# Table of Contents

**Chapter 11. Creating Bindable Grids of Data** ................................. 1
  - The DataGrid Control ........................................................................................................ 2
  - The GridView Control ....................................................................................................... 16
  - Conclusion .......................................................................................................................... 57
Chapter 11
Creating Bindable Grids of Data

In this chapter:
The DataGrid Control ................................................. 510
The GridView Control ................................................. 524
Conclusion .......................................................... 565

Data-bound controls play a key role in the development of ASP.NET applications. Data-driven controls allow you to associate the whole interface, or individual properties, with one or more columns read out of a .NET-compliant data source. We already mentioned data-bound controls in Chapter 10 and reviewed their basics. In this chapter, we’ll delve into the details of a couple of extremely versatile data-bound controls that are a fixed presence in any real-world ASP.NET application—the DataGrid control in ASP.NET 1.x and the GridView control in ASP.NET 2.0 and newer versions.

Both controls render a multicolumn, templated grid and provide a largely customizable user interface with read/write options. In spite of the rather advanced programming interface and the extremely rich set of attributes, DataGrid and GridView controls simply generate an HTML table with interspersed hyperlinks to provide interactive functionalities such as sorting, paging, selection, and in-place editing.

Although they are customizable at will, grid controls feature a relatively rigid and inflexible graphical model. The data bound to a DataGrid or GridView is always rendered like a table, therefore, in terms of rows and columns. As we’ll see later in the chapter, though, the contents of the cells in a column can be customizable to some extent using system-provided as well as user-defined templates.

The DataGrid is the principal control of most data-driven ASP.NET 1.x applications. Like all ASP.NET 1.x controls, the DataGrid is fully supported in ASP.NET 2.0 and newer versions, but it is partnered with a newer control that is meant to replace it in the long run. The new grid control, GridView, is complemented by other view controls, such as DetailsView, FormView, and only in ASP.NET 3.5 the ListView control. (We’ll cover ListView in the next chapter and DetailsView and FormView in Chapter 13.) The GridView is a major upgrade of the ASP.NET 1.x DataGrid control. It provides the same basic set of capabilities, plus a long list of extensions and improvements.
In this chapter, we’ll first take a look at the DataGrid capabilities and try to identify and discuss its major shortcomings and limitations. Then we’ll consider the GridView control and its modified programming interface. For brand new ASP.NET 3.5 applications, choosing the GridView over the DataGrid is a no-brainer. For ASP.NET 1.x applications that are being maintained, a move to the GridView doesn’t present any significant difficulties and such a move positions you well for future enhancements.

Note
I mentioned that the output of grid controls is an HTML table. It should be said that this represents just the default and most common option. By installing the CSS-friendly adapter toolkit, you can have grid controls to output HTML markup that makes no use of tables but resorts to cascading style sheet (CSS) style to define the data layout. For more information, check out the following white paper: http://www.asp.net/cssadapters/whitepaper.aspx. Note, though, that only the GridView control supports CSS adapters; the DataGrid always outputs HTML tables.

The DataGrid Control

The DataGrid is a column-based control that supports various types of data-bound columns, including text, templated, and command columns. You associate the control with a data source using either the DataSource or, in ASP.NET 2.0 and beyond, the DataSourceID property. The simplest way of displaying a table of data using the ASP.NET grid is as follows:

```xml
<asp:DataGrid runat="server" id="grid" />
```

Once the control has been placed into the page, you bind it to the data source and have it display the resulting markup.

The DataGrid Object Model

The DataGrid control provides a grid-like view of the contents of a data source. Each column represents a data source field, and each row represents a record. The DataGrid control supports several style and visual properties; in a more realistic scenario, the markup required to embed the control in a page is significantly larger and more complex so as to include all those attributes. In ASP.NET 2.0 and newer versions, themes can wrap control-specific visual settings and apply them seamlessly while leaving the markup as slim as possible.

Properties of the DataGrid Control

Table 11-1 lists the properties of the control, except those the control inherits from Control and WebControl.
### TABLE 11-1 Properties of the DataGrid Class

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllowCustomPaging</td>
<td>Gets or sets whether custom paging is enabled. AllowPaging must be set to true for this setting to work.</td>
</tr>
<tr>
<td>AllowPaging</td>
<td>Gets or sets whether paging is enabled.</td>
</tr>
<tr>
<td>AllowSorting</td>
<td>Gets or sets whether sorting is enabled.</td>
</tr>
<tr>
<td>AlternatingItemStyle</td>
<td>Gets the style properties for alternating rows.</td>
</tr>
<tr>
<td>AutoGenerateColumns</td>
<td>Gets or sets whether columns are automatically created and displayed for each field in the data source. True by default.</td>
</tr>
<tr>
<td>BackImageUrl</td>
<td>Gets or sets the URL of the image to display as the background of the control.</td>
</tr>
<tr>
<td>Caption</td>
<td>The text to render in the control’s caption. Not available in ASP.NET 1.x.</td>
</tr>
<tr>
<td>CaptionAlign</td>
<td>Alignment of the caption. Not available in ASP.NET 1.x.</td>
</tr>
<tr>
<td>CellPadding</td>
<td>Gets or sets the space (in pixels) remaining between the cell’s border and the embedded text.</td>
</tr>
<tr>
<td>CellSpacing</td>
<td>Gets or sets the space (in pixels) remaining, both horizontally and vertically, between two consecutive cells.</td>
</tr>
<tr>
<td>Columns</td>
<td>Gets a collection of DataGridColumn objects.</td>
</tr>
<tr>
<td>CurrentPageIndex</td>
<td>Gets or sets the index of the currently displayed page.</td>
</tr>
<tr>
<td>DataKeyField</td>
<td>Gets or sets the key field in the bound data source.</td>
</tr>
<tr>
<td>DataKeys</td>
<td>Gets a collection that stores the key values of all records displayed as a row in the grid. The column used as the key is defined by the DataKeyField property.</td>
</tr>
<tr>
<td>DataSource</td>
<td>Gets or sets the data source object that contains the values to populate the control.</td>
</tr>
<tr>
<td>DataSourceID</td>
<td>Indicates the data source object to populate the control. Not available in ASP.NET 1.x.</td>
</tr>
<tr>
<td>EditItemIndex</td>
<td>Gets or sets the index of the grid’s item to edit.</td>
</tr>
<tr>
<td>EditItemStyle</td>
<td>Gets the style properties for the item being edited.</td>
</tr>
<tr>
<td>FooterStyle</td>
<td>Gets the style properties for the footer section of the grid.</td>
</tr>
<tr>
<td>GridLines</td>
<td>Gets or sets whether all cells must have the border drawn.</td>
</tr>
<tr>
<td>HeaderStyle</td>
<td>Gets the style properties for the heading section of the grid.</td>
</tr>
<tr>
<td>HorizontalAlign</td>
<td>Gets or sets the horizontal alignment of the text in the grid.</td>
</tr>
</tbody>
</table>
Property Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Items</td>
<td>Gets the collection of the currently displayed items.</td>
</tr>
<tr>
<td>ItemStyle</td>
<td>Gets the style properties for the items in the grid.</td>
</tr>
<tr>
<td>PageCount</td>
<td>Gets the number of pages required to display all bound items.</td>
</tr>
<tr>
<td>PagerStyle</td>
<td>Gets the style properties for the paging section of the grid.</td>
</tr>
<tr>
<td>PageSize</td>
<td>Gets or sets the number of items to display on a single page.</td>
</tr>
<tr>
<td>SelectedIndex</td>
<td>Gets or sets the index of the currently selected item.</td>
</tr>
<tr>
<td>SelectedItem</td>
<td>Gets a DataGridItem object representing the selected item.</td>
</tr>
<tr>
<td>SelectedItemStyle</td>
<td>Gets the style properties for the currently selected item.</td>
</tr>
<tr>
<td>ShowFooter</td>
<td>Indicates whether the footer is displayed. False by default.</td>
</tr>
<tr>
<td>ShowHeader</td>
<td>Indicates the header is displayed. True by default.</td>
</tr>
<tr>
<td>UseAccessibleHeader</td>
<td>Indicates whether the control’s header is rendered in an accessible format—that is, using &lt;th&gt; tags instead of &lt;td&gt;. Not available in ASP.NET 1.x</td>
</tr>
<tr>
<td>VirtualItemCount</td>
<td>Gets or sets the virtual number of items in the DataGrid control when custom paging is used.</td>
</tr>
</tbody>
</table>

The characteristic traits of the DataGrid control are the Columns and Items collections, the style and data-binding properties. All columns in the grid are represented by an object with its own set of properties and methods. Several types of columns are available to implement the most common tasks. In general, not all rows in the bound data source are included in the HTML code for the client. The Items collection returns a collection of DataGridItem objects, one per each displayed row. The DataGridItem class is a specialized version of the TableRow class.

Note

In ASP.NET 2.0, a bunch of new properties make their debut to improve the usability of the control, especially for users with accessibility problems. Caption, CaptionAlign, and UseAccessibleHeader let you tweak the markup the control generates to make it easier for users of Assistive Technology devices to work with.

Constituent Elements of a DataGrid

The output of a DataGrid control is made of several constituent elements grouped in the ListItemType enumeration. Each element plays a clear role and has a precise location in the user interface of the control, as Figure 11-1 shows.
The **DataGrid** user interface comprises the logical elements listed in Table 11-2. Each element has its own style property—that is, the set of graphical settings that are automatically applied by the control.

### TABLE 11-2 Graphical Elements that Form a Data Grid

<table>
<thead>
<tr>
<th>Item Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlternatingItem</td>
<td>Represents a data-bound row placed in an odd position. Useful if you want to use different styles for alternating rows. AlternatingItemStyle is the property that lets you control the look and feel of the element.</td>
</tr>
<tr>
<td>EditItem</td>
<td>Represents the item currently displayed in edit mode. EditItemStyle lets you control the look and feel of the element.</td>
</tr>
<tr>
<td>Footer</td>
<td>Represents the grid’s footer. The element can’t be bound to a data source and is styled using the settings in the FooterStyle property.</td>
</tr>
<tr>
<td>Header</td>
<td>Represents the grid’s header. The element can’t be bound to a data source and is styled using the settings in the HeaderStyle property.</td>
</tr>
<tr>
<td>Item</td>
<td>Represents a data-bound row placed in an even position. Styled through the ItemStyle property.</td>
</tr>
<tr>
<td>Pager</td>
<td>Represents the pager element you use to scroll between pages. The element can’t be bound to a data source and is styled using the settings in the PagerStyle property. The pager can be placed at the top or bottom of the grid’s table and even in both places.</td>
</tr>
<tr>
<td>SelectedItem</td>
<td>Represents the item, or alternating item, currently selected. The property that defines its look and feel is SelectedItemStyle.</td>
</tr>
</tbody>
</table>

Each time one of the constituent elements is about to be created, the grid fires an ItemCreated event for you to perform some application-specific tasks. We’ll return to grid events in a moment.
Data Source Rows and Displayed Rows

By design, the DataGrid control displays the data stored in a data source object—be it an enumerable data object or a data source control. Each row in the bound data source is potentially a row in the grid. However, this one-to-one mapping doesn’t always correspond to reality. Each displayed grid row finds a place in the Items collection. Each element in the Items collection is an instance of the DataGridItem class—a slightly richer table row object—and supplies a DataItem property set to the object that corresponds to the row in the bound data source. Note that only bindable items are contained in the Items collection. The header, footer, and pager are not included in the collection.

The index properties of the DataGrid refer to the rows displayed rather than to the underlying data source. When the item with an index of 1 is selected, the second displayed item is selected, but this piece of information says nothing about the position of the corresponding source record. The data source index for the item object is stored in the DataSetIndex property on the DataGridItem class. DataSetIndex returns the absolute position in the overall data source of the record represented by the current item. Although functional, this method isn’t especially handy in some common scenarios, such as when you want to select a row and retrieve a bunch of associated records. In such a case, you need to know the value of the key field in the underlying data source row.

The DataKeys collection and the DataKeyField property provide an effective shortcut designed specifically to work in similar situations. When you configure a DataGrid, you can store the name of a key field in the DataKeyField property. During the data-binding phase, the control extracts from the data source the values for the specified key field that correspond to the rows being displayed. As a result, the index of the selected row in the Items collection can be used with DataKeys to get the key value for the underlying data source row. Let’s consider the following declaration, which refers to a grid that displays information about the employees of a company:

```csharp
<asp:DataGrid runat="server" id="grid"
    DataKeyField="employeeid" ...>

    // empID is the key of the currently selected item
    int empID = grid.DataKeys[grid.SelectedIndex];

    The DataKeys collection is automatically filled by the control based on the value of the DataKeyField property and the bound data source.
```
Events of the DataGrid Control

The DataGrid control has no specific methods worth mentioning. Table 11-3 lists the events that the control fires during its life cycle.

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CancelCommand</td>
<td>The user clicked to cancel any updates made on the current item being edited.</td>
</tr>
<tr>
<td>DeleteCommand</td>
<td>The user clicked to start a delete operation on the current item.</td>
</tr>
<tr>
<td>EditCommand</td>
<td>The user clicked to put the current item in edit mode.</td>
</tr>
<tr>
<td>ItemCommand</td>
<td>The user clicked a command button within the grid control.</td>
</tr>
<tr>
<td>ItemCreated</td>
<td>This event occurs after a new grid item is created.</td>
</tr>
<tr>
<td>ItemDataBound</td>
<td>This event occurs after a grid item is bound to data.</td>
</tr>
<tr>
<td>PageIndexChanged</td>
<td>The user clicked to see a new page of data.</td>
</tr>
<tr>
<td>SelectedIndexChanged</td>
<td>The user clicked to select a different item.</td>
</tr>
<tr>
<td>SortCommand</td>
<td>The user clicked to start a sort operation on a column.</td>
</tr>
<tr>
<td>UpdateCommand</td>
<td>The user clicked to save any updates made on the current item being edited.</td>
</tr>
</tbody>
</table>

The CancelCommand and UpdateCommand events are fired under special circumstances—that is, when an item is being edited. (We'll cover the DataGrid in-place editing feature later in the chapter.) The CancelCommand event signals that the user clicked the Cancel button to cancel all pending changes. The UpdateCommand event denotes the user’s intention to persist all the changes. The other command events—EditCommand, DeleteCommand, and SortCommand—indicate that the user required a particular action by clicking on command buttons within the user interface of the grid.

In addition to the events just listed, the DataGrid control fires all the standard events of Web controls, including Load, Init, PreRender, and DataBinding. In particular, you might want to write a handler for PreRender if you need to modify the HTML code generated for the grid. The DataBinding event, on the other hand, is the entry point in the grid’s binding process. The event is fired as the first step before the whole binding process begins regardless of the type of object bound—be it an enumerable object or a data source control.

Note These command events mark a key difference between the DataGrid and the newer GridView control. While the DataGrid is limited to firing an event to let the page know the user’s intention, the GridView proactively handles the event by executing the configured command through the bound data source control. The DataGrid supports data source controls too, but the support is limited to showing read-only data.
Part II Adding Data in an ASP.NET Site

Binding Data to the Grid

A DataGrid control is formed by data-bindable columns. By default, the control includes all the data source columns in the view. You can change this behavior by setting the AutoGenerateColumns property to false. In this case, only the columns explicitly listed in the Columns collection are displayed. The DataGrid control supports a variety of column types, which mostly differ from one another in how each represents the data. You are required to indicate the type of the column if you add it to the Columns collection; otherwise, if automatic generation is used, all columns are of the simplest type—the BoundColumn column type. Table 11-4 details the various types of columns supported.

<table>
<thead>
<tr>
<th>Column Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoundColumn</td>
<td>The contents of the column are bound to a field in a data source. Each cell displays as plain text.</td>
</tr>
<tr>
<td>ButtonColumn</td>
<td>Displays a button for each item in the column. The text of the button can be data-bound, and buttons have a common command name.</td>
</tr>
<tr>
<td>EditCommandColumn</td>
<td>Particular type of button column associated with a command named Edit. When in edit mode, the whole row is drawn using text boxes rather than literals.</td>
</tr>
<tr>
<td>HyperLinkColumn</td>
<td>Displays the contents of each item in the column as a hyperlink. The text of the hyperlink can be bound to a column in the data source or it can be static text. The target URL can be data-bound too. Clicking a hyperlink column causes the browser to jump to the specified URL.</td>
</tr>
<tr>
<td>TemplateColumn</td>
<td>This type displays each cell of the column following a specified ASP.NET template. It also allows you to provide custom behaviors.</td>
</tr>
</tbody>
</table>

Note that the AutoGenerateColumns property and the Columns collection are not mutually exclusive. If both properties are set to true and the collection is not empty, the grid will show the user-defined columns followed by all the ones that auto-generation would produce.

You normally bind columns using the <columns> tag in the body of the <asp:datagrid> server control, as the following code demonstrates:

```xml
<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>
```
Alternately, you can create a new column of the desired class, fill its member properly, and then add the class instance to the Columns collection. Here is some code to add a `BoundColumn` object to a grid:

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

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

**Note** The Columns collection doesn't persist its contents to the view state, and it is empty whenever the page posts back. To preserve any dynamically added column, you need to re-add it on each and every postback.

### Data-Bound Columns

All grid column types inherit from the `DataGridColumn` class and have a few common properties, such as the header text, footer and item style, and visibility flag. Table 11-5 details the properties shared by all types of columns.

**Table 11-5 Common Properties for All Column Types**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FooterStyle</td>
<td>Gets the style properties for the footer of the column</td>
</tr>
<tr>
<td>FooterText</td>
<td>Gets or sets the static text displayed in the footer of the column</td>
</tr>
<tr>
<td>HeaderImageUrl</td>
<td>Gets or sets the URL of an image to display in the header</td>
</tr>
<tr>
<td>HeaderStyle</td>
<td>Gets the style properties for the header of the column</td>
</tr>
<tr>
<td>HeaderText</td>
<td>Gets or sets the static text displayed in the header of the column</td>
</tr>
<tr>
<td>ItemStyle</td>
<td>Gets the style properties for the item cells of the column</td>
</tr>
<tr>
<td>SortExpression</td>
<td>Gets or sets the expression to sort the data in the column</td>
</tr>
<tr>
<td>Visible</td>
<td>Gets or sets whether the column is visible</td>
</tr>
</tbody>
</table>

The `BoundColumn` class represents a column type that is bound to a data field. The key properties to set up a grid column are `DataField`, which represents the name of the column to bind, and `DataFormatString`, which allows you to format the displayed text to some extent. The `ReadOnly` property has an effect only if an edit command column is added to the grid. In this case, the cells in the column are switched to edit mode according to the value of the property.
Part II Adding Data in an ASP.NET Site

The following code snippet inserts two columns and specifies for each the header text and the source column. In addition, the second column is given a format string to make it look like a currency value with right alignment.

```html
<asp:boundcolumn runat="server" datafield="quantityperunit"
    headertext="Packaging"/>
<asp:boundcolumn runat="server" datafield="unitprice"
    headertext="Price" dataformatstring="{0:c}"/>
    <itemstyle width="80px" horizontalalign="right" />
</asp:boundcolumn>
```

Graphical settings for a column must be specified using a child style element.

HyperLink Columns

The `HyperLinkColumn` class is a column type that contains a hyperlink for each cell. The programmer can control the text of the hyperlink and the URL to navigate. Both fields can be bound to a column in the data source. The `DataTextField` takes the name of the field to use for the text of the hyperlink. `DataNavigateUrlField`, on the other hand, accepts the field that contains the URL. Another property, named `DataNavigateUrlFormatString`, defines the format of the final URL to use. By combining the two properties, you can redirect users to the same page, passing row-specific information on the query string, as shown in the following code:

```html
<asp:hyperlinkcolumn runat="server" datatextfield="productname"
    headertext="Product"
    datanavigateurlfield="productid"
    datanavigateurfformatstring="productinfo.aspx?id={0}" target="ProductView">
    <itemstyle width="200px" />
</asp:hyperlinkcolumn>
```

The hyperlinks will point to the same page—`productinfo.aspx`—each with the product ID associated with the corresponding row of the bound data. The column class is responsible for building the real URL correctly.

**Note** By using the `DataNavigateUrlField` and `DataNavigateUrlFormatString` property together, you can make the URL of the hyperlink parametric. However, by default you are limited to just one parameter—the value of the field bound through the `DataNavigateUrlField` property. To use a hyperlink bound to any number of arguments, you should resort to templated columns or use a `GridView`.

Button Columns

The `ButtonColumn` class represents a command column and contains a user-defined button for each cell in the column. Functionally similar to hyperlink columns, button columns are different because they generate a postback event on the same URL. Although the caption of

---

Chapter 11. Creating Bindable Grids of Data
Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.
Each button can be bound to a data-source column, more often than not a button column has static text displayed through all the cells.

The key idea behind the button column is that you execute a particular action after the user clicks on a row. All buttons in the column are associated with some script code that posts the page back and executes the `ItemCommand` server-side procedure. Within that procedure, you use the command name (CommandName property) to distinguish between multiple button columns and you use the `ItemIndex` property of the `DataGridItem` class to know about the particular row that was clicked. A reference to a `DataGridItem` object is passed through the `ItemClick` event.

A special type of button column is the select column. It is a normal button column with the command name of select. When you click on such a column, the `DataGrid` automatically redraws the selected row using a different class of settings—those determined by the `SelectedItemStyle` property. There is no need for you to write an `ItemCommand` handler; the described behavior is built in.

```html
<asp:ButtonColumn runat="server" text="Select" CommandName="Select" />
```

The style of the selected row—with, at most, one row selected at a time—is set using the `SelectedItemStyle` property. It can be as easy as the following code:

```html
<SelectedItemStyle backcolor="cyan" />
```

The change of the selected item is signaled with the `SelectedIndexChanged` event. However, before this event is fired, the application can handle the related `ItemClick` event. When `SelectedIndexChanged` reaches the application, the `SelectedIndex` property indicates the new selected index.

### Templated Columns

Templated columns allow you to create combinations of HTML text and server controls to design a custom layout for any cells in the column. The controls within a templated column can be bound to any combination of fields in the data source. In particular, you can group more fields in a single expression and even embellish it with HTML attributes such as boldface or italic style. Templates are column-specific and cannot be applied to auto-generated columns. If you want more columns to share the same template, you can duplicate the code only in the ASP.NET page for each column.

A templated column is recognized by the `<TemplateColumn>` tag and rendered by the `TemplateColumn` class. The body of the tag can contain up to four different templates: `ItemTemplate`, `EditItemTemplate`, `HeaderTemplate`, and `FooterTemplate`. Just as with any other column type, a templated column can have header text and a sort expression. Templated columns, though, do not have an explicit data source field to bind. To bind a templated column to one or more data fields, you use a data-binding expression. (See Chapter 10.)
you use the `Eval` method to evaluate data-bound expressions at run time and return the value properly cast. For example, the following code snippet shows a templated column that mimics the behavior of a `BoundColumn` object associated with the lastname column:

```xml
<asp:templatecolumn runat="server" headertext="Last Name">
    <itemtemplate>
        <asp:label runat="server" Text='@DataBinder.Eval(Container.DataItem, "lastname")'>
        </itemtemplate>
    </asp:templatecolumn>
```

By using `DataBinder.Eval` (or simply `Eval` in ASP.NET 2.0 and beyond), you can access any number of fields in the currently bound data source. In addition, you can combine them in any order to obtain any sort of expression, which is otherwise impossible using a simpler bound or button column.

**Working with the `DataGrid`**

The `DataGrid` control is not simply a tool to display static data; it also provides advanced functionalities to page, sort, and edit bound data. The interaction that is established between a `DataGrid` and the host page is limited to exchanging notifications in the form of postback events. The `DataGrid` lets the page know that something happened and leaves the page free to react as appropriate. This pattern is common to most supported operations, with the notable exception of item selection. As mentioned, in fact, if you add a Select button column to the grid and define a proper style for selected items, clicking on a select button makes the page post back and forces the `DataGrid` to change the appearance of the corresponding row.

Other operations for which the `DataGrid` simply fires an event to the page are paging, sorting, and in-place editing.

As you can see, these are relatively common operations that plenty of pages need to accomplish. If you choose to use a `DataGrid` control, be ready to write much more boilerplate code than you would with the newer `GridView` control.

**Paging Through the Data Source**

In real-world scenarios, the size of a data source easily exceeds the real estate of the page. Data paging is the contrivance that many applications adopt to both gain in scalability and present a more helpful page to the user. Especially on the Web, displaying only a few rows at a time is a more effective approach than downloading hundreds of records that stay hidden most of the time. The `DataGrid` control provides some built-in facilities to let the programmer easily switch to a new page according to the user’s clicking.

The control needs to know how many items should be displayed per page, what type of functionality is required for the pager, and the data source to page through. In return for this, the control tracks the current page index, extracts the rows that fit into the particular
page, and refreshes the user interface. Whenever the page index changes, an event is fired to the application—the `PageIndexChanged` event.

Note, however, that the host page is still responsible for ensuring that all the rows that fit into the new page are bound to the control. This holds true even if the `DataGrid` is bound to a data source control or a classic enumerable object. With a `DataGrid`, a handler for the `PageIndexChanged` event is always required. What you do in the handler might be different, though, depending on the actual data source. Here's the code you need to use if the `DataGrid` is bound to a data source control:

```csharp
protected void grid_PageIndexChanged(object sender, DataGridPageChangedEventArgs e)
{
    grid.CurrentPageIndex = e.NewPageIndex;
    // Must be repeated to force a refresh
    grid.DataSourceID = "SqlDataSource1";
}
```

Note that you still need to reassign `DataSourceID` to trigger an internal data source changed event and cause the control to load its new data set. If the grid is bound to an enumerable object, you simply assign a new bunch of rows to the `DataSource` property.

Overall, paging is a tough feature and a potential scalability killer. If you leave grid controls in charge of handling paging more or less automatically, caching data is a must. A data source control makes it as easy as turning on the `EnableCaching` property, as you saw in Chapter 10.

Caching a lot of data, though, might pose a serious problem, especially if you have to do that for each user.

`DataGrid` controls also support custom paging, an alternative and cost-effective approach to paging that binds to the control only the records that fit in the current page:

```csharp
protected void grid_PageIndexChanged(object sender, DataGridPageChangedEventArgs e)
{
    grid.CurrentPageIndex = e.NewPageIndex;
    grid.DataSource = GetRecordsInPage(grid.CurrentPageIndex);
}
```

```csharp
protected object GetRecordsInPage(int pageIndex)
{
    // Retrieve and return data that fits in the given page
    ...
}
```

As we'll see later, the `GridView` doesn't explicitly support custom paging. On the other hand, the `GridView` doesn't prevent server paging from working if it is supported by the underlying data source control or the data access layer (DAL).
Part II  Adding Data in an ASP.NET Site

Sorting Columns of Data

The AllowSorting property enables sorting on all or some of the DataGrid’s displayed columns. Just as for paging, clicking to sort data by a column doesn’t really produce any visible effect unless you add a handler for the SortCommand event. Here’s a simple handler you can use if the DataGrid is bound to a data source control:

```csharp
protected void grid_SortCommand(
    object sender,
    DataGridSortCommandEventArgs e)
{
    SqlDataSource1.SelectCommand += “ ORDER BY ” + e.SortExpression;
    grid.DataSourceID = “SqlDataSource1”;
}
```

Sorting is a potentially slow operation to accomplish and can have significant repercussions on scalability. For this reason, it is important to understand how it really works in the context of grids. In ASP.NET 1.x, you can employ in the SortCommand event handler only your own logic to sort. You can sort in memory using the Sort method of the DataView object (which is a very slow process, indeed); you can rely on the database sort capabilities (which is typically the fastest approach to sort data, but communication latency and network bandwidth might slow things down from the user’s perspective); and sometimes you can also maintain presorted caches of data. Whatever approach you choose, you need to know what you’re doing.

In ASP.NET 2.0 and beyond, data source controls tend to hide some details. If the data source control is configured to retrieve data via a DataSet (the default setting), sorting happens in memory via the Sort method. This approach is not really efficient, and it should be avoided unless you have only a few records to sort. If the data source control works via data readers and stored procedures, sorting can take place on the server and data will be returned in the correct order. In the end, sorting is a delicate operation no matter which controls you use. Only careful benchmarks and an application-specific combination of tools and options can deliver the perfect result. To get this, you need to understand how controls work internally.

Editing Existing Rows

A DataGrid control displays mostly read-only data. If editing is needed, you select the row to update and post a request for editing. The new page contains an edit form with input fields and links to persist or reject the changes. This pattern is probably the most effective one for editing data over the Web, and it’s certainly the pattern that provides the highest level of flexibility. With DataGrid controls, though, a simpler model of data editing is possible. The new model is known as in-place editing and mimics the behavior of a Microsoft Office Excel worksheet. When you trigger the event that begins the editing phase, the visible part of the grid is redrawn and—like cells in Excel—the row selected for editing is rendered in a different way, using text-box controls instead of literals and labels. At the same time, the DataGrid control completes its own user interface with a couple of button links to allow you to commit or roll back changes.
In-place editing does not require much work to be completely set up, but at the same time it is not appropriate for all types of applications and not functional in all operating contexts. All in all, if you have to edit the content of single and relatively small tables that have no special validation or business logic to apply, in-place editing is extremely handy and powerful.

The key object for in-place editing is the `EditCommandColumn` class. The column adds a link button to all rows of the grid. When the link is clicked, the page posts back and the cells of the row are drawn in edit mode. How a column behaves in edit mode depends on the column type. For example, button and hyperlink columns are completely ignored in edit mode. Bound and templated columns, on the other hand, change their rendering when the row is being edited. In particular, bound columns are rendered using text boxes in place of literals, whereas templated columns display the contents of the `<EditItemTemplate>` section, if any.

As with paging and sorting, code is required to have the `DataGrid` complete an in-place editing operation too. You typically need to write three event handlers—`EditCommand`, to put the grid in edit mode; `CancelCommand`, to put the grid back in read-only mode; and `UpdateCommand`, to persist changes and refresh the grid. Handlers for `EditCommand` and `CancelCommand` are relatively simple and standard. Writing a handler for `UpdateCommand` might not be that easy, though.

Basically, the `UpdateCommand` handler must accomplish two key operations—retrieving input data and persisting changes. Both operations are hard-coded in the `GridView`, performed in collaboration with the underlying data source control, and mostly configured at design-time by the page author.

---

**Important** Admittedly, this section about `DataGrid` controls didn’t get into the nitty-gritty details of how the control works and deliberately avoided describing how to implement paging, sorting, and editing properly in real-world scenarios. The reason for this approach lies in the structural difference that exists between `DataGrid` and `GridView` controls. To a large extent, the two controls provide the same set of abstract features—grid-like display, paging, sorting, editing, and templates. How each control implements individual features and binds to data is radically different. In one word, the philosophy behind each control is different. Now, the `GridView` control is newer, richer, and smarter, and it would probably have been the only grid control starting with ASP.NET 2.0 if it weren’t for compatibility issues.

If you have an existing ASP.NET application to maintain, and you don’t feel like leaping to `GridView`, you already know all the details and techniques omitted here. If you’re building a new application and want to take advantage of grids, you don’t need to know about `DataGrid` controls and are better off focusing entirely on `GridView` controls. The purpose of this section is to help people in the middle make a decision about which control to use while explaining why Microsoft decided to go with a new control that is designed to complement the changes in the data-binding model we explored in Chapter 10. The `GridView` control is also complemented by other view controls—specifically, `FormView`, `DetailsView`, and `ListView`—that we’ll cover in the upcoming chapters.
The GridView Control

The GridView is the successor to the ASP.NET 1.x DataGrid control. It provides the same base set of capabilities, plus a long list of extensions and improvements. As mentioned, the DataGrid—which is still fully supported in ASP.NET 2.0—is an extremely powerful and versatile control. However, it has one big drawback: it requires you to write a lot of custom code, even to handle relatively simple and common operations such as paging, sorting, editing, or deleting data. The GridView control was designed to work around this limitation and make two-way data binding happen with as little code as possible. The control is tightly coupled to the family of new data source controls, and it can handle direct data source updates as long as the underlying data source object supports these capabilities.

This virtually codeless two-way data binding is by far the most notable feature of the new GridView control, but other enhancements are numerous. The control is an improvement over the DataGrid control because it has the ability to define multiple primary key fields, new column types, and style and templating options. The GridView also has an extended eventing model that allows you to handle or cancel events.

The GridView Object Model

The GridView control provides a tabular grid-like view of the contents of a data source. Each column represents a data source field, and each row represents a record. The class is declared as follows:

```csharp
public class GridView : CompositeDataBoundControl, ICallbackContainer, ICallbackEventHandler
```

The base class ensures data-binding and naming-container support. The ICallbackContainer and ICallbackEventHandler interfaces provide more effective paging and sorting than is now supported. It does this through client-side, out-of-band calls that use the new script callback technology. (I'll talk more about this later, even though with the advent of AJAX it has lost most, if not all, of its original appeal.) Let's begin our tour of the GridView control by looking at the control's programming interface.

Properties of the GridView Control

The GridView supports a large set of properties that fall into the following broad categories: behavior, visual settings, style, state, and templates. Table 11-6 details the properties that affect the behavior of the GridView.
### TABLE 11-6 Behavior Properties of the GridView Control

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllowPaging</td>
<td>Indicates whether the control supports paging.</td>
</tr>
<tr>
<td>AllowSorting</td>
<td>Indicates whether the control supports sorting.</td>
</tr>
<tr>
<td>AutoGenerateColumns</td>
<td>Indicates whether columns are automatically created for each field in the data source. The default is true.</td>
</tr>
<tr>
<td>AutoGenerateDeleteButton</td>
<td>Indicates whether the control includes a button column to let users delete the record that is mapped to the clicked row.</td>
</tr>
<tr>
<td>AutoGenerateEditButton</td>
<td>Indicates whether the control includes a button column to let users edit the record that is mapped to the clicked row.</td>
</tr>
<tr>
<td>AutoGenerateSelectButton</td>
<td>Indicates whether the control includes a button column to let users select the record that is mapped to the clicked row.</td>
</tr>
<tr>
<td>DataMember</td>
<td>Indicates the specific table in a multimember data source to bind to the grid. The property works in conjunction with DataSource. If DataSource is a DataSet object, it contains the name of the particular table to bind.</td>
</tr>
<tr>
<td>DataSource</td>
<td>Gets or sets the data source object that contains the values to populate the control.</td>
</tr>
<tr>
<td>DataSourceID</td>
<td>Indicates the bound data source control</td>
</tr>
<tr>
<td>EnableSortingAndPagingCallbacks</td>
<td>Indicates whether sorting and paging are accomplished using script callback functions. Disabled by default.</td>
</tr>
<tr>
<td>RowHeaderColumn</td>
<td>Name of the column to use as the column header. This property is designed for improving accessibility.</td>
</tr>
<tr>
<td>SortDirection</td>
<td>Gets the direction of the column's current sort.</td>
</tr>
<tr>
<td>SortExpression</td>
<td>Gets the current sort expression.</td>
</tr>
<tr>
<td>UseAccessibleHeader</td>
<td>Specifies whether to render <code>&lt;th&gt;</code> tags for the column headers instead of default <code>&lt;td&gt;</code> tags.</td>
</tr>
</tbody>
</table>

The **SortDirection** and **SortExpression** properties specify the direction and the sort expression on the column that currently determine the order of the rows. Both properties are set by the control’s built-in sorting mechanism when users click a column’s header. The whole sorting engine is enabled and disabled through the **AllowSorting** property. The **EnableSortingAndPagingCallbacks** property toggles on and off the control’s capability of using script callbacks to page and sort without doing roundtrips to the server and changing the entire page.
Each row displayed within a GridView control corresponds to a special type of grid item. The list of predefined types of items is nearly identical to that of the DataGrid, and it includes items such as the header, rows and alternating rows, footer, and pager. These items are static in the sense that they remain in place for the lifetime of the control in the application. Other types of items are active for a short period of time—the time needed to accomplish a certain operation. Dynamic items are the edit row, selected row, and EmptyData item. EmptyData identifies the body of the grid when the grid is bound to an empty data source.

Note The GridView control provides a few properties specifically designed for accessibility. They are UseAccessibleHeader, Caption, CaptionAlign, and RowHeaderColumn. When you set RowHeaderColumn, all the column cells will be rendered with the default header style (boldface type). However, ShowHeader, HeaderStyle, and other header-related properties don’t affect the column indicated by RowHeaderColumn.

Table 11-7 details the style properties available on the GridView control.

### Table 11-7 Style Properties of the GridView Control

<table>
<thead>
<tr>
<th>Style</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlternatingRowStyle</td>
<td>Defines the style properties for every other row in the table</td>
</tr>
<tr>
<td>EditRowStyle</td>
<td>Defines the style properties for the row being edited</td>
</tr>
<tr>
<td>FooterStyle</td>
<td>Defines the style properties for the grid’s footer</td>
</tr>
<tr>
<td>HeaderStyle</td>
<td>Defines the style properties for the grid’s header</td>
</tr>
<tr>
<td>EmptyDataRowStyle</td>
<td>Defines the style properties for the empty row, which is rendered when the</td>
</tr>
<tr>
<td></td>
<td>GridView is bound to empty data sources</td>
</tr>
<tr>
<td>PagerStyle</td>
<td>Defines the style properties for the grid’s pager</td>
</tr>
<tr>
<td>RowStyle</td>
<td>Defines the style properties for the rows in the table</td>
</tr>
<tr>
<td>SelectedRowStyle</td>
<td>Defines the style properties for the currently selected row</td>
</tr>
</tbody>
</table>

Table 11-8 lists most of the properties that affect the appearance of the control, and Table 11-9 details the templating properties.
### TABLE 11-8 Appearance Properties of the GridView Control

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackImageUrl</td>
<td>Indicates the URL to an image to display in the background</td>
</tr>
<tr>
<td>Caption</td>
<td>The text to render in the control’s caption</td>
</tr>
<tr>
<td>CaptionAlign</td>
<td>Alignment of the caption text</td>
</tr>
<tr>
<td>CellPadding</td>
<td>Indicates the amount of space (in pixels) between the contents of a cell and the border</td>
</tr>
<tr>
<td>CellSpacing</td>
<td>Indicates the amount of space (in pixels) between cells</td>
</tr>
<tr>
<td>EmptyDataText</td>
<td>Indicates the text to render in the control when it is bound to an empty data source</td>
</tr>
<tr>
<td>GridLines</td>
<td>Indicates the gridline style for the control</td>
</tr>
<tr>
<td>HorizontalAlign</td>
<td>Indicates the horizontal alignment of the control on the page</td>
</tr>
<tr>
<td>PagerSettings</td>
<td>References an object that lets you set the properties of the pager buttons</td>
</tr>
<tr>
<td>ShowFooter</td>
<td>Indicates whether the footer row is displayed</td>
</tr>
<tr>
<td>ShowHeader</td>
<td>Indicates whether the header row is displayed</td>
</tr>
</tbody>
</table>

The `PagerSettings` object groups together all the visual properties you can set on the pager. Many of these properties should sound familiar to `DataGrid` programmers. The `PagerSettings` class also adds some new properties to accommodate new predefined buttons (first and last pages), and it uses images instead of text in the links. (You need to figure out a trick to do the same with a `DataGrid`.)

### TABLE 11-9 Templating Properties of the GridView Control

<table>
<thead>
<tr>
<th>Template</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EmptyDataTemplate</td>
<td>Indicates the template content to be rendered when the control is bound to an empty source. This property takes precedence over EmptyDataText if both are set. If neither is set, the grid isn’t rendered if bound to an empty data source.</td>
</tr>
<tr>
<td>PagerTemplate</td>
<td>Indicates the template content to be rendered for the pager. This property overrides any settings you might have made through the <code>PagerSettings</code> property.</td>
</tr>
</tbody>
</table>

The final block of properties—the state properties—is shown in Table 11-10. State properties return information about the internal state of the control.
### TABLE 11-10 State Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bottom Pager Row</td>
<td>Returns a GridViewRow object that represents the bottom pager of the grid.</td>
</tr>
<tr>
<td>Columns</td>
<td>Gets a collection of objects that represent the columns in the grid. The</td>
</tr>
<tr>
<td></td>
<td>collection is always empty if columns are auto-generated.</td>
</tr>
<tr>
<td>Data Key Names</td>
<td>Gets an array that contains the names of the primary key fields for the</td>
</tr>
<tr>
<td></td>
<td>currently displayed items.</td>
</tr>
<tr>
<td>Data Keys</td>
<td>Gets a collection of DataKey objects that represent the values of the</td>
</tr>
<tr>
<td></td>
<td>primary key fields set in Data Key Names for the currently displayed records.</td>
</tr>
<tr>
<td>Edit Index</td>
<td>Gets and sets the 0-based index that identifies the row currently rendered</td>
</tr>
<tr>
<td></td>
<td>in edit mode.</td>
</tr>
<tr>
<td>Footer Row</td>
<td>Returns a GridViewRow object that represents the footer.</td>
</tr>
<tr>
<td>Header Row</td>
<td>Returns a GridViewRow object that represents the header.</td>
</tr>
<tr>
<td>Page Count</td>
<td>Gets the number of pages required to display the records of the data source.</td>
</tr>
<tr>
<td>Page Index</td>
<td>Gets and sets the 0-based index that identifies the currently displayed</td>
</tr>
<tr>
<td></td>
<td>page of data.</td>
</tr>
<tr>
<td>Page Size</td>
<td>Indicates the number of records to display on a page.</td>
</tr>
<tr>
<td>Rows</td>
<td>Gets a collection of GridViewRow objects that represent the data rows</td>
</tr>
<tr>
<td></td>
<td>currently displayed in the control.</td>
</tr>
<tr>
<td>Selected Data Key</td>
<td>Returns the DataKey object for the currently selected record.</td>
</tr>
<tr>
<td>Selected Index</td>
<td>Gets and sets the 0-based index that identifies the row currently selected.</td>
</tr>
<tr>
<td>Selected Row</td>
<td>Returns a GridViewRow object that represents the currently selected row.</td>
</tr>
<tr>
<td>Selected Value</td>
<td>Returns the explicit value of the key as stored in the DataKey object.</td>
</tr>
<tr>
<td></td>
<td>Similar to Selected Data Key.</td>
</tr>
<tr>
<td>Top Pager Row</td>
<td>Returns a GridViewRow object that represents the top pager of the grid.</td>
</tr>
</tbody>
</table>

The GridView is designed to leverage the new data source object model, and it works best when bound to a data source control via the DataSourceID property. The GridView also supports the classic DataSource property, but if you bind data in that way, some of the features (such as built-in updates and paging) become unavailable.

### Events of the GridView Control

The GridView control doesn’t have methods other than DataBind. As mentioned, though, in many situations you don’t need to call methods on the GridView control. The data-binding process is started implicitly when you bind the GridView to a data source control.
Starting with ASP.NET 2.0, many controls, and the `Page` class itself, feature pairs of events of the type `doing/done`. Key operations in the control life cycle are wrapped by a pair of events—one firing before the operation takes place, and one firing immediately after the operation is completed. The `GridView` class is no exception. The list of events is shown in Table 11-11.

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PageIndexChanging</code>,</td>
<td>Both events occur when one of the pager buttons is clicked. They fire before and after the grid control handles the paging operation, respectively.</td>
</tr>
<tr>
<td><code>PageIndexChanged</code></td>
<td></td>
</tr>
<tr>
<td><code>RowCancelingEdit</code></td>
<td>Occurs when the Cancel button of a row in edit mode is clicked, but before the row exits edit mode.</td>
</tr>
<tr>
<td><code>RowCommand</code></td>
<td>Occurs when a button is clicked.</td>
</tr>
<tr>
<td><code>RowCreated</code></td>
<td>Occurs when a row is created.</td>
</tr>
<tr>
<td><code>RowDataBound</code></td>
<td>Occurs when a data row is bound to data.</td>
</tr>
<tr>
<td><code>RowDeleting</code>, <code>RowDeleted</code></td>
<td>Both events occur when a row’s Delete button is clicked. They fire before and after the grid control deletes the row, respectively.</td>
</tr>
<tr>
<td><code>RowEditing</code></td>
<td>Occurs when a row’s Edit button is clicked but before the control enters edit mode.</td>
</tr>
<tr>
<td><code>RowUpdating</code>, <code>RowUpdated</code></td>
<td>Both events occur when a row’s Update button is clicked. They fire before and after the grid control updates the row, respectively.</td>
</tr>
<tr>
<td><code>SelectedIndexChanging</code>,</td>
<td>Both events occur when a row’s Select button is clicked. The two events occur before and after the grid control handles the select operation, respectively.</td>
</tr>
<tr>
<td><code>SelectedIndexChanged</code>,</td>
<td></td>
</tr>
<tr>
<td><code>Sorting</code>, <code>Sorted</code></td>
<td>Both events occur when the hyperlink to sort a column is clicked. They fire before and after the grid control handles the sort operation, respectively.</td>
</tr>
</tbody>
</table>

`RowCreated` and `RowDataBound` events are the same as the `DataGrid`’s `ItemCreated` and `ItemDataBound` events, with new names. They behave exactly as they do in ASP.NET 1.x. The same is true of the `RowCommand` event, which is the same as the `DataGrid`’s `ItemCommand` event.

The availability of events that announce a certain operation significantly enhances your programming power. By hooking the `RowUpdating` event, you can cross-check what is being updated and validate the new values. Likewise, you might want to handle the `RowUpdating` event to HTML-encode the values supplied by the client before they are persisted to the underlying data store. This simple trick helps you to fend off script injections.
Simple Data Binding

The following code demonstrates the simplest way to bind data to a GridView control. The data source object keeps the page virtually code-free.

```xml
<asp:ObjectDataSource ID="MySource" runat="server"
    TypeName="Core35.DAL.Customers"
    SelectMethod="LoadAll">
</asp:ObjectDataSource>
<asp:GridView runat="server" id="grid" DataSourceID="MySource" />
```

Setting the DataSourceID property triggers the binding process, which runs the data source query and populates the user interface of the grid. You need not write any binding code. (Note that you still have to write the LoadAll method and the data access layer it calls.)

By default, the GridView control auto-generates enough columns to contain all the data coming through the data source. In other cases, you might want to control and style each column individually. For this to happen, the binding process should be refined a little bit.

Binding Data to a GridView Control

If no data source property is set, the GridView control doesn't render anything. If an empty data source object is bound and an EmptyDataTemplate template is specified, the results shown to the user have a more friendly look:

```xml
<asp:gridview runat="server" datasourceid="MySource">
    <emptydatatemplate>
        <asp:label runat="server">
            There's no data to show in this view.
        </asp:label>
    </emptydatatemplate>
</asp:gridview>
```

The EmptyDataTemplate property is ignored if the bound data source is not empty. Figure 11-2 shows the output generated by the empty template.

![The GridView in action on an empty data source.](image)
When you use a declared set of columns, the AutoGenerateColumns property of the grid is typically set to false. However, this is not a strict requirement—a grid can have declared and auto-generated columns. In this case, declared columns appear first. Note also that auto-generated columns are not added to the Columns collection. As a result, when column auto-generation is used, the Columns collection is typically empty.

Configuring Columns

The Columns property is a collection of DataControlField objects. The DataControlField object is akin to the DataGrid's DataGridColumn object, but it has a more general name because these field objects can be reused in other data-bound controls that do not necessarily render columns. (For example, in the DetailsView control, the same class is used to render a row.)

You can define your columns either declaratively or programmatically. In the latter case, you just instantiate any needed data field objects and add them to the Columns collection. The following code adds a data-bound column to the grid:

```csharp
BoundField field = new BoundField();
field.DataField = "companyname";
field.HeaderText = "Company Name";
grid.ColumnFields.Add(field);
```

Columns of data are displayed in the order that the column fields appear in the collection. To statically declare your columns in the .aspx source file, you use the <Columns> tag, as shown here:

```xml
<columns>
    <asp:boundfield datafield="customerid" headertext="ID" />
    <asp:boundfield datafield="companyname" headertext="Company Name" />
</columns>
```

Table 11-12 lists the column field classes that can be used in a GridView control. All the classes inherit DataControlField.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoundField</td>
<td>Default column type. Displays the value of a field as plain text.</td>
</tr>
<tr>
<td>ButtonField</td>
<td>Displays the value of a field as a command button. You can choose the link or the push button style.</td>
</tr>
<tr>
<td>CheckBoxField</td>
<td>Displays the value of a field as a check box. It is commonly used to render Boolean values.</td>
</tr>
<tr>
<td>CommandField</td>
<td>Enhanced version of ButtonField, represents a special command such as Select, Delete, Insert, or Update. It's rarely useful with GridView controls; the field is tailor-made for DetailsView controls. (GridView and DetailsView share the set of classes derived from DataControlField)</td>
</tr>
</tbody>
</table>
Part II Adding Data in an ASP.NET Site

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HyperLinkField</td>
<td>Displays the value of a field as a hyperlink. When the hyperlink is clicked, the browser navigates to the specified URL.</td>
</tr>
<tr>
<td>ImageField</td>
<td>Displays the value of a field as the <code>Src</code> property of an <code>&lt;img&gt;</code> HTML tag. The content of the bound field should be the URL to the physical image.</td>
</tr>
<tr>
<td>TemplateField</td>
<td>Displays user-defined content for each item in the column. You use this column type when you want to create a custom column field. The template can contain any number of data fields combined with literals, images, and other controls.</td>
</tr>
</tbody>
</table>

Table 11-13 lists the main properties shared by all column types.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccessibleHeaderText</td>
<td>The text that represents abbreviated text read by screen readers of Assistive Technology devices.</td>
</tr>
<tr>
<td>FooterStyle</td>
<td>Gets the style object for the column's footer.</td>
</tr>
<tr>
<td>FooterText</td>
<td>Gets and sets the text for the column's footer.</td>
</tr>
<tr>
<td>HeaderImageUrl</td>
<td>Gets and sets the URL of the image to place in the column's header.</td>
</tr>
<tr>
<td>HeaderStyle</td>
<td>Gets the style object for the column's header.</td>
</tr>
<tr>
<td>HeaderText</td>
<td>Gets and sets the text for the column's header.</td>
</tr>
<tr>
<td>InsertVisible</td>
<td>Indicates whether the field is visible when its parent data-bound control is in insert mode. This property does not apply to GridView controls.</td>
</tr>
<tr>
<td>ItemStyle</td>
<td>Gets the style object for the various columns' cells.</td>
</tr>
<tr>
<td>ShowHeader</td>
<td>Indicates whether the column's header is rendered.</td>
</tr>
<tr>
<td>SortExpression</td>
<td>Gets and sets the expression used to sort the grid contents when the column's header is clicked. Typically, this string property is set to the name of the bound data field.</td>
</tr>
</tbody>
</table>

The properties listed in the table represent a subset of the properties that each column type actually provides. In particular, each type of column defines a tailor-made set of properties to define and configure the bound field. Refer to the MSDN documentation for details on the programming interface of GridView's column types.
Bound Fields

The `BoundField` class represents a field that is displayed as plain text in a data-bound control such as `GridView` or `DetailsView`. To specify the field to display, you set the `DataField` property to the field's name. You can apply a custom formatting string to the displayed value by setting the `DataFormatString` property. The `NullDisplayText` property lets you specify alternative text to display should the value be null. Finally, by setting the `ConvertEmptyStringToNull` property to `true`, you force the class to consider empty strings as null values.

A `BoundField` can be programmatically hidden from view through the `Visible` property, while the `ReadOnly` property prevents the displayed value from being modified in edit mode. To display a caption in the header or footer sections, set the `HeaderText` and `FooterText` properties, respectively. You can also choose to display an image in the header instead of text. In this case, you set the `HeaderImageUrl` property.

Button Fields

A button field is useful to put a clickable element in a grid's column. You typically use a button field to trigger an action against the current row. A button field represents any action that you want to handle through a server-side event. When the button is clicked, the page posts back and fires a `RowCommand` event. Figure 11-3 shows a sample.

![Figure 11-3 Button fields in a GridView control.](file.png)
The following listing shows the markup code behind the grid in the figure:

```html
<asp:GridView ID="GridView1" runat="server" DataSourceID="SqlDataSource1"
 AutoGenerateColumns="false" AllowPaging="true"
 OnRowCommand="GridView1_RowCommand">
 <HeaderStyle backcolor="gray" font-bold="true" height="200%" />
 <PagerStyle backcolor="gray" font-bold="true" height="200%" />
 <PagerSettings Mode="NextPreviousFirstLast" />
 <Columns>
 <asp:BoundField datafield="productname"
  headerText="Product" />
 <asp:BoundField datafield="quantityperunit"
  headerText="Packaging" />
 <asp:BoundField datafield="unitprice"
  headerText="Price"
  htmlEncode="false"
  DataFormatString="{0:c}">
   <ItemStyle Width="80px" HorizontalAlign="right" />
 </asp:BoundField>
 <asp:ButtonField buttontype="Button" text="Add" CommandName="Add" />
 </Columns>
</asp:GridView>
```

Product information is displayed using a few `BoundField` objects. The sample button column allows you to add the product to the shopping cart. When users click the button, the `RowCommand` server event is fired. In case multiple button columns are available, the `CommandName` attribute lets you figure out which button was clicked. The value you assign to `CommandName` is any unique string that the code-behind class can understand. Here's an example:

```csharp
void GridView1_RowCommand(object sender, GridViewCommandEventArgs e)
{
 if (e.CommandName.Equals("Add"))
  {
    // Get the index of the clicked row
    int index = Convert.ToInt32(e.CommandArgument);
    // Create a new shopping item and add it to the cart
    AddToShoppingCart(index);
  }
}
```

In the sample, the button column shows fixed text for all data items. You get this by setting the `Text` property on the `ButtonField` class. If you want to bind the button text to a particular field on the current data item, you set the `DataTextField` property to the name of that field.

You can choose different styles for the button—push, link, or image. To render the button as an image, do as follows:

```html
<asp:ButtonField buttontype="Image" CommandName="Add"
  ImageUrl="/core35/images/cart.gif" />
```
To add a ToolTip to the button (or the image), you need to handle the `RowCreated` event. (I’ll discuss this in more detail later in the chapter.)

**Note** The `DataFormatString` property of the `BoundField` class doesn’t work properly without the additional attribute `HtmlEncode="false"`. The reason is because ASP.NET first HTML-encodes the value of bound field and then applies the formatting. But at that point, the bound value is no longer affected by the specified format string. Enabling HTML-encoding earlier in the cycle is a security measure aimed at preventing cross-site scripting attacks.

**HyperLink Fields**

Hyperlink columns point the user to a different URL, optionally displayed in an inner frame. Both the text and URL of the link can be obtained from the bound source. In particular, the URL can be set in either of two ways: through a direct binding to a data source field or by using a hard-coded URL with a customized query string. You choose the direct binding if the URL is stored in one of the data source fields. In this case, you set the `DataNavigateUrlFields` property to the name of the column. In some situations, though, the URL to access is application specific and not stored in the data source. In this case, you can set the `DataNavigateUrlFormatString` property with a hard-coded URL and with an array of parameters in the query string, as follows:

```xml
<asp:HyperLinkField DataTextField="productname"
  HeaderText="Product"
  DataNavigateUrlFields="productid"
  DataNavigateUrlFormatString="productinfo.aspx?id={0}"
  Target="ProductView" />
```

When the user clicks, the browser fills the specified frame window with the contents of the `productinfo.aspx?id=xxx` URL, where `xxx` comes from the `productid` field. The URL can include multiple parameters. To include more data-bound values, just set the `DataNavigateUrlFields` property to a comma-separated list of field names. This behavior extends that of the `DataGrid`’s hyperlink column in that it supports multiple parameters.

The text of the hyperlink can be formatted too. The `DataTextFormatString` property can contain any valid markup and uses the `{0}` placeholder to reserve space for the data-bound value. (See Figure 11-4.)
Part II Adding Data in an ASP.NET Site

FIGURE 11-4 Hyperlink fields in a GridView control.

Tip When choosing a target for the hyperlinked pages, you can also use any of the following standard targets: _self, _parent, _new.

CheckBox Fields

Although renamed, the column types we hitherto considered maintain an overall behavior that is very akin to that of analogous column types for DataGrids. The CheckBoxField type, on the other hand, is a new entry in ASP.NET 2.0 and beyond and is limited to GridView and other view controls. The simplest way in which you can get a check-box column in ASP.NET 1.x (or in general for DataGrids) is through templates.

The CheckBoxField column is a relatively simple bound column that displays a check box. You can bind it only to a data field that contains Boolean values. A valid Boolean value is a value taken from a column of type Bit in a SQL Server table (and analogous types in other databases) or a property of type bool if the control is bound to a custom collection. Any other form of binding will result in a parsing exception. In particular, you get an exception if you bind a CheckBoxField column to an integer property, thus implicitly assuming that 0 is false and a nonzero value is true. Figure 11-5 shows the CheckBoxField field type active within a GridView control.
Image Fields

The `ImageField` column type represents a field that is displayed as an image in a data-bound control. The cell contains an `<img>` element, so the underlying field must reference a valid URL. You can compose the URL at will, though. For example, you can use the `DataImageUrlField` to perform a direct binding where the content of the field fills the `Src` attribute of the `<img>` tag. Alternately, you can make the column cells point to an external page (or HTTP handler) that retrieves the bytes of the image from any source and passes them down to the browser. The following code illustrates this approach:

```xml
<Columns>
  <asp:ImageField DataImageUrlField="employeeid"
    DataImageUrlFormatString="showemployepicture.aspx?id={0}"
    DataAlternateTextField="lastname">
    <ControlStyle Width="120px" />
  </asp:ImageField>
  <asp:TemplateField headerText="Employee">
    <ItemStyle Width="220px" />
    <ItemTemplate>
      <b><%# Eval("titleofcourtesy") + " " + Eval("lastname") + ", " + Eval("firstname") %></b> <br />
      <%# Eval("title")%>
      <hr />
      <i><%# Eval("notes")%></i>
    </ItemTemplate>
  </asp:templatefield>
</Columns>
```
Cells in the ImageField column are filled with the output of the next URL:

```
ShowEmployeePicture.aspx?id=xxx
```

Needless to say, xxx is the value in the employeeid field associated with DataImageUrlField.

Interestingly enough, the alternate text can also be data bound. You use the DataAlternateTextField property. Figure 11-6 gives a sneak preview of the feature. The page in Figure 11-6 employs a template column to render the employee's information. I'll return to template columns in a moment.

![Figure 11-6 Image fields in a GridView control.](image)

The following code demonstrates the world's simplest page to retrieve and then serve an image out of a database table:

```csharp
void Page_Load(object sender, EventArgs e)
{
    int id = Convert.ToInt32(Request.QueryString["id"]);
    string connString = "..."
    string cmdText = "SELECT photo FROM employees WHERE employeeid='\"empID\"';
    using (SqlConnection conn = new SqlConnection(connString))
    {
        SqlCommand cmd = new SqlCommand(cmdText, conn);
        cmd.Parameters.AddWithValue("\"empID\"", id);
        byte[] img = null;
        conn.Open();
        try
        {
            img = (byte[])cmd.ExecuteScalar();
            if (img != null)
```
The preceding code serves a standard image if the value of the field specified is `null`. You can obtain the same result by setting the `NullImageUrl` property if you're using direct binding—that is, not passing through an external page or handler.

**Note** The `EMP_IMG_OFFSET` constant in the code snippet should normally be just 0. However, given the particular structure of the photo column of the Northwind's Employees database, it has to be 78. But, again, this is required only with that particular table.

### Templated Fields

Figure 11-6 shows a customized column where the values of several fields are combined. This is exactly what you can get by using templates. A `TemplateField` column gives each row in the grid a personalized user interface that is completely defined by the page developer. You can define templates for various rendering stages, including the default view, in-place editing, the header, and the footer. The supported templates are listed in Table 11-14.

<table>
<thead>
<tr>
<th>Template</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlternatingItemTemplate</td>
<td>Defines the contents and appearance of alternating rows. If these items are not specified, the <code>ItemTemplate</code> is used.</td>
</tr>
<tr>
<td>EditItemTemplate</td>
<td>Defines the contents and appearance of the row currently being edited. This template should contain input fields and possibly validators.</td>
</tr>
<tr>
<td>FooterTemplate</td>
<td>Defines the contents and appearance of the row's footer.</td>
</tr>
<tr>
<td>HeaderTemplate</td>
<td>Defines the contents and appearance of the row's header.</td>
</tr>
<tr>
<td>ItemTemplate</td>
<td>Defines the default contents and appearance of the rows.</td>
</tr>
</tbody>
</table>

A templated view can contain anything that makes sense to the application you're building—server controls, literals, and data-bound expressions. Data-bound expressions allow you to insert values contained in the current data row. You can use as many fields as needed in a template. Notice, though, that not all templates support data-bound expressions. The header
and footer templates are not data-bound, and any attempt to use expressions will result in an exception.

The following code shows how to define the item template for a product column. The column displays on two lines and includes the name of the product and some information about the packaging. You use data-bound expressions (which are discussed in Chapter 10) to refer to data fields.

```csharp
<asp:templatefield headertext="Product">
    <itemtemplate>
        <b><%# Eval("productname")%></b> <br />
        available in <%# Eval("quantityperunit")%>
    </itemtemplate>
</asp:templatefield>
```

Figure 11-7 demonstrates template fields in action.

**FIGURE 11-7** Template fields in a GridView control.

**Note** The TemplateField class also features an InsertTemplate property. However, this type of template is never used by the GridView control. InsertTemplate is used by the FormView control instead. As mentioned earlier, in ASP.NET, view controls share some field classes, such as TemplateField. As a result, TemplateField (and a few more classes) provides a superset of properties that serves the needs of multiple view controls. We’ll cover the FormView control in Chapter 13.
Paging Data

The GridView is designed to take advantage of specific capabilities of the underlying data source control. In this way, the grid control can handle common operations on data such as sorting, paging, updating, and deleting. In general, not all data source components support all possible and feasible data operations. Data source components expose Boolean properties (such as the CanSort property) to signal whether they can perform a given operation.

Important If a GridView control is bound to its data source through the DataSource property—that is, it doesn’t leverage data source controls—its overall behavior as far as paging and other operations (for example, sorting, editing) are concerned is nearly identical to the DataGrid control. In this case, the GridView fires events and expects the binding code in the page to provide instructions and fresh data. In the remainder of the chapter, unless explicitly mentioned, we refer to a GridView bound to a data source control.

To some extent, the GridView makes transparent for the page developer the implementation of commonly required features such as sorting and paging. In most cases, you need only a fraction of the code you need with DataGrid; in some cases, no code at all is required. This said, don’t forget what one old and wise proverb says—not all that glitters is gold. Put another way, be aware that the less code you write, the more you rely on the existing infrastructure to get things done. In doing so, you let the system make important decisions on your behalf. Paging and sorting are key operations in Web applications. You can still accept what the GridView does by default, but if you get to know exactly what happens under the hood, you have a better chance of diagnosing and fixing in a timely manner any performance problems that show up in the lifetime of the application.

Codeless Data Paging

The ability to scroll a potentially large set of data is an important but challenging feature for modern, distributed applications. An effective paging mechanism allows customers to interact with a database without holding resources. To enable paging on a GridView control, all you do is set the AllowPaging property to true. When the AllowPaging property is set to true, the grid displays a pager bar and prepares to detect a user’s pager button clicks.

When a user clicks to see a new page, the page posts back, but the GridView traps the event and handles it internally. This marks a major difference between GridView and the DataGrid and programming model you might know from previous versions of ASP.NET. With the GridView, there’s no need to write a handler for the PageIndexChanged event. The event is still exposed (and partnered with PageIndexChanging), but you should handle it only to perform extra actions. The GridView knows how to retrieve and display the requested new page. Let’s take a look at the following control declaration:

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

To some extent, the GridView makes transparent for the page developer the implementation of commonly required features such as sorting and paging. In most cases, you need only a fraction of the code you need with DataGrid; in some cases, no code at all is required. This said, don’t forget what one old and wise proverb says—not all that glitters is gold. Put another way, be aware that the less code you write, the more you rely on the existing infrastructure to get things done. In doing so, you let the system make important decisions on your behalf. Paging and sorting are key operations in Web applications. You can still accept what the GridView does by default, but if you get to know exactly what happens under the hood, you have a better chance of diagnosing and fixing in a timely manner any performance problems that show up in the lifetime of the application.

Codeless Data Paging

The ability to scroll a potentially large set of data is an important but challenging feature for modern, distributed applications. An effective paging mechanism allows customers to interact with a database without holding resources. To enable paging on a GridView control, all you do is set the AllowPaging property to true. When the AllowPaging property is set to true, the grid displays a pager bar and prepares to detect a user’s pager button clicks.

When a user clicks to see a new page, the page posts back, but the GridView traps the event and handles it internally. This marks a major difference between GridView and the DataGrid and programming model you might know from previous versions of ASP.NET. With the GridView, there’s no need to write a handler for the PageIndexChanged event. The event is still exposed (and partnered with PageIndexChanging), but you should handle it only to perform extra actions. The GridView knows how to retrieve and display the requested new page. Let’s take a look at the following control declaration:

```xml
<asp:GridView ID="GridView1" runat="server" 
    DataSourceID="SqlDataSource1" AllowPaging="true" />
```
Any data SqlDataSource1 binds to the grid is immediately pageable. As in Figure 11-8, the control displays a pager with a few predefined links (first, previous, next, and last) and automatically selects the correct subset of rows that fit in the selected page.

The default user interface you get with the GridView doesn’t include the page number. Adding a page number label is as easy as writing a handler for the PageIndexChanged event:

```csharp
protected void GridView1_PageIndexChanged(object sender, EventArgs e)
{
    ShowPageIndex();
}
private void ShowPageIndex()
{
    CurrentPage.Text = (GridView1.PageIndex + 1).ToString();
}
```

Once again, note that the PageIndexChanged handler is not involved with data binding or page selection as it is with DataGrids. If you don’t need any post-paging operation, you can blissfully omit it altogether.

What’s the cost of this apparently free (and magical) paging mechanism?

The GridView control doesn’t really know how to get a new page. It simply asks the bound data source control to return the rows that fit in the specified page. Paging is ultimately up to the data source control. When a grid is bound to a SqlDataSource control, paging requires that the whole data source be bound to the control. When a grid is bound to an ObjectDataSource control, paging depends on the capabilities of the business object you’re connecting to.
Let's tackle SqlDataSource first. It is mandatory that you set DataSourceMode to DataSet (the default setting). This means that the whole data set is retrieved and only the few records that fit in the current page size are displayed. In an extreme scenario, you might end up downloading 1000 records for each postback to show only 10. Things go much better if you enable caching on SqlDataSource by setting EnableCaching to true. In this case, the whole data set is downloaded only once and stored in the ASP.NET cache for the specified duration. As long as the data stays cached, any page is displayed almost for free. However, a potentially large chunk of data is stored in memory. This option is therefore recommended only for relatively small sets of records shared by all users.

Tip
If you want to page records at the database level, the best that you can do is code the desired behavior in a stored procedure and bind the stored procedure to the SelectCommand property of the SqlDataSource control. In this case, turn caching off.

Moving the Burden of Paging to the DAL
As we discussed in Chapter 10, the ObjectDataSource control supplies a rather generic interface for paging that heavily relies on the capabilities of the underlying business and data access layers (DALs).

The key point is that you should have a paging-enabled business object. You configure the ObjectDataSource control based on the characteristics of your business object method. Once you have identified the select method, you overload it with a version that takes two extra parameters—the page size and start index for the page. In the end, the select method must be able to retrieve pages of records. In the declaration of the ObjectDataSource control, you set the StartRowIndexParameterName and MaximumRowsParameterName properties to the name of the method parameter that denotes the start index and page size, respectively.

One more step is needed to enable the GridView to page the data source provided by the ObjectDataSource control. You also need to set the Enable Paging property of ObjectDataSource to true:

```xml
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server" EnablePaging="true" TypeName="Core35.DAL.Customers"
    StartRowIndexParameterName="firstRow"
    MaximumRowsParameterName="totalRows"
    SelectMethod="LoadByCountry">
    <SelectParameters>
        <asp:ControlParameter Name="country" ControlID="Countries"
            PropertyName="SelectedValue" />
    </SelectParameters>
</asp:ObjectDataSource>
```
In the preceding code, you explicitly specify only the parameters whose contents are important for the method to work. The two paging-related parameters are left to the GridView to set. The page size parameter is automatically bound to the PageSize property of the GridView; the first index to retrieve is determined by multiplying page size by page index. Here are the prototypes of the LoadByCountry method:

```csharp
public static CustomerCollection LoadByCountry(string country) {
    LoadByCountry(country, -1, 0);
}

public static CustomerCollection LoadByCountry(string country, int totalRows, int firstRow) {
    // Retrieve the specified subset of records
}
```

The mechanics of ObjectDataSource doesn't say much about the effectiveness of the paging algorithm. How the business object actually retrieves the records in the requested page is an implementation and application-specific detail. In the sample code, LoadByCountry runs the original query and retrieves a data reader to the whole data set. Next, it discards all the records that don't fit in the specified range. This implementation is a good compromise between simplicity and effectiveness. It is not the best solution possible, but it's easy to implement and demonstrate. The memory consumption is limited to one record at a time, but the database returns the whole data set.

**Paging Algorithms**

The GridView doesn't support the AllowCustomPaging property you find on DataGrids. However, customizing the paging algorithm is definitely possible. At its core, a custom paging algorithm provides a way to extract pages of records that minimizes caching of records. Ideally, you would ask the database to page the results of a particular query. Very few databases, though, support this feature. Several alternative approaches exist, with pros and cons.

A possible strategy entails creating temporary tables to select only the subset of records you really need. You build a stored procedure and pass it parameters to indicate the page...
Chapter 11 Creating Bindable Grids of Data

size and index. Alternately, you can use nested SELECT commands and the TOP statement to retrieve all the records up to the last record in the requested page, reverse the order, and discard unneeded records. Again, the TOP clause is not common to all databases. Another possible approach based on dynamically built SQL code is discussed in the following blog post: http://weblogs.sqlteam.com/jeffs/archive/2004/03/22/1085.aspx.

If you can collaborate with the database administrator (DBA), you can require that an ad hoc column be added to index the queries. In this case, the DAL must guarantee that the values in the column form a regular succession of values and can be computable. The simplest way of accomplishing this is by giving the column progressive numbers.

Configuring the Pager

When the AllowPaging property is set to true, the grid displays a pager bar. You can control the characteristics of the pager to a large extent, through the <PagerSettings> and <PagerStyle> tags or their equivalent properties. The pager of the GridView control also supports first and last page buttons and lets you assign an image to each button. (This is also possible for DataGrids, but it requires a lot of code.) The pager can work in either of two modes—displaying explicit page numbers, or providing a relative navigation system. In the former case, the pager contains numeric links, one representing a page index. In the latter case, buttons are present to navigate to the next or previous page and even to the first or last page. The Mode property rules the user interface of the pager. Available modes are listed in the Table 11-15.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NextPrevious</td>
<td>Displays next and previous buttons to access the next and previous pages of the grid</td>
</tr>
<tr>
<td>NextPreviousFirstLast</td>
<td>Displays next and previous buttons plus first and last buttons to directly access first and last pages of the grid</td>
</tr>
<tr>
<td>Numeric</td>
<td>Displays numeric link buttons corresponding to the pages of the grid</td>
</tr>
<tr>
<td>NumericFirstLast</td>
<td>Displays numeric link buttons corresponding to the pages of the grid plus first and last buttons to directly access first and last pages of the grid</td>
</tr>
</tbody>
</table>

Ad hoc pairs of properties—xxxPageText and xxxPageImageUrl—let you set the labels for these buttons as desired. The xxx stands for any of the following: First, Last, Next, or Previous. Figure 11-9 shows a sample page in action.
Part II Adding Data in an ASP.NET Site

FIGURE 11-9 A pageable GridView with two pagers.

Depending on the size of the grid, the first and last rows in a grid might not necessarily fit in the screen real estate. To make it easier for users to page regardless of the scrollbar position, you can enable top and bottom pagers for a grid. You do this by setting the Position attribute on the <PagerSettings> element:

```xml
<PagerSettings Position="TopAndBottom" />
```

Other options are to display the pager only at the top or only at the bottom of the grid.

The pager of the GridView control can be entirely replaced with a new one, in case of need. (See Figure 11-10.) You do this by adding the <PagerTemplate> element to the control's declaration. Here's an example:

```xml
<PagerTemplate>
    <asp:Button ID="BtnFirst" runat="server" commandname="First"
               Text="First" />
    <asp:Button ID="BtnPrev" runat="server" commandname="Prev"
               Text="<<" />
    <asp:Button ID="BtnNext" runat="server" commandname="Next"
               Text=">>" />
    <asp:Button ID="BtnLast" runat="server" commandname="Last"
               Text="Last" />
</PagerTemplate>
```
To handle clicks on the buttons, you write a RowCommand event handler and set the page index explicitly:

```csharp
void GridView1_RowCommand(object sender, GridViewCommandEventArgs e)
{
    if (e.CommandName == "Last")
        GridView1.PageIndex = GridView1.PageCount - 1;
    if (e.CommandName == "First")
        GridView1.PageIndex = 0;
    if (e.CommandName == "Next")
        GridView1.PageIndex ++;
    if (e.CommandName == "Prev")
        GridView1.PageIndex --;
}
```

Admittedly, this code is quite simple and should be fleshed out a little bit, at least to make it capable of disabling buttons when the first or last index is reached.

**FIGURE 11-10** A pageable GridView with a custom pager.

### Sorting Data

Sorting is a delicate, nonlinear operation that normally is quite expensive if performed on the client. Generally speaking, in fact, the best place to sort records is in the database environment because of the super-optimized code you end up running most of the time. Be aware of this as we examine the sorting infrastructure of the GridView control and data source controls. The GridView doesn't implement a sorting algorithm; instead, it relies on the data source control (or the page, if bound to an enumerable object) to provide sorted data.
Codeless Data Sorting

To enable the GridView’s sorting capabilities, you set the AllowSorting property to true. When sorting is enabled, the GridView gains the ability of rendering the header text of columns as links. You can associate each column with a sorting expression by using the SortExpression property. A sorting expression is any comma-separated sequence of column names. Each column name can be enriched with an order qualifier such as DESC or ASC. DESC indicates a descending order, while ASC denotes the ascending order. The ASC qualifier is the default; if the order qualifier value is omitted, the column is sorted in ascending order. The following code sets up the GridView column for sorting on the productname data source column:

```xml
<asp:GridView runat="server" id="MyGridView" DataSourceID="MySource" AllowSorting="true" AutoGenerateColumns="false">
    <Columns>
        <asp:BoundField datafield="productname" headertext="Product" sortexpression="productname" />
        <asp:BoundField datafield="quantityperunit" headertext="Packaging" />
    </Columns>
</asp:GridView>
```

Just as for paging, with a GridView no manually written code is required to make sorting work. If properly configured, the GridView’s sorting infrastructure works without further intervention and in a bidirectional way—that is, if you click on a column sorted in descending order, it is sorted in ascending order and vice versa. You need to add some custom code only if you want to implement more advanced capabilities such as showing a glyph in the header to indicate the direction. (I’ll say more about that in a moment.)

Just as for paging, the main snag with sorting is how the underlying data source control implements it. Let’s see what happens when the grid is bound to a SqlDataSource object. Other than setting AllowSorting to true and adding the sort expression to the sortable columns, no other action is required. (See Figure 11-11.)

When the user clicks to sort, the grid asks the SqlDataSource control to return sorted data. As mentioned, the SqlDataSource control returns a DataSet by default. If this is the case, the control retrieves the data, builds a DataView out of it, and calls the DataView’s Sort method. This approach works fine, but it’s not exactly the fastest way you have to sort. You might still find it to be a good fit for your application, but be aware that sorting is performed using the Web server’s memory. Combined with caching, both paging and sorting in memory are a feasible solution for shared and relatively small sets of records.
Is there any chance to get pre-sorted data from the database server? The first step is to set the `DataSourceMode` property of the `SqlDataSource` control to `DataReader`. If you leave it set to `DataSet`, sorting will occur in memory. The second step requires you to write a stored procedure to retrieve data. To get data sorted, you also set the `SortParameterName` property of the data source control to the name of the stored procedure parameter that indicates the sort expression. Obviously, you need the stored procedure to build its command text dynamically to incorporate the proper `ORDER BY` clause. Here’s how to modify the Northwind database’s `CustOrderHist` stored procedure to make its results sortable at will:

```sql
CREATE PROCEDURE CustOrderHistSorted
    @CustomerID nchar(5), @SortedBy varchar(20)='total'  AS
SET QUOTED_IDENTIFIER OFF
IF @SortedBy = ''
BEGIN
    SET @SortedBy = 'total'
END
EXEC (  
    'SELECT ProductName, Total=SUM(Quantity)  
    FROM Products P, [Order Details] OD, Orders O, Customers C  
    WHERE C.CustomerID = ''' + @CustomerID + '''  
    AND C.CustomerID = O.CustomerID AND O.OrderID = OD.OrderID  
    AND OD.ProductID = P.ProductID GROUP BY ProductName  
    ORDER BY ' + @SortedBy)  
GO
```

FIGURE 11-11 A sortable GridView control
At this point, the grid is ready to show sorted columns of data and the burden of sorting has moved to the database management system (DBMS).

```<asp:SqlDataSource ID="SqlDataSource1" runat="server"
    DataSourceMode="DataReader"
    ConnectionString='<%$ ConnectionStrings:NWind %>'
    SortParameterName="SortedBy"
    SelectCommand="CustOrderHistSorted"
    SelectCommandType="StoredProcedure">
    <SelectParameters>
    <asp:ControlParameter ControlID="CustList"
    Name="CustomerID" PropertyName="SelectedValue" />
    </SelectParameters>
    </asp:SqlDataSource>
```

It is essential to know that sorting data on the database, as shown here, is incompatible with caching. You need to set `EnableCaching` to false, otherwise an exception is thrown. As a result, you go back to the database every time the user clicks to sort.

If you use the `DataSet` mode and enable caching, you initially get data from the database, sorted as expected, but successive sorting operations are resolved in memory. Finally, if you use the `DataSet` mode and disable caching, you still go down to the database for sorting each time. Note that this option is mentioned only for completeness: the effect is the same as using `DataReader`, but a data reader is a more efficient approach when caching is not required.

In general, the availability of the `SortParameterName` property opens up a world of possibility for sorting the contents of other data-bound controls (for example, `Repeater` and custom controls) that mostly consume data and don't require paging or caching.

### Moving the Burden of Sorting to the DAL

What if you use an `ObjectDataSource` control instead? In this case, the burden of sorting should be moved to the DAL or business layer and exposed to the data source control by the programming interface of the bound business object. Let's modify the `LoadByCountry` method we considered earlier for paging and add to it a new parameter to indicate the sort expression:

```csharp
public static CustomerCollection LoadByCountry(
    string country, int totalRows, int firstRow,
    string sortExpression)
{
    CustomerCollection coll = new CustomerCollection();
    using (SqlConnection conn = new SqlConnection(ConnectionString))
    {
        SqlCommand cmd;
```
The `cmdLoadByCountry` constant represents the SQL command or stored procedure we use to retrieve data. As you can see, this implementation of the method simply adds an optional `ORDER BY` clause to the existing command. This might not be the best approach ever devised, but it certainly fits the bill of having the burden of sorting moved down to the DAL and from there to the database. At this point, you set the `SortParameterName` on the `ObjectDataSource` control to the method’s parameter that determines the sorting—in this case, `sortExpression`:

```csharp
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server" EnablePaging="true"
    TypeName="Core35.DAL.Customers"
    SortParameterName="sortExpression"
    StartRowIndexParameterName="firstRow"
    MaximumRowsParameterName="totalRows"
    SelectMethod="LoadByCountry">
    <SelectParameters>
        ...
    </SelectParameters>
</asp:ObjectDataSource>
```

The advantage of this approach is that you take full control of the sorting machinery, and you can decide how, where, and when to implement it. You might have to write some code in your DAL for sorting, but consider that you only write highly focused code. In fact, no infrastructural code is required, as the machinery is set up for you by ASP.NET.

---

**Note** One more item worth mentioning about sorting on a GridView control is that you can cancel the sorting operation if need be. To do this, you write a handler for the `Sorting` event, get the event argument data (of type `GridViewSortEventArgs`), and set its `Cancel` property to `true`. 
Give Users Feedback

The GridView control doesn't automatically add any visual element to the output that indicates the direction of the sorting. This is one of the few cases in which some coding is needed to complete sorting:

```csharp
<script runat="server">
void GridView1_RowCreated (object sender, GridViewRowEventArgs e) {
if (e.Row.RowType == DataControlRowType.Header)
AddGlyph(sender as GridView, e.Row);
}

void AddGlyph(GridView grid, GridViewRow item) {
    Label glyph = new Label();
glyph.EnableTheming = false;
    // This does assume you have this font installed. You might want to
    // consider using images in reality
    glyph.Font.Name = "webdings";
glyph.Font.Size = FontUnit.Small;
glyph.Text = (grid.SortDirection==SortDirection.Ascending ?"5" :"6");
    // Find the column you sorted by
    for(int i=0; i<grid.Columns.Count; i++) {
        string colExpr = grid.Columns[i].SortExpression;
        if (colExpr != null && colExpr == grid.SortExpression)
        item.Cells[i].Controls.Add (glyph);
    }
}
</script>
```

The idea is that you write a handler for the RowCreated event and look for the moment when the header is created. Next you create a new Label control that represents the glyph you want to add. Where should the Label control be added?

The newly created Label control has font and text adequately set to generate a glyph (typically ⬆ and ⬇) that indicates the direction of the sorting. (The glyphs correspond to 5 and 6 in the Webdings font.) You must add it alongside the header text of the clicked column. The index of the column can be stored to the view state during the Sorting event. Alternatively, it can simply be retrieved, comparing the current sort expression—the grid’s SortExpression property—to the column’s sort expression. Once you know the index of the column, you retrieve the corresponding table cells and add the Label:

```csharp
item.Cells[1].Controls.Add (glyph);
```

The results are shown in Figure 11-12. If your page is based on a theme, the font of the Label control—essential for rendering the glyph correctly—might be overridden. To avoid that,
you should disable theming support for the label control. The EnableThemimg property does just that.

![Figure 11-12: Enhancing the sorting capabilities of the GridView control.]

**Using Callbacks for Paging and Sorting**

Both sorting and paging operations require a postback with subsequent full refresh of the page. In most cases, this is a heavy operation, as the page usually contains lots of graphics. To provide the user with a better experience, wouldn’t it be nice if the grid could go down to the Web server, grab the new set of records, and update only a portion of the interface? Thanks to ASP.NET script callbacks—which I cover in greater detail in my other book, *Programming Microsoft ASP.NET 2.0 Applications: Advanced Topics* (Microsoft Press, 2006)—the GridView control is capable of offering this feature. All that you have to do is turn on the Boolean property `EnableSortingAndPagingCallbacks`.

As mentioned, the feature relies on the services of the ASP.NET script callback engine, which is designed to work also with non–Internet Explorer browsers, including Firefox, Netscape 6.x and newer, Safari 1.2, and the latest Opera browser.

---

**Important** As we’ll see later in Chapter 19 and Chapter 20, the AJAX library makes this feature obsolete. Either via partial rendering or through direct calls to script services, flicker-free page updates are better implemented via AJAX than by using the ASP.NET Script Callback engine.
Part II Adding Data in an ASP.NET Site

**SqlDataSource vs. ObjectDataSource**

A few considerations will help clarify when to use SqlDataSource and ObjectDataSource controls. First, remember that these data source controls are not the only two choices for developers to do sane data binding. By far, though, they are the most popular and commonly used. It is also essential to bear in mind that data binding in ASP.NET 2.0 and beyond is in no way limited to using data source controls. This said, SqlDataSource and ObjectDataSource are just tools in the ASP.NET toolbox and should be used if they’re right for the job.

As I see things, SqlDataSource is optimized for a disconnected approach to data binding. It works at its best if you retrieve data through a DataSet. Only in this case are paging, sorting, and caching capabilities enabled. Of these three functionalities, only sorting is somehow replicable in data reader mode. If using DataSets is fine for your application, using SqlDataSource is an excellent choice. It gives you ready-made solutions with mostly declarative code that is simple to write, but it’s not necessarily effective in a real-world application. Put another way, using SqlDataSource in an application might be good for certain features, but it’s hardly sufficient to power the whole DAL.

Should you instead realize that you need more control over paging and sorting operations (such as custom paging or server-side sorting), switching to ObjectDataSource appears to me to be a sounder idea. In this case, you start by designing and implementing a fully fledged DAL and, optionally, a business layer too. In this layer, you craft any capabilities you need to be supported from the grid—paging, sorting, or even data caching.

In addition, the ObjectDataSource control fully supports custom entity classes and custom collections. The support for generics in the .NET Framework makes writing custom collections a snap, and it significantly reduces the cost of writing a fully custom DAL built on made-to-measure and domain-specific objects. LINQ and related data source controls doesn’t diminish the importance of ObjectDataSource and, especially, what lies behind it. You can use the LINQ syntax to work on data within the DAL or use LINQ as a replacement for the SqlDataSource control. LINQ doesn’t kill the DAL; instead, it is a smart option to use to build parts of it.

**Editing Data**

A major strength of the GridView control—which makes up for a major shortcoming of the DataGrid—is the ability to handle updates to the data source. The DataGrid control provides only an infrastructure for data editing. The GridView provides the necessary user interface elements and fires appropriate events when the user modifies the value of a certain data field, but it does not submit those changes back to the data source. Developers are left with...
the disappointing realization that they have to write a huge amount of boilerplate code to really persist changes.

With the GridView control, when the bound data source supports updates, the control can automatically perform this operation, thus providing a truly out-of-the-box solution. The data source control signals its capability to update through the CanUpdate Boolean property.

Much like the DataGrid, the GridView can render a column of command buttons for each row in the grid. These special command columns contain buttons to edit or delete the current record. With the DataGrid, you must explicitly create an edit command column using a special column type—the EditCommandColumn class. The GridView simplifies things quite a bit for update and delete operations.

In-Place Editing and Updates

In-place editing refers to the grid’s ability to support changes to the currently displayed records. You enable in-place editing on a grid view by turning on the AutoGenerateEditButton Boolean property:

```xml
<asp:gridview runat="server" id="GridView1" datasourceid="MySource"
    autogeneratecolumns="false" autogenerateeditbutton="true">
    ...
</asp:gridview>
```

When the AutoGenerateEditButton property is set to true, the GridView displays an additional column, like that shown in Figure 11-13. By clicking the Edit button, you put the selected row in edit mode and can enter new data at will.

![Figure 11-13 A GridView that supports in-place editing.](image-url)
Part II  Adding Data in an ASP.NET Site

To abort editing and lose any changes, users simply click the Cancel button. The GridView can handle this click without any external support; the row returns to its original read-only state; and the EditIndex property takes back its -1 default value—meaning no row is currently being edited. But what if users click the update link? The GridView first fires the RowUpdating event and then internally checks the CanUpdate property on the data source control. If CanUpdate returns false, an exception is thrown. CanUpdate returns false if the data source control has no update command defined.

Suppose your grid is bound to a SqlDataSource object. To persist changes when the user updates, you have to design your code as follows:

```xml
<asp:sqlsource runat="server" ID="EmployeesSource"
    ConnectionString="<%$ ConnectionStrings:NWind %>">
    SelectCommand="SELECT employeeid, firstname, lastname FROM employees"
    UpdateCommand="UPDATE employees SET
                 firstname=@firstname, lastname=@lastname
                 WHERE employeeid=@original_employeeid"
</asp:sqlsource>
<asp:gridview runat="server" id=".GridView1" datasourceid="EmployeesSource"
    AutoGenerateColumns="false"
    DataKeyNames="employeeid"
    AutoGenerateEditButton="true">
    <columns>
    <asp:boundfield datafield="firstname" headertext="First" />
    <asp:boundfield datafield="lastname" headertext="Last" />
    </columns>
</asp:gridview>
```

The UpdateCommand attribute is set to the SQL command to use to perform updates. When you write the command, you declare as many parameters as needed. However, if you stick with a particular naming convention, parameter values are automatically resolved. Parameters that represent fields to update (such as firstname) must match the name of DataField property of a grid column. The parameter used in the WHERE clause to identify the working record must match the DataKeyNames property—the key for the displayed records. The original_XXX format string is required for identity parameters. You can change this scheme if you want through the OldValuesParameterFormatString property on the data source control.

The successful completion of an update command is signaled throughout the grid via the RowUpdated event.

**Note** The GridView collects values from the input fields and populates a dictionary of name/value pairs that indicate the new values for each field of the row. The GridView also exposes a RowUpdating event that allows the programmer to validate the values being passed to the data source object. In addition, the GridView automatically calls Page.IsValid before starting the update operation on the associated data source. If Page.IsValid returns false, the operation is canceled. This is especially useful if you’re using a custom template with validators.
If the grid is bound to an ObjectDataSource control, things go a bit differently. The bound business object must have an update method. This method will receive as many arguments as it needs to work. You can decide to pass parameters individually or grouped in a unique data structure. This second option is preferable if you have a well-done DAL. Here's an example:

```xml
<asp:ObjectDataSource ID="CustomersSource" runat="server" 
    TypeName="Core35.DAL.Customers" 
    SelectMethod="LoadAll" 
    UpdateMethod="Save" 
    DataObjectTypeName="Core35.DAL.Customer">
</asp:ObjectDataSource>
<asp:GridView ID="GridView1" runat="server" DataSourceID="CustomersSource" 
    DataKeyNames="id" AutoGenerateColumns="false" 
    AutoGenerateEditButton="true" 
    Columns>
    <asp:BoundField DataField="companyname" HeaderText="Company" />
    <asp:BoundField DataField="street" HeaderText="Address" />
    <asp:BoundField DataField="city" HeaderText="City" />
</Columns>
</asp:GridView>
```

The `Save` method can have the following prototype and implementation:

```csharp
public static void Save(Customer cust) {
    using (SqlConnection conn = new SqlConnection(ConnectionString)) {
        SqlCommand cmd = new SqlCommand(cmdSave, conn);
        cmd.Parameters.AddWithValue("@id", cust.ID);
        cmd.Parameters.AddWithValue("@companyname", cust.CompanyName);
        cmd.Parameters.AddWithValue("@city", cust.City);
        cmd.Parameters.AddWithValue("@address", cust.Street);
        ...
        conn.Open();
        cmd.ExecuteNonQuery();
        conn.Close();
        return;
    }
}
```

The physical SQL command (or stored procedure) to run is nothing more than a classic UPDATE statement with a list of SET clauses. The `DataObjectTypeName` attribute indicates the name of a class that the `ObjectDataSource` uses for a parameter in a data operation.

---

**Note** If you set the `DataObjectTypeName` property, all data methods can either be parameterless or accept an object of the specified type. This happens regardless of whether you declaratively fill the parameters collection for the method. The `DataObjectTypeName` property takes precedence over parameter collections.
Deleting Displayed Records

From the GridView's standpoint, deleting records is not much different from updating. In both cases, the GridView takes advantage of a data source's ability to perform data operations. You enable record deletion by specifying a value of true for the AutoGenerateDeleteButton property. The GridView renders a column of buttons that, if clicked, invokes the delete command for the row on the bound data source control. The data source method is passed a dictionary of key field name/value pairs that are used to uniquely identify the row to delete.

```xml
<asp:sqldatasource runat="server" ID="EmployeesSource"
    ConnectionString="@%$ ConnectionStrings:NWind %"
    SelectCommand="SELECT employeeid, firstname, lastname FROM employees"
    UpdateCommand="UPDATE employees SET
        firstname=@firstname, lastname=@lastname
    WHERE employeeid=@original_employeeid"
    DeleteCommand="DELETE employees WHERE
        employeeid=@original_employeeid" />
```

The GridView doesn’t provide any feedback about the operation that will take place. Before proceeding, it calls Page.IsValid, which is useful if you have a custom template with validators to check. In addition, the RowDeleting event gives you another chance to programmatically control the legitimacy of the operation.

The delete operation fails if the record can’t be deleted because of database-specific constraints. For example, the record can’t be deleted if child records refer to it through a relationship. In this case, an exception is thrown.

To delete a record through an ObjectDataSource control, you give your business object a couple of methods, as follows:

```csharp
public static void Delete(Customer cust)
{
    Delete(cust.ID);
}

public static void Delete(string id)
{
    using (SqlConnection conn = new SqlConnection(ConnectionString))
    {
        SqlCommand cmd = new SqlCommand(cmdDelete, conn);
        cmd.Parameters.AddWithValue("@id", id);
        conn.Open();
        cmd.ExecuteNonQuery();
        conn.Close();
        return;
    }
}
```

Overloading the delete method is not mandatory, but it can be useful and certainly make your DAL more flexible and easier to use.


Inserting New Records

In its current form, the GridView control doesn't support inserting data against a data source object. This omission is a result of the GridView implementation and not the capabilities and characteristics of the underlying data source. In fact, all data source controls support an insert command property. As you'll see in Chapter 13, the insertion of new records is a scenario fully supported by the DetailsView and FormView control.

In older versions of ASP.NET, a common practice to make DataGrid controls support record insertions entails that you modify the footer or the pager to make room for empty text boxes and buttons. The GridView supports the same model and makes it slightly simpler through the PagerTemplate property as far as the pager is concerned. Modifying the contents of the footer is possible through the RowCreated event (which I'll say more about in a moment). Note, though, that if the grid is bound to an empty data set, the footer bar is hidden. What if you want your users to be able to add a new record to an empty grid? Resort to the EmptyDataTemplate, as follows:

```xml
<emptydatatemplate>
  <asp:label ID="Label1" runat="server">
    There's no data to show in this view.
  </asp:label>
  <asp:Button runat="server" ID="btnAddNew" CommandName="AddNew"
    Text="Add New Record" />
</emptydatatemplate>
```

To trap the user's click of the button, you write a handler for the RowCommand event:

```csharp
void Gridview1_RowCommand(object sender, GridViewCommandEventArgs e) {
  if (e.CommandName == "AddNew") {
  [...]
}
```

Advanced Capabilities

To complete the overview of the GridView control, we just need to take a look at a couple of common programming scenarios—drill-down and row customization. A grid presents a list of items to the user; in many cases, the user needs to select one of those items and start an operation on it. As discussed earlier, button columns exist to facilitate this task. We'll delve deeper into this topic in a moment. Row customization is another common feature, which gives you a chance to modify the standard rendering of the grid. You can edit the row layout, add or remove cells, or modify visual attributes on a per-row basis so that certain rows show up distinctly from others (for example, rows representing negative values).
Let's return to a problem that we briefly mentioned earlier in the chapter while discussing button columns. Imagine you're building an e-commerce application; one of your pages shows a grid of products with buttons for users to add products to their shopping cart. You add a button column and write a handler for the RowCommand event:

```csharp
void GridView1_RowCommand(object sender, GridViewCommandEventArgs e)
{
    if (e.CommandName.Equals("Add"))
    {
        // Get the index of the clicked row
        int index = Convert.ToInt32(e.CommandArgument);
        // Create a new shopping item and add it to the cart
        AddToShoppingCart(index);
    }
}
```

This is where we left off earlier. Let's go one step further now and expand the code for AddToShoppingCart. What's the purpose of this method? Typically, it retrieves some information regarding the clicked product and stores that in the data structure that represents the shopping cart. In the sample code, the shopping cart is a custom collection named ShoppingCart:

```csharp
public class ShoppingCart : List<ShoppingItem>
{
    public ShoppingCart()
    {
    }
}
```

ShoppingItem is a custom class that describes a purchased product. It contains a few properties—product ID, product name, price per unit, and quantity purchased. The shopping cart is stored in the session state and exposed through a pagewide property named MyShoppingCart:

```csharp
protected ShoppingCart MyShoppingCart
{
    get
    {
        object o = Session["ShoppingCart"];
        if (o == null) {
            InitShoppingCart();
            return (ShoppingCart) Session["ShoppingCart"];
        }
        return (ShoppingCart) o;
    }
}
```
private void InitShoppingCart()
{
    ShoppingCart cart = new ShoppingCart();
    Session["ShoppingCart"] = cart;
}

At its core, the goal of AddToShoppingCart is merely that of creating a ShoppingItem object filled with the information of the clicked product. How would you retrieve that information?

As you can see, the GridView stores the index of the clicked row in the CommandArgument property of the GridViewCommandEventArgs structure. This information is necessary but not sufficient for our purposes. We need to translate that index into the key of the product behind the grid’s row. Better yet, we need to translate the grid row index into a data set index to retrieve the data item object rendered in the clicked grid’s row.

The DataKeyNames property of the GridView indicates the names of the data fields to persist in the view state to be retrieved later during postback events, such as RowCommand. Implemented as a string array, DataKeyNames is the GridView’s counterpart of the DataKeyField of DataGrid controls. It carries the value of the primary key for a displayed row in a DataGrid and a slew of properties for a GridView.

<asp:GridView ID="GridView1" runat="server"
  DataSourceID="SqlDataSource1"
  DataKeyNames="productid,productname,unitprice" ... />

How many fields should you list in DataKeyNames? Consider that every field you list there takes up some view-state space. On the other hand, if you limit yourself to storing only the primary key field, you need to run a query to retrieve all the data you need. Which approach is better depends on what you really need to do. In our sample scenario, we need to make a copy of a product that is already cached in the Web server’s memory. There’s no need to run a query to retrieve data we already know. To fill a ShoppingItem object, you need the product ID, name, and unit price:

private void AddToShoppingCart(int rowIndex)
{
    DataKey data = GridView1.DataKeys[rowIndex];
    ShoppingItem item = new ShoppingItem();
    item.NumberOfItems = 1;
    item.ProductID = (int) data.Values["productid"];
    item.ProductName = data.Values["productname"].ToString();
    item.UnitPrice = (decimal) data.Values["unitprice"];  
    MyShoppingCart.Add(item);
    ShoppingCartGrid.DataSource = MyShoppingCart;
    ShoppingCartGrid.DataBind();
}
The values of the fields listed in `DataKeyNames` are packed in the `DataKeys` array—an old acquaintance for `DataGrid` developers. `DataKeys` is an array of `DataKey` objects. `DataKey`, in turn, is a sort of ordered dictionary. You access the values of the persisted fields through the `Values` collection as shown in the preceding code.

For user-interface purposes, the contents of the shopping cart are bound to a second `GridView` so that users can see what’s in their basket at any time. The binding takes place through the classic `DataSource` object. Look back to Figure 11-3 for a view of this feature.

Caution

Each grid row gets bound to a data item—a row from the data source—only when the control is rendered out. A postback event such as `RowCommand` fires before this stage is reached. As a result, the `DataItem` property of the clicked `GridViewRow` object—where the data we need is expected to be—is inevitably null if accessed from within the `RowCommand` handler. That’s why you need `DataKeyNames` and the related `DataKeys` properties.

Selecting a Given Row

A more general mechanism to select clicked rows can be implemented through a special command button—the select button. As with delete and edit buttons, you bring it on by setting the `AutoGenerateSelectButton` Boolean property. To fully take advantage of the selection feature, it is recommended that you also add a style for selected rows:

```xml
<asp:GridView ID="GridView1" runat="server" ...>
    <SelectedRowStyle BackColor="cyan" />
    ...
</asp:GridView>
```

When users click a select-enabled button, the page receives a more specific `SelectedIndexChanged` event. Some properties such as `SelectedIndex`, `SelectedRow`, and `SelectedDataKey` are updated too. For completeness, note that when a row is selected the page first receives a `RowCommand` event, and later it is reached by the `SelectedIndexChanged` event. When `RowCommand` fires, though, none of the select properties is updated yet.

The following code shows how to rewrite the previous example to add to the cart the product being selected:

```csharp
protected void GridView1_SelectedIndexChanged(object sender, EventArgs e)
{
    AddToShoppingCart();
}
private void AddToShoppingCart()
{
    DataKey data = GridView1.SelectedDataKey;
    AddToShoppingCart(object sender, EventArgs e)
    { 
        AddToShoppingCart();
    }
private void AddToShoppingCart()
{ 
    DataKey data = GridView1.SelectedDataKey;
```
Chapter 11 Creating Bindable Grids of Data

... for the task—RowCreated and RowDataBound. The former is fired when any grid row is being created—whether it's a...

Row Customization

Want a quick example of why it's often important to render grid rows in a customized way? Take a look at Figure 11-14. The user just added to the cart a product that has been discontinued. Wouldn't it be nice if you could disable any rows matching certain criteria or, more simply, customize the row layout according to runtime conditions? Let's see how to do it.

There are two GridView events that are essential for the task—RowCreated and RowDataBound. The former is fired when any grid row is being created—whether it's a...
header, footer, item, alternating item, pager, and so on. The latter fires when the newly created row is bound to its data item—that is, the corresponding record in the bound data source. The **RowDataBound** event is not fired for all rows in the grid, but only for those that represent bound items. No event fires for the header, footer, and pager.

As a first example, let's see how to disable the Select link for rows where the `Discontinued` field returns `true`. In this case, you need a handler to **RowDataBound** because the required customization depends on the values on the bound data row. As mentioned, this information is not available yet when **RowCreated** fires.

```csharp
void GridView1_RowDataBound(object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        object dataItem = e.Row.DataItem;
        bool discontinued = (bool) DataBinder.Eval(dataItem, "discontinued");
        e.Row.Enabled = !discontinued;
    }
}
```

In general, you start by checking the type of the row. To be precise, this test is not strictly necessary for a **RowDataBound** event, which fires only for data rows. The data item—that is, the corresponding record—is retrieved through the **DataItem** property of the **GridViewRow** object. Next, you retrieve the field of interest and apply your logic. You might not know in advance the type of the data object bound to the row. The **DataBinder.Eval** method is a generic accessor that works through reflection and regardless of the underlying object. If you want to disable the whole row (and contained controls), you can turn off the `Enabled` property of the grid row object. To access a particular control, you need to find your way in the grid’s object model. Here’s how to access (and disable) the Select link alone:

```csharp
((WebControl)e.Row.Cells[0].Controls[0]).Enabled = !discontinued;
```

This code works because in the sample grid the Select link is always the first control in the first cell of each data row. Figure 11-15 shows the previous product list with discontinued products disabled.

Once you gain access to the grid row object model, you can do virtually whatever you want.
Chapter 11 Creating Bindable Grids of Data

In this chapter, we examined the grid controls available in ASP.NET—DataGrid and the newer GridView. Grids are a type of component that all Web applications need to employ in one shape or another. All Web applications, in fact, at a certain point of their life cycle are called to display data. More often than not, this data is in tabular format.

As long as the data to be displayed can be articulated in rows and columns, a grid is ideal for displaying it. Such controls provide facilities to select and edit single rows, page through a bound data source, and sort views. In addition, you can customize all the cells in a column by using any data-bound template made of any combination of HTML and ASP.NET text. To top it off, a fair number of events signal to user applications the key events in the control's life cycle.

The in-place editing feature is a piece of cake to use, as it is powerful and easy to configure. Even though this type of editing—designed to resemble Excel worksheets—is not appropriate for all applications and pages, as long as you can functionally afford the feature, in-place editing can save you a lot of coding and increase productivity by at least one order of magnitude.

Why are there two grid controls in ASP.NET 2.0 and beyond? Let’s state the answer clearly—the DataGrid control is supported mostly for backward compatibility. If you are writing a new ASP.NET application, choosing to use the GridView is a no-brainer. The GridView has a newer and more effective design and totally embraces the data-binding model of newest versions.
Part II Adding Data in an ASP.NET Site

ASP.NET. The key shortcoming of ASP.NET 1.x data binding is that it requires too much code for common, relatively boilerplate operations. This has been addressed with the introduction of data source controls. But data source controls require richer data-bound controls that are capable of working with the new model. This explains why ASP.NET now offers a brand-new control—the GridView—rather than just enhancing the existing DataGrid.

In the upcoming chapters, we’ll cover controls named ListView, DetailsView, and FormView—the perfect complements to the GridView. The latter two controls, in particular, fill another hole in the ASP.NET data toolbox, as they offer a smart interface for displaying individual records.

Just The Facts

- ASP.NET comes with two grid controls—DataGrid and GridView. The DataGrid works like in ASP.NET 1.x, whereas the GridView has a newer and more effective design and totally embraces the new data-binding model of ASP.NET.
- The GridView supports more column types including check box and image columns.
- The GridView provides paging, sorting, and editing capabilities, and it relies on the bound data source control for effective implementation. If bound to an enumerable data source object (ASP.NET 1.x-style binding), it behaves like a DataGrid control.
- If bound to a SqlDataSource control, the GridView heavily relies on the capabilities of the DataSet capabilities for paging and sorting data in memory.
- If bound to a LinqDataSource control, the GridView relies on the capabilities of the specified LINQ command.
- If bound to an ObjectDataSource control, the GridView requires a fully fledged DAL that contains any custom logic for paging, sorting, and, sometimes, caching.
- Compared to the DataGrid control, the GridView provides an extended eventing model: pre/post pairs of events, the possibility of canceling ongoing operations, and more events.
- To retrieve information about a clicked row, you use the CommandArgument property of the event data structure to get the index and the newest version of the DataKeys collection to access selected field on the data item. With DataGrid, you can select only the primary key field and need to run a query to access row data in drill-down scenarios.