Programming Microsoft
ASP.NET 3.5
Dino Esposito
# Table of Contents

- Chapter 4. ASP.NET Core Server Controls................................................................. 1
  - Generalities of ASP.NET Server Controls................................................................................................................................. 2
  - HTML Controls.................................................................................................................................................................................. 13
  - Web Controls.................................................................................................................................................................................. 29
  - Validation Controls........................................................................................................................................................................ 44
  - Conclusion.................................................................................................................................................................................. 58
ASP.NET pages are made of code, markup tags, literal text, and server controls. Based on the request, the server controls generate the right markup language. The ASP.NET runtime combines the output of all controls and serves the client a page to display in a browser. The programming richness of ASP.NET springs from the wide library of server controls that covers the basic tasks of HTML interaction—for example, collecting text through input tags—as well as more advanced functionalities such as calendaring, menus, tree views, and grid-based data display.

Key to ASP.NET control programming is the runat attribute. If a tag in the .aspx source is declared without the runat attribute, it is considered plain text and is output verbatim. Otherwise, the contents of the tag are mapped to a server control and processed during the page life cycle. Back in Chapter 1, we identified two main families of server controls—HTML server controls and Web server controls. HTML controls map to HTML tags and are implemented through server-side classes whose programming interface faithfully represents the standard set of attributes for the corresponding HTML tag. Web controls, in turn, are a more abstract library of controls in which adherence of the proposed API to HTML syntax is much less strict. As a result, Web and HTML controls share a large common subset of functionalities and, in spite of a few exceptions, we could say that Web controls, functionally speaking, are a superset of HTML controls. Web controls also feature a richer development environment with a larger set of methods, properties and events, and they participate more actively in the page life cycle.

As we’ll see in more detail in the following pages, a second and more thoughtful look at the characteristics of the server controls in ASP.NET reveals the existence of more than just two families of controls. In real-world ASP.NET applications, you’ll end up using controls from at least the following functional categories: HTML controls, core Web controls, validation controls, data-bound controls, user controls, mobile controls, and custom controls. Validation controls are a special subset of Web controls and deserve to be treated in a separate section.
Data-bound controls refer to data binding and therefore to the control's capability of connecting some of its properties to particular data sources. Hence, data-bound controls deserve a section of their own because of the difference in how they're used. User controls are visual aggregates of existing Web and HTML controls that appear as individual, encapsulated, programmable controls to external callers. Mobile controls are used when creating Web applications that target mobile devices. Custom controls refer to server controls you create entirely with code (not visually, as with a user control) that derive from a base control class.

In this chapter, we'll cover HTML controls, Web controls, and validation controls. Data-bound controls will be covered in Chapter 9. User controls, mobile controls, and custom controls find their place in my book — *Programming Microsoft ASP.NET 2.0 Applications: Advanced Topics* (Microsoft Press, 2006), which is written for advanced users as a companion book to this one. As mentioned, the content of my *Advanced Topics* book is not significantly affected by the release of ASP.NET 3.5.

### Generalities of ASP.NET Server Controls

All ASP.NET server controls, including HTML and Web controls plus any custom controls you create or download, descend from the `Control` class. The class is defined in the `System.Web.UI` namespace and, as we discussed in Chapter 3, it also is the foundation of all ASP.NET pages. The `Control` class is declared as follows:

```csharp
public class Control : IComponent, IDisposable, IParserAccessor, IUrlResolutionService, IDataBindingsAccessor, IControlBuilderAccessor, IControlDesignerAccessor, IExpressionsAccessor
```

The `IComponent` interface defines the way in which the control interacts with the other components running in the common language runtime (CLR), whereas `IDisposable` implements the common pattern for releasing managed objects deterministically. Table 4-1 explains the role of the other interfaces that the `Control` class implements.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>IControlBuilderAccessor</td>
<td>Internal use interface; provides members to support the page parser in building a control and the child controls it contains. <em>Not available in ASP.NET 1.x.</em></td>
</tr>
<tr>
<td>IControlDesignerAccessor</td>
<td>Internal use interface; provides members to make the control interact with the designer. <em>Not available in ASP.NET 1.x.</em></td>
</tr>
<tr>
<td>IDataBindingsAccessor</td>
<td>Makes the control capable of supporting data-binding expressions at design time.</td>
</tr>
<tr>
<td>IExpressionsAccessor</td>
<td>Internal use interface; defines the properties a class must implement to support collections of expressions. <em>Not available in ASP.NET 1.x.</em></td>
</tr>
</tbody>
</table>
### Interface Goal

**IParserAccessor**
Enables the control to work as the container of child controls and to be notified when a block of child markup is parsed.

**IUrlResolutionService**
Provides members to resolve relative URLs both at runtime and design time. **Not available in ASP.NET 1.x.**

The **IDataBindingsAccessor** interface defines a read-only collection—the **DataBindings** property—that contains all the data bindings for the controls available to rapid application development (RAD) designers such as Microsoft Visual Studio 2008. Note that the collection of data bindings exist only at design time and, as such, is useful only if you write a RAD designer for the control.

### Properties of the Control Class

The properties of the **Control** class have no user interface-specific features. The class, in fact, represents the minimum set of functionalities expected from a server control. The list of properties for the **Control** class is shown in Table 4-2.

**TABLE 4-2 Properties Common to All Server Controls**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BindingContainer</strong></td>
<td>Gets the control that represents the logical parent of the current control as far as data binding is concerned. <strong>Not available in ASP.NET 1.x.</strong></td>
</tr>
<tr>
<td><strong>ClientID</strong></td>
<td>Gets the ID assigned to the control in the HTML page. The string is a slightly different version of the UniqueID property. UniqueID can contain the dollar symbol ($), but this symbol is not accepted in ClientID and is replaced with the underscore (_).</td>
</tr>
<tr>
<td><strong>Controls</strong></td>
<td>Gets a collection filled with references to all the child controls.</td>
</tr>
<tr>
<td><strong>EnableTheming</strong></td>
<td>Indicates whether themes apply to the control. <strong>Not available in ASP.NET 1.x.</strong></td>
</tr>
<tr>
<td><strong>EnableViewState</strong></td>
<td>Gets or sets whether the control should persist its view state—and the view state of any child controls across multiple requests—to the configured medium (for example, HTML hidden field, session state, and server-side databases or files).</td>
</tr>
<tr>
<td><strong>ID</strong></td>
<td>Gets or sets the name that will be used to programmatically identify the control in the page.</td>
</tr>
<tr>
<td><strong>NamingContainer</strong></td>
<td>Gets a reference to the control's naming container. The naming container for a given control is the parent control above it in the hierarchy that implements the <strong>INamingContainer</strong> interface. If no such control exists, the naming container is the host page.</td>
</tr>
<tr>
<td><strong>Page</strong></td>
<td>Gets a reference to the <strong>Page</strong> instance that contains the control.</td>
</tr>
<tr>
<td><strong>Parent</strong></td>
<td>Gets a reference to the parent of the control in the page hierarchy.</td>
</tr>
</tbody>
</table>
Part I Building an ASP.NET Page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Site</td>
<td>Gets information about the container that hosts the current control when rendered on a design surface. For example, you use this property to access the Visual Studio designer when the control is being composed in a Web form.</td>
</tr>
<tr>
<td>SkinID</td>
<td>Gets or sets the name of the skin to apply to the control. A skin is a particular subset of attributes in a theme. Not available in ASP.NET 1.x.</td>
</tr>
<tr>
<td>TemplateControl</td>
<td>Gets a reference to the template that contains the current control. Not available in ASP.NET 1.x.</td>
</tr>
<tr>
<td>TemplateSourceDirectory</td>
<td>Gets the virtual directory of the host page.</td>
</tr>
<tr>
<td>UniqueID</td>
<td>Gets a hierarchically qualified ID for the control.</td>
</tr>
<tr>
<td>Visible</td>
<td>Gets or sets whether ASP.NET has to render the control.</td>
</tr>
</tbody>
</table>

The **Control** class is the ideal base class for new controls that have no user interface and don't require style information.

Identifying a Server Control

The client ID of a control is generated from the value of the **UniqueId** property—the truly server-side identifier that ASP.NET generates for each control. The contents of the **ClientId** property differ from **UniqueId** simply in that all occurrences of the dollar symbol ($), if any, are replaced with the underscore (_). Dollar symbols in the **UniqueId** string are possible only if the control belongs to a naming container different from the page.

ASP.NET generates the value for the **UniqueId** property based on the value of the **ID** property that the programmer indicates. If no **ID** has been specified, ASP.NET auto-generates a name such as _ctlX, where X is a progressive 0-based index. If the control's naming container is the host page, **UniqueId** simply takes the value of **ID**. Otherwise, the value of **ID** is prefixed with the string representing the naming container and the result is assigned to **UniqueId**.

Naming Containers

A naming container is primarily a control that acts as a container for other controls. In doing so, the naming container generates a sort of virtual namespace so that ASP.NET roots the actual ID of contained controls in the ID of the naming container. To fully understand the role and importance of naming containers, consider the following example.

Imagine you have a composite control, such as a user control, that includes a child control like a button. Entirely wrapped by the user control, the button is not directly accessible by the page code and can't be given a distinct and per-instance ID. In the end, the ID of the button is hard-coded in the outermost control that creates it. What happens when two or more instances of the composite control are placed on a page? Are you going to have two
button child controls with the same ID? This is exactly what will happen unless you configure the composite control to be a naming container.

The importance of naming containers doesn’t end here. Imagine you have an instance of a composite control named `Control1`. Imagine also that the embedded button is named `Trigger`. The full name of the child button will be `Control1$Trigger`. Suppose you click on the button and cause the page to post back. If the name of the posting control contains the `$` symbol, the ASP.NET runtime recognizes a known pattern: tokenize the name and locate the postback control correctly, no matter its depth in the page tree.

On the other hand, if the button is contained in a control not marked to be a naming container, the ID of the clicked button is not prefixed and will simply be, say, `Trigger`. In this case, the ASP.NET runtime will look for it as a direct child of the form. The search will obviously fail—the button is a child of a top-level control—and the postback event will pass unnoticed.

**Note** Starting with version 2.0, ASP.NET uses the dollar (`$`) symbol to separate the various parts to form the ID of a control rooted in a naming container. In ASP.NET 1.x, the colon (`:`) symbol is used for the same purpose.

**Binding Containers**

Starting with ASP.NET 2.0, a new kind of container is introduced—the binding container. The binding container—the `BindingContainer` property—indicates which control in the page hierarchy represents the parent of a control as far as data binding is concerned. In other words, the binding container is the control that receives bound data from the host (typically, the page) and that passes it down to child controls.

As you can easily imagine, binding and naming containers often coincide. The only exception is when the control is part of a template. In that case, the `NamingContainer` property is generally set to the physical parent of the control, namely a control in the template. `BindingContainer`, instead, will point to the control that defines the template.

**Visibility of a Server Control**

If you set `Visible` to `false`, ASP.NET doesn’t generate any markup code for the control. However, having `Visible` set to `false` doesn’t really mean that no path in the control’s code can output text. The control is still an active object that exposes methods and handles events. If a method, or an event handler, sends text directly to the output console through `Response.Write`, this text will be displayed to the user anyway. A control with the `Visible` attribute set to `false` is still part of the page and maintains its position in the control tree.
Methods of the Control Class

The methods of the Control class are listed and described in Table 4-3.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplyStyleSheetSkin</td>
<td>Applies the properties defined in the page style sheet to the control. The skin properties used depend on the SkinID property. Not available in ASP.NET 1.x.</td>
</tr>
<tr>
<td>DataBind</td>
<td>Fires the OnDataBinding event and then invokes the DataBind method on all child controls.</td>
</tr>
<tr>
<td>Dispose</td>
<td>Gives the control a chance to perform clean-up tasks before it gets released from memory.</td>
</tr>
<tr>
<td>Focus</td>
<td>Sets the input focus to the control. Not available in ASP.NET 1.x.</td>
</tr>
<tr>
<td>FindControl</td>
<td>Looks for the specified control in the collection of child controls. Child controls not in the Controls collection of the current controls—that is, not direct children—are not retrieved.</td>
</tr>
<tr>
<td>HasControls</td>
<td>Indicates whether the control contains any child controls.</td>
</tr>
<tr>
<td>RenderControl</td>
<td>Generates the HTML output for the control.</td>
</tr>
<tr>
<td>ResolveClientUrl</td>
<td>Use the method to return a URL suitable for use by the client to access resources on the Web server, such as image files, links to additional pages, and so on. Can return a relative path. The method is sealed and can’t be overridden in derived classes.</td>
</tr>
<tr>
<td>ResolveUrl</td>
<td>Resolves a relative URL to an absolute URL based on the value passed to the TemplateSourceDirectory property.</td>
</tr>
<tr>
<td>SetRenderMethodDelegate</td>
<td>Internal use method, assigns a delegate to render the control and its content into the parent control.</td>
</tr>
</tbody>
</table>

Each control can have child controls. All children are stored in the Controls collection, an object of type ControlCollection. This collection class has a few peculiarities. In particular, it post-processes controls that are added to, and removed from, the collection. When a control is added, its view state is restored if needed and view state tracking is turned on. When a control is removed, the Unload event is fired.

Events of the Control Class

The Control class also defines a set of base events that all server controls in the .NET Framework support.
TABLE 4-4 Events of a Server Control

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataBinding</td>
<td>Occurs when the DataBind method is called on a control and the control is binding to a data source.</td>
</tr>
<tr>
<td>Disposed</td>
<td>Occurs when a control is released from memory—the last stage in the control life cycle.</td>
</tr>
<tr>
<td>Init</td>
<td>Occurs when the control is initialized—the first step in the life cycle.</td>
</tr>
<tr>
<td>Load</td>
<td>Occurs when the control is loaded into the page. Occurs after Init.</td>
</tr>
<tr>
<td>PreRender</td>
<td>Occurs when the control is about to render its content.</td>
</tr>
<tr>
<td>Unload</td>
<td>Occurs when the control is unloaded from memory.</td>
</tr>
</tbody>
</table>

All server controls are rendered to HTML using the RenderControl method and, when this happens, the PreRender event is fired.

Other Features

Starting with ASP.NET 2.0, server controls gained some new features that are more architectural than related to programming capabilities. These features are the offspring of significant changes in the underpinnings of the controls.

Adaptive Rendering

Adaptive rendering is the process that enables controls to generate different markup for individual browsers. This result is obtained by delegating the generation of the markup to an external component—the adapter. When each control is about to render, it figures out its current adapter and hands the request over to that adapter. Nicely enough, a control adapter is a configurable component that you can declaratively unplug in any application to roll your own.

The selected adapter depends on the current browser. The adapter for a control is resolved by looking at the browser capabilities as configured in the ASP.NET browser database. If the browser record includes an adapter class for a given control, the class is instantiated and used. Otherwise, the default adapter for the control is used, which is an instance of the ControlAdapter class. The ControlAdapter class is a generic adapter and simply generates the markup for a control by calling the rendering methods on the control itself.

Note: The ASP.NET database used for storing browser information is not a real database. It is, instead, a list of text files with a .browser extension located under the ASP.NET installation folder on the Web server. The exact path is the following: %WINDOWS%\Microsoft.NET\Framework\[version]\CONFIG\Browsers. The data located in this folder is used to return browser capabilities.
A control holds a reference to the mapped adapter instance through the (protected) `Adapter` property. Each control has an associated adapter unless it is a composite control that defers to its children for rendering.

All ASP.NET controls have an entry point into the rendering engine in the `Render` method. Here’s the method’s signature:

```csharp
protected virtual void Render(HtmlTextWriter writer) {
...
}
```

The `Render` method ends up calling into an internal method whose implementation is nearly identical to the following pseudocode:

```csharp
void RenderControlInternal(HtmlTextWriter writer, ControlAdapter adapter) {
    if (adapter != null) {
        adapter.BeginRender(writer);
        adapter.Render(writer);
        adapter.EndRender(writer);
    } else {
        this.Render(writer);
    }
}
```

As you can see, if defined, a control adapter is used to generate the markup for the control. The adapter can be declaratively specified and is an external component that can be made to measure for your needs. Using an adapter to alter the markup of a given class of controls is an unobtrusive option that doesn’t require any changes to existing pages using the control. It only requires you to add a browser definition file.

Browser definition files have a `.browser` extension and contain definitions that apply to a specific browser. At run time, ASP.NET determines the browser being used, uses the configuration file to determine the capabilities of the browser, and based on that figures out how to render markup to that browser. Here's a snippet that illustrates how to register a control adapter for the `Menu` for whatever browsers the user will employ:

```xml
<browsers>
  <browser refID="Default">
    <controlAdapters>
      <adapter controlType="System.Web.UI.WebControls.Menu" adapterType="Core35.MenuAdapter" />
      ...
    </controlAdapters>
  </browser>
</browsers>
```
Saved to a .browser file, the preceding snippet is deployed to the App_Browsers folder of an ASP.NET application, version 2.0 or newer.

An adapter class looks like the following class:

```csharp
public class MenuAdapter : 
{
    ...
}
```

The class commonly overrides methods such as Init, RenderBeginTag, RenderEndTag, and RenderContents.

To write an adapter effectively, though, you must reasonably know a lot of details about the internal workings of the control you’re hooking up. For more information on the architecture of control adapters, you might want to take a look at http://msdn2.microsoft.com/en-us/library/67276kc5.aspx.

Note
The markup that too many ASP.NET server controls return makes excessive use of `<table>` tags (often nested) and limited use of CSS styling. Based on the community feedback, the ASP.NET team released a free toolkit to enable a few built-in controls to output CSS-friendly markup where the `<table>` tag is not used or used less and in accordance with XHTML rules. The CSS Control Adapter Toolkit (CSSCAT) can be downloaded from http://www.asp.net/cssadapters. It comes with full source code and a permissive license that allows for unlimited further customization of the code. CSSCAT is built atop the control adapter architecture that in ASP.NET 2.0, and newer versions, makes it possible for developers to unplug the default rendering engine to roll their own. For more information on the CSSCAT logic and internal architecture, pay a visit to http://www.asp.net/cssadapters/whitepaper.aspx.

Browser-Sensitive Rendering
In ASP.NET 2.0 and newer versions, you can declaratively assign a browser-specific value to all control properties. Here’s a quick example:

```xml
<asp:Button ID="Button1" runat="server" Text="I’m a Button"

    ie:Text=IE Button

    moz:Text=Firefox Button

/>  
```

The Text property of the button will contain “IE button” if the page is viewed through Internet Explorer and “Firefox button” if the page goes through Firefox. If another browser is used, the value of the unprefixed Text attribute is used. All properties you can insert in a tag...
declaration can be flagged with a browser ID. Each supported browser has a unique ID. As in the preceding code, ie is for Internet Explorer and mozilla is for Firefox. Unique IDs exist for various versions of Netscape browsers and mobile devices.

Browser-specific filtering is supported also for master pages. We’ll return to this feature in Chapter 6 and present a table with the most common browser IDs. However, browser IDs are interspersed in .browser files, which you can find at this path:

%windows%\Microsoft.NET\Framework\[version]\CONFIG\Browsers

XHTML Compliance

XHTML is a World Wide Web Consortium (W3C) standard that defines Web pages as XML documents. This approach guarantees that the elements in the pages are well formed and more forward-compatible with browsers in the near future. By default, the markup produced by ASP.NET controls conforms to the XHTML standard with very few exceptions. This compliance with standards produces a number of observable features in the final markup served to browsers. For example, each element either includes an explicit closing tag or is self-closing (with />) and is always enclosed in a container element. For example, the view state hidden field is surrounded by a <div> tag and the name attribute has been removed from the <form> element:

<form method="post" action="default.aspx" id="MainForm">
  <div>
    <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="..." />
  </div>
  ...
</form>

In addition, any script tags rendered into the page include an appropriate type attribute and are rendered in CDATA elements.

It’s clear that some of these changes might break existing old pages as you upgrade to ASP.NET 3.5. What if, say, you have a page that relies on the name attribute on the form? To smooth migration of ASP.NET 1.x pages, you can add the following setting to the web.config file, which forces ASP.NET to render controls as in ASP.NET 1.x:

<system.web>
  <XHTML11Conformance enableObsoleteRendering="true" />
</system.web>

The option to disable XHTML rendering is provided primarily to assist you in upgrading existing pages. You should not abuse it, as it might not be supported in future versions of ASP.NET. Moreover, you should be migrating to XHTML anyway; ASP.NET 3.5 just gives you one more reason to do it now, if possible.
The generation of XHTML-compliant output is guaranteed only for the vast majority of core ASP.NET server controls. Controls such as `HyperLink`, `BulletedList`, and `AdRotator` generate non-XHTML-compliant markup regardless of the settings you choose. `GridView` and `TreeView` controls are also at risk if they incorporate `HyperLinkColumn` and `TreeNode` components. You should avoid using these controls in pages where XHTML compliance is a strict requirement. If you make use of third-party controls, you should always check with the vendor to see whether they generate XHTML markup. Finally, note that ASP.NET is unable to fix XHTML errors that occur in the literal part of the pages. If your page contains static text or HTML elements, the responsibility of ensuring that they are XHTML compliant is entirely yours.

How can you make sure that a given page, or a given custom control, renders XHTML markup? You must use a service that runs the page and checks its output. For example, you can use the W3C Markup Validation Service at http://validator.w3.org. You can use the validator in two ways: by entering the URL of your page and having it request and check the page, or by uploading the page to the validator’s site.

### Themeable Controls

In the ASP.NET jargon, a **theme** is a named collection of property settings that can be applied to controls to make them look consistent across pages. You can apply theme settings to an entire Web site, to a page and its controls, or to an individual control. A theme is identified by name and consists of cascading style sheet (CSS) files, images, and control skins. A control **skin** is a text file that contains predefined values for some control properties. Applied together, these settings contribute to change the look and feel of the control and give the whole site a consistent (and, you hope, appealing) user interface. In addition, because themes are a sort of monolithic attribute, you can easily export that look from one application to the next. With themes enabled, if the developer adds, say, a `DataGrid` control to a page, the control is rendered with the default appearance defined in the currently selected theme.

Server controls can dynamically accept or deny theming through a Boolean property named `EnableTheming`, set to `true` by default. As a general rule, themes affect only properties that relate to the control’s appearance. Properties that explicitly specify a behavior or imply an action should not be made themeable. Each control has the power to state which properties are themeable and which are not. This happens at compile time through attributes—in particular, the `Themeable` attribute. We’ll return to themes in Chapter 6. I cover custom control development in *Programming Microsoft ASP.NET 2.0 Applications: Advanced Topics* (Microsoft Press, 2006).

### Control State

Some ASP.NET controls require that some state be kept across requests. Examples of this type of state information include the current page of a paged control and the current sort order of a sortable data control. In ASP.NET 1.x, there is only one container in which this data can...
Part I Building an ASP.NET Page

be stored—the view state. However, the view state is mainly designed to maintain settings set by the application and, more importantly, it can be turned off. What would happen to control-specific state in this case? For this reason, starting with ASP.NET 2.0 Microsoft introduced the notion of the "control state" and managed to keep it separate from the view state. So it's clear that control state is a vital piece of the control infrastructure.

Control state is a collection of critical view-state data that controls need to function. Because of its critical role, control state data is contained in separate member variables from normal view state and is not affected when view state is disabled. Unlike view state, control state requires extra implementation steps to use.

For one thing, each control needs to signal to the page that it requires control state. Next, there's no unique container to store data, such as ViewState, but the data can be retrieved from any object you want—arrays, collections, or a slew of instance variables. Each control persists and loads its control state using a pair of overridable methods, as shown here:

protected override object SaveControlState()
protected override void LoadControlState(object state)

Control state works similarly to view state and is saved and loaded at the same stage of the pipeline that view state is processed. Ultimately, control state is persisted in the same hidden field as the view state.

Input Focus

A useful feature that ASP.NET 1.x lacks is the ability to quickly assign the input focus to a particular control when the page is displayed. This feature can be coded in not much time by a seasoned developer and can be easily engineered into a company-wide framework for building controls and pages.

As we saw in Chapter 3, the Page class of ASP.NET 2.0 provides the SetFocus method to assign the input focus to any control you want. The following code shows how to set the focus to a TextBox control named txtLastName:

www.Page.Load(object sender, EventArgs e)
{
  if (!IsPostBack)
    SetFocus("txtLastName");
}

The SetFocus method caches the ID of the control and forces the Page class to generate ad hoc script code when the page is rendered. Each control can also reclaim the input focus for itself by calling its new Focus method. Starting with version 2.0, all ASP.NET controls benefit from this feature.
HTML Controls

At first sight, HTML server controls look like HTML tags except for the extra runat=server attribute. Although it’s true that they look the same, the additional runat attribute makes a huge difference. As mentioned, in ASP.NET by simply adding the runat attribute, you can bring to life otherwise-dead HTML text. Once transformed into a living instance of a server-side component, the original tag can be configured programmatically using an object-oriented approach. By design, HTML controls expose a set of methods and properties that carefully reflect the HTML syntax. For example, to set the default text of an input form field you use a property named Value instead of the more expressive Text. The name of the server control is determined by the value of the ID attribute. The following code snippet shows how to define a server-side input tag named lastName:

```html
<input runat="server" id="lastName" type="text" />
```

The tag declaration does not include an explicit and static value for the Value attribute, which can be configured programmatically as follows:

```csharp
void Page_Load(object sender, EventArgs e)
{
    lastName.Value = "Esposito";
}
```

After being processed by the ASP.NET runtime, the preceding declaration generates the following HTML code:

```html
<input name="myName" id="myName" type="text" value="Esposito" />
```

Notice that a server-side ID attribute expands to a pair of HTML attributes—Name and ID. Be aware that this happens for browser compatibility. In no way does this mean that on the server Name and ID can be interchangeably used to name the server instance of the control. The name of the server control instance is given by ID. If you specify both Name and ID on a server-side tag, the value assigned to Name will be silently overridden.

Generalities of HTML Controls

The .NET Framework provides predefined server controls for commonly used HTML elements such as `<form>`, `<input>`, and `<select>`, as well as for tables, images, and hyperlinks. All the predefined HTML server controls inherit from the same base class—the `HtmlControl` class. In addition, each control then provides its own set of specific properties and its own events.

Controls typically supply properties that allow you to manipulate the HTML attributes programmatically from within server code. HTML controls integrate well with data-binding and the ASP.NET state maintenance, and they also provide full support for postback events and client scripting. For example, for a button that gets clicked, you can have some JavaScript code...
Part I Building an ASP.NET Page

code running on the client responding to the onclick event as well as some code that handles
the event on the server if the page posts back as the result of that event.

HTML controls are defined in the System.Web.UI.HtmlControls namespace. Most, but not
all, HTML tags have a direct class counterpart in the .NET Framework. HTML elements that
don’t map to a made-to-measure class are rendered through the HtmlGenericControl class
and have attributes set using generic collections rather than direct properties. Generic con-
tROLS include <iframe>, <hr>, <font>, and <body>. in general, you should bear in mind that
every element that can appear in an HTML page can be marked as runat=’server’ and pro-
grammed and styled on the server.

The HtmlControl Base Class

The HtmlControl class inherits from Control and defines the methods, properties, and events
common to all HTML controls. Actually, many properties and all methods and events are
simply inherited from the base class. Table 4-5 shows the list of properties specific to HTML
controls.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
<td>Gets a collection object representing all the attributes set on the control with the corresponding value</td>
</tr>
<tr>
<td>Disabled</td>
<td>Gets or sets a Boolean value, which indicates whether the HTML control is disabled</td>
</tr>
<tr>
<td>Style</td>
<td>Gets a collection object representing all CSS properties applied to the control</td>
</tr>
<tr>
<td>TagName</td>
<td>Gets the name of the HTML tag behind the control</td>
</tr>
</tbody>
</table>

A disabled HTML server control is visible and always gets generated as HTML code. If the
Disabled property is set to true, the disabled HTML attribute is inserted in the HTML output
for the control. As mentioned earlier, if the Visible property is set to false, HTML is not gener-
ated for the control.

Note The disabled HTML attribute applies only to HTML input elements. It has no effect on, say, anchor tags.

Working with HTML Attributes

Each HTML control features more properties than those listed in Table 4-5. Properties of
HTML server controls map to HTML attributes, and the values assigned to the properties are
replicated in the HTML output. For controls that don’t have an HTML direct counterpart, the
Attributes collection is used to set attributes on the resulting HTML tag. This collection can
also be used to set properties not mapped by the control's interface and, if needed, to define custom HTML attributes. Any content of the Attributes collection is managed as a string.

Given the following HTML code snippet, let's see how to programmatically set some attributes on the <body> tag:

```html
<script>
    function Init() {
        alert("Hello");
    }
</script>
<script runat=server language="C#">
    void Page_Load(object sender, EventArgs e) {
        theBody.Attributes["onload"] = "Init()";
    }
</script>
<html>
<body runat="server" id="theBody">
</body>
</html>
```

You bind a JavaScript script to the onload attribute of the <body> tag. The resulting HTML code that the browser displays is as follows:

```html
<script>
    function Init() {
        alert("Hello");
    }
</script>
<html>
<body id="theBody" onload="Init()"/>
</body>
</html>
```

The Attributes property is rendered through a special type of class named AttributeCollection. In spite of the name, the content of the class is not directly enumerable using the for...each statement because the IEnumerable interface is not supported. The AttributeCollection class provides ad hoc methods to render attributes of a text writer object and to add and remove elements. Interestingly, if you add an attribute named Style, the class is smart enough to reroute the assigned content to the Style collection.

Hierarchy of HTML Controls

Most HTML controls can be grouped into two main categories—container and input controls. A few controls, though, cannot be easily catalogued in either of the two groups. They are HtmlImage, HtmlLink, HtmlMeta, and HtmlTitle, and they are the ASP.NET counterpart of the <img>, <link>, <meta>, and <title> tags. Figure 4-1 shows the tree of HTML controls.
FIGURE 4-1 A diagram that groups all HTML controls by looking at their base class. Controls in boldface type require ASP.NET version 2.0 or later.

The input controls category includes all possible variations of the `<input>` tag, from submit buttons to check boxes and from text fields to radio buttons. The container controls category lists anchors, tables, forms, and in general, all HTML tags that might contain child elements.

**HTML Container Controls**

The base class for container controls is the `HtmlContainerControl` class, which descends directly from `HtmlControl`. The HTML elements addressed by this tag are elements that must have a closing tag—that is, forms, selection boxes, and tables, as well as anchors and text areas. Compared to the `HtmlControl` class, a container control features a couple of additional string properties—`InnerHtml` and `InnerText`.

Both properties manipulate the reading and writing of literal content found between the opening and closing tags of the element. Note that you cannot get the inner content of a control if the content includes server controls. `InnerHtml` and `InnerText` work only in the presence of all literal content. The tag itself is not considered for the output. Unlike `InnerText`, though, `InnerHtml` lets you work with HTML rich text and doesn’t automatically encode and decode text. In other words, `InnerText` retrieves and sets the content of the tag as plain text, whereas `InnerHtml` retrieves and sets the same content but in HTML format.

Table 4-6 lists the HTML container controls defined in ASP.NET.
### TABLE 4-6 HTML Container Controls

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HtmlAnchor</td>
<td>Represents an HTML anchor—specifically, the <code>&lt;a&gt;</code> tag.</td>
</tr>
<tr>
<td>HtmlButton</td>
<td>Represents the <code>&lt;button&gt;</code> tag. The <code>&lt;button&gt;</code> element is defined in the HTML 4.0 specification and supported only in Internet Explorer version 4.0 and later.</td>
</tr>
<tr>
<td>HtmlForm</td>
<td>Represents the <code>&lt;form&gt;</code> tag, but can be used only as a container of interactive server controls on a Web page. Cannot really be used to create HTML forms programmable on the server.</td>
</tr>
<tr>
<td>HtmlGenericControl</td>
<td>Represents an HTML tag for which the .NET Framework does not provide a direct class. Sample tags include <code>&lt;font&gt;</code>, <code>&lt;hr&gt;</code>, and <code>&lt;iframe&gt;</code>. You program these controls by using the Attributes collection and set attributes indirectly.</td>
</tr>
<tr>
<td>HtmlHead</td>
<td>Represents the <code>&lt;head&gt;</code> tag, and allows you to control meta tags, the style sheet, and the page title programmatically. Not available in ASP.NET 1.x.</td>
</tr>
<tr>
<td>HtmlSelect</td>
<td>Represents the <code>&lt;select&gt;</code> tag—that is, an HTML selection box.</td>
</tr>
<tr>
<td>HtmlTable</td>
<td>Represents an HTML table—specifically, the <code>&lt;table&gt;</code> tag.</td>
</tr>
<tr>
<td>HtmlTableCell</td>
<td>Represents the <code>&lt;td&gt;</code> HTML tag—that is, a cell in a table.</td>
</tr>
<tr>
<td>HtmlTableRow</td>
<td>Represents the <code>&lt;tr&gt;</code> HTML tag—that is, a row in a table.</td>
</tr>
<tr>
<td>HtmlTextArea</td>
<td>Represents a multiline text box, and maps the <code>&lt;textarea&gt;</code> HTML tag.</td>
</tr>
</tbody>
</table>

Note that the `HtmlButton` control is different than `HtmlInputButton`, which represents the button variation of the `<input>` tag. The `HtmlButton` control represents the HTML 4.0–specific `<button>` tag. We’ll say more about buttons in the next section while discussing the Web controls.

Server-side forms play a key role in the economy of ASP.NET applications, as they are the means for implementing postbacks and guaranteeing state maintenance. For this reason, the `HtmlForm` control is not simply a form element you can program on the server. In particular, the `HtmlForm` hides the `Action` property and cannot be used to post content to a page different than the content that generated the HTML for the browser. We will cover HTML forms in great detail in Chapter 5.

### Managing Header Information

An instance of the `HtmlHead` control is automatically created if the page contains a `<head>` tag marked with the attribute `runat=server`. Note that this setting is the default when you add a new page to a Visual Studio 2008 Web project, as shown in the following snippet:

```html
<head runat="server">
    <title>Untitled Page</title>
</head>
```
The header of the page is returned through the new `Header` property of the `Page` class. The property returns `null` if the `<head>` tag is missing, or if it is present but lacks the `runat` attribute.

The `HtmlHead` control implements the `IPageHeader` interface, which consists of three collection properties—`Metadata`, `LinkedStylesheet`, and `Stylesheet`—and a string property—`Title`. The `Metadata` property is a dictionary that collects all the desired child `<meta>` tags of the header:

```csharp
Header.Metadata.Add("CODE-LANGUAGE", "C#");
```

The code results in the following markup:

```html
<meta name="CODE-LANGUAGE" content="C#" />
```

To express other common metadata such as `Http-Equiv`, you can resort to the newest `HtmlMeta` control, as shown here:

```csharp
void Page_Init(object sender, EventArgs e)
{
    HtmlMeta meta = new HtmlMeta();
    meta.HttpEquiv = "refresh";
    meta.Content = Int32.Parse(TextBox1.Text).ToString();
    ((Control)Header).Controls.Add(meta);
}
```

The preceding code creates a `<meta>` tag dynamically and adds it to the `<head>` section of the page during the initialization phase. You can also manipulate an existing `<meta>` programmatically, as long as it is flagged with the `runat` attribute.

**Tip** In Internet Explorer only, the `<meta>` tag can be used to smooth the transition from one page to the next, and also when you move back to a previously visited page. When navigating from page to page in the browser, the current page usually disappears all of a sudden and the new page shows up in its place. By using the following two meta tags, you can make them fade away smoothly:

```html
<meta http-equiv="Page-Enter"
    content="progid:DXImageTransform.Microsoft.Fade(duration=.5)" />
<meta http-equiv="Page-Exit"
    content="progid:DXImageTransform.Microsoft.Fade(duration=.5)" />
```

Needless to say, the tags can be created and managed programmatically in ASP.NET 2.0 and newer versions.
To add a `<meta>` tag programmatically in ASP.NET 1.x, you must resort to a trick. You create the string as a literal control and add it to the `Controls` collection of the header.

```csharp
string meta = "<meta http-equiv='refresh' content='3' />";
LiteralControl equiv = new LiteralControl(meta);
((Control) Header).Controls.Add(equiv);
```

Notice that you must explicitly cast the object returned by the `Header` property to `Control`. This is because the `Header` property is declared as type `IPageHeader`, which has no `Controls` property defined.

To link a style sheet file, you use the following code:

```csharp
Header.LinkedStyleSheets.Add("MyStyles.css");
```

Alternatively, you can resort to the `HtmlLink` control. The `HtmlLink` control represents the `<link>` element. Unlike `<a>`, the `<link>` tag can appear only in the `<head>` section of a document, although it might appear any number of times.

Finally, the `HtmlHead` control features the `Title` property, through which you can retrieve and set the title of the page:

```csharp
Header.Title = "This is the title";
```

Note that this property returns the correct page title only if the `<title>` tag is correctly placed within the `<head>` tag. Some browsers, in fact, are quite forgiving on this point and allow developers to define the title outside the header. To manipulate the `<title>` tag independently from the header, use the `HtmlTitle` control and mark the `<title>` tag with the `runat` attribute.

**Navigating to a URL**

The `HtmlAnchor` class is the programmatic way of accessing and configuring the `<a>` tag. With respect to the other container controls, the `HtmlAnchor` class provides a few extra properties such as `HRef`, `Name`, `Target`, and `Title`. The `HRef` property sets the target of the hyperlink and can be used to navigate to the specified location. The `Name` property names a section in the ASP.NET page that can be reached from anywhere on the same page through `#`-prefixed `HRefs`. The following code demonstrates a bookmarked anchor named `MoreInfo`:

```csharp
<a name="MoreInfo" />
```

This anchor can be reached using the following hyperlink:

```csharp
<a href="#MoreInfo">Get More Info</a>
```

The `Target` property identifies the target window or the frame where the linked URL will be loaded. Common values for `Target` are `self`, `_top`, `blank`, and `parent`, as well as any other name that refers to a page-specific frame. Although the feature is mostly browser depen-
dent, you should always consider these special names as lowercase. Finally, the Title property contains the text that virtually all browsers display as a ToolTip when the mouse hovers over the anchor’s area.

**Handling Events on the Server**

In addition to being used for navigating to a different page, the anchor control—as well as the HtmlButton control—can be used to post back the page. Key to this behavior is the ServerClick event, which lets you define the name of the method that will handle, on the server, the event generated when the user clicks the control. The following code demonstrates an anchor in which the click event is handled on both the client and server:

```html
<a runat=server onclick="Run()" onserverclick="DoSomething">Click</a>
```

The onclick attribute defines the client-side event handler written using JavaScript; the onserverclick attribute refers to the server-side code that will run after the page posts back. Of course, if both event handlers are specified, the client-side handler executes first before the post back occurs.

**The HtmlSelect Control**

The HtmlSelect control represents a list of options from which you choose one or more. You control the appearance and behavior of the control by setting the Size and Multiple properties. The Size property specifies the number of rows to be displayed by the control, whereas the Multiple property indicates whether more than one item can be selected in the control. Internal items are grouped in the Items collection, and each element is represented by a ListItem object. Interestingly, the ListItem class is not defined in the HtmlControls namespace but lives instead in the WebControls namespace. To specify the text for each selectable item, you can either set the Text property of the ListItem or simply define a series of <option> tags within the opening and closing tags of the <select> element.

By default, the HtmlSelect control shows up as a drop-down list. However, if multiple selections are allowed or the height is set to more than one row, the control is displayed as a list box. The index of the selected item in a single-selection control is returned through the SelectedIndex property. If the multiple selection is enabled, you just loop through the Items collection and check the Selected property on individual list items.

The HtmlSelect control supports data binding through additional properties. The DataSource property lets you set the data source, which can be any .NET object that implements the IEnumerable interface. If the data source contains multiple bindable tables (for example, a Dataset object), by using theDataMember property you can choose a particular one. Finally, the DataTextField and DataValueField properties are used to bind the list item’s Text and Value properties to columns in the data source. (We’ll cover data binding in Chapter 9.)
HTML Tables

In ASP.NET, HTML tables provide a minimum set of functions when rendered using the `HtmlTable` control. In most cases, you don't need to use server-side tables because you typically rely on richer list and grid controls to do the job of displaying tables or records. So you resort to tables when you need to define a fixed layout for graphical elements of the page, but this is not a feature that requires a server-side table.

Until ASP.NET 3.5, server-side tables were not as powerful as pure HTML tables—which are created by using the `<table>` tag. The main limitation was that the `HtmlTable` class did not support HTML elements such as `<caption>`, `<col>`, `<colgroup>`, `<tbody>`, `<thead>`, and `<tfoot>`. If you used these elements in your ASP.NET 2.0 code, no run-time exception or compile error was ever thrown, but those elements were silently removed from the HTML code being generated.

In ASP.NET 3.5, table sections like `<tbody>`, `<thead>`, and `<tfoot>` are fully supported by all flavors of table controls—both `HtmlTable` and `Table`—the control behind the `<asp:Table>` server tag.

The `HtmlTextArea` Control

The `HtmlTextArea` control corresponds to the `<textarea>` HTML element and allows you to programmatically create and configure a multiline text box. The `HtmlTextArea` class provides the `Rows` and `Cols` properties to control the number of rows and columns of the text box. The `Value` property can be used to assign some text to display in the control area.

The `HtmlTextArea` class also provides a `ServerChange` event that fires during a postback and allows you to validate on the server the data contained in the control. Note that the `HtmlTextArea` control does not fire the event itself and does not directly cause the page to post back. Rather, when the page posts back in response to a click on a link or submit button, the `HtmlTextArea` control intervenes in the server-side chain of events and gives the programmer a chance to run some code if the internal content of the control is changed between two successive postbacks.

All ASP.NET controls that, like `HtmlTextArea`, implement the `IPostBackDataHandler` interface can invoke user-defined code when the control's internal state changes. As discussed in Chapter 3, controls can fire custom events by overriding the `RaisePostDataChangedEvent` method on the aforementioned interface. The following pseudocode shows what happens in the method's implementation of `HtmlTextArea`:

```csharp
void System.Web.UI.IPostBackDataHandler.RaisePostDataChangedEvent()
{
    this.OnServerChange(EventArgs.Empty);
}
```
Finally, note that the control raises the event only if the state has changed between two successive posts. To determine whether that has happened, the control needs to track the content it had the time before. This value can be stored only in the view state. Of course, the `ServerChange` event won’t fire if you disable the view state for the host page or the control.

### HTML Input Controls

In HTML, the `<input>` element has several variations and can be used to provide a submit button as well as a check box or text box. In ASP.NET, each possible instance of the `<input>` element is mapped to a specific class. All input classes derive from the `HtmlInputControl` class. `HtmlInputControl` is the abstract class that defines the common programming interface for all input controls. The class inherits from `HtmlControl` and simply adds three custom properties—`Name`, `Type`, and `Value`—to the inherited interface.

The `Name` property returns the name assigned to the control. In ASP.NET, this property is peculiar because, although it’s marked as read/write, it actually works as a read-only property. The `get` accessor returns the control’s `UniqueId` property, while the `set` accessor is just void. As a result, whatever value you assign to the property, either programmatically or declaratively, is just ignored and no exception or compile error is ever thrown.

The `Type` property mirrors the `type` attribute of the HTML input elements. The property is read-only. Finally, the `Value` property is read/write and represents the content of the input field.

Table 4-7 lists the HTML input controls defined in ASP.NET.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>HtmlInputButton</code></td>
<td>Represents the various flavors of a command button supported by HTML. Feasible values for the <code>Type</code> attribute are <code>button</code>, <code>submit</code>, and <code>reset</code>.</td>
</tr>
<tr>
<td><code>HtmlInputCheckBox</code></td>
<td>Represents an HTML check box—that is, the <code>&lt;input&gt;</code> tag with a type equal to <code>checkbox</code>.</td>
</tr>
<tr>
<td><code>HtmlInputFile</code></td>
<td>Represents the file uploader—that is, the <code>&lt;input&gt;</code> tag with a type equal to <code>file</code>.</td>
</tr>
<tr>
<td><code>HtmlInputHidden</code></td>
<td>Represents a hidden buffer of text data—that is, the <code>&lt;input&gt;</code> tag with a type equal to <code>hidden</code>.</td>
</tr>
<tr>
<td><code>HtmlInputImage</code></td>
<td>Represents a graphic button—that is, the <code>&lt;input&gt;</code> tag with a type equal to <code>image</code>. Note that this tag is supported by all browsers.</td>
</tr>
<tr>
<td><code>HtmlInputPassword</code></td>
<td>Represents a protected text field—that is, the <code>&lt;input&gt;</code> tag with a type of <code>password</code>. Not available in ASP.NET 1.x.</td>
</tr>
</tbody>
</table>
Class Description

**HtmlInputRadioButton** Represents a radio button—that is, the `<input>` tag with a type equal to radio.

**HtmlInputReset** Represents a reset command button. Not available in ASP.NET 1.x.

**HtmlInputSubmit** Represents a submit command button. Not available in ASP.NET 1.x.

**HtmlInputText** Represents a text field—that is, the `<input>` tag with a type of either password or text.

The hidden and text input controls are nearly identical, and the contents of both are posted back. Essentially, they differ only in that hidden fields are not displayed and, subsequently, they don't provide some UI-related properties such as MaxLength and Size.

**Command Buttons**

The **HtmlInputButton** class is the most flexible button class in the .NET Framework. It differs from the **HtmlButton** class in that it renders through the `<input>` tag rather than the Internet Explorer–specific `<button>` tag. This fact ensures for the control much wider support from browsers.

The HTML input button controls support the `ServerClick` event, which allows you to set the code to run on the server after the button is clicked. Note that if you set the button type to `Button` and the `ServerClick` event handler is specified, the control automatically adds the postback script code to the `onclick` HTML attribute. In this way, any click causes the page to post back and the code to execute. Let's consider the following ASP.NET code:

```html
<input runat="server" type="button" id="btn" value="Click" onserverclick="buttonClicked" />
```

The corresponding HTML code is as follows:

```html
<input language="javascript" onclick="__doPostBack('btn','')" name="btn" type="button" value="Click" />
```

The client-side `__doPostBack` script function is the standard piece of code generated by ASP.NET to implement the postback. If the button type is set to `Submit`—that is, a value that would always cause a postback—no client-side script code is generated and the `onclick` attribute is not set.

In ASP.NET 2.0 and newer versions, more specific controls have been added to render submit and reset buttons. The controls are **HtmlInputSubmit** and **HtmlInputReset**.
Controlling Validation

The HtmlInputImage class supports a nearly identical pattern for handling server-side events and validation. The HtmlInputImage control features a few more properties specific to the image it shows. In particular, you can set the alternate text for the image, the border, and the alignment with respect to the rest of the page. The ServerClick event handler has a slightly different form and looks like the following:

```csharp
void ImageClickEventHandler(object sender, ImageClickEventArgs e);
```

When an image button is clicked, the coordinates of the click are determined by using the X and Y properties of the ImageClickEventArgs data structure.

Detecting State Changes of Controls

Earlier in this chapter, while discussing the features of the HtmlTextArea control, we ran into the ServerChange event and described it as the mechanism to detect and validate changes in the control's state between two successive postbacks. The ServerChange event is not an exclusive feature of the HtmlTextArea control but is also supported by other input controls such as HtmlInputCheckBox, HtmlInputRadioButton, HtmlInputHidden, and HtmlInputText. Let's look at an example in which we use the ServerChange event to detect which elements have been checked since last time the control was processed on the server.

We build a page with a list of check boxes and a button to let the user post back to the server when finished. Notice, in fact, that neither the HtmlInputCheckBox control nor any other input control except buttons, post back to the server when clicked. For this reason, you must provide another control on the Web page that supports posting to the server—for example,
an HtmlButton or an HtmlInputButton control. The following code implements the page shown in Figure 4-2:

```csharp
<%@ Page Language="C#" %>
<script runat="server">
    public void DetectChange(object sender, EventArgs e) {
        HtmlInputCheckBox cb = (HtmlInputCheckBox) sender;
        Response.Write("Control <b>" + cb.UniqueID + "</b> changed<br>" + cb.OnServerChange("DetectChange"));
    }
</script>
<body>
<form runat="server">
  <input runat="server" type="checkbox" id="one" OnServerChange="DetectChange" />
  <input runat="server" type="checkbox" id="two" OnServerChange="DetectChange" />
  <input runat="server" type="checkbox" id="three" OnServerChange="DetectChange" />
  <input runat="server" type="submit" value="Submit" />
</form>
</body>
</html>
```

**FIGURE 4-2** The ServerChange event fires only if the status of the control has changed since the last time the control was processed on the server.

The ServerChange event is fired only if the state of the control results changed after two postbacks. To get the first screen shot, you select the element and then submit. Next, if you submit again without selecting or deselecting anything, you get the second screen shot.

As mentioned in Chapter 3, by implementing the IPostBackDataHandler interface, each server control gets a chance to update its current state with data posted by the client. I cover this interface in detail in the *Programming Microsoft ASP.NET 2.0 Applications: Advanced Topics* (Microsoft Press, 2006).
Part I Building an ASP.NET Page

Uploading Files

The `HtmlInputFile` control is the HTML tool for uploading files from a browser to the Web server. To take advantage of the `HtmlInputFile` control, you should first ensure that the server form’s `Enctype` property is set to `multipart/form-data`. Note, though, that starting with ASP.NET 2.0, the proper `Enctype` is automatically set care of the `HtmlInputFile` control before the control’s markup is rendered:

```html
<form runat="server" enctype="multipart/form-data">
  <input runat="server" type="file" id="upLoader" />
  <input runat="server" type="submit" value="Upload..." />
</form>
```

The way in which the `HtmlInputFile` control is rendered to HTML is browser-specific, but it normally consists of a text box and a Browse button. The user selects a file from the local machine and then clicks the button to submit the page to the server. When this occurs, the browser uploads the selected file to the server, as shown in Figure 4-3.

![Figure 4-3: A new file has been uploaded to the Web server and copied to the destination folder.](image)

**Note** Prior to ASP.NET, a server-side process—the posting acceptor—was required to run in the background to handle `multipart/form-data` submissions. In ASP.NET, the role of the posting acceptor is no longer necessary, as it is carried out by the ASP.NET runtime itself.

On the server, the file is parked into an object of type `HttpPostedFile` and stays there until explicitly processed—for example, saved to disk or to a database. The `HttpPostedFile` object...
provides properties and methods to get information on an individual file and to read and save the file. The following code shows how to save a posted file to a particular folder to disk:

```csharp
<%@ Page language="C#" %>
<%@ Import Namespace="System.IO" %>
<script runat="server">
    void UploadButton_Click(object sender, EventArgs e)
    {
        // *** ASSUME THE PATH EXISTS ***
        string savePath = @"c:\temp\pictures\";
        if (!Directory.Exists(savePath)) {
            string msg = "<h1>Upload path doesn't exist: {0}</h1>")
                Response.Write(String.Format(msg, savePath));
                Response.End();
        }
        // Verify that a file has been posted
        if (FileUpload1.PostedFile != null)
        {
            // Save the uploaded file to the specified path
            string fileName = Path.GetFileName(FileUpload1.Value);
            savePath += fileName;
            FileUpload1.PostedFile.SaveAs(savePath);
            // Notify the user of the name the file was saved under.
            UploadStatusLabel.InnerText = "File saved as: " + savePath;
        }
        else
        {
            // Notify the user that a file was not uploaded.
            UploadStatusLabel.InnerText = "No file specified."
        }
    }
</script>
<html>
<head runat="server">
<title>File Upload</title>
</head>
<body>
<form runat="server">
<h3>Select a picture to upload:</h3>
<br />
<b>Picture to upload</b><br />
<input type="file" id="FileUpload1" runat="server" />
<br />
<input runat="server" id="UploadButton" type="submit" value="Upload" onserverclick="UploadButton_Click" />
<br />
<span runat="server" id="UploadStatusLabel" />
</form>
</body>
</html>
```
You can also use the InputStream property of the HttpPostedFile object to read the posted data before persisting or processing. The HttpInputFile control also allows you to restrict the file types that can be uploaded to the server. You do this by setting the Accept property with a comma-separated list of MIME types.

**Caution** When you use the SaveAs method, you should pay attention to specify the full path to the output file. If a relative path is provided, ASP.NET attempts to place the file in the system directory. This practice can result in an access denied error. Furthermore, make sure to provide write permission for the account used by ASP.NET for the directory where you want to store the file.

ASP.NET exercises some control on the amount of data being uploaded. The maxRequestLength attribute in the <httpRuntime> section of the configuration file sets the maximum allowable file size. An error is generated in the browser when the file exceeds the specified size—4 MB by default. Uploading large files might also generate another runtime error as a result of an excessive consumption of system memory.

### The HtmlImage Control

The HtmlImage class is the ASP.NET counterpart of the `<img>` tag. You can use it to configure on the server the display of an image. Possible parameters you can set are the size of the image, the border, and the alternate text. An instance of HtmlImage is created only when the runat attribute is added to the `<img>` tag. If you simply need to display an image within a page, and the image is not dynamically determined or configured, there is no need to resort to the HtmlImage control, which would add unnecessary overhead to the page.

The following code snippet shows how to configure a server-side `<img>` tag called to display an image whose name is determined based on run-time conditions:

```csharp
theImg.Width = 100;
theImg.Height = 100;
theImg.Src = GetImageUrl(Request); // assume GetImageUrl is a method of yours
```

The HtmlImage control should be used to programmatically manipulate the image to change the source file, the width and height, or the alignment of the image relative to other page elements. The majority of properties of the HtmlImage control are implemented as strings, including Src—the URL of the image—and Align. Feasible values of Align are only a small set of words such as left, right, top, and so forth. These words would have been more appropriately grouped in a custom enumerated type, thus providing for a strongly typed programming model. If you think so, too, you just got the gist of the difference between HTML and Web server controls! HTML controls just mirror HTML tags; Web controls attempt to provide a more consistent and effective programming interface by exploiting the characteristics of the .NET Framework.
Literal Controls

Literal controls are a special type of server control that ASP.NET creates and uses whenever it encounters plain text that doesn’t require server-side processing. In general, everything that appears in the context of an ASP.NET page is treated like a control. If a tag includes the runat="server" attribute, ASP.NET creates an instance of a specific class; otherwise, if no runat attribute has been specified, the text is compiled into a LiteralControl object. Literal controls are simple text holders that are added to and removed from pages using the same programming interface defined for other server controls.

Note that a literal control is created for each sequence of characters placed between two successive server controls, including carriage returns. Using a new line to separate distinct server controls and increase code readability actually affects the number of server controls being created to serve the page. Writing the page as a single string without carriage returns produces the smallest number of server controls.

Web Controls

Web controls are defined in the System.Web.UI.WebControls namespace and represent an alternative approach to HTML server controls. Like HTML controls, Web controls are server-side components that spring to life thanks to the runat="server" attribute. Unlike HTML controls, Web controls provide a programming interface that refactors the classic set of HTML attributes and events. For this reason, Web controls sometimes appear to be more consistent and abstract in the API design and richer in functionality, but they still generate valid markup. When hosted in .aspx pages, Web controls are characterized by the asp namespace prefix.

To a large degree, Web controls and HTML controls overlap and generate almost the same markup, although they do it through different programming interfaces. For example, the Web controls namespace defines the TextBox control and makes it available through the <asp: textbox> tag; similarly, the HTML controls namespace provides the HtmlInputText control and declares it using the <input> tag. Using either is mostly a matter of preference; only in a few cases will you run into slight functionality differences.

Generalities of Web Controls

The WebControl class is the base class from which all Web controls inherit. WebControl inherits from Control. The class defines several properties and methods that are shared, but not necessarily implemented, by derived controls. Most properties and methods are related to the look and feel of the controls (font, style, colors, CSS) and are subject to browser and...
Properties of Web Controls

Table 4-8 lists the properties available on the WebControl class.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccessKey</td>
<td>Gets or sets the letter to press (together with Alt) to quickly set focus to the control in a Web form. Supported on Internet Explorer 4.0 and newer.</td>
</tr>
<tr>
<td>Attributes</td>
<td>Gets the collection of attributes that do not correspond to properties on the control. Attributes set in this way will be rendered as HTML attributes in the resulting page.</td>
</tr>
<tr>
<td>BackColor</td>
<td>Gets or sets the background color of the Web control.</td>
</tr>
<tr>
<td>BorderColor</td>
<td>Gets or sets the border color of the Web control.</td>
</tr>
<tr>
<td>BorderStyle</td>
<td>Gets or sets the border style of the Web control.</td>
</tr>
<tr>
<td>BorderWidth</td>
<td>Gets or sets the border width of the Web control.</td>
</tr>
<tr>
<td>ControlStyle</td>
<td>Gets the style of the Web server control. The style is an object of type Style.</td>
</tr>
<tr>
<td>ControlStyleCreated</td>
<td>Gets a value that indicates whether a Style object has been created for the ControlStyle property.</td>
</tr>
<tr>
<td>CssClass</td>
<td>Get or sets the name of the cascading style sheet (CSS) class associated with the client.</td>
</tr>
<tr>
<td>Enabled</td>
<td>Gets or sets whether the control is enabled.</td>
</tr>
<tr>
<td>Font</td>
<td>Gets the font properties associated with the Web control.</td>
</tr>
<tr>
<td>ForeColor</td>
<td>Gets or sets the foreground color of the Web control mostly used to draw text.</td>
</tr>
<tr>
<td>Height</td>
<td>Gets or sets the height of the control. The height is expressed as a member of type Unit.</td>
</tr>
<tr>
<td>Style</td>
<td>Gets a CssStyleCollection collection object made of all the attributes assigned to the outer tag of the Web control.</td>
</tr>
<tr>
<td>TabIndex</td>
<td>Gets or sets the tab index of the control.</td>
</tr>
<tr>
<td>ToolTip</td>
<td>Gets or sets the text displayed when the mouse pointer hovers over the control.</td>
</tr>
<tr>
<td>Width</td>
<td>Gets or sets the width of the control. The width is expressed as a member of type Unit.</td>
</tr>
</tbody>
</table>
The \texttt{ControlStyle} and \texttt{ControlStyleCreated} properties are used primarily by control developers, while the \texttt{Style} property is what application developers would typically use to set CSS attributes on the outer tag of the control. The \texttt{Style} property is implemented using an instance of the class \texttt{CssStyleCollection}. The \texttt{CssStyleCollection} class is a simple collection of strings like those you would assign to the HTML style attribute.

\section*{Styling Web Controls}

The \texttt{ControlStyle} property evaluates to an object of type \texttt{Style}—a class that encapsulates the appearance properties of the control. The \texttt{Style} class groups together some of the properties that were shown in Table 4-8, and it works as the repository of the graphical and cosmetic attributes that characterize all Web controls. The grouped properties are \texttt{BackColor}, \texttt{BorderColor}, \texttt{BorderStyle}, \texttt{BorderWidth}, \texttt{CssClass}, \texttt{Font}, \texttt{ForeColor}, \texttt{Height}, and \texttt{Width}. All properties of the \texttt{Style} class are strongly typed. The properties just mentioned are not persisted to the view state individually, but they benefit from the serialization machinery supported by the \texttt{Style} object.

It should be clear by now that the \texttt{Style} class is quite different from the \texttt{Style} property, whose type is \texttt{CssStyleCollection}. Note that style values set through the \texttt{Style} property are not automatically reflected by the (strongly typed) values in the \texttt{Style} object. For example, you can set the CSS border-style through the \texttt{Style} property, but that value won't be reflected by the value of the \texttt{BorderStyle} property.

\begin{verbatim}
// Set the border color through a CSS attribute
MyControl.Style["border-color"] = "blue";

// Set the border color through an ASP.NET style property
MyControl.BorderColor = Color.Red;
\end{verbatim}

So what happens if you run the preceding code snippet? Which setting would win? When a control is going to render, the contents of both \texttt{ControlStyle} and \texttt{Style} properties are rendered to HTML style attributes. The \texttt{ControlStyle} property is processed first, so in case of overlapping settings the value stuffed in \texttt{Style}, which is processed later, would win, as shown by the following markup:

\begin{verbatim}
style="border-color:Red;border-color:blue;    
\end{verbatim}

\section*{Managing the Style of Web Controls}

The style properties of a Web control can be programmatically manipulated to some extent. For example, in the \texttt{Style} class, you can count on a \texttt{CopyFrom} method to duplicate the object and on the \texttt{MergeWith} method to combine two style objects.

\begin{verbatim}
currentStyle.MergeStyle(newStyle);
\end{verbatim}
The MergeWith method joins the properties of both objects. In doing so, it does not replace any property that is already set in the base object but limits itself to defining uninitialized properties. Finally, the Reset method clears all current attributes in the various properties of the style object.

Methods of Web Controls

The WebControl class supports a few additional methods that are not part of the base Control class. These methods are listed in Table 4-9.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplyStyle</td>
<td>Copies any nonempty elements of the specified style object to the control. Existing style properties are overwritten.</td>
</tr>
<tr>
<td>CopyBaseAttributes</td>
<td>Imports from the specified Web control the properties AccessKey, Enabled, ToolTip, TabIndex, and Attributes. Basically, it copies all the properties not encapsulated in the Style object.</td>
</tr>
<tr>
<td>MergeStyle</td>
<td>Like ApplyStyle, copies any nonempty elements of the specified style to the control. Existing style properties are not overwritten, though.</td>
</tr>
<tr>
<td>RenderBeginTag</td>
<td>Renders the HTML opening tag of the control into the specified writer. The method is called right before the control's RenderControl method.</td>
</tr>
<tr>
<td>RenderEndTag</td>
<td>Renders the HTML closing tag of the control into the specified writer. The method is called right after the control's RenderControl method.</td>
</tr>
</tbody>
</table>

All these methods are rarely of interest to application developers. They are mostly designed to support control developers.

Core Web Controls

The set of Web controls can be divided into various categories according to the provided functionality—input and button controls, validators, data-bound controls, security-related controls, grid and view controls, plus a few miscellaneous controls that provide ad hoc functions and are as common on the Web as they are hard to catalogue (for example, calendar, ad rotator, and so forth).

In this chapter, we're focused on covering the most common and essential Web controls, such as the controls for capturing and validating the user's input and posting data to the server. We'll cover the various types of data-bound controls in Chapter 11, Chapter 12, and Chapter 13. Security-related controls, on the other hand, are slated for Chapter 17. Table 4-10 details the core server controls of ASP.NET.
## TABLE 4-10 Core Web Controls

<table>
<thead>
<tr>
<th>Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
<td>Implements a push button through the <code>&lt;input&gt;</code> tag.</td>
</tr>
<tr>
<td>CheckBox</td>
<td>Implements a check box through the <code>&lt;input&gt;</code> tag.</td>
</tr>
<tr>
<td>FileUpload</td>
<td>Allows users to select a file to upload to the server. <em>Not available in ASP.NET 1.x.</em></td>
</tr>
<tr>
<td>HiddenField</td>
<td>Implements a hidden field. <em>Not available in ASP.NET 1.x.</em></td>
</tr>
<tr>
<td>HyperLink</td>
<td>Implements an anchor <code>&lt;a&gt;</code> tag, and lets you specify either the location to jump to or the script code to execute.</td>
</tr>
<tr>
<td>Image</td>
<td>Implements a picture box through the <code>&lt;img&gt;</code> tag.</td>
</tr>
<tr>
<td>ImageButton</td>
<td>Displays an image and responds to mouse clicks on the image like a real button.</td>
</tr>
<tr>
<td>ImageMap</td>
<td>Displays an image and optionally defines clickable hot spots on it. <em>Not available in ASP.NET 1.x.</em></td>
</tr>
<tr>
<td>Label</td>
<td>Represents a static, nonclickable piece of text. Implemented through the <code>&lt;span&gt;</code> tag.</td>
</tr>
<tr>
<td>LinkButton</td>
<td>Implements an anchor <code>&lt;a&gt;</code> tag that uses only the ASP.NET postback mechanism to post back. It is a special type of hyperlink where the programmer can't directly set the target URL.</td>
</tr>
<tr>
<td>MultiView</td>
<td>Represents a control that acts as a container for a group of child View controls. <em>Not available in ASP.NET 1.x.</em></td>
</tr>
</tbody>
</table>
| Panel     | Implements an HTML container using the `<div>` block element. In ASP.NET 2.0, the container supports scrolling. Note that in down-level browsers the control renders out as a `<table>`.
| RadioButton| Implements a single radio button through the `<input>` tag.                 |
| Table     | Implements the outer table container. Equivalent to the HTML `<table>` element. |
| TableCell | A table cell, is equivalent to the HTML `<td>` element.                     |
| TableRow  | A table row, is equivalent to the HTML `<tr>` element.                     |
| TextBox   | Implements a text box using the `<input>` or `<textarea>` tag as appropriate and according to the requested text mode. Can work in single-line, multilime, or password mode. |
| View      | Acts as a container for a group of controls. A View control must always be contained within a MultiView control. *Not available in ASP.NET 1.x.* |

Most controls in Table 4-10 look like HTML controls. Compared to HTML controls, their programming model is certainly richer and more abstract, but in the end it still generates valid and legal markup. If a given feature can't be obtained with raw HTML, there's no way a custom Web control can provide it. No matter how complex the programming model is, all Web controls must produce valid HTML for both up-level and down-level browsers.
Part I Building an ASP.NET Page

Button Controls

Starting with ASP.NET 2.0, controls that provide button functions are characterized by a new interface—IButtonControl. Core controls that implement the interface are Button, ImageButton, and LinkButton. In general, by implementing IButtonControl any custom control can act like a button on a form.

The IButtonControl interface is a clear example of the refactoring process that the entire ASP.NET Framework went through in the transition from 1.x to 2.0. The interface now groups a few properties that most button controls (including some HTML button controls) support since ASP.NET 1.x. In addition to this, a few new properties heralding new functions have been added, such as PostBackUrl and ValidationGroup. Table 4-11 details the IButtonControl interface.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CausesValidation</td>
<td>Boolean value, indicates whether validation is performed when the control is clicked.</td>
</tr>
<tr>
<td>CommandArgument</td>
<td>Gets or sets an optional parameter passed to the button's Command event along with the associated CommandName.</td>
</tr>
<tr>
<td>CommandName</td>
<td>Gets or sets the command name associated with the button that is passed to the Command event.</td>
</tr>
<tr>
<td>PostBackUrl</td>
<td>Indicates the URL that will handle the postback triggered through the button control. This feature is known as cross-page postback. (We'll cover this further in Chapter 5.)</td>
</tr>
<tr>
<td>Text</td>
<td>Gets or sets the caption of the button.</td>
</tr>
<tr>
<td>ValidationGroup</td>
<td>Gets or sets the name of the validation group that the button belongs to.</td>
</tr>
<tr>
<td>Visible</td>
<td>Boolean value, indicates whether the button control is rendered.</td>
</tr>
</tbody>
</table>

In addition to the properties defined by the IButtonControl interface, the Button class features two new properties in ASP.NET 2.0—OnClientClick and UseSubmitBehavior. The former standardizes a common practice that many developers used countless times in ASP.NET 1.x projects. OnClientClick lets you define the name of the JavaScript function to run when the client-side onclick event is fired. The following two statements are perfectly legal and equivalent:

```
// For ASP.NET 2.0 and newer versions
Button1.OnClientClick = "ShowMessage()";

// Equivalent in ASP.NET 1.x
Button1.Attributes["onclick"] = "ShowMessage()";
```

The OnClientClick property is also available on LinkButton and ImageButton controls.
By default, the Button class is rendered through an `<input type=submit>` tag. In this way, it takes advantage of the browser's submit mechanism to post back. The `UseSubmitBehavior` property allows you to change the default behavior. Set the `UseSubmitBehavior` property to `false` and the control will render out through an `<input type=button>` tag. Also in this case, though, the Button control remains a postback button. When `UseSubmitBehavior` is `false`, the control's `onclick` client event handler is bound to a piece of JavaScript code (the `_doPostBack` function) that provides the ASP.NET postback mechanism just like for `LinkButton` or `ImageButton` controls.

**Important** Buttons are not the only controls that can trigger a postback. Text boxes and check boxes (plus a few more data-bound list controls, which we'll see in Chapter 9) also can start a postback if their `AutoPostBack` property is set to `true`. (Note that the default setting is `false`.) When this happens, the control wires up to a client-side event—`onchange` for text boxes and `onclick` for check boxes—and initiates a postback operation via script. In fact, because this mechanism is available to server-side code, virtually any control can be modified to post back.

### HyperLinks

The `HyperLink` control creates a link to another Web page and is typically displayed through the text stored in the `Text` property. Alternatively, the hyperlink can be displayed as an image: in this case, the URL of the image is stored in the `ImageUrl` property. Note that if both the `Text` and `ImageUrl` properties are set, the `ImageUrl` property takes precedence. In this case, the content of the `Text` property is displayed as a ToolTip when the mouse hovers over the control's area.

The `NavigateUrl` property indicates the URL the hyperlink is pointing to. The `Target` property is the name of the window or frame that will contain the output of the target URL.

### Images and Image Buttons

The `Image` control displays an image on the Web page. The path to the image is set through the `ImageUrl` property. Image URLs can be either relative or absolute, with most programmers showing a clear preference for relative URLs because they make a Web site inherently easier to move. You can also specify alternate text to display when the image is not available or when the browser doesn't render the image for some reason. The property to use in this case is `AlternateText`. The image alignment with respect to other elements on the page is set by using the `ImageAlign` property. Feasible values are taken from the homonymous enum type (i.e.: `ImageAlign.Left`, `ImageAlign.Middle`, and so forth).

The `Image` control is not a clickable component and is simply limited to displaying an image. If you need to capture mouse clicks on the image, use the `ImageButton` control instead. The `ImageButton` class descends from `Image` and extends it with a couple of events—`Click` and `Command`—that are raised when the control is clicked. The `OnClick` event handler provides...
you with an ImageClickEventArgs data structure that contains information about the coordinates for the location at which the image is clicked.

The OnCommand event handler makes the ImageButton control behave like a command button. A command button has an associated name that you can control through the CommandName property. If you have multiple ImageButton controls on the same page, the command name allows you to distinguish which one is actually clicked. The CommandArgument property can be used to pass additional information about the command and the control.

Another new entry in ASP.NET 2.0 is the ImageMap control. In its simplest and most commonly used form, the control displays an image on a page. However, when a hot-spot region defined within the control is clicked, the control either generates a post back to the server or navigates to a specified URL. The hot spot is a clickable region within the displayed image. The hot spot is implemented with a class that inherits from the HotSpot class. There are three predefined types of hot spots—polygons, circles, and rectangles.

Check Boxes and Radio Buttons

Check boxes and radio buttons are implemented through the <input> tag and the type attribute set to checkbox or radio. Unlike the HTML control versions, the Web control versions of check boxes and radio buttons let you specify the associated text as a property. The HTML elements and corresponding HTML controls lack an attribute whose content becomes the text near the check box or radio button. In HTML, to make the text near the check box or radio button clickable, you have to resort to the <label> tag with the for attribute:

```html
<input type="checkbox" id="ctl" />
<label for="ctl">Check me</label>
```

Neither the HtmlInputCheckBox nor the HtmlInputRadioButton control adds a label, which leaves you responsible for doing that. The counterparts to these Web controls, on the other hand, are not bound to the HTML syntax and do precisely that—they automatically add a Text property, which results in an appropriate <label> tag. For example, consider the following ASP.NET code:

```csharp
<asp:checkbox runat="server" id="ctl" text="Check me" />
```

It results in the following HTML code:

```html
<input type="checkbox" id="ctl" />
<label for="ctl">Check me</label>
```

Scrollable Panels

The Panel control groups controls in a <div> tag. It allows developers to add and remove controls, and it supports style information. Panels support horizontal and vertical scrollbars
implemented through the overflow CSS style. Here’s an example that demonstrates a scrollable panel:

```html
<asp:Panel ID="Panel1" runat="server" Height="85px" Width="400px"
ScrollBars="Auto" BorderStyle="Solid">
<h2>Choose a technology</h2>
<br />
<asp:CheckBox ID="ChkBox1" runat="server" Text="ASP.NET" />
<br />
<asp:CheckBox ID="ChkBox2" runat="server" Text="AJAX" />
<br />
<asp:CheckBox ID="ChkBox3" runat="server" Text="Web Services" />
<br />
<asp:CheckBox ID="ChkBox4" runat="server" Text="XML" />
<br />
<asp:CheckBox ID="ChkBox5" runat="server" Text="WCF Services" />
<br />
<asp:CheckBox ID="ChkBox6" runat="server" Text="Silverlight" />
</asp:Panel>
```

Figure 4-4 shows the page in action.

FIGURE 4-4 A page that uses a scrollable panel.

Text Controls

The fastest way to insert text in a Web page is through literals—that is, static text inserted directly in the .aspx source. This text will still be compiled to a control but, at least, the number of dynamically created literal controls is the minimum possible because any sequence of consecutive characters are grouped into a single literal. If you need to identify and manipulate particular strings of text programmatically, you can resort to a Literal control or, better yet, to the richer Label control. Modifiable text requires a TextBox.

Some minor changes occurred to these controls starting with ASP.NET 2.0. First, a few new interfaces have been introduced to logically group capabilities. They are ITextControl and IEditableTextControl. The former includes the sole Text property and is implemented by
Literal, Label, TextBox, and list controls. The latter interface defines the TextChanged event and is specific to TextBox and list controls.

It is worth mentioning a new accessibility feature of the Label control—the AssociatedControlID property. The property takes the ID of a control in the page—typically, an input control such as a TextBox—that you want to associate with the label. AssociatedControlID changes the way the Label control renders out. It is a <span> tag if no associated control is specified; it is a <label> tag otherwise. Let's consider the following example:

```html
<asp:Label ID="Label1" runat="server" Text="Sample text" />
<asp:TextBox ID="TextBox1" runat="server" />
```

As is, it generates the following markup:

```
<span id="Label1">Sample text</span>
<input name="TextBox1" type="text" id="TextBox1" />
```

If you set the label’s AssociatedControlID property to TextBox1, the markup changes as shown here:

```
<label for="TextBox1" id="Label1">Sample text</label>
<input name="TextBox1" type="text" id="TextBox1" />
```

The runtime behavior changes a bit because now any click on the label text will be extended to the associated control. For example, clicking on the label will move the input focus to a text box, or it will select or deselect a check box.

AssociatedControlID is a feature designed to improve the accessibility of the resulting page. In Visual Studio 2008, you can check any page for accessibility rules (both WCAG and Section 508) by clicking on the Tools|Check Accessibility menu item.

### Hidden Fields and File Upload

If you're looking for a more comfortable programming interface to create hidden fields and upload files, two Web controls might help. The HiddenField and FileUpload controls add no new functionality to the ASP.NET programmer’s bag, but they have been added to the toolbox for completeness. A hidden field can be created in two other ways that work with ASP.NET 1.x too. For example, you can use the RegisterHiddenField method on the Page class:

```csharp
// Works in ASP.NET 1.x but is obsolete starting with 2.0
RegisterHiddenField("HiddenField1", "Great book!");
```
Note that the `RegisterHiddenField` method has been flagged as obsolete in ASP.NET 2.0. The recommended code analogous to the previous snippet is shown next:

```csharp
// Recommended code in ASP.NET 2.0 and beyond
ClientScriptManager.RegisterHiddenField("HiddenField1", "Great book!");
```

In addition, to create a hidden field you can resort to the HTML markup, adding a `runat` attribute if you need to set the value programmatically:

```html
<input runat="server" id="HiddenField1" type="hidden" value="..." />
```

Analogous considerations can be made for the `FileUpload` control, which provides the same capabilities as the `HtmlInputFile` control that we discussed earlier. In this case, though, the programming interface is slightly different and perhaps more intuitive. The `HasFile` property and `SaveAs` method hide any reference to the object that represents the posted file. Likewise, the `FileName` property provides a more immediate name for the name of the posted file. The code to upload a file can be rewritten as follows:

```csharp
if (FileUpload1.HasFile)
{
    // Get the name of the file to upload.
    string fileName = FileUpload1.FileName;
    string targetPath = GetSavePath(fileName);
    FileUpload1.SaveAs(targetPath);
}
```

Whether you use `FileUpload` or `HtmlInputFile` is mostly a matter of preference.

**Miscellaneous Web Controls**

The `WebControls` namespace also includes a few controls that provide useful functionality that is common in Web applications. In particular, we’ll examine the `AdRotator` control, which works like an advertisement banner, and the `Calendar` control, which is a flexible and highly interactive control used to specify a date.

**The `AdRotator` Control**

Abstractly speaking, the `AdRotator` control displays an automatically sized image button and updates both the image and the URL each time the page refreshes. The image to display and other information is read from an XML file written according to a specific schema. More concretely, you use the `AdRotator` control to create an advertisement banner on a Web Forms page. The control actually inserts an image and hyperlink in the page and makes them point to the advertisement page selected. The image is sized by the browser to the dimensions of
the AdRotator control, regardless of its actual size. The following code shows a typical XML advertisement file:

```xml
<Advertisements>
  <Ad>
    <ImageUrl>6235.gif</ImageUrl>
    <NavigateUrl>www.microsoft.com/MSPress/books/6235.asp</NavigateUrl>
    <AlternateText>Introducing ASP.NET AJAX</AlternateText>
    <Impressions>50</Impressions>
  </Ad>
  <Ad>
    <ImageUrl>5727.gif</ImageUrl>
    <NavigateUrl>www.microsoft.com/MSPress/books/5727.asp</NavigateUrl>
    <AlternateText>Programming ASP.NET Applications</AlternateText>
    <Impressions>50</Impressions>
  </Ad>
</Advertisements>
```

The `<Advertisement>` root node contains multiple `<Ad>` elements, one for each image to show. The advertisement file must reside in the same application as the AdRotator control.

The syntax of the AdRotator control is as follows:

```csharp
<%@ Page Language="C#" %>
<html>
<head>
<title>Ad Rotators</title>
</head>
<body>
<form runat="server">
<h1>Dino Esposito's Books</h1>
<asp:AdRotator runat="server" id="bookRotator" AdvertisementFile="MyBooks.xml" />
</form>
</body>
</html>
```

In the XML advertisement file, you use the `<ImageUrl>` node to indicate the image to load and the `<NavigateUrl>` node to specify where to go in case of a click. The `<AlternateText>` node indicates the alternate text to use if the image is unavailable, whereas `<Impressions>` indicates how often an image should be displayed in relation to other images in the advertisement file. The higher the impression value (as compared to the other values in the advertisement file), the higher the frequency its associated ad image is displayed. The sum of all the impressions in the advertisement file may not exceed 2,047,999,999 or the AdRotator control will throw an exception. Finally, each image can also be associated with a keyword through the `<Keyword>` node. Of all the elements, only `<ImageUrl>` is required.

Once per roundtrip, the AdRotator control fires the server-side AdCreated event. The event occurs before the page is rendered. The event handler receives an argument of type AdCreatedEventArgs, which contains information about the image, navigation URL, alternate text, and any custom properties associated with the advertisement. The AdCreated event can be used to programmatically select the image to show. The XML schema of the advertisement is not fixed and can be extended with custom elements. All nonstandard elements
associated with the selected advertisement will be passed to the AdCreated event handler stuffed in the AdProperties dictionary member of the AdCreatedEventArgs class.

Note Starting with ASP.NET 2.0, the AdRotator control has undergone a significant change. It is derived from WebControl in ASP.NET 1.x, but it inherits from DataBoundControl in ASP.NET 2.0 and beyond. Among other things, this means that the advertisement feed can also be provided through an XML or a relational data source. Image and navigation URLs, as well as the alternate text, can be read from fields belonging to the data source. The control cannot be bound to more than one data source at a time. If more than one property—AdvertisementFile, DataSourceID, or DataSource—is set, an exception will be thrown.

The Calendar Control

The Calendar control (shown in Figure 4-5) displays a one-month calendar and allows you to choose dates and navigate backward and forward through the months of the year. The control is highly customizable both for appearance and functionality. For example, by setting the SelectionMode property, you can decide what the user can select—that is, whether a single date, week, or month can be selected.

```html
<asp:calendar runat="server" id="hireDate"
    SelectedDate="2007-08-16" VisibleDate="2007-08-16" />
```

![The Calendar control in action.](image)

The VisibleDate property sets a date that must be visible in the calendar, while SelectedDate sets with a different style the date that is rendered as selected. The control also fires three ad hoc events: DayRender, SelectionChanged, and VisibleMonthChanged. The DayRender event signals that the control has just created a new day cell. You can hook the event if you think you need to customize the cell output. The SelectionChanged event fires when the selected...
date changes, while VisibleMonthChanged is raised whenever the user moves to another
month using the control's selector buttons.

The Calendar control originates a roundtrip for each selection you make. Although it is cool
and powerful on its own, for better performance you might also want to provide a plain text
box for manually typing dates.

The Xml Control

The Xml control, defined by the <asp:Xml> tag, is used to inject the content of an XML docu-
ment directly into an ASP.NET page. The control can display the source XML as-is or as the
results of an XSL transformation (XSLT). The Xml control is a sort of declarative counterpart
for the XslTransform class and can make use of the .NET Framework XSLT transform class
internally.

You use the Xml control when you need to embed XML documents in a Web page. For ex-
ample, the control is extremely handy when you need to create XML data islands for the
client to consume. The control lets you specify a document to work with and, optionally, a
transformation to apply. The XML document can be specified in a variety of formats—an
XML document object model, string, or file name. The XSLT transformation can be defined
through either an already configured instance of the .NET Framework XslTransform class or a
file name.

<asp:xml runat="server"
documentsource="document.xml"
transformsource="transform.xsl" />

If you're going to apply some transformation to the XML data, you could also embed it inline
between the opening and closing tags of the control. The control also makes it easier to ac-
complish a common task: apply browser-dependent transformations to portions of the page
expressed in an XML meta language. In this case, you exploit the programming interface of
the control as follows:

<asp:xml runat="server" id="theXml" documentsource="document.xml" />

In the Page_Load event, you just check the browser capabilities and decide which transfor-
mation should be applied.

void Page_Load(object sender, EventArgs e)
{
    if (IsInternetExplorer(Request.Browser))
        theXml.TransformSource = "ie5.xsl";
    else
        theXml.TransformSource = "downlevel.xsl";
}
The PlaceHolder Control

The PlaceHolder control is one of the few controls in the WebControls namespace that isn't derived from the WebControl class. It inherits from Control and is used only as a container for other controls in the page. The PlaceHolder control does not produce visible output of its own and is limited to containing child controls dynamically added through the Controls collection. The following code shows how to embed a placeholder control in a Web page:

```csharp
<asp:placeholder runat="server" id="theToolbar" />
```

Once you have a placeholder, you can add controls to it. As mentioned, the placeholder does not add extra functionality, but it provides for grouping and easy and direct identification of a group of related controls. The following code demonstrates how to create a new button and add it to an existing placeholder:

```csharp
Button btn = new Button();
btn.Text = "Click me";
theToolbar.Controls.Add(btn);
```

The PlaceHolder control reserves a location in the control tree and can be extremely helpful in identifying specific areas of the page to customize and extend by adding controls programmatically.

Important Note that each control dynamically added to the Controls collection of a parent control is not restored on postback. If the control generates some input elements on the client, the client data is regularly posted but there will be no server-side control to handle that. To avoid this, you must "remember" that you created a certain control dynamically and re-create it while the page loads on postbacks. To remember that a certain control was added to a parent, you can create a custom entry in the view state or use a hidden field.

View Controls

ASP.NET provides two related controls to create a group of interchangeable panels of child controls. The MultiView control defines a group of views, each represented with an instance of the View class. Only one view is active at a time and rendered to the client. The View control can't be used as a standalone component and can only be placed inside a MultiView control. Here's an example:

```csharp
<asp:MultiView runat="server" id="Tables">
    <asp:View runat="server" id="Employees">
        ...
    </asp:View>
    <asp:View runat="server" id="Products">
        ...
    </asp:View>
</asp:MultiView>
```
Part I  Building an ASP.NET Page

```csharp
<asp:View runat="server" id="Customers">
    ...
</asp:View>
</asp:MultiView>
```

You change the active view through postback events when the user clicks buttons or links embedded in the current view. To indicate the new view, you can either set the `ActiveViewIndex` property or pass the view object to the `SetActiveView` method.

Figure 4-6 shows a sample page in action. You select the page from the drop-down list and refresh the view.

```csharp
void Page_Load(object sender, EventArgs e)
{
    // Views is an auto-postback drop-down list
    Tables.ActiveViewIndex = Views.SelectedIndex;
}
```

**FIGURE 4-6** A multiview control in action.

The combination of `View` and `MultiView` controls lends itself very well to implementing wizards. In fact, the new ASP.NET `Wizard` control uses a `MultiView` control internally. We’ll cover the `Wizard` control in Chapter 6.

Validation Controls

A key rule for writing more secure applications is to get the data right, before you use it. Getting the data right requires you to apply a validation step to any external input. In ASP.NET, validation controls provide an easy-to-use mechanism to perform a variety of validation tasks, including testing for valid types, values within a given range, or required fields.
Validation controls inherit from the `BaseValidator` class which, in turn, descends from `Label`. All validators defined on a page are automatically grouped in the `Validators` collection of the `Page` class. You can validate them all in a single shot using the `Validate` method in the page class or individually by calling the `Validate` method on each validator. The `Validate` method sets the `IsValid` property both on the page and on the individual validator. The `IsValid` property indicates whether the user’s entries match the requirements of the validators. Other than explicitly using the `Validate` method, the user’s entry is also automatically validated whenever the page posts back.

**Note** Typical control members involved with input validation have been grouped in the `IValidator` interface that the `BaseValidator` class implements. The interface includes the `Validate` method and the `IsValid` and `ErrorMessage` properties.

The .NET Framework also provides complete client-side implementation for validation controls. This allows Dynamic HTML–enabled browsers (such as Internet Explorer version 4.0 and later) to perform validation on the client as soon as the user tabs out of a monitored input field.

### Generalities of Validation Controls

Each validation control references an input control located elsewhere on the page. When the page is going to be submitted, the contents of the monitored server control is passed to the validator for further processing. Each validator would perform a different type of verification. Table 4-12 shows the types of validation supported by the .NET Framework.

<table>
<thead>
<tr>
<th>Validator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompareValidator</td>
<td>Compares the user’s entry against a fixed value by using a comparison operator such as <code>LessThan</code>, <code>Equal</code>, or <code>GreaterThan</code>. Can also compare against the value of a property in another control on the same page.</td>
</tr>
<tr>
<td>CustomValidator</td>
<td>Employs a programmatically defined validation logic to check the validity of the user’s entry. You use this validator when the other validators cannot perform the necessary validation and you want to provide custom code that validates the input.</td>
</tr>
<tr>
<td>RangeValidator</td>
<td>Ensures that the user’s entry falls within a specified range. Lower and upper boundaries can be expressed as numbers, strings, or dates.</td>
</tr>
<tr>
<td>RegularExpressionValidator</td>
<td>Validates the user’s entry only if it matches a pattern defined by a regular expression.</td>
</tr>
<tr>
<td>RequiredFieldValidator</td>
<td>Ensures that the user specifies a value for the field.</td>
</tr>
</tbody>
</table>
Multiple validation controls can be used with an individual input control to validate according to different criteria. For example, you can apply multiple validation controls on a text box that is expected to contain an e-mail address. In particular, you can impose that the field is not skipped (RequiredFieldValidator) and that its content matches the typical format of e-mail addresses (RegularExpressionValidator).

Table 4-12 lacks a reference to the ValidationSummary control. The control does not perform validation tasks itself. Instead, it displays a label to summarize all the validation error messages found on a Web page as the effect of other validators. We’ll cover the ValidationSummary control later in the chapter.

**The BaseValidator Class**

Table 4-13 details the specific properties of validation controls. Some properties—such asForeColor, Enabled, and Text—are overridden versions of base properties on base classes.

**TABLE 4-13 Basic Properties of Validators**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ControlToValidate</td>
<td>Gets or sets the input control to validate. The control is identified by</td>
</tr>
<tr>
<td></td>
<td>name—that is, by using the value of the ID attribute.</td>
</tr>
<tr>
<td>Display</td>
<td>If client-side validation is supported and enabled, gets or sets how the</td>
</tr>
<tr>
<td></td>
<td>space for the error message should be allocated—either statically or</td>
</tr>
<tr>
<td></td>
<td>dynamically. In case of server-side validation, this property is ignored. A</td>
</tr>
<tr>
<td></td>
<td>Static display is possible only if the browser supports the display CSS style.</td>
</tr>
<tr>
<td></td>
<td>The default is Dynamic.</td>
</tr>
<tr>
<td>EnableClientScript</td>
<td>True by default; gets or sets whether client-side validation is enabled.</td>
</tr>
<tr>
<td>Enabled</td>
<td>Gets or sets whether the validation control is enabled.</td>
</tr>
<tr>
<td>ErrorMessage</td>
<td>Gets or sets the text for the error message.</td>
</tr>
<tr>
<td>ForeColor</td>
<td>Gets or sets the color of the message displayed when validation fails.</td>
</tr>
<tr>
<td>IsValid</td>
<td>Gets or sets whether the associated input control passes validation.</td>
</tr>
<tr>
<td>SetFocusOnError</td>
<td>Indicates whether the focus is moved to the control where validation</td>
</tr>
<tr>
<td></td>
<td>failed. Not available in ASP.NET 1.x.</td>
</tr>
<tr>
<td>Text</td>
<td>Gets or sets the description displayed for the validator in lieu of the</td>
</tr>
<tr>
<td></td>
<td>error message. Note, though, this text does not replace the contents of</td>
</tr>
<tr>
<td></td>
<td>ErrorMessage in the summary text.</td>
</tr>
<tr>
<td>ValidationGroup</td>
<td>Gets or sets the validation group that this control belongs to. Not available</td>
</tr>
<tr>
<td></td>
<td>in ASP.NET 1.x.</td>
</tr>
</tbody>
</table>

All validation controls inherit from the BaseValidator class except for compare validators, for which a further intermediate class—the BaseCompareValidator class—exists. The BaseCompareValidator class serves as the foundation for validators that perform typed comparisons. An ad hoc property, named Type, is used to specify the data type the values are
converted to before being compared. The `CanConvert` static method determines whether the user's entry can be converted to the specified data type. Supported types include string, integer, double, date, and currency. The classes acting as compare validators are `RangeValidator` and `CompareValidator`.

**Associating Validators with Input Controls**

The link between each validator and its associated input control is established through the `ControlToValidate` property. The property must be set to the ID of the input control. If you do not specify a valid input control, an exception will be thrown when the page is rendered.

The association validator/control is between two controls within the same container—be it a page, user control, or template.

Not all server controls can be validated, only those that specify their validation property through an attribute named `[ValidationProperty]`. The attribute takes the name of the property that contains the user's entry to check. For example, the validation property for a `TextBox` is `Text` and is indicated as follows:

```csharp
[ValidationProperty("Text")]
public class TextBox : WebControl, ITextControl
{
    ...
}
```

The list of controls that support validation includes `TextBox`, `DropDownList`, `ListBox`, `RadioButtonList`, `FileUpload`, `HtmlInputFile`, `HtmlInputText`, `HtmlInputPassword`, `HtmlTextArea`, and `HtmlSelect`. Custom controls can be validated too, as long as they are marked with the aforementioned `[ValidationProperty]` attribute.

> **Note** If the validation property of the associated input control is left empty, all validators accept the value and pass the test. The `RequiredFieldValidator` control represents a rather natural exception to this rule, as it has been specifically designed to detect fields the user skipped and left blank.

**Gallery of Controls**

Let's take a closer look at the various types of validation controls that you'll use in ASP.NET Web forms.
Part I Building an ASP.NET Page

The CompareValidator Control

The CompareValidator control lets you compare the value entered by the user with a constant value or the value specified in another control in the same naming container. The behavior of the control is characterized by the following additional properties:

- **ControlToCompare** Represents the ID of the control to compare with the current user's entry. You should avoid setting the ControlToCompare and ValueToCompare properties at the same time. They are considered mutually exclusive; if you set both, the ControlToCompare property takes precedence.

- **Operator** Specifies the comparison operation to perform. The list of feasible operations is defined in the ValidationCompareOperator enumeration. The default operator is Equal; feasible operators are also LessThan, GreaterThan, and their variations. The DataTypeCheck operator is useful when you want to make sure that certain input data can be converted to a certain type. When the DataTypeCheck operator is specified, both ControlToCompare and ValueToCompare are ignored. In this case, the test is made on the type of the input data and succeeds if the specified data can be converted to the expected type. Supported types are expressed through the following keywords: String, Integer, Double, Date, and Currency (decimal).

- **ValueToCompare** Indicates the value to compare the user's input against. If the Type property is set, the ValueToCompare property must comply with it.

The following code demonstrates the typical markup of the CompareValidator control when the control is called to validate an integer input from a text box representing someone's age:

```xml
<asp:CompareValidator runat="server" id="ageValidator"
ControlToValidate="ageTextBox"
ValueToCompare="18"
Operator="GreaterThanEqual"
Type="Integer"
ErrorMessage="Must specify an age greater than 17." />
```

The CustomValidator Control

The CustomValidator control is a generic and totally user-defined validator that uses custom validation logic to accomplish its task. You typically resort to this control when none of the other validators seems appropriate or, more simply, when you need to execute your own code in addition to that of the standard validators.

To set up a custom validator, you can indicate a client-side function through the ClientValidationFunction property. If client-side validation is disabled or not supported, simply omit this setting. Alternatively, or in addition to client validation, you can define some managed code to execute on the server. You do this by defining a handler for the ServerValidate event. The code will be executed when the page is posted back in response to a click on a
button control. The following code snippet shows how to configure a custom validator to check the value of a text box against an array of feasible values:

```html
<asp:CustomValidator runat="server" id="membershipValidator"
    ControlToValidate="membership"
    ClientValidationFunction="CheckMembership"
    OnServerValidate="ServerValidation"
    ErrorMessage="Membership can be Normal, Silver, Gold, or Platinum." />
```

If specified, the client validation function takes a mandatory signature and looks like this:

```javascript
function CheckMembership(source, arguments) {
    // ... 
}
```

The `source` argument references the HTML tag that represents the validator control—usually, a `<span>` tag. The `arguments` parameter references an object with two properties, `IsValid` and `Value`. The `Value` property is the value stored in the input control to be validated. The `IsValid` property must be set to `false` or `true` according to the result of the validation.

The `CustomValidator` control is not associated in all cases with a single input control in the current naming container. For this type of validator, setting the `ControlToValidate` property is not mandatory. For example, if the control has to validate the contents of multiple input fields, you do not simply set the `ControlToValidate` property and the `arguments.Value` variable evaluates to the empty string. In this case, you write the validation logic so that any needed values are dynamically retrieved. With client-side script code, this can be done by accessing the members of the document's form, as shown in the following code:

```javascript
function CheckMembership(source, arguments) {
    // Retrieve the current value of the element
    // with the specified ID
    var membership = document.getElementById("membership").value;
    // ... 
}
```

**Warning** Setting only a client-side validation code opens a security hole because an attacker could work around the validation logic and manage to have invalid or malicious data sent to the server. By defining a server event handler, you have one more chance to validate data before applying changes to the back-end system.

To define a server-side handler for a custom validator, use the `ServerValidate` event.

```csharp
void ServerValidation(object source, ServerValidateEventArgs e) {
    // ... 
}
```
The **ServerValidateEventArgs** structure contains two properties—**IsValid** and **Value**—with the same meaning and goal as in the client validation function. If the control is not bound to a particular input field, the **Value** property is empty and you retrieve any needed value using the ASP.NET object model. For example, the following code shows how to check the status of a check box on the server:

```csharp
void ServerValidation (object source, ServerValidateEventArgs e) {
    e.IsValid = (CheckBox1.Checked == true);
}
```

The **CustomValidator** control is the only option you have to validate controls that are not marked with the **[ValidationProperty]** attribute—for example, calendars and check-box controls.

The **RegularExpressionValidator** Control

Regular expressions are an effective way to ensure that a predictable and well-known sequence of characters form the user's entry. For example, using regular expressions you can validate the format of zip codes, Social Security numbers, e-mail addresses, phone numbers, and so on. When using the **RegularExpressionValidator** control, you set the **ValidationExpression** property with the regular expression, which will be used to validate the input.

The following code snippet shows a regular expression validator that ensures the user's entry is an e-mail address:

```xml
<asp:RegularExpressionValidator runat="server" id="emailValidator" ControlToValidate="email" ValidationExpression="^[a-zA-Z_0-9.-]+@[a-zA-Z_0-9.-]+\.[\w]+$" ErrorMessage="Must be a valid email address." />
```

The regular expression just shown specifies that valid e-mail addresses are formed by two nonzero sequences of letters, digits, dashes, and dots separated by an @ symbol and followed by a dot (.) and an alphabetic string. (This might not be the perfect regular expression for e-mail addresses, but it certainly incorporates the majority of e-mail address formats.)

**Note** The regular expression validation syntax is slightly different on the client than on the server. The **RegularExpressionValidator** control uses JScript regular expressions on the client and the .NET Framework **Regex** object on the server. Be aware that the JScript regular expression syntax is a subset of the **Regex** model. Whenever possible, try to use the regular expression syntax supported by JScript so that the same result is obtained for both the client and server.
The **RangeValidator Control**

The `RangeValidator` control lets you verify that a given value falls within a specified range. The type of the values involved in the check is specified dynamically and picked from a short list that includes strings, numbers, and dates. The following code shows how to use a range validator control:

```xml
<asp:RangeValidator runat="server" id="hiredDateValidator"
    ControlToValidate="hired"
    MinimumValue="2000-1-4"
    MaximumValue="9999-12-31"
    Type="Date"
    ErrorMessage="Must be a date after <b>Jan 1, 1999</b>." />
```

The key properties are `MinimumValue` and `MaximumValue`, which together clearly denote the lower and upper boundaries of the interval. Note that an exception is thrown if the strings assigned to `MinimumValue` or `MaximumValue` cannot be converted to the numbers or dates according to the value of the `Type` property.

If the type is set to `Date`, but no specific culture is set for the application, you should specify dates using a culture-neutral format, such as `yyyy-MM-dd`. If you don’t do so, the chances are good that the values will not be interpreted correctly.

---

**Note** The `RangeValidator` control extends the capabilities of the more basic `CompareValidator` control by checking for a value in a fixed interval. In light of this, the `RangeValidator` control might raise an exception if either `MinimumValue` or `MaximumValue` is omitted. Whether the exception is thrown or not depends on the type chosen and its inherent ability to interpret the empty string. For example, an empty string on a `Date` type causes an exception. If you want to operate on an unbound interval—whether lower or upper unbound—either you resort to the `GreaterThan` (or `LessThan`) operator on the `CompareValidator` control or simply use a virtually infinite value such as the `9999-12-31` value.

---

The **RequiredFieldValidator Control**

To catch when a user skips a mandatory field in an input form, you use the `RequiredFieldValidator` control to show an appropriate error message:

```xml
<asp:RequiredFieldValidator runat="server" id="lnameValidator"
    ControlToValidate="lname"
    ErrorMessage="Last name is mandatory" />
```

As long as you’re using an up-level browser and client-side scripting is enabled for each validator, which is the default, invalid input will display error messages without performing a postback.
Important Note that just tabbing through the controls is not a condition that raises an error; the validator gets involved only if you type blanks or if the field is blank when the page is posted back.

How can you determine whether a certain field is really empty? In many cases, the empty string is sufficient, but this is not a firm rule. The InitialValue property specifies the initial value of the input control. The validation fails only if the value of the control equals InitialValue upon losing focus. By default, InitialValue is initialized with the empty string.

Special Capabilities

The primary reason why you place validation controls on a Web form is to catch errors and inconsistencies in the user's input. But how do you display error messages? Are you interested in client-side validation and, if you are, how would you set it up? Finally, what if you want to validate only a subset of controls when a given button is clicked? Special capabilities of validation controls provide a valid answer to all these issues.

Displaying Error Information

The ErrorMessage property determines the static message that each validation control will display in case of error. It is important to know that if the Text property is also set, it would take precedence over ErrorMessage. Text is designed to display inline where the validation control is located, ErrorMessage is designed to display in the validation summary. (Strategies for using Text and ErrorMessage will be discussed more in the next section, “The ValidationSummary Control.”) Because all validation controls are labels, no other support or helper controls are needed to display any message. The message will be displayed in the body of the validation controls and, subsequently, wherever the validation control is actually placed. The error message is displayed as HTML, so it can contain any HTML formatting attribute.

Validators that work in client mode can create the <span> tag for the message either statically or dynamically. You can control this setting by using the Display property of the validator. When the display mode is set to Static (the default), the <span> element is given the following style:

```
style="color:Red;visibility:hidden;"
```

The CSS visibility style attribute, when set to Hidden, causes the browser not to display the element but reserves space for it. If the Display property contains Dynamic, the style string changes as follows:

```
style="color:Red;display:none;"
```
The CSS display attribute, when set to none, simply hides the element, which will take up space on the page only if displayed. The value of the Display property becomes critical when you have multiple validators associated with the same input control. (See Figure 4-7)

As you can see, the hire text box is first validated to ensure it contains a valid date and then to verify the specified date is later than 1-1-1999. If the Display property is set to Static for the first validator, and the date is outside the specified range, you get a page like the one shown in Figure 4-8.
The full source code of the page in the figure is available on the Web at the following address: http://www.microsoft.com/mspress/companion/9780735625273/

Note You can associate multiple validators with a single input control. The validation takes place in order, and each validation control generates and displays its own error message. The content of the input control is considered valid if all the validators return true. If an input control has multiple valid patterns—for example, an ID field can take the form of a Social Security number or a European VAT number—you can either validate by using custom code or regular expressions.

The ValidationSummary Control
The ValidationSummary control is a label that summarizes and displays all the validation error messages found on a Web page after a postback. The summary is displayed in a single location formatted in a variety of ways. The DisplayMode property sets the output format, which can be a list, a bulleted list, or a plain text paragraph. By default, it is a bulleted list. The feasible values are grouped in the ValidationSummaryDisplayMode enumeration.

Whatever the format is, the summary can be displayed as text in the page, in a message box, or in both. The Boolean properties ShowSummary and ShowMessageBox let you decide. The output of the ValidationSummary control is not displayed until the page posts back no matter what the value of the EnableClientScript property is. The HeaderText property defines the text that is displayed atop the summary.

```
<asp:ValidationSummary runat="server" 
    ShowMessageBox="true"
    ShowSummary="true"
    HeaderText="The following errors occurred:"
    DisplayMode="BulletList" />
```

This code snippet originates the screen shown in Figure 4-9.

The validation summary is displayed only if there’s at least one pending error. Notice that, in the default case, the labels near the input controls are updated anyway, along with the summary text. In summary, you can control the error information in the following ways:

- **Both in-place and summary information**  This is the default scenario. Use the ValidationSummary control, and accept all default settings on the validator controls. If you want to leverage both places to display information, a recommended approach consists of minimizing the in-place information by using the Text property rather than ErrorMessage. If you set both, Text is displayed in-place while ErrorMessage shows up in the validation summary. For example, you can set Text with a glyph or an exclamation mark and assign ErrorMessage with more detailed text.
FIGURE 4-9 After the page posts back, the validation summary is updated and a message box pops up to inform the user of the errors.

- **Only in-place information** Do not use the ValidationSummary control, and set the ErrorMessage property in each validation control you use. The messages appear after the page posts back.

- **Only summary information** Use the ValidationSummary control, and set the ErrorMessage property on individual validation controls. Set the Display property of validators to None so that no in-place error message will ever be displayed.

- **Custom error information** You don’t use the ValidationSummary control, and you set the Display property of the individual validators to None. In addition, you collect the various error messages through the ErrorMessage property on the validation controls and arrange your own feedback for the user.

### Enabling Client Validation

As mentioned earlier, the verification normally takes place on the server as the result of the postback event or after the Validate method is called. If the browser supports Dynamic HTML, though, you can also activate the validation process on the client, with a significant gain in responsiveness. To be precise, ASP.NET automatically enables client-side validation if it detects a browser with enough capabilities. While ASP.NET 1.x limits its client-side support only to Internet Explorer 4.0 or higher, starting with ASP.NET 2.0 validation controls also work fine on the client with Mozilla Firefox, Netscape 6.x, and Safari 1.2. Figure 4-10 shows the previous sample page in action in Mozilla Firefox.
If client-side validation is turned on, the page won’t post back until all the input fields contain valid data. To run secure code and prevent malicious and underhanded attacks, you might want to validate data on the server too. Consider also that not all types of validation can be accomplished on the client. In fact, if you need to validate against a database, there’s no other option than posting back to the server. (AJAX facilities which we’ll explore in Chapter 19 and Chapter 20 may provide a relief for this problem.)

Client validation can be controlled on a per-validation control basis by using the `EnableClientScript` Boolean property. By default, the property is set to `true`, meaning client validation is enabled as long as the browser supports it. By default, the code in the `BaseValidator` class detects the browser’s capabilities through the `Request.Browser` property. If the browser is considered up-level, the client validation will be implemented. Browsers and client devices that are considered up-level support at least the following:

- ECMAScript (including JScript and JavaScript) version 1.2
- HTML version 4.0
- The Microsoft Document Object Model
- Cascading style sheets

For down-level browsers, the only requirement is HTML version 3.2. You can also control the client validation at the page level by using the `ClientTarget` attribute on the `@Page` directive.
The following code disables client validation by specifying that any code in the page should target a down-level browser:

```csharp
@Page ClientTarget="DownLevel"
```

The `ClientTarget` attribute overrides the type of browser that ASP.NET should target when generating the page. When the `ClientTarget` attribute is set, ASP.NET doesn't detect the actual browser's capabilities but loads the capabilities for the specified browser from the browser database.

**Validation Groups**

In ASP.NET 1.x, control validation occurs in an all-or-nothing kind of way. For example, if you have a set of input and validation controls and two buttons on the form, clicking either button will always validate all controls. In other words, there's no way to validate some controls when one button is clicked, and some others when the other button is clicked. The `CausesValidation` property on button controls allows you to disable validation on a button, but that is not the point here. What is missing is the ability to do validation on a group of controls. This is exactly what the `ValidationGroup` property provides in ASP.NET 2.0 and newer versions. The property is available on validators, input controls, and button controls.

Using the `ValidationGroup` property is simple; just define it for all the validation controls that you want to group together, and then assign the same name to the `ValidationGroup` property of the button that you want to fire the validation. Here's an example:

```csharp
<asp:textbox runat="server" id="TextBox1"/>
<asp:RequiredFieldValidator runat="server" ValidationGroup="Group1" ControlToValidate="TextBox1" ErrorMessage="TextBox1 is mandatory"/>
<asp:textbox runat="server" id="TextBox2"/>
<asp:RequiredFieldValidator runat="server" ValidationGroup="Group2" ControlToValidate="TextBox2" ErrorMessage="TextBox2 is mandatory"/>
<asp:Button runat="server" Text="Check Group1" ValidationGroup="Group1"/>
<asp:Button runat="server" Text="Check Group2" ValidationGroup="Group2"/>
```

The two `RequiredFieldValidator` controls belong to distinct validation groups—`Group1` and `Group2`. The first button validates only the controls defined within `Group1`, the second button takes care of the input associated with `Group2`. In this way, the validation process can be made as granular as needed.
Important: The ValidationGroup property can also be defined optionally on input controls. This is required only if you use the CustomValidator control as a way to check whether a given input control belongs to the right validation group.

The validation group feature gets especially helpful when combined with cross-page postbacks—a feature that we’ll cover in the next chapter. Cross-page postback allows a button to post the contents of the current form to another page, in a certain way overriding the single-form model of ASP.NET. Imagine you have a search box in your page, and you want to post its contents directly to a search page without passing through the classic postback mechanism and an additional redirect. Validation groups allow you to check only the contents of the search text box prior to posting to the search page.

Validation groups are also reflected on the server-side, where the Validate method of the Page class now features an overload that lets you select the group according to which the page must be validated.

Conclusion

In ASP.NET pages, server controls are vital components and transform the programming model of ASP.NET from a mere factory of HTML strings to a more modern and effective component-based model. ASP.NET features a long list of control classes. Looking at the namespaces involved, we should conclude that only two families of controls exist—HTML and Web controls. Controls in the former group simply mirror the set of elements in the HTML syntax. Each constituent control has as many properties as there are attributes in the corresponding HTML tag. Names and behavior have been kept as faithful to the originals as possible. The ultimate goal of the designers of HTML controls is to make the transition from ASP to ASP.NET as seamless as possible—just add runat="server" and refresh the page.

The overall design of Web controls is more abstract and much less tied to HTML. In general, Web controls do not promote a strict one-to-one correspondence between controls and HTML tags. However, the capabilities of Web and HTML controls overlap. All ASP.NET server controls render in HTML, but Web controls render to more complex HTML representation than HTML controls.

In the family of core Web controls, we can identify interesting and powerful families of controls—for example, validators. Validators let you put declarative boundaries around input controls so that any user’s input is filtered and validated both on the client and server. This alone is not sufficient to certify an application as secure, but it is a quantum leap in the right direction.
Just The Facts

- In ASP.NET, there are two big families of controls: HTML controls and Web controls. The former group includes controls that are in 1:1 correspondence with HTML elements. The controls in the latter group offer a more abstract programming model and richer functionalities not specifically bound to one HTML element.
- If made invisible, ASP.NET controls don’t generate any markup code but are activated and processed nonetheless.
- Adaptive rendering is the process that enables controls to generate different markup for individual browsers.
- ASP.NET controls let you declaratively assign a browser-specific value to properties. For example, you can use one style for Internet Explorer and another one for Mozilla Firefox.
- The vast majority of ASP.NET controls can generate XHTML-compliant markup. Non XHTML mode is supported for backward compatibility.
- New controls let you fully manage programmatically the `<head>` tag of a page.
- Everything you put on a page is ultimately processed as a control, including literal text, blanks, and carriage returns. Contiguous characters are conveyed to a single control instance.
- Validation controls let you test for valid types, values within a given range, regular expressions, and required fields.
- Validators let you put declarative boundaries around input controls so that any user’s input is filtered and validated both on the client and server.
- Group validation allows you to validate only certain controls when the page posts back.