
            // Only 1 cell at this time ...
            TableCell c = e.Item.Cells[0];
            c.ColumnSpan = nTotalCols;
            c.Text = grid.PageCount.ToString() + " pages found.";
        }
    }
```

In most cases, you need a footer made up of a single cell that works as a summary row. To obtain this, you must remove at run time all cells but one, as shown in the preceding code example. Next, you set the *ColumnSpan* property of the remaining cell to the total number of the columns and display the summary text.

Custom Pagination

Every time the *DataGrid* control moves to a different page, all contents in the associated data source have to be reloaded. No matter how you retrieve the data—whether from a database, an XML file, or the *Session* object—as long as you use default pagination, you must have all that data stored in memory to successfully refresh the grid. When the data source is very large, storing it can consume a lot of resources on the Web server.

Custom pagination enables you to load only the segment of data needed to display a single page. To enable custom pagination, you set both the *AllowPaging* and *AllowCustomPaging* properties to *true*. Next, you provide code that handles the *PageIndexChanged* event.

So how is coding actually different when you enable custom pagination? The only significant difference between default paging and custom paging is the way in which the *DataGrid* control processes what's stored in the *Items* property. In the case of default paging, the *DataGrid* control assumes to have in memory all possible items and, based on the page size and current index, extracts a subset of items that will fit in the selected page. With custom paging, the *DataGrid* control always displays all contents of the *Items* property, so you are responsible for loading only the records that fit in the selected page. This approach clearly requires less Web server memory and looks inherently more scalable—at least for large blocks of data. Let's see custom pagination in action.

Setting the Virtual Item Count

When you use custom paging, you don't need to handle the pager bar yourself. As long as you inform the *DataGrid* control about the virtual number of items you plan to display through all pages, you have pager bar functionality served for free, as usual. You let the *DataGrid* control know about the virtual item count by setting the *VirtualItemCount* property. You should set the property before the *DataGrid* control is displayed. A good place to set it is in *Page_Load* when *IsPostBack* is *False*.

```
if (!Page.IsPostBack)
    SetVirtualItemCount();
```

If your *DataGrid* will show the entire contents of a table—say, the *Products* table of the SQL Server 2000 Northwind database—the virtual item count is simply the total number of records found in the table.

```
public void SetVirtualItemCount()
{
    // Set up the connection
    String strConn = "DATABASE=Northwind;SERVER=localhost;UID=sa;PWD=;";
    SqlConnection conn = new SqlConnection(strConn);

    // Set up the command
    String strCmd = "SELECT COUNT(*) FROM products";
    SqlCommand cmd = new SqlCommand(strCmd, conn);

    // Execute the command
    conn.Open();
    int nItemCount = (int) cmd.ExecuteScalar();
    conn.Close();

    // Set the grid's virtual item count
    grid.VirtualItemCount = nItemCount;
    return;
}
```

> **Note** *ExecuteScalar* is a method of the ADO.NET command classes, including in particular, *SqlCommand*. It executes a SQL command or a stored procedure, and it returns only the scalar value on column 1, row 1 in the result set. *ExecuteScalar* returns the value boxed in an *Object* class, so casting is necessary to obtain, for example, a true integer value.

Obtaining the Page Content

With custom paging, the amount of memory allocated is limited to the number of items that fit in a single grid page. The rub is, how can you get *all* and *only* the records that fit in a particular page? Once again, the way you handle this issue is strictly application-specific. The SQL language provides no support for pagination. The only way to get records from a SQL-based data management system is by using the *SELECT* statement. The only way to restrict the set of rows returned is by using the *WHERE* clause. Therefore, you must figure out a way to retrieve the contents of a particular page based on some sort of field condition. Because no official SQL-based solution exists, any assumption you can make in your own project given the structure of your own databases is

valid. In the remainder of this section, I'll examine a SQL case scenario. Consider the following SQL statement:

```
SELECT TOP n Fields FROM Table WHERE Key > page_related_info
```

> **Caution** Very few databases support the *TOP* clause in the *SELECT* statement. The *TOP* clause was introduced with SQL Server version 7 and has no counterpart in SQL Server version 6.5 and Oracle. Informix SQL does provide a similar behavior through the *FIRST* clause. Although less flexible, the *SET ROWCOUNT* statement can be used as a rougher counterpart for *TOP* when this clause is not supported.

With SQL Server 7 and later versions, *SELECT* with a *TOP* clause returns from the specified table the first *n* rows that match the Boolean comparison in the *WHERE* clause. It goes without saying that, for our purposes, *n* is the page size of the *DataGrid* control.

The role of *page_related_info* is less defined. Its function has to be such that, based on the page number, you can directly access the first record entered in that page. If the table has a unique key column set with a regular series of numbers, you can calculate the first value for the key field on a given page with a simple arithmetic expression, based on the first record to display.

For example, the Northwind *Products* table's *ProductID* field contains incrementing numbers: 1, 2, 3, and so on. You can then use this field as the key column for your query. The statement that loads the records for page 4, then, looks like this:

```
SELECT TOP n * FROM Products WHERE productid > (n-1)*4
```

Page 4 contains records from 31 through 40. Even if your table does not have a column like this, you should consider adding a new one *ad hoc*. Doing so is not manna from heaven, though, as you must regenerate the column every time a row is deleted. Putting this all together, you can retrieve the rows needed for display as follows:

```
SqlDataReader dr;
SqlDataReader CreateDataSource(int nPageIndex)
{
    // Page index is assumed to be 0-based
    int nPageSize = grid.PageSize;
    int nBaseProductID = nPageSize * nPageIndex;

    // Set up the connection
    String strConn = "DATABASE=Northwind;SERVER=localhost;UID=sa;PWD=;";
    SqlConnection conn = new SqlConnection(strConn);
```

```
    // Set up the command
    String strCmd = "SELECT TOP " + nPageSize + " " +
            "productid, productname, quantityperunit, unitsinstock " +
            "FROM products " +
            "WHERE productid >" + nBaseProductID;
    SqlCommand cmd = new SqlCommand(strCmd, conn);

    // Execute the command
    conn.Open();
    dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);
    return dr;
}
```

When you retrieve data frequently, the *SqlDataReader* class is a better option than *SqlDataAdapter* because the former results in a structure—the data reader—that is faster to load than the data set. While the data reader is in use, the associated connection is busy. The argument passed to *ExecuteReader* specifies that the connection must be closed when the *SqlDataReader* object is closed. *CreateDataSource* passes the *SqlDataReader* object to the *DataGrid* control, which stores the object in its *DataSource* property. At this point in the code, the data reader and the connection are still open. Setting *DataSource* does not automatically transfer the content of the data source to the *DataGrid* control's memory (the *Items* property). Until the content is bound, you cannot close the data reader and release the database connection. The following code performs this binding and closes the data reader:

```
grid.DataSource = CreateDataSource(0);
grid.DataBind();
dr.Close();
```

Figure 2-10 shows a snapshot from the sample application illustrating custom paging. The full source code for the CustomPaging.aspx application is available on the companion CD.

productid	productname	quantityperunit	unitsinstock
51	Manjimup Dried Apples	50 - 300 g pkgs.	20
52	Filo Mix	16 - 2 kg boxes	38
53	Perth Pasties	48 pieces	0
54	Tourtière	16 pies	21
55	Pâté chinois	24 boxes x 2 pies	115
56	Gnocchi di nonna Alice	24 - 250 g pkgs.	21
57	Ravioli Angelo	24 - 250 g pkgs.	36
58	Escargots de Bourgogne	24 pieces	62
59	Raclette Courdavault	5 kg pkg.	79
60	Camembert Pierrot	15 - 300 g rounds	19

[1] [2] [3] [4] [5] **Page 6** [7] [8]

Figure 2-10 Custom pagination was used to create the scrolling mechanism in this database.

Using Generic SQL

Let's explore two more general approaches to implementing custom pagination that are not dependent on any application-specific assumption or precondition. One is based on T-SQL cursors and the *FETCH* statement. The other is based on a generic and parametric SQL command made of three nested *SELECT* statements.

The first approach, which is based on server-side cursors, has two major drawbacks. First and foremost, server-side cursors cause a loss in performance, which is why they are a last resort solution. The second drawback is that each row you fetch via cursors is stored in a separate result set. To bind these rows to the *DataGrid* control, you need to merge the result sets into a single *DataTable* object.

A much better (but less generic) approach can be inferred from the following SQL statement, which retrieves the rows displayed on page 3, where pages have 2 records each and are sorted by the *lastname* field:

```
SELECT TOP 2 * FROM
    (SELECT TOP 2 employeeid, lastname FROM Employees
        WHERE lastname IN
        (SELECT TOP 6 lastname FROM Employees ORDER BY lastname)
        ORDER BY lastname DESC) AS tmp
    ORDER BY tmp.lastname
```

As you can see, the statement comprises three nested *SELECT* commands. Despite the apparent complexity, the statement takes significant advantage of the query optimization engine that comes with SQL Server 2000. The key operations that the query performs can be more easily described in the temporary tables shown below:

```
SELECT TOP 6 employeeid, lastname INTO t1 FROM Employees ORDER BY lastname
SELECT TOP 2 employeeid, lastname INTO t2 FROM t1 ORDER BY lastname DESC
SELECT * FROM t2 ORDER BY lastname
```

The first statement, which corresponds to the innermost nested *SELECT* command, copies into the temporary table *t1* a subset of rows that, once sorted, will contain the rows on the page to be displayed. For example, if 2 is the page size, and you want to access page 3, then the last 2 rows of the first 2*3 rows are what you need. These 2 rows are moved to another table *t2* in the reverse order and then returned.

Using temporary tables is not really effective and, more important, can create additional conflicts in a Web scenario where many concurrent requests must be processed. Nested queries provide the same outcome with much better performance, though at the price of a less readable SQL syntax.

To create a more generic SQL statement, you can parameterize three elements: the number of rows to return, the number of rows to fetch, and the field to sort by. The number of rows to return is the grid's page size (or less if the

total number of rows doesn't exactly match a multiple of the page size). The number of rows to fetch is calculated by multiplying the page size by the 1-based index of the page to show. The *TOP* and the *ORDER BY* clauses of the T-SQL language do not accept parameters, so a *DataGrid* that wants custom pagination should prepare its query command through string formatting rather than regular SQL parameters:

```
// Prepare the SQL command
StringBuilder sb = new StringBuilder("");
sb.Append("SELECT TOP {0} * FROM ");
sb.Append("(SELECT TOP {0} employeeid, firstname, lastname ");
sb.Append("FROM Employees WHERE {1} IN ");
sb.Append("(SELECT TOP {2} {1} FROM Employees ORDER BY {1}) ");
sb.Append("ORDER BY {1} DESC) AS tmp ");
sb.Append("ORDER BY {1}");
String strCmd = sb.ToString();
```

The *{n}* placeholders are replaced with actual data through the *String.Format* method.

```
int nRowsToDisplay = grid.PageSize;
int nMod = grid.VirtualItemCount % grid.PageSize;
if (nPageIndex == grid.PageCount && nMod > 0)
    nRowsToDisplay = nMod;
strCmd = String.Format(strCmd,
    nRowsToDisplay, strCurrentSortExpr, grid.PageSize * nPageIndex);
```

If the page to show is the last one, then the number of rows to display is the page size or the modulus of the total number of items and the page size. Finally, the sort expression is the current sort expression of the *DataGrid*, or the primary key if you don't support sorting. The full source code for the UsingGenericSql.aspx application is available on the companion CD.

Sorting Columns

The *DataGrid* control does not actually sort rows, but it provides good support for sorting as long as the sorting capabilities of the underlying data source are adequate. The data source is always responsible for returning a sorted set of records based on the sort expression selected by the user through the *DataGrid* control's user interface. The built-in sorting mechanism is triggered by setting the *AllowSorting* property to *true*.

```
<asp:datagrid id="grid" runat="server"
    ⋮
    AllowSorting="True"
    OnSortCommand="SortCommand">
```

When *AllowSorting* is set to *true*, the *DataGrid* control offers a link button for rendering the column caption. When users click one of these sortable columns, the *SortCommand* event is fired. In the body of the *SortCommand* event handler, you have to refresh the contents of the *DataGrid* control so that it reflects the requested sort.

Setting Up Sorting

You make a column sortable by setting the *SortExpression* property. You can set it declaratively in the ASP.NET page layout or you can set it programmatically. *SortExpression* evaluates to any expression that the data source understands as a sort command. Typically, *SortExpression* is the name of a field in the data source. If needed, though, you can sort by multiple fields by separating them with a comma. You can specify *DESC* (or *DESCENDING*) and *ASC* (or *ASCENDING*) to indicate a direction for sorting. By default, sorting is ascending.

```
<asp:BoundColumn runat="server" DataField="lastname"
    HeaderText="Last Name" SortExpression="lastname, firstname" />
<asp:BoundColumn runat="server" DataField="country"
    HeaderText="Last Name"
    SortExpression="country DESC, lastname, firstname" />
```

After you set up the columns to sort by and the relative expressions, you'd better take care of the *SortCommand* event handler. The typical logic for this handler is to sort the list of items and then rebind the data to the *DataGrid* control.

```
void SortCommand(Object sender, DataGridSortCommandEventArgs e)
{
   ViewState["SortExpression"] = e.SortExpression;
   UpdateDataView();
}
```

The *SortCommand* event handler knows about the sort expression through the *SortExpression* property, which is provided by the *DataGridSortCommandEventArgs* class. In the preceding code, the sort information is persisted because it is stored in a slot in the page's *ViewState* collection. The current sort expression, in fact, is yet another piece of information that the *DataGrid* control does not automatically persist in the control's view state. If you need it to be persistent, you must do the work yourself. Figure 2-11 shows sorting in action. The full source code for the SortColumns.aspx application is available on the companion CD.

ID	First Name	Last Name	Position	Country
9	Anne	Dodsworth	Sales Representative	UK
2	Andrew	Fuller	Vice President, Sales	USA
7	Robert	King	Sales Representative	UK
[1] Page 2 [3]				

Figure 2-11 A *DataGrid* control with sortable columns.

The *DataGrid* control in Figure 2-11 shows rows sorted by the *lastname* field, but you can't figure that out easily just by looking at the grid, even with such simple data. You would probably want to modify the *DataGrid* control's user interface to indicate the column used for sorting.

Auto-Reverse Sorting

The typical way to identify the sorted column is by means of a little glyph in the header. This glyph also reflects the direction of the sort (ascending or descending). To add such a glyph to the column header, you again need to use our old acquaintance *ItemCreated* and look up the *Header* item type. A marker that quickly identifies the sort column makes especially good sense when you implement an auto-reverse sort. In an auto-reverse sort, the column reverses the direction of the data when the user clicks the marker twice in sequence. In this situation, persisting the current sort expression across multiple page requests is vital. To implement the auto-reverse sorting feature, the *SortCommand* handler changes slightly, as follows:

```
void SortCommand(Object sender, DataGridSortCommandEventArgs e)
{
    // Caches the current information
    String strSortBy = (String) ViewState["SortExpression"];
    String strSortAscending = (String) ViewState["SortAscending"];

    // Sets the new sort expression
    ViewState["SortExpression"] = e.SortExpression;

    // If you click on the sorted column, the order reverses
    if (e.SortExpression == strSortBy)
        ViewState["SortAscending"] = (strSortAscending=="yes"
            ? "no" :"yes");
    else
        // Defaults to ascending order
        ViewState["SortAscending"] = "yes";

    UpdateDataView();
}
```

The *DataGrid* control now stores two pieces of information in *ViewState*: the sort expression and a yes/no flag that indicates whether or not the direction is ascending. When the sort expression of the column just clicked matches the current sort expression, the same column has been clicked twice consecutively, and the order reverses. Note that this procedure has not performed the actual sorting. The *ViewState* collection has just been holding up-to-date information about how to sort data.

Sorting would not be possible without the help of the ADO.NET *DataView* class. The *DataView* class is a view class built on top of a *DataTable class*. It enables filtering and sorting. You can obtain a default, unfiltered, and unsorted view object from any *DataTable* object simply by accessing the *DefaultView* property. Once you have a *DataView* object, you just set the *Sort* property, assign it to the *DataGrid* control, and rebind.

```
void UpdateDataView()
{
    DataSet ds = (DataSet) Session["MyDataSet"];
    DataView dv = ds.Tables["MyList"].DefaultView;

    // Apply sort information to the view
    dv.Sort = (String) ViewState["SortExpression"];
    if (ViewState["SortAscending"] == "no")
        dv.Sort += " DESC";

    // Rebind data
    grid.DataSource = dv;
    grid.DataBind();
}
```

The *Sort* property is set with the sort expression taken from *ViewState*. Then the sort expression is appended with *DESC* if the required order is descending. By default, the *DataView* class sorts in ascending order.

> **Note** If you don't use the *DataView* object to sort, both the sort expression and the use of *DESC* and *ASC* can be radically different, depending on the query syntax of the database you use for sorting. Although I don't wholeheartedly recommend requerying the database directly for performance reasons, using it to sort data is still a valid option.

Notice that in our code example, I used the *Session* object to persist on the Web server the *DataGrid* control's data source. Later in the chapter, I'll review the various options available for data persistence.

Marking the Sort Column

To add a glyph near the column caption, you handle the *ItemCreated* event and look for an element type of *ListItemType.Header*. The *ItemCreated* event fires when the grid heading has been completely set up. To locate the column to mark, you have no other choice but to loop through the *Columns* collection.

```
if (elemType == ListItemType.Header)
{
    String strSortBy = (String) ViewState["SortExpression"];
    String strSortAscending = (String) ViewState["SortAscending"];
    String strOrder = (strSortAscending=="yes" ? " 5" : " 6");

    for (int i=0; i<grid.Columns.Count; i++)
    {
        // Draw the glyph to reflect sorting
        if (strSortBy == grid.Columns[i].SortExpression)
        {
            TableCell cell = e.Item.Cells[i];
            Label lblSorted = new Label();
            lblSorted.Font.Name = "webdings";
            lblSorted.Font.Size = FontUnit.XSmall;
            lblSorted.Text = strOrder;
            cell.Controls.Add(lblSorted);
        }
    }
}
```

The column to mark is the column whose *SortExpression* matches the current sort expression. When the column is found, you create a new label control, set its font to Webdings, select the font size and text (usually the fifth and the sixth characters), and finally add the label to the table cell that hosts the column heading. Figure 2-12 shows the results. The full source code for the AutoReverse.aspx application is available on the companion CD.

ID	First Name	Last Name ▲	Position	Country
9	Anne	Dodsworth	Sales Representative	UK
2	Andrew	Fuller	Vice President, Sales	USA
7	Robert	King	Sales Representative	UK
[1] Page 2 [3]				

ID	First Name	Last Name ▼	Position	Country
7	Robert	King	Sales Representative	UK
2	Andrew	Fuller	Vice President, Sales	USA
9	Anne	Dodsworth	Sales Representative	UK
[1] Page 2 [3]				

Figure 2-12 A *DataGrid* control with the auto-reverse sort feature.

Despite appearances, the code in AutoReverse.aspx has severe drawbacks, not the least of which is that it assumes you never use *ASC* in the sort expression to indicate ascending order. I will explain what else is wrong with the code in the next section.

Sorting Multiple Fields

Auto-reverse sorting, at least as we've implemented it so far, poses a serious issue when you use it with expressions that involve multiple columns. Suppose that in a database of products you have a column sorted by *unitprice, productname*. When the user reverses the direction, the user *expects* to sort by this:

```
unitprice DESC, productname DESC
```

The grid does sort the data correctly in ascending mode, but when the user reverses the order, the sort expression becomes rather anomalous and clearly wrong:

```
unitprice, productname DESC
```

The incriminating code is located in the *UpdateDataView* subroutine and is as follows:

```
if (ViewState["SortAscending"] == "no")
    dv.Sort += " DESC";
```

The code clearly assumes that the sort expression is composed of exactly one field, even without a trailing *ASC* sort order. Let's see how to fix it to fully support both auto-reverse and any valid sort expression with as many fields and sort orderings as you need.

Auto-Reverse Sorting for Multiple Columns

To work around the insidious bug I just mentioned, you need to step back from the details of sorting and look at the bigger picture. In particular, you need to keep separate the fields to sort by and their respective directions. A simple string comparison between the clicked columns and the current sort expression is insufficient to determine whether the order has to be reversed. Due to the sorting syntax supported by *DataView* objects and the majority of database systems, the sort expression must have direction information embedded. On the other hand, auto-reverse just means that if the fields involved in the sorting are the same, you change their order. This clearly implies a separation between fields and directions.

When the *SortCommand* event occurs, you process the sort information, taking into account auto-reverse sorting. The sort expression is composed from two comma-separated strings that are both stored in *ViewState*. The first string

contains the names of the columns involved in the sorting. In the simplest case, this string contains a single field name, as in the preceding example. The second string contains, for each column, the sort order direction. The *ASC* sort order is always used whether or not you specified it in the sort expression. You modify your *SortCommand* event handler as follows:

```
void SortCommand(Object s, DataGridSortCommandEventArgs e)
{
    // Processes the sort expression. Determines whether auto-reverse is
    // needed and stores in ViewState two comma-separated strings. One
    // contains the names of the involved columns and one contains - in the
    // same order - the respective sort order directions.
    ProcessSortExpression(e.SortExpression);

    // Refreshes the view
    UpdateDataView();
}
```

The *ProcessSortExpression* routine performs boilerplate code that stores in *ViewState* a comma-separated string with the names of the fields involved in the sort, and a comma-separated string with the order required for each field by the current state of the grid (auto-reverse). The former string uses a slot named *SortingFields*, and the latter string uses a slot named *SortingOrders*. (Of course, these names are totally arbitrary.) *ProcessSortExpression* also fills a third slot, named *ColumnSortExpression*, that contains the original sort expression associated with the clicked column, that is, *e.SortExpression*. This information will be useful later when you add a glyph to the column's header.

When the grid is going to be refreshed, you process the information stored in *ViewState* and end up with a tailor-made string that the *DataView* object can easily understand.

```
void UpdateDataView()
{
    DataSet ds = (DataSet) Session["MyDataSet"];
    DataView dv = ds.Tables["MyList"].DefaultView;

    // Apply sort information to the view
    dv.Sort = PrepareSortExpression();

    // Rebind data
    grid.DataSource = dv;
    grid.DataBind();
}
```

The *PrepareSortExpression* routine retrieves from *ViewState* the information about sorting fields and orders, and it merges that information in a string that the *DataView* object knows how to process. The following is a sample string:

```
unitprice asc, productname desc
```

This final sort expression is also explicitly stored in *ViewState* in light of another future enhancement that I'll discuss in a moment. When you add the glyph, you recognize the column that is used to sort the *DataGrid* control by comparing the column's sort expression to the string stored in *ViewState["ColumnSortExpression"]*. Next, you get the right WebDing character (indicating ascending or descending order) based on the order requested for the primary column in the sort expression. If a column has to be sorted by *unitprice, productname*, you might want to consider the order of only the first primary column to decide which glyph to display. Figure 2-13 shows that multicolumn, auto-reverse sorting now is working just fine.

ID	Product	Price ▲	Stock
67	Laughing Lumberjack Lager	$14.00	52
25	NuNuCa Nuß-Nougat-Creme	$14.00	76
34	Sasquatch Ale	$14.00	111
42	Singaporean Hokkien Fried Mee	$14.00	26
70	Outback Lager	$15.00	15
73	Röd Kaviar	$15.00	101
15	Genen Shouyu	$15.50	39
50	Valkoinen suklaa	$16.25	65
66	Louisiana Hot Spiced Okra	$17.00	4
16	Pavlova	$17.45	29

[1] [2] **Page 3** [4] [5] [6] [7] [8]

ID	Product	Price ▼	Stock
15	Genen Shouyu	$15.50	39
73	Röd Kaviar	$15.00	101
70	Outback Lager	$15.00	15
42	Singaporean Hokkien Fried Mee	$14.00	26
34	Sasquatch Ale	$14.00	111
25	NuNuCa Nuß-Nougat-Creme	$14.00	76
67	Laughing Lumberjack Lager	$14.00	52
58	Escargots de Bourgogne	$13.25	62
77	Original Frankfurter grüne Soße	$13.00	32
48	Chocolade	$12.75	15

[1] [2] [3] [4] [5] **Page 6** [7] [8]

Figure 2-13 A *DataGrid* control that supports multicolumn, auto-reverse sorting.

Showing Sorting Information

Showing a little bitmap in the column heading helps users understand how the column is sorted, but users of more sophisticated applications might find this contrivance insufficient. When you need to reveal more information, you can provide a tooltip that shows the column's sort expression. In this tooltip, you can also distinguish between the currently sorted columns and all the other sortable ones. Let's see how to accomplish this, bearing in mind that you can use the next trick to show any type of information, not just information about sorting.

You add a tooltip to an ASP.NET control by setting its *ToolTip* property. The tooltip cannot be added to the column as a whole; instead it must be

defined only for the link button used to render the caption. To get your hands on this control, once again you have to resort to the *ItemCreated* event handler.

```
if (elemType == ListItemType.Header)
{
    for (int i=0; i<grid.Columns.Count; i++)
    {
        // Grab the cell with the link button
        TableCell cell = e.Item.Cells[i];

        // Add a tooltip with the sort expression
        if (grid.Columns[i].SortExpression != "")
            cell.ToolTip = "Sort by: " + grid.Columns[i].SortExpression;

            // Draw the glyph to reflect sorting
            String strSortBy = (String)ViewState["ColumnSortExpression"];
            if (strSortBy == grid.Columns[i].SortExpression)
            {
                cell.ToolTip = "Sorted by: " +
                    ViewState["ColumnSortExpression"]ToString();

                // Code that draws the glyph ...
            }
    }
}
```

Figure 2-14 shows the final result. The full source code for the MfieldSorting.aspx application is available on the companion CD.

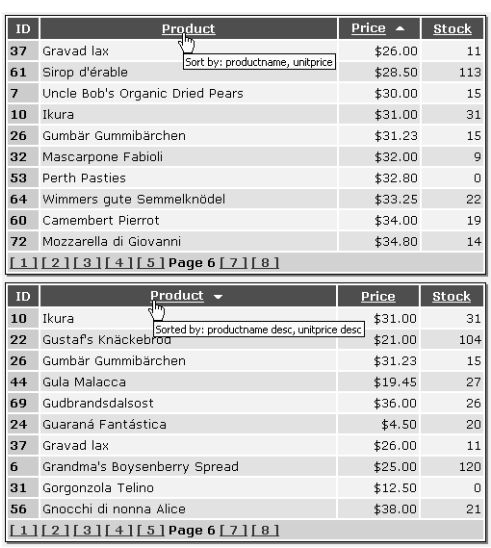


Figure 2-14 Tooltips let users know about the sort expression of each column.

When you use custom pagination, you are responsible for providing all the records that fit in every page. This task is even more complex if you have to provide sorting too.

DataGrid Controls and Data Persistence

ASP.NET applications are modeled after the Web Forms model, which is the typical Visual Basic form-based, client/server model of interaction delivered on the Web. The ASP.NET run time shields you from the structural differences between the two models. It takes care of serializing and deserializing the state of the form. Any server-side processing takes place in an environment that maintains the state of the client browser. When you use *DataGrid* controls, you need to retrieve and process the control's data source every time you execute postback code on the server. As mentioned earlier in the chapter, the *DataGrid* control does not cache its data source in the control's view state. This makes sense because the data source can be too large to be effectively transferred back and forth between the Web server and the browser. Bear in mind that all the information Web controls store in their *ViewState* properties makes the size of the HTML page larger. This information is then posted back from the browser to the server when a postback event occurs. Basically, you have two options for repopulating the *DataGrid* control's *DataSource* property:

- Cache the data source, as a whole or in part, on the Web server and read it back
- Reload all the records from the physical data storage (typically, a database)

When you cache the data source, data is retrieved from storage only once, stored in a cache, and subsequent postback events read from that cache. You can use in-memory global objects such as *Session* and *Cache*, or alternatively you can use XML files stored on the Web server or another accessible share.

If you plan to reload the records each time a postback event occurs, consider that using a *DataReader* class is more efficient than using a data adapter. Don't forget to close both the reader and the connection as soon as possible, as I previously explained.

Scalability? What Was That?

The way in which you decide to retrieve the grid's data source might seriously affect the overall scalability of the application. However, years of real-world experience should have taught you scalability is affected by many factors. Scalability is precious like a diamond and, like a diamond, can have many facets that contribute to its value. Scalability can be described as the system's ability to maintain, or improve, its responsiveness as the number of clients grows. The theory of queuing states that a queue forms itself when the frequency of the requests tend to overtake the system's response time. For the sake of the application, you cannot take measures to reduce the user requests, but you can try to lessen the response time.

You normally adjust the scalability level of Web applications by mixing together, in application-specific doses, heterogeneous and even contradictory measures such as the following:

- Limiting the number of calls to the database
- Delegating as many tasks as possible to the database
- Limiting the occupation of the Web server's memory
- Using relatively simple and stateless components

Writing fast and optimized code would also certainly help a lot! For a good result, each ingredient is extremely important and, with the right doses, even otherwise lethal ingredients are acceptable. Limiting the number of calls to the database implies that you are *not* delegating data processing to it and are therefore placing a load on the Web server. Limiting the server memory occupation implies that you don't cache data and, consequently, that you have to call the database whenever data is required. And the list could go on and on. Scalability is a sort of philosopher's stone, and programmers, like medieval alchemists, can only try remedies again and again, learning from their errors and fine-tuning their skills.

Let's review the most common options you have for persisting the *DataGrid* control's content on the server.

Using the *Session* Object

In ASP and ASP.NET, the *Session* object is a global repository for data and objects that belong to the session. The visibility of the data is limited to the pages invoked within the session. Using the *Session* object is critical any way you look at it. It guarantees quick and prompt access to data and returns ready-to-use objects, but all the session data is duplicated per each active session and

connected user. In general, you should be extremely careful when it comes to using the *Session* object in production code. But my advice about being careful does not mean that Microsoft would have been better off dropping the *Session* object. Using *Session* is still the fastest way to access session-specific data, but try to keep the amount of data stored in *Session* under strict control.

The ASP.NET *Session* object has two major advantages over its ASP counterpart. First, any .NET object can now be safely stored in a *Session* slot. This overcomes the thread-affinity problem you might have experienced with ASP and Visual Basic COM components. Second, the *Session* object is the programming interface for a module—the Session Manager—that can work in-process and out-of-process, and can even rely on SQL Server for data storage. This is probably the best reason to opt for *Session*: it now works well with Web farm architectures.

Using the *Cache* Object

The majority of ASP.NET applications will take advantage of the *Cache* object for all of their caching needs. The *Cache* object is new to ASP.NET and provides unique and powerful features. It is a global, thread-safe object that does not store information on a per-session basis. In addition, the *Cache* object is designed to ensure it does not tax the server's memory. If low memory does become an issue, the *Cache* object will automatically purge its least recently used items based on a priority defined by the developer. Like the familiar *Application* object, the *Cache* object does not share its state across the machines of a Web farm. In terms of the programming interface, using the *Cache* object is not at all different from using *Session* or *Application* objects.

What really differentiates the *Cache* object from *Application* is its ability to automatically remove least-used items when the memory is low. To help the built-in scavenging routines of the *Cache* object, you can assign some of your cache items with a priority and even a decay factor that lowers the priority of the items that have limited use. When working with the *Cache* object, you should never assume that an item is there when you need it. Always be ready to handle exceptions caused by null or invalid values. If your application needs to be notified of an item's removal, then register for the *OnRemove* event. You can do this by creating an instance of the *CacheItemRemovedCallback* delegate and passing it to the *Cache* object's *Insert* or *Add* method.

In addition, some of the items stored in the *Cache* object can be bound to the timestamp of one or more files or other cached items. When any of these linked resources change, the cached item becomes obsolete and is removed from the cache. By using a *try/catch* block you can catch the invalid item and reload it from persistent storage.

Aside from Web farms, in resource-constrained scenarios you might want to consider alternatives to *Cache*. Even when you have large data sets to store on a per-session basis, storing and reloading them from memory will be much faster than any other approach. With many users connected at the same time, each storing large blocks of data, you might want to consider helping the *Cache* object do its job better. An application-specific, layered caching system built around the *Cache* object is an option to evaluate. In this case, hot and sensitive data will go into the *Cache* and be efficiently managed by the ASP.NET run time. The rest of the data could be cached in a slower, but memory-free, storage such as session-specific XML files.

Using XML Files

ADO.NET classes, and the *DataSet* class in particular, are tightly integrated with XML. This means that saving the content of *DataSet* to a disk-based XML document is a snap. Also, rebuilding a living instance of a *DataSet* object from a persistent XML file is not particularly hard. If you don't want to re-read the *DataGrid* control's data out of a database every time it's needed and if you don't want to load the data only once and leave it stored to *Session* or *Cache*, persisting the data to XML files is an interesting option to consider. The more memory the Web server has, the more quickly it can serve new requests.

Earlier in the chapter, I showed a page that based persistence of the *DataGrid* control's content on the *Session* object. Let's rewrite the code to use XML files instead.

```
<script runat="server">
void Page_Load(Object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        DataFromSourceToMemory("MyDataSet");
        UpdateDataView();
    }
}


void DataFromSourceToMemory(String strDataSessionName)
{
    // Gets rows from the data source
    DataSet oDS = PhysicalDataRead();

    // Stores it in the session cache
    SerializeDataSource(oDS);
}
```

When the page is first loaded, the control calls *DataFromSourceToMemory* which, in turn, performs a physical read from the storage medium and then, instead of storing information to *Session*, calls a new routine named *SerializeDataSource*.

```
void SerializeDataSource(DataSet ds)
{
    String strFile = Server.MapPath(Session.SessionID + ".xml");
    XmlTextWriter xtw = new XmlTextWriter(strFile, null);
    ds.WriteXml(xtw);
    xtw.Close();
}
```

The code creates a new XML file whose name is based on the session ID. The file is populated by using the *DataSet* object's *WriteXml* method. Reading information back is easy, too. The task is accomplished by the *DeserializeDataSource* function.

```
private DataSet DeserializeDataSource()
{
    String strFile = Server.MapPath(Session.SessionID + ".xml");
    XmlTextReader xtr = new XmlTextReader(strFile);
    DataSet ds = new DataSet();
    ds.ReadXml(xtr);
    xtr.Close();
    return ds;
}
```

Instead of restoring *DataSet* from *Session*, your code would use *DeserializeDataSource*.

Using Data Readers and Adapters

Another, more radical choice for data persistence is re-reading the data from the database whenever needed. This choice is good when you have to deal with a very large data set and plan to implement a custom pagination service. Now might be a good time to refresh your memory about the differences between ADO.NET data adapters and data readers. You should use adapters when you want to work on your data in a disconnected manner. You should use readers if you don't plan to persist the retrieved data.

A data adapter populates a *DataSet* structure, which provides the main tools you need to work in the absence of a database. It provides for filters, sorting, indexing, searching, cloning, and even in-memory relations. Given its arsenal of programming tools, the *DataSet* class is not optimized for a simple read of a few records. The *DataSet* class can be persisted, either on disk or in memory (it does not matter), and survive across multiple page requests.

But if your goal is to read only the records that fill up one grid page, you are much better off using any of the data reader classes. Not only does ADO.NET provide a data reader for each supported .NET–managed data provider, but every .NET data provider is expected to expose a data adapter and a data reader class. So you use *SqlDataReader* when your target database is SQL Server version 7 and later versions. You use *OleDbDataReader* when your target is an OLE DB provider. You use *OdbcDataReader* when you are working against the ODBC .NET–managed data provider. The data reader works like a firehose—it provides an open channel through which read-only records flow, and the client reads them, one after the next, in a forward direction.

The Paradox of Pagination

Let's review what happens when you decide to retrieve records to populate a *DataGrid* control at every postback event using a data adapter object. You typically use a *CreateDataSource* function that returns a *DataSet* object, and you assign the result to the grid and rebind. If you don't cache *DataSet* in any way, you end up loading *all* the records available in the data source for *each* page scrolled. Get the point? Now do you see the paradox? You load, say, 100 records to display only the 10 or fewer that fit in the page real estate. And this happens for each page and for each postback event triggered. You probably architect things this way to exploit the *DataGrid* control's built-in pagination but, in doing so, you pay for it without ever using it. The only concrete gain is that the *DataGrid* control selects for you the 10 or fewer records to display for the current page—and that is at least something.

> **Note** The paradox of pagination described here applies to both data readers and adapters. The only difference is that data readers are more lightweight objects that, if not closed properly, can cause even more serious damage to the scalability of the application.

Conclusion

So what is the moral of the story? Always try to cache the data of the *DataGrid* control on the server. The *Session* and *Cache* objects have been fine-tuned to work well in the most common programming scenarios, such as when dynamically

changing information is tossed into and out of shopping carts. In addition, the ASP.NET *Session* object can work without cookies and support both Web farm and Web garden architectures.

If you don't want to use in-memory caching, consider caching to a disk. In this case nothing is better than XML, not because it's cool and trendy, but because it has excellent support from ADO.NET classes. You can use the *Session ID* to keep each user's information separated. If you can't afford caching and applications built according to the data pagination paradox, custom paging is the only way to go.

In this chapter, you received an in-depth exposure to basic *DataGrid* control issues. You learned how to bind columns and exploit the built-in pagination services. Paging without sorting, and sorting without auto-reverse, is simply inadequate for today's applications. You now know how to code sorted columns with *DataGrid* controls. But the saga of the *DataGrid* control does not end here. In the next chapter, you learn about templates.

3

Templated *DataGrid* Controls

As you learned in the first two chapters, the *DataGrid* control has a table-based structure and is flexible enough to let you determine the structure and contents of columns. Specifically in Chapter 2, you saw *DataGrid* controls in action and learned how to use their basic yet powerful characteristics. You also learned how to bind special columns representing actions and links. In this chapter, I'll show you how to use HTML templates to take item customization to the next level. You'll learn how to design the content of a column so that it closely reflects the meaning and the structure of your data.

Templated columns enable you to use a combination of HTML and server controls to design a custom layout for a column. The controls in a templated column can be bound to any combination of fields in the data source. You can group multiple fields in a single expression and even embellish fields with HTML attributes such as bold or italic. Templates are column-specific and cannot be applied to auto-generated columns. If you want more than one column to share the same template, you can either duplicate the code in the ASP.NET page for each column, or save some code by loading the template from an external file.

Binding Templated Columns

A templated column is recognized by the <*TemplateColumn*> tag. The body of the tag contains from one through four different templates: *ItemTemplate*, *EditItemTemplate*, *HeaderTemplate*, and *FooterTemplate*. You can tell from these names that the tag lacks a specific template for the alternating item and selected

item. Alternating and selected items typically differ from other items only in terms of graphical settings such as background and foreground color and font styles, and such graphical settings can be easily set for any item at the *DataGrid* control level by using the <*AlternatingItemStyle*> and <*SelectedItemStyle*> tags. You will typically not need to change the layout of a column when one of its cells is selected or when it needs to be redrawn with alternating items. But when you do, for this sort of "genetic" manipulation of the grid, nothing works better than hooking the *ItemCreated* event.

The following code shows how to bind a templated column to a *DataGrid* control. Notice that a templated column, like any other column type, can have header text as well as a sorting expression. It does not, however, have an explicit data source field to bind to. Among the members of the *TemplateColumn* class, you will not find any *DataField* or *DataTextField* properties.

```
<asp:TemplateColumn runat="server"
    HeaderText="heading" SortExpression="field">
    <itemtemplate>
        HTML and/or ASP.NET layout code
    </itemtemplate>
</asp:TemplateColumn>
```

To bind a template column to one or more data fields, you use a data-binding expression and the *DataBinder* class, which was fully described in Chapter 1. To render a column, you could use a *Label* control that has the *Text* property, but you could also choose a drop-down list control (more on this later) or an image, both of which do not have anything like the *Text* property. As a result, you must always use data-binding expressions to bind data, which provides you unprecedented flexibility, albeit with more verbose code. The following code snippet is valid content for an item template:

```
<asp:label runat="server" Text='<%#
    DataBinder.Eval(Container.DataItem, "lastname") %>' />
```

By using *DataBinder.Eval*, you can access any number of fields in the currently bound data source. In addition, you can combine them in any order to obtain an expression that would otherwise be impossible using a simple bound column or button column. The key advantage of templated columns is that any ASP.NET control can be used to populate the column's cells. How those controls are filled is completely up to you, providing an advantage over binding to data using only the rigid *DataField* or *DataTextField* properties.

ItemTemplate is the property that lets you define the layout and the contents of each cell in the column. Other templates let you define the structure of the column header (*HeaderTemplate*) and footer (*FooterTemplate*). You can determine the behavior and appearance of the column when a cell is being edited via the *EditItemTemplate* template. (I'll have more to say about in-place

column editing inChapter 4.) Table 3-1 summarizes the column template properties supported by the *DataGrid* Web control and hints for using them.

Table 3-1 Column Template Properties Supported by the *DataGrid* Control

Template Property	Usage

ItemTemplate	The template for the items in a *DataGrid* control's column.
	`<ItemTemplate>`
	`  <asp:label runat="server"`
	`  text= '<%# ... %>' />`
	`</ItemTemplate>`
	You can use any combination of HTML and ASP.NET controls to populate the column.

EditItemTemplate	Controls the contents of the item selected for editing in the *DataGrid* control's column.
	`<EditItemTemplate>`
	`  <asp:textbox runat="server"`
	`  text= '<%# ... %>' />`
	`</EditItemTemplate>`

HeaderTemplate	Contains information for the heading section.
	`<HeaderTemplate>`
	`  <asp:label runat="server"`
	`  text= "Header" />`
	`</HeaderTemplate>`
	If you omit this template, the column header is rendered with a label or with a link if sorting is enabled. When you specify a custom template, you are responsible for providing the user interface tools to enable sorting on the column.

FooterTemplate	Contains information for the footer section of the column. This property value is null by default.
	`<FooterTemplate>`
	`  <asp:label runat="server" text= "..."  />`
	`</FooterTemplate>`
	The footer is displayed only if the *ShowFooter* property of the *DataGrid* control is set to *true*.

Caution Unless you need special functionality, you are better off leaving the header template as is. Changing the template is an easy and smooth process as long as you don't need to establish a direct interaction between the elements in the header and the user. More on this in a moment.

The template properties in Table 3-1 are exposed by the *TemplateColumn* class as data members of a type that inherits from the *ITemplate* interface. Template-based properties are declaratively set with plain text in the layout of ASP.NET pages. During page processing, the ASP.NET run time takes care of loading that text into a data member of the proper type.

In addition to template properties, the *TemplateColumn* class provides a few style properties—*ItemStyle*, *HeaderStyle*, and *FooterStyle*—which you can use to customize the appearance of items in individual columns. You use these properties in the same way you use the properties for the column types discussed in Chapter 2.

Templated Columns in Action

As powerful as it is, a *DataGrid* control handles data only in terms of columns, and it can display only information that can be expressed by using a data field name. More often than not, these restrictions are too limiting to build a really user-friendly interface. For example, suppose you have three fields to display: first name, last name, and a title of courtesy, such as "Mr.", "Mrs.", or "Dr." The query selects three distinct fields, and your simplest option is to display that data in a *DataGrid* control that uses three distinct columns. Figure 3-1 shows the results.

ID	Title	First Name	Last Name
1	Ms.	Nancy	Davolio
2	Dr.	Andrew	Fuller
3	Ms.	Janet	Leverling
4	Mrs.	Margaret	Peacock
5	Mr.	Steven	Buchanan
6	Mr.	Michael	Suyama
7	Mr.	Robert	King
8	Ms.	Laura	Callahan
9	Ms.	Anne	Dodsworth

Figure 3-1 The *DataGrid* control must display even simple information in a column format.

Figure 3-1 is clear, but the full name looks unusual spread over three columns, and the layout does not transmit that feeling of informality that makes users—especially nonexpert users—feel comfortable with the application. (You'll see a marked improvement in Figure 3-2, which I discuss later.) A better approach would be tying together the three fields in a single string, such as "Ms. Davolio, Nancy".

You can ask the database to return a string that is already in the desired format, but such a request typically results in increased network traffic and

more work for the database engine. It can also leave you with less useful information at hand. For example, if you return only the concatenated string, you lose any ready-to-use information such as the first and last names and, among other options, your ability to sort by these fields.

In Chapter 1, I discussed the pros and the cons of the database returning data fields in a format that is ready to be displayed. That discussion focused on the *CheckBoxList* control, but the pros and cons are really the same for the *DataGrid* control. You obtain preformatted data more effectively by building precalculated columns that are created in memory in a *DataTable* object that you then associate with a grid.

When you have to create expressions based on data fields, using templates gives you more flexibility than any other solution. Unfortunately, templated columns do not come free. The *TemplateColumn* class does have parsing costs and occupies more memory than, say, the *BoundColumn* class. My advice is to avoid using templated columns for relatively simple tasks that you can accomplish in other ways. For example, if you have a cached *DataTable* object that already has all fields in a displayable format, the page will regenerate even faster if you use *BoundColumn* only. Let's review the code for creating and using templated columns.

Concatenating Data Fields

The following code is template code that mimics the standard behavior of bound columns:

```
<asp:TemplateColumn runat="server" HeaderText="Last Name">
    <itemtemplate>
        <%# DataBinder.Eval(Container.DataItem, "lastname") %>
    </itemtemplate>
</asp:TemplateColumn>
```

You can also use a *Label* control and fill its *Text* property with any string, data bound or not. The following code creates a column like the Employee Name column in Figure 3-2. The full source code for the TemplateColumns.aspx application is available on the companion CD.

```
<asp:TemplateColumn runat="server" HeaderText="Employee Name">
    <itemtemplate>
        <asp:label runat="server" Text='<%#
            DataBinder.Eval(Container.DataItem, "TitleOfCourtesy") +
            "<b> " +
            DataBinder.Eval(Container.DataItem, "LastName") +
            "</b>" + ", " +
            DataBinder.Eval(Container.DataItem, "FirstName") %>' />
    </itemtemplate>
</asp:TemplateColumn>
```

ID	Employee Name	Position	From
1	Ms. **Davolio**, Nancy	Sales Representative	USA
2	Dr. **Fuller**, Andrew	Vice President, Sales	USA
3	Ms. **Leverling**, Janet	Sales Representative	USA
4	Mrs. **Peacock**, Margaret	Sales Representative	USA
5	Mr. **Buchanan**, Steven	Sales Manager	UK
6	Mr. **Suyama**, Michael	Sales Representative	UK
7	Mr. **King**, Robert	Sales Representative	UK
8	Ms. **Callahan**, Laura	Inside Sales Coordinator	USA
9	Ms. **Dodsworth**, Anne	Sales Representative	UK

Figure 3-2 The Employee Name column is created by combining multiple fields.

The three fields—*TitleOfCourtesy*, *LastName*, and *FirstName*—have been merged into a single string containing some additional HTML attributes. Compared with Figure 3-1, you can see that the resulting single column displays logically related information in a more readable, user-friendly way.

> **Tip** Every page element in ASP.NET evaluates to a Web control, but this does not mean that all controls work in the same way and have the same impact on overall page performance. The *LiteralControl* control is the most lightweight of all controls because it does not require any special server-side processing. Use literals instead of labels whenever they happen to provide you with the same capabilities. In particular, you use them to render static text that you never update across postback events.

Sorting Templated Columns

Sorting templated columns is similar to sorting any other type of column. You just set the *SortExpression* property to a comma-separated list of fields and make sure the *DataGrid* control is properly configured for sorting. (See Chapter 2 for more information on this.)

Of course, you can sort the template column by one or more data source fields, so in the case of Figure 3-2, the Employee Name column can be sorted by any combination of columns present in the grid's data source but not according to the plain text displayed. If you need to sort the text as it appears in the column, you need to have an in-memory column with that content. Figure 3-3 shows the Employee Name column sorted by *lastname*. The full source code for the SortableTemplateColumns.aspx application is available on the companion CD.

ID	Employee Name	Position	From
5	Mr. **Buchanan**, Steven	Sales Manager	UK
8	Ms. **Callahan**, Laura	Inside Sales Coordinator	USA
1	Ms. **Davolio**, Nancy	Sales Representative	USA
9	Ms. **Dodsworth**, Anne	Sales Representative	UK
2	Dr. **Fuller**, Andrew	Vice President, Sales	USA
7	Mr. **King**, Robert	Sales Representative	UK
3	Ms. **Leverling**, Janet	Sales Representative	USA
4	Mrs. **Peacock**, Margaret	Sales Representative	USA
6	Mr. **Suyama**, Michael	Sales Representative	UK

Figure 3-3 The Employee Name column sorted by the *lastname* field.

> **Caution** If you omit setting the *HeaderText* attribute for a column or set it to a blank string, sorting will not be activated on that column. This could become a rather significant issue if you plan to use templated headers.

Grouping Columns Under a Single Header

Figure 3-3 demonstrates a little bug in the output of the grid: the last names are not aligned, resulting in a display problem. Nothing is wrong with the code—this is the natural consequence of the string format. Is there a way to ensure that, within the same column, a given data field begins at an absolute position? Using a single-row table to wrap up all the data fields just doesn't work.

```
<itemtemplate>
    <table><tr>
        <td>
            <asp:label runat="server" Text='<%#
                DataBinder.Eval(Container.DataItem, "TitleOfCourtesy") %>'
            /> </td>
        <td>
            <asp:label runat="server" Text='<%#
                "<b>" +
                DataBinder.Eval(Container.DataItem, "lastname") %>' +
                "</b>, " +
                DataBinder.Eval(Container.DataItem, "firstname") %>'
            /> </td>
    </tr></table>
</itemtemplate>
```

The alignment problem arises between rows, and only a table that encompasses all the cells in the column can solve it effectively. Using child tables poses the subsequent problem of bubbling down font and color settings. A

much better approach is using distinct columns placed under the umbrella of a common header. Figure 3-4 demonstrates what I mean.

ID	Employee Name	Position	From
1	Ms. **Davolio**, Nancy	Sales Representative	USA
2	Dr. **Fuller**, Andrew	Vice President, Sales	USA
3	Ms. **Leverling**, Janet	Sales Representative	USA
4	Mrs. **Peacock**, Margaret	Sales Representative	USA
5	Mr. **Buchanan**, Steven	Sales Manager	UK
6	Mr. **Suyama**, Michael	Sales Representative	UK
7	Mr. **King**, Robert	Sales Representative	UK
8	Ms. **Callahan**, Laura	Inside Sales Coordinator	USA
9	Ms. **Dodsworth**, Anne	Sales Representative	UK

Figure 3-4 A *DataGrid* control that groups two columns under the same heading section.

To group even more columns under the same header, you need to manipulate the grid structure—something that can be accomplished only by handling the *ItemCreated* event. Here's what the grid's collection of columns looks like in code:

```
<asp:DataGrid id="grid" runat="server"
    ⋮
    OnItemCreated="ItemCreated">
    ⋮
<columns>
    <asp:BoundColumn runat="server" DataField="employeeid" HeaderText="ID">
        <itemstyle backcolor="lightblue" font-bold="true" />
    </asp:BoundColumn>
    <asp:BoundColumn runat="server" DataField="titleofcourtesy" />
    <asp:TemplateColumn runat="server" HeaderText="Employee Name">
        <itemtemplate>
            <asp:label runat="server" Text='<%# "<b> " +
                DataBinder.Eval(Container.DataItem, "LastName") +
                "</b>" + ", " +
                DataBinder.Eval(Container.DataItem, "FirstName") %>'
                />
        </itemtemplate>
    </asp:TemplateColumn>
    ⋮
</columns>
</asp:DataGrid>
```

The columns to group are the *BoundColumn* column that is currently linked to the *titleofcourtesy* field, and the templated column whose caption is Employee Name. The *BoundColumn* column has an empty caption and occupies the second position in the collection.

The *ItemCreated* event is fired when the control is done preparing the given item. You check the item type by using the *ItemType* property of the event data structure, as the following code shows. (The full source code for the GroupingColumns.aspx application is available on the companion CD.)

```
public void ItemCreated(Object sender, DataGridItemEventArgs e)
{
    ListItemType lit = e.Item.ItemType;
    if (lit == ListItemType.Header)
    {
        // Each cell corresponds to a column header
        TableCell cell;

        // One cell must be dropped because we have one too many.
        // It can be the second or the third, one of the two we
        // want to incorporate.
        cell = (TableCell) e.Item.Cells[1];
        e.Item.Cells.Remove(cell);

        // The second heading (the first of the two) spans to
        // cover two columns
        cell = (TableCell) e.Item.Cells[1];
        cell.ColumnSpan = 2;
    }
}
```

After you make sure the event fired because the header of the *DataGrid* control is being processed, you access the cells that form the header. The cells are contained in the *e.Item.Cells* collection. Your next goal is to fuse the headers of columns 2 and 3. To do this, you must remove the extra cell from the table row and then create a header cell that spans the other two columns. You can delete the header of either column 2 or column 3. (Bear in mind that the *Cells* collection is 0-based, just like any other collection in the .NET Framework.) So you might want to drop the column without a caption to save an extra line of code that you would have needed to restore the caption for the column on the left—in this case, column 2. Once the extra cell has been dropped from the table row, you take what is now column 2 and span it over two table columns by using the *ColumnSpan* property.

Adjusting Column Margins

To make the page more readable, you might want to add a few empty pixels to both horizontal sides of the cell text. The *DataGrid* control provides the *CellPadding* property to handle this adjustment. The *CellPadding* property, which maps to the *cellpadding* attribute of the HTML *<table>* tag, has two significant effects, however: it indiscriminately applies to all cells in the grid, and it provides both horizontal and vertical padding.

Other cell formatting options include *CellSpacing*, which indicates the space between columns, and *GridLines*, which lets you specify whether you want horizontal lines or vertical lines, or both. By combining the values of properties such as *CellPadding*, *CellSpacing*, and *GridLines*, you end up with a nice report table in which the cell text is easily readable. Figure 3-5 shows the result of setting *CellPadding* to 5, *CellSpacing* to 0, and leaving *GridLines* to its default value of *Both*.

ID	Employee Name	Position	From
1	Ms. **Davolio**, Nancy	Sales Representative	USA
2	Dr. **Fuller**, Andrew	Vice President, Sales	USA
3	Ms. **Leverling**, Janet	Sales Representative	USA
4	Mrs. **Peacock**, Margaret	Sales Representative	USA
5	Mr. **Buchanan**, Steven	Sales Manager	UK
6	Mr. **Suyama**, Michael	Sales Representative	UK
7	Mr. **King**, Robert	Sales Representative	UK
8	Ms. **Callahan**, Laura	Inside Sales Coordinator	USA
9	Ms. **Dodsworth**, Anne	Sales Representative	UK

Figure 3-5 A good mix of values for cell padding, cell spacing, and grid lines improves readability.

Having empty pixels between columns is helpful, especially when you have right-aligned columns. If you set *CellSpacing* to a nonzero value, the result is worse, as Figure 3-6 shows.

ID	Employee Name	Position	From
1	Ms. **Davolio**, Nancy	Sales Representative	USA
2	Dr. **Fuller**, Andrew	Vice President, Sales	USA
3	Ms. **Leverling**, Janet	Sales Representative	USA
4	Mrs. **Peacock**, Margaret	Sales Representative	USA
5	Mr. **Buchanan**, Steven	Sales Manager	UK
6	Mr. **Suyama**, Michael	Sales Representative	UK
7	Mr. **King**, Robert	Sales Representative	UK
8	Ms. **Callahan**, Laura	Inside Sales Coordinator	USA
9	Ms. **Dodsworth**, Anne	Sales Representative	UK

Figure 3-6 Setting *CellSpacing* to a nonzero value gives you a more cluttered effect.

The root of the problem is that both the *CellSpacing* and *GridLines* properties apply to all cells horizontally and vertically. The ideal solution is to set both properties to 0 and pad as needed at the cell level by using the various margin CSS styles, as shown in this example:

```
style="margin-left:5;margin-right:5;"
```

The preceding code sets the horizontal distance between two side-by-side cells to 5 pixels. As you know, the *DataGrid* control is a table made of <*tr*> and <*td*> elements. These tags are not affected by the value of the margin CSS style. The margin style produces an effect only if applied to the content of the cell. This behavior is hard coded in the CSS definition; the margin is calculated from the parent control. As a result, if you want the cell text drawn 5 pixels from the left border, you have to wrap the cell text in an HTML tag and set the margin style attributes for the tag.

There is a trick to accomplish this in a declarative way for all columns without resorting to templates. When you use templates, padding text is trivial because the *TemplateColumn* class requires you to define the exact layout of the cells, as the following code shows:

```
<asp:TemplateColumn runat="server" HeaderText="...">
<itemstyle ... />
<itemtemplate>
    <span style="margin-left:5;margin-right:5;">
    <%# DataBinder.Eval(...) %>
    </span>
</itemtemplate>
</asp:TemplateColumn>
```

Notice that using a <*span*> tag is more efficient than using a *Label* control because the tag evaluates to a *LiteralControl* control and therefore does not require any server-side processing. For other types of columns, the key to padding the text effortlessly is to use the *DataFormatString* property:

```
<asp:BoundColumn runat="server"
    DataField="employeeid" HeaderText="ID"
    DataFormatString="<span style=\"margin-left:5;\">{0}</span>">
</asp:BoundColumn>
```

The default cell text is represented by the *{0}* placeholder, which is wrapped by a <*span*> tag that has appropriate margin values set. The *DataFormatString* property is not available (and not needed) for templated columns.

Customizing Column Headers

A templated column allows you to define a custom layout for the header and the footer sections. Changing the layout of the header can be problematic if you need to sort that column by an expression. The sorting mechanism is triggered by a *HyperLink* control that the *DataGrid* control automatically embeds in the column heading. The *href* attribute rendered by this *HyperLink* control generates a postback event when the user clicks the element. The target of the link is a piece of client-side JavaScript code whose internals have not been fully documented yet.

```
<a href="javascript:__doPostBack(...)" style="color:White;">
    Caption of the column goes here
</a>
```

What this code does, at the highest level of abstraction, is clear: it posts back to the Web server the form contents of the ASP.NET page. In doing so, it passes to the HTTP runtime some extra information—the parameters of *__doPostBack*—which is responsible for processing the page. At the time this book is being written, you cannot safely make assumptions about the structure and the role of these parameters.

In summary, if you want to change the template of a column heading, by all means do so as long as you don't need sorting capabilities. If you can't just eliminate the sorting, use the *ItemCreated* event to *add* extra controls to the header. The following code dynamically adds a drop-down list to the header of a template column, allowing you to choose the expression to sort by. Figure 3-7 shows the final result.

```
public void ItemCreated(Object sender, DataGridItemEventArgs e)
{
    ListItemType lit = e.Item.ItemType;
    if (lit == ListItemType.Header)
    {
        // Create and fill a drop-down list control
        DropDownList dd = new DropDownList();
        dd.ID = "ddSort";
        ListItem li1, li2, li3;

        // ListItem constructor takes Text and Value for the item
        li1 = new ListItem("Title of courtesy", "titleofcourtesy");
        dd.Items.Add(li1);

        li2 = new ListItem("Last Name", "lastname");
        dd.Items.Add(li2);

        li3 = new ListItem("First Name", "firstname");
        dd.Items.Add(li3);

        // Select the item, if any, that was selected last time
        dd.SelectedIndex = Convert.ToInt32(grid.Attributes["FieldIndex"]);

        // Add the drop-down list to the header of the 2nd column
        TableCell cell = (TableCell) e.Item.Controls[1];
        cell.Controls.Add(dd);
    }
}
```

ID	Sort by Title of courtesy	Position	From
2	Dr. **Fuller**, Andrew	Vice President, Sales	USA
5	Mr. **Buchanan**, Steven	Sales Manager	UK
6	Mr. **Suyama**, Michael	Sales Representative	UK
7	Mr. **King**, Robert	Sales Representative	UK
4	Mrs. **Peacock**, Margaret	Sales Representative	USA
1	Ms. **Davolio**, Nancy	Sales Representative	USA
3	Ms. **Leverling**, Janet	Sales Representative	USA
8	Ms. **Callahan**, Laura	Inside Sales Coordinator	USA
9	Ms. **Dodsworth**, Anne	Sales Representative	UK

Figure 3-7 A custom layout for a column header. You pick up a sort expression from the drop-down list and click Sort By to sort.

Customizing the header is useful when you have templated columns that group together several fields. The preceding code creates a dynamic drop-down list with the available sort expressions. Next it retrieves the currently selected expression when users click the header's link. The header text of the template column must be set to a non-empty string so that the standard infrastructure for sorting can be built to work properly.

```
<asp:TemplateColumn runat="server"
    HeaderText="Sort by" SortExpression="*">
```

Notice the unusual value assigned to the *SortExpression* property. It plays a critical role in that it allows the sort command handler to recognize that the user clicked a column requiring special treatment for sorting. The *SortExpression* attribute, in fact, is the only element you have for recognizing the clicked column.

```
public void SortCommand(Object sender, DataGridSortCommandEventArgs e)
{
    // Code that retrieves the grid's data source GOES HERE
    ⋮
    if (e.SortExpression != "*")
        dv.Sort = e.SortExpression;
    else
    {
        // Retrieves the drop-down list control through its ID
        DataGridItem dgi = (DataGridItem) e.CommandSource;
        DropDownList dd = (DropDownList) dgi.FindControl("ddSort");

        // Retrieves the sorting expression from the list
        dv.Sort = dd.SelectedItem.Value;
```

(continued)

```
        // Persists the currently selected drop-down item
        ViewState["FieldIndex"] = dd.SelectedIndex.ToString();
    }

    // Refreshes the grid
    grid.DataBind();
}
```

The expression you assign to *SortExpression* does not matter as long as it allows you to identify the column.

Creating Templates Dynamically

Typically, the layout code of templated columns is defined at design time, but you might face situations in which using predefined templates is not the optimal solution and you need a dynamic template. For example, if you know in advance that a lot of changes must be applied at run time via events such as *ItemCreated* and *ItemDataBound*, there is no reason to define a static template, forcing the control to support a double effort: processing the template first and the changes next. Also, when users can change among different views of the same data, a dynamic template is preferable. Whatever your reason for adding templated columns dynamically, you face the problem of how to create a template programmatically. In such a situation, using an external ASCX file can help.

Loading Templates from Files

A template for a column property is a class that implements the *ITemplate* interface. An instance of such an object can be created using the *LoadTemplate* method of the *Page* class. *LoadTemplate* takes only one argument: the name of the text file that describes the template. The file must have an ASCX extension—ASCX is the typical extension of user control files, formerly known as *pagelets*. You create a file-based template column using the following code:

```
TemplateColumn tc = new TemplateColumn();
tc.ItemTemplate = Page.LoadTemplate("template.ascx");
```

The template file can be written in any .NET language and not necessarily in the language of the page. The *Page.LoadTemplate* method can be used to load the layout code for any template property of the column, including *EditItemTemplate* and *HeaderTemplate*.

> **Note** There is no way to load a column template from an external file at design time. The *<ItemTemplate>* ASP.NET tag does not support an *src* attribute or an attribute similar to *src*.

The ASCX user control you use to populate the templates of a column defines the HTML and the ASP.NET controls you want to employ. The following code shows the ASCX file that concatenates *TitleOfCourtesy*, *LastName*, and *FirstName*:

```
<%@ Language="C#" %>
<%#
    DataBinder.Eval(((DataGridItem) Container).DataItem, "TitleOfCourtesy")
    + " " + "<b>" +
    DataBinder.Eval(((DataGridItem) Container).DataItem, "LastName") +
    "</b>, " +
    DataBinder.Eval(((DataGridItem) Container).DataItem, "FirstName")
%>
```

The user control must indicate its language, even if the language is the same one being used in the hosting page. The following code shows the same user control written in Visual Basic .NET. You can use this code interchangeably with the previous code.

```
<%@ Language="VB" %>
<%#
    DataBinder.Eval( _
        CType(Container, DataGridItem).DataItem, "TitleOfCourtesy") + _
        "<b>" + _
    DataBinder.Eval( _
        CType(Container, DataGridItem).DataItem, "LastName") + _
        "</b>, " + _
    DataBinder.Eval( _
        CType(Container, DataGridItem).DataItem, "FirstName")
%>
```

Managing Multiple Views for a Column

The ability to dynamically load templates from disk can be exploited to build an interesting application that lets users change the view of a templated column. Figure 3-8 shows what I mean.

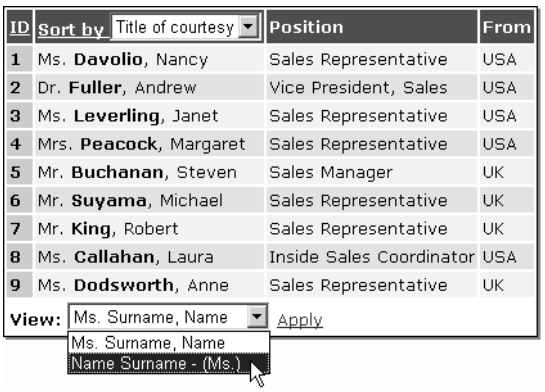


Figure 3-8 This application uses dynamically loaded templates to allow users to change the column view.

The *DataGrid* control in Figure 3-8 shows a footer that has been dynamically modified to span all columns. The footer contains a drop-down list with the available views for the templated column. The user selects the desired view and then clicks the Apply link button to enable it.

Setting Up the Footer

You can place the controls for selecting the view mode anywhere on the page. A good place is in the footer of the templated column. In this case, you can use the <*FooterTemplate*> tag:

```
<FooterTemplate>
    <b>View:</b>
    <asp:dropdownlist runat="server" id="ddViews" />
    <asp:linkbutton runat="server"
        Text="Apply" CommandName="ApplyView" />
<FooterTemplate>
```

By default, the *DataGrid* control's footer is disabled, and you turn it on by setting the *ShowFooter* attribute to *true*. Usually the footer is a blank row added at the bottom of the grid's current page and needs a different structure from the rest of the rows. Let's say you want the footer to span all the columns in the grid to become a table row with a single cell. Once again you have to resort to the *ItemCreated* event to accomplish this kind of structural manipulation of the grid items. The following code shows the strategy for creating the footer shown in Figure 3-9.

```
if (lit == ListItemType.Footer)
{
    // Remove 1st and 3rd columns in the original schema
    e.Item.Cells.RemoveAt(0);
    e.Item.Cells.RemoveAt(1);
    e.Item.Cells[0].ColumnSpan = 3;
```

```
    // Populate the drop-down list with available views
    // Each view corresponds to an ASCX file in the current folder
    DropDownList ddViews = (DropDownList) e.Item.FindControl("ddViews");
    ListItem l;
    l = new ListItem("Ms. Surname, Name", "courtesylastfirst.ascx");
    ddViews.Items.Add(l);
    l = new ListItem("Name Surname - (Ms.)", "firstlastcourtesy.ascx");
    ddViews.Items.Add(l);

    // Select the previously selected element, if any
    // Need to preserve list state across postback events
    ddViews.SelectedIndex = Convert.ToInt32(ViewState["ViewIndex"]);
}
```

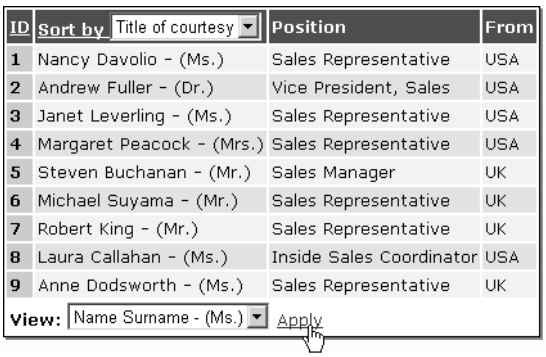


Figure 3-9 In this grid, you use the *ItemCreated* event to create a footer that becomes a table row with a single cell.

Applying the View

The view in Figure 3-9 is enabled when the user clicks the Apply link button. This button has been assigned a *CommandName* property. Any clickable element within the body of the *DataGrid* control raises an *ItemCommand* event when the user clicks it. This event is handled in much the same way it is for the *DataList* control, which I covered in Chapter 1.

When you handle the *ItemCommand* event, you distinguish among the various elements that might have fired the event by using the *CommandName* attribute. In the example of Figure 3-9, I defined a link button with a command name of *ApplyView*. To handle the button click, you need code such as the following:

```
public void ItemCommand(Object sender, DataGridCommandEventArgs e)
{
    if (e.CommandName == "ApplyView")
    {
        DropDownList ddViews = (DropDownList)e.Item.FindControl("ddViews");
        String strFile = ddViews.SelectedItem.Value;
```

(continued)

```
        ViewState["CurrentViewFile"] = strFile;
        ViewState["ViewIndex"] = ddViews.SelectedIndex.ToString();
        UpdateView();
    }
}
```

The *ItemCommand* handler is registered through the *OnItemCommand* attribute of the *<asp:datagrid>* tag.

```
<asp:datagrid id="grid" runat="server"
    ⋮
    OnItemCommand="ItemCommand"
    ⋮
/>
```

You first retrieve the instance of the drop-down list control. The control *ddViews* is not accessible at the page level because of the implementation of ASP.NET templates. Each template runs in a separate naming container that makes it impossible for the run time processing the ASP.NET page to retrieve the drop-down list control by name within the page scope. So you cannot use *ddViews*, which is the ID of the drop-down list, as the programmatic identifier of the control in the page code. The ID *ddViews* makes sense only in the context of the template—a sort of child control with its own namespace—that contains it. To retrieve a valid instance of the drop-down list control, then, you need to resort to the following code:

```
DropDownList ddViews = (DropDownList) e.Item.FindControl("ddViews");
```

Notice that *FindControl* called from the *Page* object or the *DataGrid* object will not work, because *FindControl* knows how to locate a control only in the current naming container. In the body of *ItemCommand*, when the command name is *ApplyView*, *e.Item* represents the footer. Calling *FindControl* within the range of the footer turns out to be successful.

In *ItemCommand*, once you hold a reference to the drop-down list control, you pick up the name of the currently selected view and store it, as well as its index, in the *Attributes* collection of the grid. The drop-down list's *Value* property contains the name of the ASCX file that represents the view, whereas its *Text* property points to a display name. The name of the file and its index need to be persisted across multiple invocations of the same page to guarantee that the control's state can be correctly restored.

Refreshing the View

Once you know the name of the file to load the template from, you are pretty much finished. The only remaining work is letting the *DataGrid* control know about the new template for one of its columns. In our running example, the templated column is the second column. The next code snippet explains how to change the item template dynamically. The full source code for the ColumnView.aspx application is available on the companion CD.

```
TemplateColumn tc = (TemplateColumn) grid.Columns[1];
tc.ItemTemplate = Page.LoadTemplate((String)ViewState["CurrentViewFile"]);
grid.DataBind();
```

The final call to *DataBind* causes the grid to redraw its user interface, taking into account all the changes to the columns.

> **Note** The working directory for *LoadTemplate* is the root of the current Web application, so you don't need to indicate a fully qualified URL. However, *LoadTemplate* requires a virtual path and throws an exception if you pass it an absolute file system path with drive and directory information.

Loading Templates from Strings

Unlike many other methods that are expected to read from disk, the *Page* object's *LoadTemplate* method does not support streams and writers. If this support were possible, then in-memory strings could have been used to create dynamic templates. If you don't want to, or can't afford to, make your application dependent on external ASCX files, how can you create dynamic templates? Typically, you don't want to deal with ASCX files because having several file names hard coded in the source is too restrictive. Another good reason to avoid disk-based templates is that it results in the template code being just one of the configuration parameters of the application. In this case, in fact, you might have all this information stored in a centralized medium such as a SQL Server table or an XML file.

Is there a way to dynamically create a template from a string? I haven't found any documentation that says it is possible or how to do it, but it may be that the documentation hasn't been written yet. After looking at the programming interface of the involved classes, I haven't been able to discover a way to do this yet. However, nothing really prevents you from creating a temporary file, writing the string that represents the layout, and loading a template from the temporary file. When you create temporary files from within an ASP.NET application, make sure that the file name is really unique for each concurrent session. For this purpose, use the Session ID or create a unique temporary file through the static method, *Path.GetTempFileName*. Bear in mind that the *LoadTemplate* method assumes it has been given a virtual path. On the other hand, stream and writer classes require absolute paths and don't know how to cope with virtual paths. As a result, you come up with the following code to create and load a string-based template.

```
// Create the column object
TemplateColumn bc = new TemplateColumn();

// Create the temp file and write the template code
String tmp = Session.SessionID + ".ascx";
StreamWriter sw = new StreamWriter(Server.MapPath(tmp));
sw.Write(strLayoutCode);
sw.Close();

// Load the template from the temp file and add the column
bc.ItemTemplate = Page.LoadTemplate(tmp);
grid.Columns.Add(bc);

// Delete the temp file
File.Delete(Server.MapPath(tmp));
```

You need to use *Server.MapPath* to map a URL from a virtual to a physical path. You need to do this only when working with streams and files.

> **Tip** Use the *StringBuilder* class to build the text of the template instead of concatenating strings with the plus sign (+) operator. The *Append* method (part of the *StringBuilder* class) really appends a given string to the internal buffer, whereas the plus sign (+) operator creates a brand new string that is the sum of the two.

Implementing *ITemplate*

When the ASP.NET run time processes a template, it parses the string, extracting the definition of the various controls that actually make it. These controls are then instantiated and added to the *Controls* collection of the naming container—typically a *DataGridItem* object. You can implement this pattern yourself by writing a made-to-measure class that inherits from *ITemplate*. A living instance of this class can then be assigned to any template property such as *ItemTemplate*.

The *ITemplate* interface has only one method, which is named *InstantiateIn*. The method is called to populate the user interface of the container control with instances of child controls in accordance with the expected template. You can certainly come up with a flexible and configurable class that reads from an external source the controls to create and bind to data. More simply, though, you might want to write ad-hoc classes—one for each needed template. Figure 3-10 shows just this.

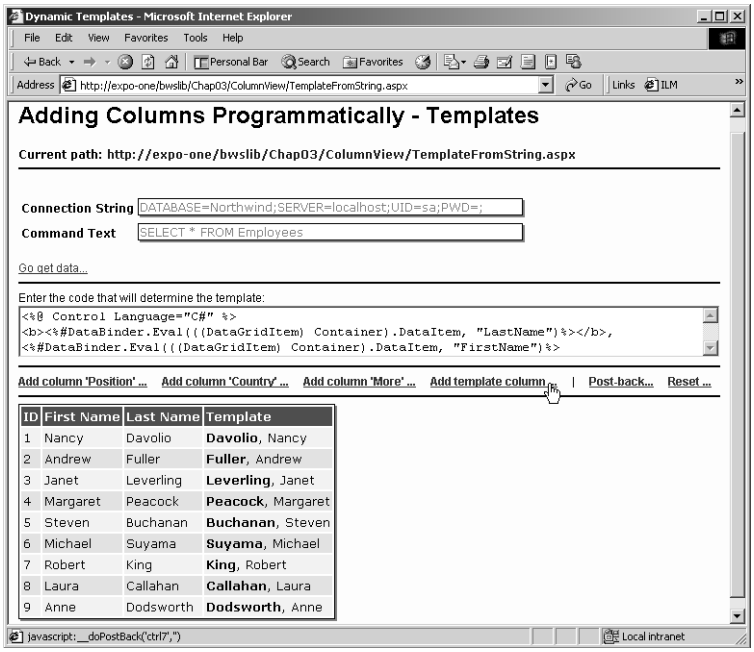


Figure 3-10 Write the template text and generate a new column dynamically.

Structure of *ITemplate* Classes

The *ITemplate* class can be defined in the *<script>* section of an ASP.NET page as well as in separate C# or Visual Basic .NET class files that you link to the project. Another good place for this kind of code is in the code-behind file for the ASP.NET pages that use it. An *ITemplate*-based class looks like in the following code.

```
class LastFirstNameTemplate : ITemplate
{
    public void InstantiateIn(Control container)
    {...}

    private void BindLastName(Object s, EventArgs e)
    {...}

    private void BindFirstName(Object s, EventArgs e)
    {...}
}
```

In the body of *InstantiateIn*, you create instances of controls and add them to the specified container. For *DataGrid* controls, the container is an object of type *DataGridItem*. It will be *DataListItem* for a *DataList* control. In general, a container is any class that implements the *INamingContainer* interface.

If the control being added to the container's *Controls* collection has to be bound to a data source column, then you also register your own handler for the *DataBinding* event. When the event occurs, you retrieve the text out of the data source and refresh the user interface of the control. When defined for a server control, the *DataBinding* event handler is expected to resolve all data-binding expressions in the server control and in any of its children. Let's consider the layout that we repeatedly encountered earlier:

```
<%# "<b>" + DataBinder.Eval(Container.DataItem, "lastname") +
    "</b>, " + DataBinder.Eval(Container.DataItem, "firstname") %>
```

The following code demonstrates a template class that obtains the same results.

```
public class LastFirstNameTemplate : ITemplate
{
    public void InstantiateIn(Control container)
    {
        container.Controls.Add(new LiteralControl("<b>"));

        Label lblLName = new Label();
        lblLName.DataBinding += new EventHandler(this.BindLastName);
        container.Controls.Add(lblLName);

        container.Controls.Add(new LiteralControl("</b>, "));

        Label lblFName = new Label();
        lblFName.DataBinding += new EventHandler(this.BindFirstName);
        container.Controls.Add(lblFName);
    }

    private void BindLastName(Object sender, EventArgs e)
    {
        Label l = (Label) sender;
        DataGridItem container = (DataGridItem) l.NamingContainer;
        l.Text = ((DataRowView) container.DataItem)["lastname"].ToString();
    }

    private void BindFirstName(Object sender, EventArgs e)
    {
        Label l = (Label) sender;
        DataGridItem container = (DataGridItem) l.NamingContainer;
        l.Text =
            ((DataRowView) container.DataItem)["firstname"].ToString();
    }
}
```

The *DataBinding* Event Handler

A *DataBinding* event handler accomplishes two tasks. If programmed correctly, it should first take and hold the underlying data item. Second, it has to refresh the user interface of the bound control to reflect data binding.

A reference to the involved control can be obtained through the always declared, but not frequently used, *sender* parameter. The container that hosts the control is returned by the *NamingContainer* property of the control itself. At this point, you have all that you need to set up and use another well-known ASP.NET expression: *Container.DataItem*. The type of the data item depends on the data source associated with the *DataGrid*. In most real-world scenarios, it will be *DataRowView*. What remains is to access a particular column on the row and set the control's bound properties.

> **Note** Nearly identical code can be used to dynamically create templates for the *DataList* and the *Repeater* controls. The only difference is in the type casting required in the *DataBinding* event handlers.

The full source for the ITemplateClass.aspx application can be found on the companion CD.

Adapting Columns to Data

Templates are good at implementing special formatting features for columns, so you should take full advantage of their flexibility to adapt column layouts so that they really reflect the nature of your data. In most cases, plain text is good enough when you want to represent information from the data source. But in certain situations, you'll want to use a custom template composed of controls whose structure is more closely aligned with the intent of your data. Want some examples of these controls? Consider Booleans, images, and arrays.

Showing Boolean Values

A Boolean field can take two possible values: *true* or *false*. In spite of its simplicity, or maybe because of it, a Boolean value does not have a unique graphical representation. So Booleans can look more user-friendly in applications when they are rendered with a yes/no or 0/1 pair. If you don't want to render Booleans with plain text, check boxes are probably the most effective alternative. Let's examine a concrete example.

The *Employees* table in the SQL Server 2000 Northwind database has a column named *ReportsTo* that contains numeric values indicating whether a given employee reports to someone. In particular, the content of the field is the employee ID of the boss and is null if the employee reports to no one. The following ASP.NET code demonstrates a templated column that uses a check box to render the content of the *ReportsTo* field. (Notice that this code snippet uses an "unknown" field name, *boss*, instead of the expected *ReportsTo*. This name is not a typo, and I'll have more to say about it in a moment.)

```
<asp:TemplateColumn HeaderText="Reports">
    <itemtemplate>
    <asp:checkbox runat="server" enabled="false" checked='<%#
        HasBoss((int)DataBinder.Eval(Container.DataItem, "boss")) %>' />
    </itemtemplate>
</asp:TemplateColumn>
```

The *CheckBox* control is disabled to prevent users from clicking it and altering the state. The *Checked* property takes a data-binding expression that evaluates to a Boolean value returned by the user-defined function *HasBoss*. Figure 3-11 shows the output of the sample page.

```
bool HasBoss(int bossID)
{
    if (bossID != 0)
        return true;
    return false;
}
```

ID	Employee Name	Reports	Position
1	Ms. **Davolio**, Nancy	☑	Sales Representative
2	Dr. **Fuller**, Andrew	☐	Vice President, Sales
3	Ms. **Leverling**, Janet	☑	Sales Representative
4	Mrs. **Peacock**, Margaret	☑	Sales Representative
5	Mr. **Buchanan**, Steven	☑	Sales Manager
6	Mr. **Suyama**, Michael	☑	Sales Representative
7	Mr. **King**, Robert	☑	Sales Representative
8	Ms. **Callahan**, Laura	☑	Inside Sales Coordinator
9	Ms. **Dodsworth**, Anne	☑	Sales Representative

Figure 3-11 Using check boxes to render Boolean data.

Employing a user-defined function within the data-binding expression for the *Checked* property is not strictly necessary in our example. What really matters is that *Checked* is assigned a Boolean value. The *ReportsTo* field contains integer data, however, so casting is mandatory to satisfy the requirements of the strong typing characteristic of .NET. The following simpler code obtains the same effect:

```
checked='<%#
    Convert.ToBoolean(DataBinder.Eval(Container.DataItem, "boss")) %>
```

A subtle point to consider is that the *ReportsTo* field can contain nulls, but null values can't be converted to Booleans. To work around this potential problem, I created an expression-based field named *boss*, which I return instead of the original *ReportsTo*. The query command that populates the grid of Figure 3-11 looks like this:

```
SELECT employeeid, titleofcourtesy, firstname, lastname,
    title, ISNULL(reportsto,0) AS boss
    FROM Employees
```

The contents of the *ReportsTo* field are filtered by the *ISNULL* T-SQL function. The preceding expression replaces each null entry in the given field with 0, making the conversion to the Boolean type possible and safe. The full source code for the Booleans.aspx application is available on the companion CD.

Showing Images

The template of a column can also contain images that you can use in many ways, not the least of which is to make the user interface more attractive. If you have a field that stores image URLs, you can insert a *<asp:image>* tag in the template of the column and make the *ImageUrl* attribute of the element point to the field content.

In the previous example, I disabled the check boxes to prevent users from clicking them. But now they don't appear to represent valid pieces of information, and graying them out makes the user interface look inconsistent. I will fix this using images. Here's how you do it.

To start off, you create two little GIF files, each containing the bitmap that browsers use to represent checked and unchecked controls.

Next, you use these little pictures in lieu of disabled check box controls. The code you need looks like the following:

```
<itemtemplate>
    <asp:image runat="server"
        imageurl='<%# GetProperGifFile(
             (int) DataBinder.Eval(Container.DataItem, "boss")) %>'
    />
</itemtemplate>
```

As shown in Figure 3-12, the final result of the code is decidedly attractive and effective. (The full source code for the Images.aspx application is available on the companion CD.) The look and feel of the user interface is significantly improved because the check boxes are not grayed out. In addition, images

cannot be clicked or selected and are not even a valid target for the tab. The correct GIF file is selected by using a rather straightforward function:

```
String GetProperGifFile(int bossID)
{
    if (bossID != 0)
        return "checked.gif";
    return "unchecked.gif";
}
```

ID	Employee Name	Reports	Position
1	Ms. **Davolio**, Nancy	☑	Sales Representative
2	Dr. **Fuller**, Andrew	☐	Vice President, Sales
3	Ms. **Leverling**, Janet	☑	Sales Representative
4	Mrs. **Peacock**, Margaret	☑	Sales Representative
5	Mr. **Buchanan**, Steven	☑	Sales Manager
6	Mr. **Suyama**, Michael	☑	Sales Representative
7	Mr. **King**, Robert	☑	Sales Representative
8	Ms. **Callahan**, Laura	☑	Inside Sales Coordinator
9	Ms. **Dodsworth**, Anne	☑	Sales Representative

Figure 3-12 Even though the images look like check boxes, they are elements that cannot be clicked or selected.

Showing Arrays

Master-detail schemas are common in Web development: users select a record and expect to see more details about it. Typically, providing details in this way is implemented using two interrelated *DataGrid* controls or a single *DataGrid* control to list the primary records and a child form to display additional information.

In certain situations, however, you will need to display only an array of logically related information for each record in the table—for example, all the territories covered by an employee. Because a one-to-many relationship exists between employees and territories, a master-detail schema makes some sense, but setting up a master-detail schema to show only an array of strings (which is what the covered territories actually are from the code's perspective) seems like overkill. Templated columns allow you to elegantly handle this display of related information by using drop-down list controls and ADO.NET data relations.

Let's create a templated column that features a drop-down list control. How can you populate this control? Based on the structure of the Northwind database, the following SQL command returns a result set with two fields containing the ID of the employee and one of the territories she covers:

```
SELECT et.employeeid, t.territorydescription
    FROM EmployeeTerritories AS et
    INNER JOIN Territories AS t ON t.territoryid = et.territoryid
```

Figure 3-13 shows the structure of the result set.

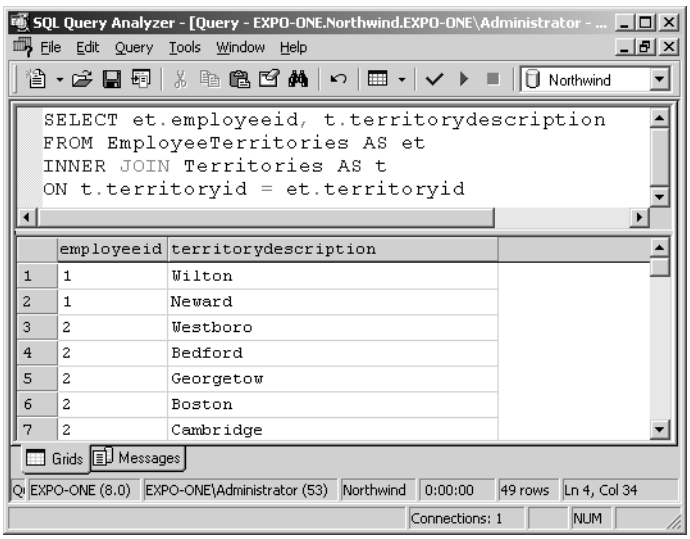


Figure 3-13 This query displays the sales territories covered by an employee.

The concept behind the code is binding the drop-down list control to the subset of the result set that applies to a given employee. The following code associates the data source of the drop-down list with a user-defined function named *GetTerritories*:

```
<asp:TemplateColumn HeaderText="Territories">
<itemtemplate>
    <asp:dropdownlist runat="server" width="120px"
    datasource='<%# GetTerritories((DataRowView) Container.DataItem) %>'
    />
</itemtemplate>
</asp:TemplateColumn>
```

The *GetTerritories* function takes an argument of type *DataRowView*. *GetTerritories* is passed *Container.DataItem*, which is a view of the data row associated with the current row in the *DataGrid* control. The *DataGrid* control, as usual, is associated with a *DataTable* object that stems from a query on the *Employees* table.

```
DataSet ds = new DataSet();
String strCmd = "SELECT * FROM Employees";
SqlDataAdapter da = new SqlDataAdapter(strCmd, conn);
da.Fill(ds, "EmployeesTable");
grid.DataSource  = ds.Tables["EmployeesTable"];
```

GetTerritories takes a *DataRowView* object, but it needs a quick way to access another *DataTable* object's rows that are logically related to the grid's data source via the employee ID field.

You add a second *DataTable* object to the same *DataSet* object used to populate the *DataGrid* control. This table, named *TerritoriesTable*, contains the results of the previous query on the *Territories* table.

```
StringBuilder sb = new StringBuilder("");
sb.Append("SELECT et.employeeid, t.territorydescription ");
sb.Append("FROM EmployeeTerritories AS et ");
sb.Append("INNER JOIN Territories AS t ");
sb.Append("ON t.territoryid = et.territoryid");

// Append a second table to the same DataSet
da = new SqlDataAdapter(sb.ToString(), conn);
da.Fill(ds, "TerritoriesTable");
```

At this point, you create a *DataRelation* object to relate the two tables. A data relation is a logical, in-memory link that you set between two tables in the same *DataSet* object. A relation is established similarly to the way it is when using the *INNER JOIN* SQL command—the relation is based on the values of a common field.

> **Note** The *DataRelation* object is a specific part of the ADO.NET object model. It has no direct counterpart in the ADO object model but shares a certain functional resemblance with ADO's data shaping technology.

The following code shows how to create a relation between *EmployeesTable* and *TerritoriesTable* according to the values of the *employeeid* field.

```
DataColumn dc1 = ds.Tables["EmployeesTable"].Columns["employeeid"];
DataColumn dc2 = ds.Tables["TerritoriesTable"].Columns["employeeid"];
DataRelation drel = new DataRelation("EmployeesAndTerritories", dc1, dc2);
ds.Relations.Add(drel);
```

A *DataRelation* object must have a name (*EmployeesAndTerritories*, in this case) and, to be effective, must be added to the *Relations* collection of the *DataSet* object. Once the data relation has been successfully set, you call the *GetChildRows* method of the *DataRow* object and obtain the array of rows that matches the current value of the key field. *GetChildRows* returns an array of *DataRow* objects. *GetTerritories* just exploits an existing data relation to do its job.

```
ArrayList GetTerritories(DataRowView drv)
{
    // Extract the underlying row from the DataRowView object
    DataRow dr = drv.Row;

    // Retrieve the child rows according to the data relation
    DataRow[] adr = dr.GetChildRows("EmployeesAndTerritories");

    // Create and return an array made of the values in the
    // TerritoryDescription column
    ArrayList a = new ArrayList();
    foreach(DataRow tmp in adr)
        a.Add(tmp["territorydescription"]);
    return a;
}
```

As discussed in Chapter 1, an array object is good at populating a drop-down list control. Figure 3-14 shows the final result. The full source code for the Arrays.aspx application is available on the companion CD.

ID	Employee Name	Position	Reports	Territories
1	Ms. **Davolio**, Nancy	Sales Representative	☑	Wilton
2	Dr. **Fuller**, Andrew	Vice President, Sales	☐	Wilton Neward
3	Ms. **Leverling**, Janet	Sales Representative	☑	Atlanta
4	Mrs. **Peacock**, Margaret	Sales Representative	☑	Rockville
5	Mr. **Buchanan**, Steven	Sales Manager	☑	Providence
6	Mr. **Suyama**, Michael	Sales Representative	☑	Phoenix
7	Mr. **King**, Robert	Sales Representative	☑	Hoffman Estates
8	Ms. **Callahan**, Laura	Inside Sales Coordinator	☑	Philadelphia
9	Ms. **Dodsworth**, Anne	Sales Representative	☑	Hollis

Figure 3-14 A templated column using a drop-down list control to show an array of related information.

Conclusion

The *TemplateColumn* class used in a *DataGrid* control lets you create a column with a customized control layout. This is extremely helpful in many circumstances. You use templates when you want to apply special formatting features not available through more direct channels such as CSS styles or the

DataFormatString property of other column classes. You also use templates to combine several fields in a single layout to improve readability and user-friendliness. Templates are also the key to a meaningful graphical representation of data.

So far you know how to retrieve and present data. In Chapter 4, we'll delve deep into the editing capabilities of the *DataGrid* control.