CHAPTER 5

Web Form Fundamentals

In this chapter, you'll learn some of the core topics that every ASP.NET developer must master. You'll begin by taking a closer look at the ASP.NET application model, and considering what files and folders belong in a web application. Then you'll take a closer look at *server controls*, the basic building block of every web form. Using server controls, you'll create your first ASP.NET application, by taking a static HTML page and transforming it into a simple, single-page currency converter.

After you have the currency converter example under your belt, you can explore the basics of ASP.NET's web form model in more detail. You'll pick up some handy skills along the way, such as the ability to create controls on the fly, navigate from one page to another, and handle special characters in HTML. Finally, you'll consider the ASP.NET configuration model, which lets you tweak the settings that govern the behavior of your web application.

Understanding the Anatomy of an ASP.NET Application

It's sometimes difficult to define exactly what a web application is. Unlike a desktop program or smartphone app, ASP.NET websites almost always include multiple web pages. This division means a user can enter an ASP.NET "application" at different points, and follow a link from one page to another part of the website or to another web server. So, does it make sense to treat an entire website as though it were a single application?

In ASP.NET, the answer is yes. Every ASP.NET application shares a common set of resources and configuration settings. Web pages from other ASP.NET applications don't share these resources, even if they're on the same web server. Technically speaking, every ASP.NET application is executed inside a separate *application domain*. Application domains are isolated areas in memory, and they ensure that even if one web application causes a fatal error, it's unlikely to affect any other application that is currently running on the same computer (in this case, that's the web server). Similarly, application domains restrict a web page in one application from accessing the in-memory information of another application. Each web application is maintained separately and has its own set of cached, application, and session data.

The standard definition of an ASP.NET application describes it as a combination of files, pages, handlers, modules, and executable code that can be invoked from a virtual directory (and, optionally, its subdirectories) on a web server. In other words, the virtual directory is the basic grouping structure that delimits an application. Figure 5-1 shows a web server that hosts four separate web applications.

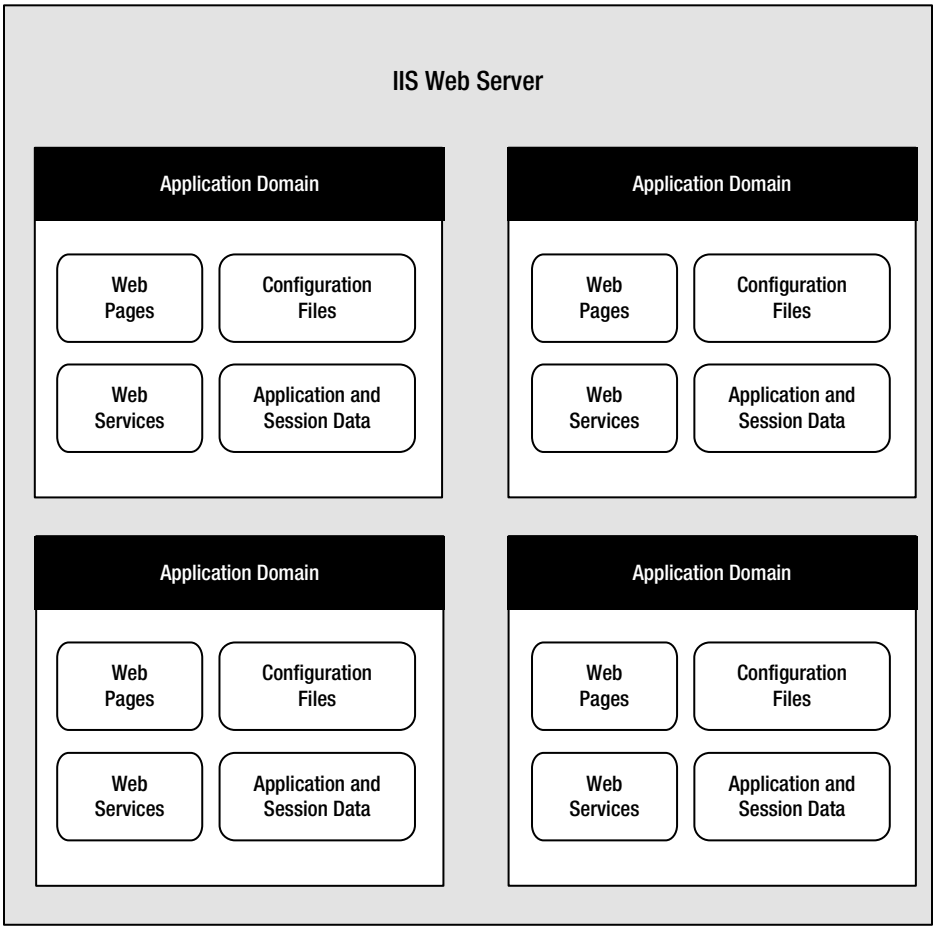

Figure 5-1. *ASP.NET applications*

■ **Note** A virtual directory is a directory that's exposed to other computers on a web server. As you'll discover in Chapter 26, you deploy your perfected ASP.NET web application by copying it to a virtual directory.

ASP.NET File Types

ASP.NET applications can include many types of files. Table 5-1 introduces the essential ingredients.

Table 5-1. *ASP.NET File Types*

File Name	Description
Ends with .aspx	These are ASP.NET web pages. They contain the user interface and, optionally, the underlying application code. Users request or navigate directly to one of these pages to start your web application.
Ends with .ascx	These are ASP.NET user controls. User controls are similar to web pages, except that the user can't access these files directly. Instead, they must be hosted inside an ASP.NET web page. User controls allow you to develop a small piece of user interface and reuse it in as many web forms as you want without repetitive code. You'll learn about user controls in Chapter 11.
web.config	This is the configuration file for your ASP.NET application. It includes settings for customizing security, state management, memory management, and much more. You'll get an introduction to the web.config file in this chapter, and you'll explore its settings throughout this book.
global.asax	This is the global application file. You can use this file to define global variables (variables that can be accessed from any web page in the web application) and react to global events (such as when a web application first starts). You'll learn about it later in this chapter.
Ends with .cs	These are code-behind files that contain C# code. They allow you to separate the application logic from the user interface of a web page. We'll introduce the code-behind model in this chapter and use it extensively in this book.

In addition, your web application can contain other resources that aren't special ASP.NET files. For example, your virtual directory can hold image files, HTML files, or CSS style sheets. These resources might be used in your ASP.NET web pages, or they might be used independently. A website can even combine static HTML pages with dynamic ASP.NET pages.

■ **Note** To access an ASP.NET website, a visitor browses to an .aspx page. The other file types that are listed in Table 5-1 cannot be accessed directly. If a visitor attempts to request one of these files, the web server will deny the request (to ensure good security). You'll learn more about how web servers deal with ASP.NET file types in Chapter 26, when you learn to deploy a completed web application.

ASP.NET Web Folders

Every web application starts with a single location, called the *root folder*. However, in a large, well-planned web application, you'll usually create additional folders inside the website's root folder. For example, you'll probably want to store images in a subfolder while you place web pages in the root folder. Or you might want to put public ASP.NET pages in one subfolder and restricted ones in another so you can apply different security settings based on the folder. (See Chapter 19 for more about how to create authorization rules like this.)

■ **Tip** To create a subfolder in Visual Studio, right-click your website in the Solution Explorer and choose Add ä New Folder.

Along with the custom folders you create, ASP.NET also uses a few specialized folders, which it recognizes by name (see Table 5-2). Keep in mind that you won't see all these folders in a typical application. In fact, in a brand

Table 5-2. *ASP.NET Folders*

Directory	Description
App_Browsers	Contains .browser files that ASP.NET uses to identify the browsers that are using your application and determine their capabilities. Usually, browser information is standardized across the entire web server, and you don't need to use this folder. For more information about ASP.NET's browser support—which is an advanced feature that most ordinary web developers can safely ignore—refer to *Pro ASP.NET 4.5 in C#* (Apress, 2012).
App_Code	Contains source code files that are dynamically compiled for use in your application.
App_GlobalResources	Stores global resources that are accessible to every page in the web application. This directory is used in localization scenarios, when you need to have a website in more than one language. Localization isn't covered in this book, and you can refer to *Pro ASP.NET 4.5 in C#* for more information.
App_LocalResources	Serves the same purpose as App_GlobalResources, except these resources are accessible to a specific page only.
App_WebReferences	Stores references to web services, which are remote code routines that a web application can call over a network or the Internet.
App_Data	Stores data, including SQL Server Express database files (as you'll see in Chapter 14). Of course, you're free to store data files in other directories.
App_Themes	Stores the themes that are used to standardize and reuse formatting in your web application. You'll learn about themes in Chapter 12.
Bin	Contains all the compiled .NET components (DLLs) that the ASP.NET web application uses. For example, if you develop a custom component for accessing a database (see Chapter 22), you'll place the component here. ASP.NET will automatically detect the assembly, and any page in the web application will be able to use it.

new blank website—such as the one you learned to create in Chapter 4—you won't have any subfolders. Instead, Visual Studio will create them when you need them.

Introducing Server Controls

In old-style web development, programmers had to master the quirks and details of HTML before they could design a dynamic web page. Pages had to be carefully tailored to a specific task, and the only way to add new content to a page was to generate raw HTML tags.

ASP.NET solves this problem with a higher-level model of *server controls*. These controls are created and configured as *objects*. They run on the web server and automatically provide their own HTML output in a way

that's conceptually similar to the simple garbage can example you saw in Chapter 2. Even better, server controls behave like their Windows counterparts by maintaining state and raising events that you can react to in code.

In Chapter 3, you built an exceedingly simple web page that incorporated a few controls you dragged in from the Visual Studio Toolbox. But before you create a more complex page, it's worth taking a step back to look at the big picture. ASP.NET actually provides *two* sets of server-side controls that you can incorporate into your web forms. These two types of controls play subtly different roles:

HTML server controls: These are server-based equivalents for standard HTML elements. These controls are ideal if you're a seasoned web programmer who prefers to work with familiar HTML tags (at least at first). They are also useful when migrating ordinary HTML pages or classic ASP pages to ASP.NET, because they require the fewest changes.

Web controls: These are similar to the HTML server controls, but they provide a richer object model with a variety of properties for style and formatting details. They also provide more events and more closely resemble the controls used for Windows development. Web controls also feature some user interface elements that have no direct HTML equivalent, such as the GridView, Calendar, and validation controls.

You'll learn about web controls in the next chapter. In this chapter, you'll take a detailed look at HTML server controls.

■ **Note** Even if you plan to use web controls exclusively, it's worth reading through this section to master the basics of HTML server controls. Along the way, you'll get an introduction to a few ASP.NET essentials that apply to all kinds of server controls, including view state, postbacks, and event handling.

HTML Server Controls

HTML server controls provide an object interface for standard HTML elements. They provide three key features:

They generate their own interface: You set properties in code, and the underlying HTML tag is created automatically when the page is rendered and sent to the client.

They retain their state: Because the Web is stateless, ordinary web pages need to do a lot of work to store information between requests. HTML server controls handle this task automatically. For example, if the user selects an item in a list box, that item remains selected the next time the page is generated. Or if your code changes the text in a button, the new text sticks the next time the page is posted back to the web server.

They fire server-side events: For example, buttons fire an event when clicked, text boxes fire an event when the text they contain is modified, and so on. Your code can respond to these events, just like ordinary controls in a Windows application. If a given event doesn't occur, the event handler won't be executed.

HTML server controls are ideal when you're performing a quick translation to add server-side code to an existing HTML page. That's the task you'll tackle in the next section, with a simple one-page web application.

Converting an HTML Page to an ASP.NET Page

Figure 5-2 shows a currency converter web page. It allows the user to convert US dollars to euros—or at least it would, if it had the code it needed to do the job. Right now, it's just a plain, inert HTML page. Nothing happens when the button is clicked.

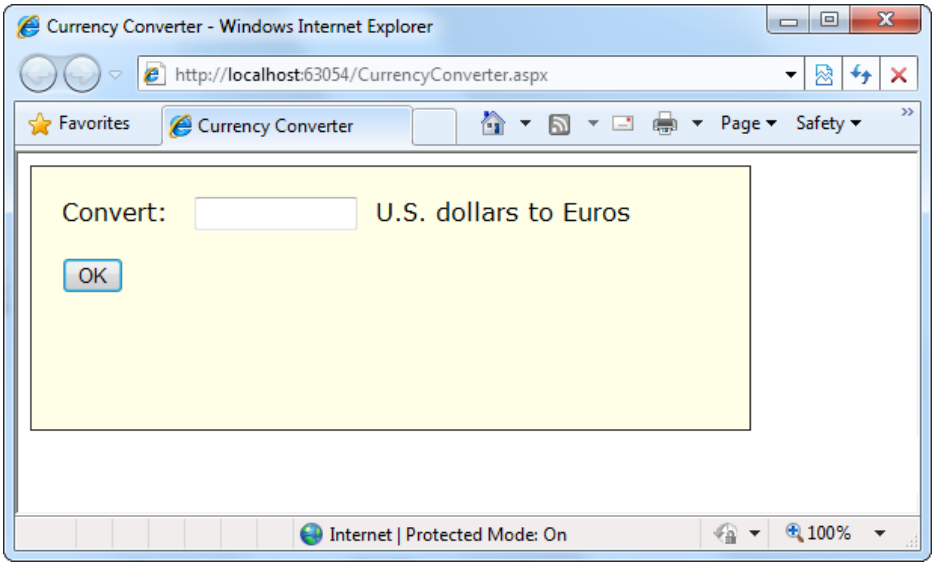

Figure 5-2. *A simple currency converter*

The following listing shows the markup for this page. To make it as clear as possible, this listing omits the style attribute of the element used for the border. This page has two <input> elements: one for the text box and one for the submit button. These elements are enclosed in a <form> tag, so they can submit information to the server when the button is clicked. The rest of the page consists of static text. The ` ` character entity is used to add an extra space to separate the controls. A doctype at the top of the page declares that it's written according to the modern HTML5 standard.

```
<!DOCTYPE html>

<html>
  <head>
    <title>Currency Converter</title>
  </head>
  <body>
    <form method="post">
      <div>
        Convert: 
        <input type="text" />
         U.S. dollars to Euros.
        <br /><br />
        <input type="submit" value="OK" />
      </div>
    </form>
  </body>
</html>
```

■ **Note** In HTML, many input controls are represented by the <input> element. You set the type attribute to indicate the type of control you want. The <input type="text"> tag is a text box, while <input type="submit"> creates a submit button for sending the web page back to the web server. This is quite a bit different from the web controls you'll see in Chapter 6, which use a different element for each type of control.

As it stands, this page looks nice but provides no functionality. It consists entirely of the user interface (HTML elements) and contains no code. It's an ordinary HTML page—not a web form.

The easiest way to convert the currency converter to ASP.NET is to start by generating a new web form in Visual Studio. To do this, follow these steps:

1. Right-click your website in the Solution Explorer and choose Add ➤ Add New Item.
2. In the Add New Item dialog box, choose Web Form (which should be first in the list).
3. Type a name for the new page (such as CurrencyConverter.aspx).
4. Make sure the Place Code in Separate File option is selected (it's in the bottom-right corner of the window).
5. Click Add to create the page.

In the new web form, delete everything that's currently in the .aspx file, except the page directive. The *page directive* gives ASP.NET basic information about how to compile the page. It indicates the language you're using for your code and the way you connect your event handlers. If you're using the code-behind approach, which is recommended, the page directive also indicates where the code file is located and the name of your custom page class.

Finally, copy all the content from the original HTML page, and paste it into the new page, right after the page directive. Here's the resulting web form, with the page directive (in bold) followed by the HTML content that's copied from the original page:

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="CurrencyConverter.aspx.cs" Inherits="CurrencyConverter" %>

<!DOCTYPE html>
<html>
  <head>
    <title>Currency Converter</title>
  </head>
  <body>
    <form method="post">
      <div>
        Convert:  
        <input type="text" />
          U.S. dollars to Euros.
        <br /><br />
        <input type="submit" value="OK" />
      </div>
    </form>
 </body>
</html>
```

Now you need to add the attribute `runat="server"` to each tag that you want to transform into a server control. You should also add an ID attribute to each control that you need to interact with in code. The ID attribute assigns the unique name that you'll use to refer to the control in code.

In the currency converter application, it makes sense to change the input text box and the submit button into HTML server controls. In addition, the <form> element must be processed as a server control to allow ASP.NET to access the controls it contains. Here's the complete, correctly modified page:

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="CurrencyConverter.aspx.cs" Inherits="CurrencyConverter" %>

<!DOCTYPE html>
<html>
  <head>
    <title>Currency Converter</title>
  </head>
  <body>
    <form runat="server">
      <div>
        Convert:  
        <input type="text" ID="US" runat="server" />
          U.S. dollars to Euros.
        <br /><br />
        <input type="submit" value="OK" ID="Convert" runat="server" />
      </div>
    </form>
  </body>
</html>
```

■ **Note** Most ASP.NET controls must be placed inside the <form> section in the page. The <form> element is a part of the standard for HTML forms, and it allows the browser to send information to the web server.

The web page still won't do anything when you run it, because you haven't written any code. However, now that you've converted the static HTML elements to HTML server controls, you're ready to work with them.

View State

To try this page, launch it in Visual Studio by clicking the Play button in the standard Visual Studio toolbar (or just press F5). Remember, the first time you run your web application, you'll be prompted to let Visual Studio modify your web.config file to allow debugging. Click OK to accept its recommendation and launch your web page in the browser. Then, right-click the page and choose View Source in your browser to look at the HTML that ASP.NET sent your way.

The first thing you'll notice is that the HTML that was sent to the browser is slightly different from the information in the .aspx file. First, the `runat="server"` attributes are stripped out (because they have no meaning to the client browser, which can't interpret them). Second, and more important, an additional hidden field has been added to the form. Here's what you'll see (in a slightly simplified form):

```
<!DOCTYPE html>
<html>
  <head>
    <title>Currency Converter</title>
  </head>
```

```
  <body>
    <form ID="form1" method="post" action="CurrencyConverter.aspx">
      <div class="aspNetHidden">
        <input type="hidden" name="__VIEWSTATE" ID="__VIEWSTATE"
         value="dDw3NDg2NTI5MDg7Oz4…" />
      </div>
      <div class="aspNetHidden">
        <input type="hidden" name="__EVENTVALIDATION" ID="__EVENTVALIDATION"
         value="/wEWAwLr3rrOBgLr797…" />
     </div>
     <div>
        Convert:  
        <input type="text" ID="US" name="US" />
          U.S. dollars to Euros.
        <br /><br />
        <input type="submit" value="OK" ID="Convert" name="Convert" />
     </div>
    </form>
  </body>
</html>
```

Keen eyes will notice that ASP.NET has added two new hidden <input> elements, each of which is nested in a separate element container. The second hidden field is used to stop certain types of web page tampering. It works behind the scenes. More interesting is the first hidden field, which stores information, in a compressed format, about the state of every control in the page:

```
<input type="hidden" name="__VIEWSTATE" ID="__VIEWSTATE"
value="dDw3NDg2NTI5MDg7Oz4..." />
```

This information is called the *view state* of the page.

Because the view state information is stored in your page, your code can change control properties at any time, and these changes will automatically "stick." For example, if you change the background color of a box, that box will keep its new color, no matter how many times the page is sent back and forth between the browser and the web server. This is a key part of the web forms programming model. Thanks to view state, you can often forget about the stateless nature of the Internet and treat your page like a continuously running application.

Even though the currency converter program doesn't yet include any code, you'll already notice one change. If you enter information in the text box and click the submit button to post the page, the refreshed page will still contain the value you entered in the text box. (In the original example that uses ordinary HTML elements, the value is cleared every time the page is submitted.) This change occurs because ASP.NET controls automatically retain their state.

The HTML Control Classes

Before you can continue any further with the currency converter, you need to know about the control objects you've created. All the HTML server controls are defined in the System.Web.UI.HtmlControls namespace. Each kind of control has a separate class. Table 5-3 describes the basic HTML server controls and shows you the related HTML element.

Table 5-3. *The HTML Server Control Classes*

Class Name	HTML Element	Description
HtmlForm	<form>	Wraps all the controls on a web page. All ASP.NET server controls must be placed inside an HtmlForm control so that they can send their data to the server when the page is submitted. Visual Studio adds the <form> element to all new pages. When designing a web page, you need to ensure that every other control you add is placed inside the <form> section.
HtmlAnchor	<a>	A hyperlink that the user clicks to jump to another page.
HtmlImage	<img>	A link that points to an image, which will be inserted into the web page at the current location.
HtmlTable, HtmlTableRow, and HtmlTableCell	<table>, <tr>, <th>, and <td>	A table that displays multiple rows and columns of static text.
HtmlInputButton, HtmlInputSubmit, and HtmlInputReset	<input type="button">, <input type="submit">, and <input type="reset">	A button that the user clicks to perform an action (HtmlInputButton), submit the page (HtmlInputSubmit), or clear all the user-supplied values in all the controls (HtmlInputReset).
HtmlButton	<button>	A button that the user clicks to perform an action. This is not supported by all browsers, so HtmlInputButton is usually used instead. The key difference is that the HtmlButton is a container element. As a result, you can insert just about anything inside it, including text and pictures. The HtmlInputButton, on the other hand, is strictly text-only.
HtmlInputCheckBox	<input type="checkbox">	A check box that the user can select or clear. Doesn't include any text of its own.
HtmlInputRadioButton	<input type="radio">	A radio button that can be selected in a group. Doesn't include any text of its own.
HtmlInputText and HtmlInputPassword	<input type="text"> and <input type="password">	A single-line text box enabling the user to enter information. Can also be displayed as a password field (which displays bullets instead of characters to hide the user input).
HtmlTextArea	<textarea>	A large text box for typing multiple lines of text.
HtmlInputImage	<input type="image">	Similar to the <img> tag, but inserts a "clickable" image that submits the page. Using server-side code, you can determine exactly where the user clicked in the image—a technique you'll consider later in this chapter.
HtmlInputFile	<input type="file">	A Browse button and text box that can be used to upload a file to your web server, as described in Chapter 17.

(*continued*)

Table 5-3. (*continued*)

Class Name	HTML Element	Description
HtmlInputHidden	<input type="hidden">	Contains text information that will be sent to the server when the page is posted back but won't be visible in the browser.
HtmlSelect	<select>	A drop-down or regular list box enabling the user to select an item.
HtmlHead and HtmlTitle	<head> and <title>	Represents the header information for the page, which includes information about the page that isn't actually displayed in the page, such as search keywords and the web page title. These are the only HTML server controls that aren't placed in the <form> section.
HtmlGenericControl	Any other HTML element.	This control can represent a variety of HTML elements that don't have dedicated control classes. For example, if you add the `runat="server"` attribute to a element, it's provided to your code as an HtmlGenericControl object. You can identify the type of element by reading the TagName property, which stores a string (for example, "div").

Remember, there are two ways to add any HTML server control. You can add it by hand to the markup in the .aspx file (simply insert the ordinary HTML element, and add the `runat="server"` attribute). Alternatively, you can drag the control from the HTML section of the Toolbox in Visual Studio, and drop it onto the design surface of a web page. However, this approach doesn't work for every HTML server control, because they don't all appear in the Toolbox.

So far, the currency converter defines three controls, which are instances of the HtmlForm, HtmlInputText, and HtmlInputButton classes, respectively. It's useful to know the class names if you want to look up information about these classes in the class library reference on Microsoft's MSDN website (http://tinyurl.com/cq63b6m). Table 5-4 gives a quick overview of some of the most important control properties.

Table 5-4. *Important HTML Control Properties*

Control	Most Important Properties
HtmlAnchor	HRef, Target
HtmlImage	Src, Alt, Width, Height
HtmlInputCheckBox and HtmlInputRadioButton	Checked
HtmlInputText	Value
HtmlTextArea	Value
HtmlInputImage	Src, Alt
HtmlSelect	Items (collection)
HtmlGenericControl	InnerText and InnerHtml

Adding the Currency Converter Code

To add some functionality to the currency converter, you need to add some ASP.NET code. Web forms are event-driven, which means every piece of code acts in response to a specific event. In the simple currency converter page example, the most useful event occurs when the user clicks the submit button (named Convert). The HtmlInputButton allows you to react to this action by handling the ServerClick event.

Before you continue, it makes sense to add another control that can display the result of the calculation. In this case, you can use an HTML <p>, or paragraph. The <p> element is one way to insert a block of formatted text into a web page. Here's the HTML that creates a <p> element named Result that starts out with no text:

```
<p style="font-weight: bold" ID="Result" runat="server">/p>
```

The style attribute applies the CSS properties used to format the text. In this example, it merely applies a bold font using the CSS font-weight property. If you're not familiar with CSS, don't worry—you'll get a closer review of the essentials in Chapter 12.

The example now has the following four server controls:

- A form (which is represented by the HtmlForm object). This is the only control you do not need to access in your code-behind class.
- An input text box named US (HtmlInputText object).
- A submit button named Convert (HtmlInputButton object).
- A <p> element named Result (HtmlGenericControl object).

Listing 5-1 shows the revised web page (CurrencyConverter.aspx), but leaves out the doctype to save space. Listing 5-2 shows the code-behind class (CurrencyConverter.aspx.vb CurrencyConverter.aspx.cs). The code-behind class includes an event handler that reacts when the Convert button is clicked. It calculates the currency conversion and displays the result.

Listing 5-1. CurrencyConverter.aspx

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="CurrencyConverter.aspx.cs" Inherits="CurrencyConverter" %>
<!DOCTYPE html>
<html>
  <head>
    <title>Currency Converter</title>
  </head>
  <body>
    <form runat="server">
      <div>
        Convert:  
        <input type="text" ID="US" runat="server" />
          U.S. dollars to Euros.
        <br /><br />
        <input type="submit" value="OK" ID="Convert" runat="server"
         OnServerClick="Convert_ServerClick" />
        <br /><br />
        <p style="font-weight: bold" ID="Result" runat="server"></p>
      </div>
    </form>
  </body>
</html>
```

Listing 5-2. CurrencyConverter.aspx.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class CurrencyConverter : System.Web.UI.Page
{
    protected void Convert_ServerClick(object sender, EventArgs e)
    {
        decimal USAmount=Decimal.Parse(US.Value);
        decimal euroAmount=USAmount * 0.85 M;
        Result.InnerText=USAmount.ToString()+" U.S. dollars=";
        Result.InnerText+= euroAmount.ToString()+" Euros.";
    }
}
```

The code-behind class is a typical example of an ASP.NET page. You'll notice the following conventions:

- It starts with several *using* statements. This provides access to all the important namespaces. This is a typical first step in any code-behind file.
- The page class is defined with the `partial` keyword. That's because your class code is merged with another code file that you never see. This extra code, which ASP.NET generates automatically, defines all the server controls that are used on the page. This allows you to access them by name in your code.
- The page defines a single event handler. This event handler retrieves the value from the text box, converts it to a numeric value, multiplies it by a preset conversion ratio (which would typically be stored in another file or a database), and sets the text of the <p> element. You'll notice that the event handler accepts two parameters (sender and e). This is the .NET standard for all control events. It allows your code to identify the control that sent the event (through the sender parameter) and retrieve any other information that may be associated with the event (through the e parameter). You'll see examples of these advanced techniques in the next chapter, but for now, it's important to realize that you won't be allowed to handle an event unless your event handler has the correct, matching signature.
- The event handler is connected to the control event by using the OnServerClick attribute in the <input> tag for the button. You'll learn more about how this hookup works in the next section.

■ **Note** Unlike with web controls, you can't create event handlers for HTML server controls by using the Properties window. Instead, you must type the method in by hand, making sure to include the Handles clause at the end. Another option is to use the two drop-down lists at the top of the code window. To take this approach, choose the control in the list on the left (for example, Convert), and then choose the event you want to handle in the list on the right (for example, ServerClick). Visual Studio will create the corresponding event handler. Instead, you must type the method in by hand. You must also modify the control tag to connect your event handler. For example, to connect the Convert button to the method named Convert_ServerClick, you must add `OnServerClick="Convert_ServerClick"` to the control tag.

- The += operator is used to quickly add information to the end of the label, without replacing the existing text.
- The event handler uses ToString() to convert the decimal value to text so it can be added to the InnerText property. In this particular statement, you don't need ToString(), because C# is intelligent enough to realize that you're joining together pieces of text. However, this isn't always the case, so it's best to be explicit about data type conversions.

You can launch this page to test your code. When you enter a value and click the OK button, the page is resubmitted, the event-handling code runs, and the page is returned to you with the conversion details (see Figure 5-3).

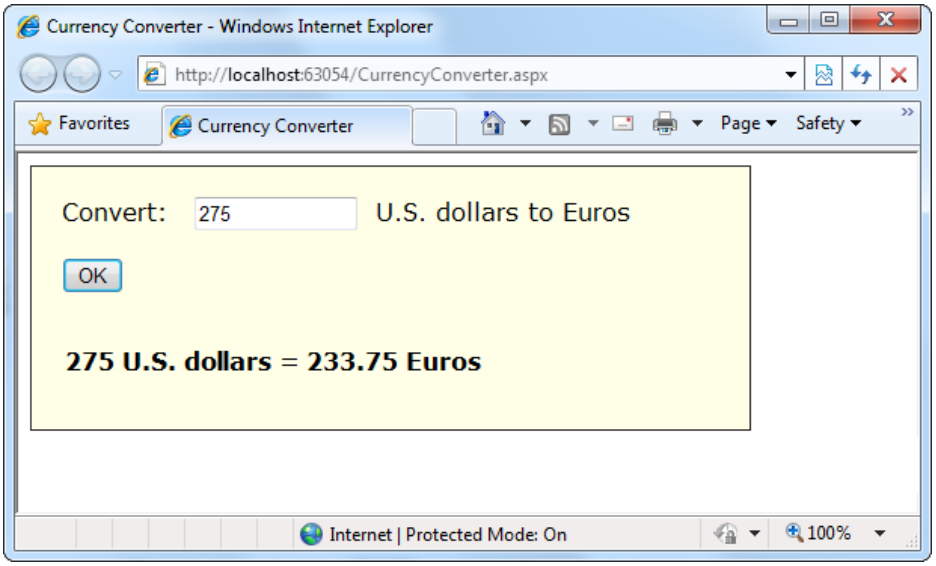

Figure 5-3. *The ASP.NET currency converter*

Event Handling

When the user clicks the Convert button and the page is sent back to the web server, ASP.NET needs to know exactly what code you want to run. To create this relationship and connect an event to an event-handling method, you need to add an attribute in the control tag.

For example, if you want to handle the ServerClick method of the Convert button, you simply need to set the OnServerClick attribute in the control tag with the name of the event handler you want to use:

```
<input type="submit" value="OK" ID="Convert"
OnServerClick="Convert_ServerClick" runat="server">
```

ASP.NET controls always use this syntax. When attaching an event handler in the control tag, you use the name of the event preceded by the word *On*. For example, if you want to handle an event named ServerChange, you'd set an attribute in the control tag named OnServerChange. When you double-click a server control on the design surface, Visual Studio adds the event handler and sets the event attribute to match. (Now that you understand this process, you'll understand the source of a common error. If you double-click a control to create an event handler, but you then delete the event-handling method, you'll end up with an event attribute that points to a nonexistent event. As a result, you'll receive an error the next time you run the page. To fix the problem, you need to remove the event attribute from the control tag.)

■ **Note** There's one ASP.NET object that doesn't use this attribute system, because it doesn't have a control tag—the web page. Instead, ASP.NET connects the web page events based on method names. In other words, if you have a method named Page_Load() in your page class, and it has the right signature (it accepts two parameters, an object representing the sender and an EventArgs object), ASP.NET connects this event handler to the Page.Load event automatically. This feature is called **automatic event wireup**.

ASP.NET allows you to use another technique to connect events—you can do it by using code, just as you saw in Chapter 3. For example, here's the code that's required to hook up the ServerClick event of the Convert button using manual event wireup:

```
Convert.ServerClick+= this.Convert_ServerClick;
```

If you're using code like this, you'd probably add it to the Page_Load() method so it connects your event handlers when the page is first initialized.

Seeing as Visual Studio handles event wireup, why should ASP.NET developers care that they have two ways to hook up an event handler? Well, most of the time you won't worry about it. But the manual event wireup technique is useful in certain circumstances. The most common example occurs if you want to create a control object and add it to a page dynamically at runtime. In this situation, you can't hook up the event handler through the control tag, because there isn't a control tag. Instead, you need to create the control and attach its event handlers by using code. (The next chapter has an example of how you can use dynamic control creation to fill in a table.)

Behind the Scenes with the Currency Converter

So, what really happens when ASP.NET receives a request for the CurrencyConverter.aspx page? The process unfolds over several steps:

1. The request for the page is sent to the web server. If you're running a live site, the web server is almost certainly IIS, which you'll learn more about in Chapter 26. If you're running the page in Visual Studio, the request is sent to the built-in test server.
2. The web server determines that the .aspx file extension is registered with ASP.NET. If the file extension belonged to another service (as it would for .html files, for example), ASP.NET would never get involved.
3. If this is the first time a page in this application has been requested, ASP.NET automatically creates the application domain. It also compiles all the web page code for optimum performance, and caches the compiled files. If this task has already been performed, ASP.NET will reuse the compiled version of the page.
4. The compiled CurrencyConverter.aspx page acts like a miniature program. It starts firing events (most notably, the Page.Load event). However, you haven't created an event handler for that event, so no code runs. At this stage, everything is working together as a set of in-memory .NET objects.
5. When the code is finished, ASP.NET asks every control in the web page to render itself into the corresponding HTML markup.

■ **Tip** In fact, ASP.NET performs a little sleight of hand and may customize the output with additional client-side JavaScript or DHTML if it detects that the client browser supports it. In the case of CurrencyConverter.aspx, the output of the page is too simple to require this type of automatic tweaking.

6. The final page is sent to the user, and the application ends.

The description is lengthy, but it's important to start with a good understanding of the fundamentals. When you click a button on the page, the entire process repeats itself. However, in step 4 the ServerClick event fires for the HtmlInputButton right after the Page.Load event, and your code runs.

Figure 5-4 illustrates the stages in a web page request.

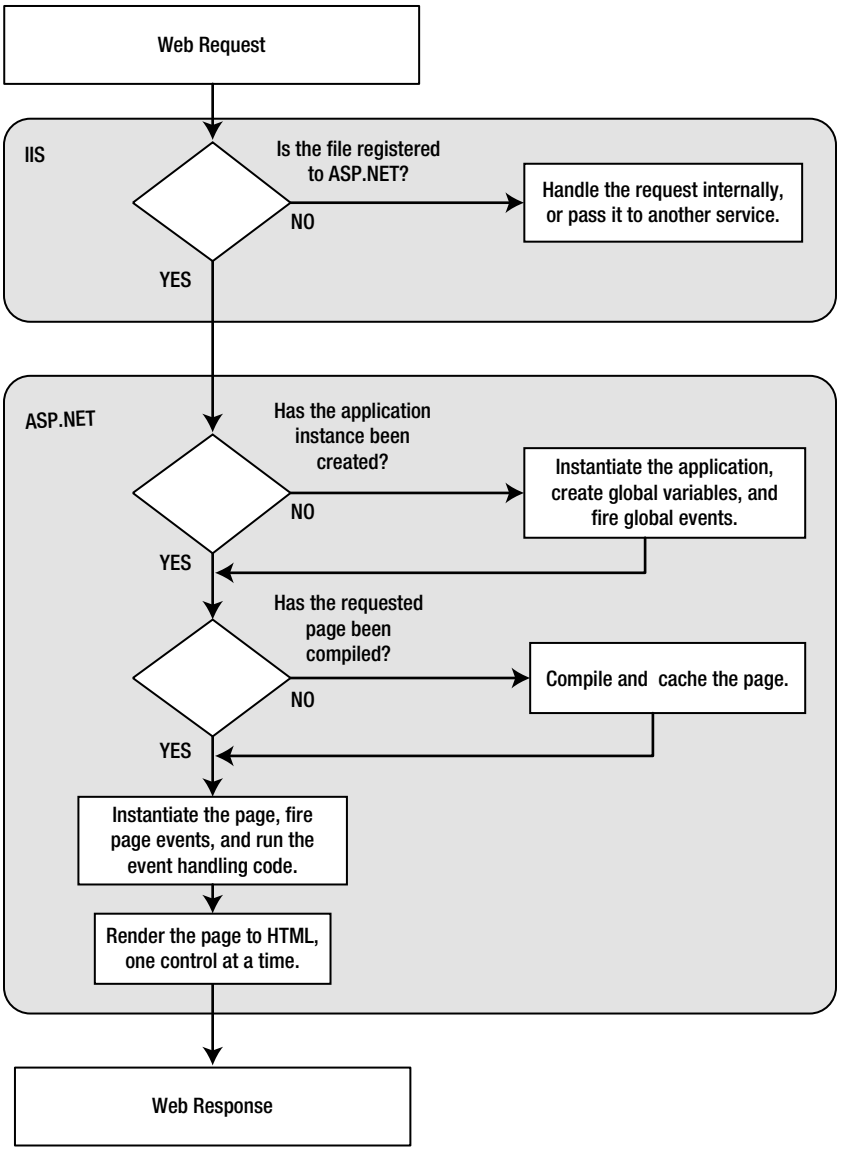

Figure 5-4. *The stages in an ASP.NET request*

The most important detail is that your code works with objects. The final step is to transform these objects into the appropriate HTML output. A similar conversion from objects to output happens with a Windows program in .NET, but it's so automatic that programmers rarely give it much thought. Also, in those environments, the code always runs locally. In an ASP.NET application, the code runs in a protected environment on the server. The client sees the results only after the web page processing has ended and the web page object has been released from memory.

Error Handling

The currency converter expects that the user will enter a number before clicking the Convert button. If the user enters something else—for example, a sequence of text or special characters that can't be interpreted as a number—an error will occur when the code attempts to use the Decimal.Parse() method. In the current version of the currency converter, this error will completely derail the event-handling code. Because this example doesn't include any code to handle errors, ASP.NET will simply send an error page back to the user describing the problem.

In Chapter 7, you'll learn how to deal with errors by catching them, neutralizing them, and informing the user more gracefully. However, even without these abilities, you can rework the code that responds to the ServerClick event to avoid potential errors. One good approach is to use the Decimal.TryParse() method instead of Decimal.Parse(). Unlike Parse(), TryParse() does not generate an error if the conversion fails—it simply informs you of the problem.

TryParse() accepts two parameters. The first parameter is the value you want to convert (in this example, US.Value). The second parameter is an output parameter that will receive the converted value (in this case, the variable named USAmount). What's special about TryParse() is its Boolean return value, which indicates whether the conversion was successful (true) or not (false).

Here's a revised version of the ServerClick event handler that uses TryParse() to check for conversion problems and inform the user:

```
protected void Convert_ServerClick(object sender, EventArgs e)
{
    decimal USAmount;

    // Attempt the conversion.
    bool success=Decimal.TryParse(US.Value, out USAmount);

    // Check if it succeeded.
    if (success)
    {
        // The conversion succeeded.
        decimal euroAmount=USAmount * 0.85M;
        Result.InnerText=USAmount.ToString()+" U.S. dollars=";
        Result.InnerText+= euroAmount.ToString()+" Euros.";
    }
    else
    {
        // The conversion failed.
        Result.InnerText="The number you typed in was not in the " +
          "correct format. Use only numbers.";
    }
}
```

Dealing with error conditions like these is an essential technique in a real-world web page.

Improving the Currency Converter

Now that you've looked at the basic server controls, it might seem that their benefits are fairly minor compared with the cost of learning a whole new system of web programming. In this section, you'll start to extend the currency converter application. You'll see how you can "snap in" additional functionality to the existing program in an elegant, modular way. As the program grows, ASP.NET handles its complexity easily, because it cleanly separates the HTML markup from the code that manipulates it.

Adding Multiple Currencies

The first task is to allow the user to choose a destination currency. In this case, you need to use a drop-down list box. In HTML, a drop-down list is represented by a <select> element that contains one or more <option> elements. Each <option> element corresponds to a separate item in the list.

To reduce the amount of HTML in the currency converter, you can define a drop-down list without any list items by adding an empty <select> tag. As long as you ensure that this <select> tag is a server control (by giving it an ID and adding the `runat="server"` attribute), you'll be able to interact with it in code and add the required items when the page loads.

Here's the revised HTML for the CurrencyConverter.aspx page:

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="CurrencyConverter.aspx.cs" Inherits="CurrencyConverter" %>
<!DOCTYPE html>
<html xmlns="
http://www.w3.org/1999/xhtml
">
  <head>
    <title>Currency Converter</title>
  </head>
  <body>
    <form runat="server">
      <div>
        Convert:  
        <input type="text" ID="US" runat="server" />
          U.S. dollars to  
        <select ID="Currency" runat="server" />
        <br /><br />
        <input type="submit" value="OK" ID="Convert"
         OnServerClick="Convert_ServerClick" runat="server" />
        <br /><br />
        <p style="font-weight: bold" ID="Result" runat="server"></p>
      </div>
    </form>
  </body>
</html>
```

The currency list can now be filled using code at runtime. In this case, the ideal event is the Page.Load event, which is fired at the beginning of the page-processing sequence. Here's the code you need to add to the CurrencyConverter page class:

```
protected void Page_Load(Object sender, EventArgs e)
{
    if (this.IsPostBack == false)
```

```
    {
        Currency.Items.Add("Euro");
        Currency.Items.Add("Japanese Yen");
        Currency.Items.Add("Canadian Dollar");
    }
}
```

Dissecting the Code . . .

This example illustrates two important points:

- You can use the Items property to get items in a list control. This allows you to append, insert, and remove <option> elements (which represent the items in the list). Remember, when generating dynamic content with a server control, you set the properties, and the control creates the appropriate HTML tags.
- Before adding any items to this list, you need to make sure this is the first time the page is being served to this particular user. Otherwise, the page will continuously add more items to the list or inadvertently overwrite the user's selection every time the user interacts with the page. To perform this test, you check the IsPostBack property of the current Page. In other words, IsPostback is a property of the CurrencyConverter class, which CurrencyConverter inherits from the generic Page class. If IsPostBack is false, the page is being created for the first time, and it's safe to initialize it.

Storing Information in the List

Of course, if you're a veteran HTML coder, you know that each item in a list also provides a value attribute that you can use to store a value for each item in the list. Because the currency converter uses a short list of hard-coded currencies, this is an ideal place to store the currency conversion rate.

To set the value attribute, you need to create a ListItem object for every item in the list and add that to the HtmlSelect control. The ListItem class provides a constructor that lets you specify the text and value at the same time that you create it, thereby allowing condensed code like this:

```
protected void Page_Load(Object sender, EventArgs e)
{
    if (this.IsPostBack == false)
    {
        // The HtmlSelect control accepts text or ListItem objects.
        Currency.Items.Add(new ListItem("Euros", "0.85"));
        Currency.Items.Add(new ListItem("Japanese Yen", "110.33"));
        Currency.Items.Add(new ListItem("Canadian Dollars", "1.2"));
    }
}
```

To complete the example, you must rewrite the calculation code to take the selected currency into account, as follows:

```
protected void Convert_ServerClick(object sender, EventArgs e)
{
    decimal oldAmount;
    bool success=Decimal.TryParse(US.Value, out oldAmount);
```

```
    if (success)
    {
        // Retrieve the selected ListItem object by its index number.
        ListItem item=Currency.Items[Currency.SelectedIndex];

        decimal newAmount=oldAmount * Decimal.Parse(item.Value);
        Result.InnerText=oldAmount.ToString()+" U.S. dollars=";
        Result.InnerText+= newAmount.ToString()+" "+item.Text;
    }
}
```

Figure 5-5 shows the revamped currency converter.

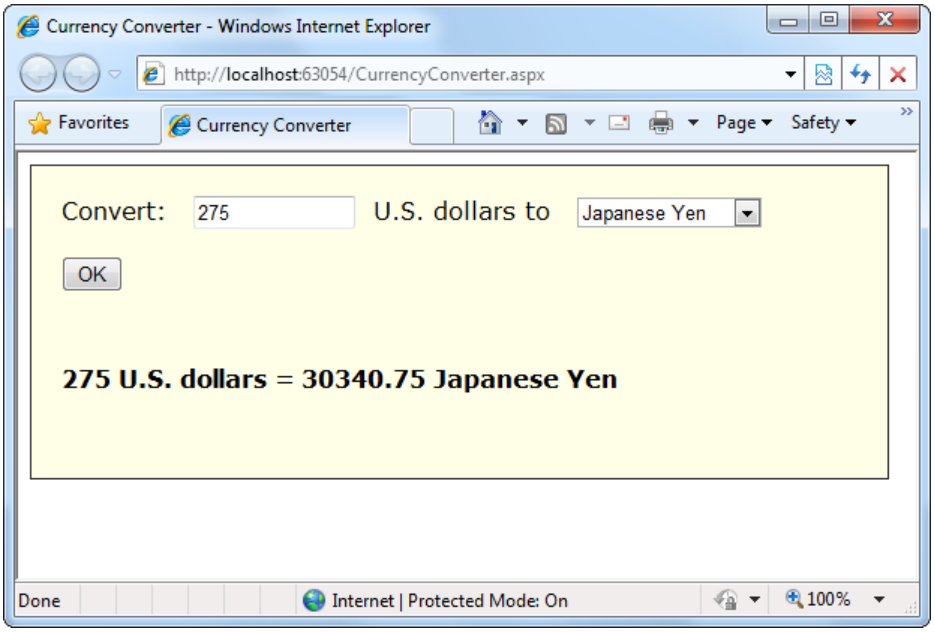

Figure 5-5. *The multicurrency converter*

All in all, this is a good example of how you can store information in HTML tags by using the value attribute. However, in a more realistic application, you wouldn't actually store the currency rate in the list. Instead, you would just store some sort of unique identifying ID value. Then, when the user submits the page, you would retrieve the corresponding conversion rate from another storage location, such as a database.

Adding Linked Images

Adding other functionality to the currency converter is just as easy as adding a new button. For example, it might be useful for the utility to display a currency conversion rate graph. To provide this feature, the program would need an additional button and image control.

Here's the revised HTML:

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="CurrencyConverter.aspx.cs" Inherits="CurrencyConverter" %>
<!DOCTYPE html>
<html>
  <head>
    <title>Currency Converter</title>
  </head>
  <body>
    <form runat="server">
      <div>
        Convert:  
        <input type="text" ID="US" runat="server" />
          U.S. dollars to  
        <select ID="Currency" runat="server" />
        <br /><br />
        <input type="submit" value="OK" ID="Convert"
         OnServerClick="Convert_ServerClick" runat="server" />
        <input type="submit" value="Show Graph" ID="ShowGraph" runat="server" />
        <br /><br />
        <img ID="Graph" src="" alt="Currency Graph" runat="server" />
        <br /><br />
        <p style="font-weight: bold" ID="Result" runat="server"></p>
      </div>
    </form>
  </body>
</html>
```

As it's currently declared, the image doesn't refer to a picture. For that reason, it makes sense to hide it when the page is first loaded by using this code:

```
protected void Page_Load(Object sender, EventArgs e)
{
     if (this.IsPostBack == false)
     {
         Currency.Items.Add(new ListItem("Euros", "0.85"));
         Currency.Items.Add(new ListItem("Japanese Yen", "110.33"));
         Currency.Items.Add(new ListItem("Canadian Dollars", "1.2"));

         Graph.Visible=false;
     }
}
```

Interestingly, when a server control is hidden, ASP.NET omits it from the final HTML page.

Now you can handle the click event of the new button to display the appropriate picture. The currency converter has three possible picture files—pic0.png, pic1.png, and pic2.png. It chooses one based on the index of the selected item (so it uses pic0.png if the first currency is selected, pic1.png for the second, and so on). Here's the code that shows the chart:

```
protected void ShowGraph_ServerClick(Object sender, EventArgs e)
{
      Graph.Src = "Pic" + Currency.SelectedIndex.ToString() + ".png";
      Graph.Visible = true;
}
```

You need to make sure you link to the event handler through the button, so modify the <input> element for the button as follows:

```
<input type="submit" value="Show Graph" ID="ShowGraph"
OnServerClick="ShowGraph_ServerClick" runat="server" />
```

You'll also need to change the graph picture when the currency is changed, using this line of code in the Convert_ServerClick() method:

```
Graph.Src = "Pic" + Currency.SelectedIndex.ToString() + ".png";
```

If this were a more complex task (for example, one that required multiple lines of code), it would make sense to put it in a separate private method. You could then call that method from both the ShowGraph_ServerClick() and Convert_ServerClick() event-handling methods, making sure you don't duplicate your code.

Already the currency converter is beginning to look more interesting, as shown in Figure 5-6.

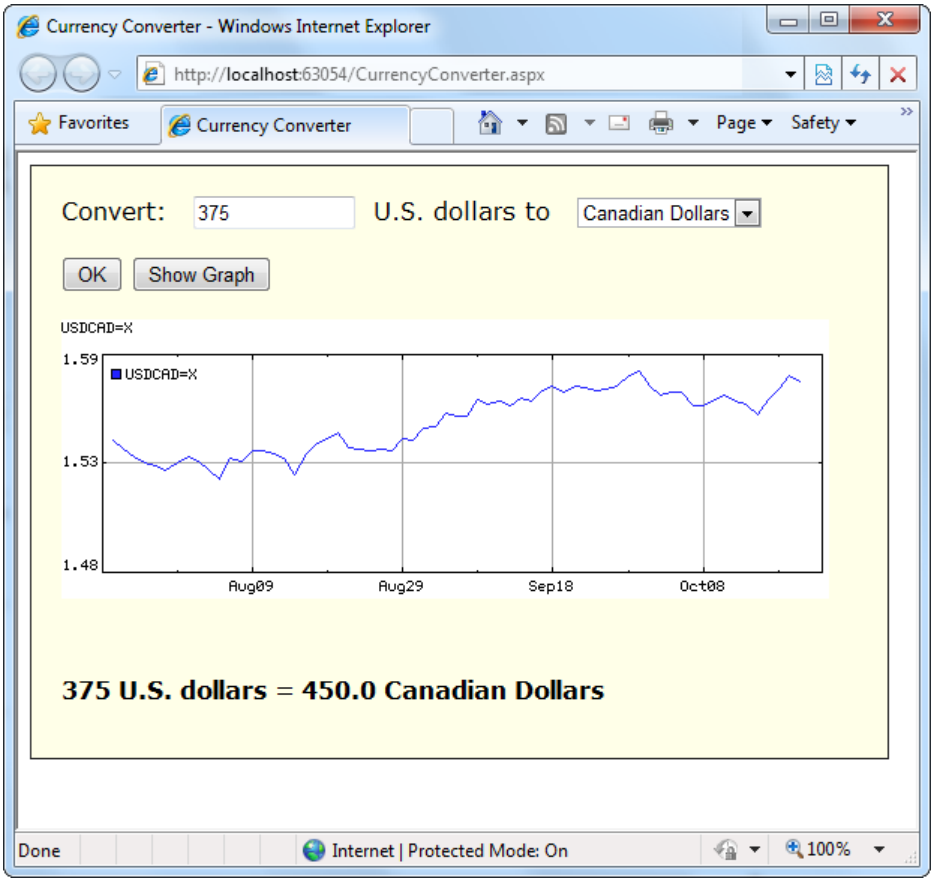

Figure 5-6. *The currency converter with an image control*

■ **Tip** Of course, in a real currency converter, a fixed picture wouldn't do. You could pull a recent image out of a database, but most likely you'd want to generate the currency chart yourself, using server-side drawing instruction. Chapter 11 explains how you would go about this somewhat tedious task.

Setting Styles

In addition to a limited set of properties, each HTML control also provides access to the CSS attributes through its Style collection. To use this collection, you need to specify the name of the CSS style attribute and the value you want to assign to it. Here's the basic syntax:

```
ControlName.Style["AttributeName"] = "AttributeValue";
```

For example, you could use this technique to emphasize an invalid entry in the currency converter with the color red. In this case, you would also need to reset the color to its original value for valid input, because the control uses view state to remember all its settings, including its style properties:

```
protected void Convert_ServerClick(object sender, EventArgs e)
{
    decimal oldAmount;
    bool success=Decimal.TryParse(US.Value, out oldAmount);

    if ((oldAmount<= 0) || (success == false))
    {
        Result.Style["color"]="Red";
        Result.InnerText="Specify a positive number";
    }
    else
    {
        Result.Style["color"]="Black";

        // Retrieve the selected ListItem object by its index number.
        ListItem item=Currency.Items[Currency.SelectedIndex];

        decimal newAmount=oldAmount * Decimal.Parse(item.Value);
        Result.InnerText=oldAmount.ToString()+" U.S. dollars=";
        Result.InnerText+= newAmount.ToString()+" "+item.Text;
    }
}
```

■ **Tip** The Style collection sets the style attribute in the HTML tag with a list of formatting options such as font family, size, and color. You'll learn more in Chapter 12. But if you aren't familiar with CSS styles, you don't **need** to use them. Instead, you could use web controls, which provide higher-level properties that allow you to configure their appearance and automatically create the appropriate style attributes. You'll learn about web controls in the next chapter.

This concludes the simple currency converter application, which now boasts automatic calculation, linked images, and dynamic formatting. In the following sections, you'll look at the building blocks of ASP.NET interfaces more closely.

Taking a Deeper Look at HTML Control Classes

Related classes in the .NET Framework use inheritance to share functionality. For example, every HTML control inherits from the base class HtmlControl. The HtmlControl class provides essential features that every HTML server control uses. Figure 5-7 shows the inheritance diagram.

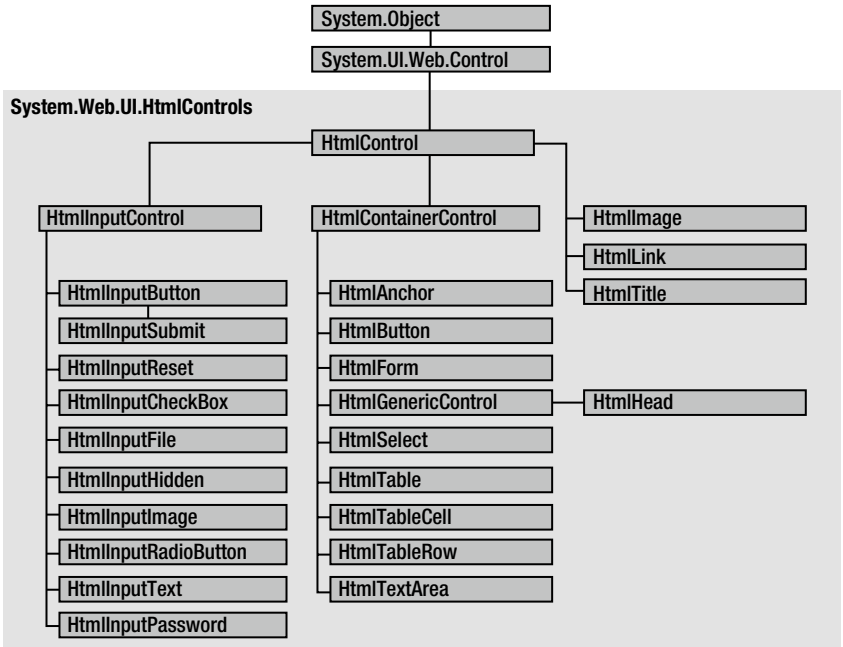

Figure 5-7. *HTML control inheritance*

This section dissects the ASP.NET classes that are used for HTML server controls. You can use this material to help understand the common elements that are shared by all HTML controls. For the specific details about each HTML control, you can refer to the class library reference on the MSDN website at `http://tinyurl.com/cq63b6m`.

HTML server controls generally provide properties that closely match their tag attributes. For example, the HtmlImage class provides Align, Alt, Border, Src, Height, and Width properties. For this reason, users who are familiar with HTML syntax will find that HTML server controls are the most natural fit. Users who aren't as used to HTML will probably find that web controls (described in the next chapter) have a more intuitive set of properties.

HTML Control Events

HTML server controls also provide one of two possible events: ServerClick or ServerChange.

The ServerClick event is simply a click that's processed on the server side. It's provided by most button controls, and it allows your code to take immediate action. For example, consider the HtmlAnchor control, which

is the server control that represents the common HTML hyperlink (the <a> element). There are two ways to use the HtmlAnchor control. One option is to set its HtmlAnchor.HRef property to a URL, in which case the hyperlink will behave exactly like the ordinary HTML <a> element (the only difference being that you can set the URL dynamically in your code). The other option is to handle the HtmlAnchor.ServerClick event. In this case, when the link is clicked, it will actually post back the page, allowing your code to run. The user won't be redirected to a new page unless you provide extra code to forward the request.

The ServerChange event responds when a change has been made to a text or selection control. This event isn't as useful as it appears, because it doesn't occur until the page is posted back (for example, after the user clicks a submit button). At this point, the ServerChange event occurs for all changed controls, followed by the appropriate ServerClick. The Page.Load event is the first to fire, but you have no way to know the order of events for other controls.

Table 5-5 shows which controls provide a ServerClick event and which ones provide a ServerChange event.

Table 5-5. *HTML Control Events*

Event	Controls That Provide It
ServerClick	HtmlAnchor, HtmlButton, HtmlInputButton, HtmlInputImage, HtmlInputReset
ServerChange	HtmlInputText, HtmlInputCheckBox, HtmlInputRadioButton, HtmlInputHidden, HtmlSelect, HtmlTextArea

Advanced Events with the HtmlInputImage Control

Chapter 4 introduced the .NET event standard, which dictates that every event should pass exactly two pieces of information. The first parameter identifies the object (in this case, the control) that fired the event. The second parameter is a special object that can include additional information about the event.

In the examples you've looked at so far, the second parameter (e) has always been used to pass an empty System.EventArgs object. This object doesn't contain any additional information—it's just a glorified placeholder. Here's one such example:

```
protected void Convert_ServerClick(Object sender, EventArgs e)
{ ... }
```

In fact, only one HTML server control sends additional information: the HtmlInputImage control. It sends an ImageClickEventArgs object (from the System.Web.UI namespace) that provides X and Y properties representing the location where the image was clicked. You'll notice that the definition for the HtmlInputImage.ServerClick event handler is a little different from the event handlers used with other controls:

```
protected void ImgButton_ServerClick(Object sender, ImageClickEventArgs e)
{ ... }
```

Using this additional information, you can replace multiple button controls and image maps with a single, intelligent HtmlInputImage control.

Here's the markup you need to create the HtmlInputImage control for this example:

```
<input type="image" ID="ImgButton" runat="server" src="button.png"
OnServerClick="ImgButton_ServerClick" />
```

The sample ImageTest.aspx page shown in Figure 5-8 puts this feature to work with a simple graphical button. Depending on whether the user clicks the button border or the button surface, a different message is displayed.

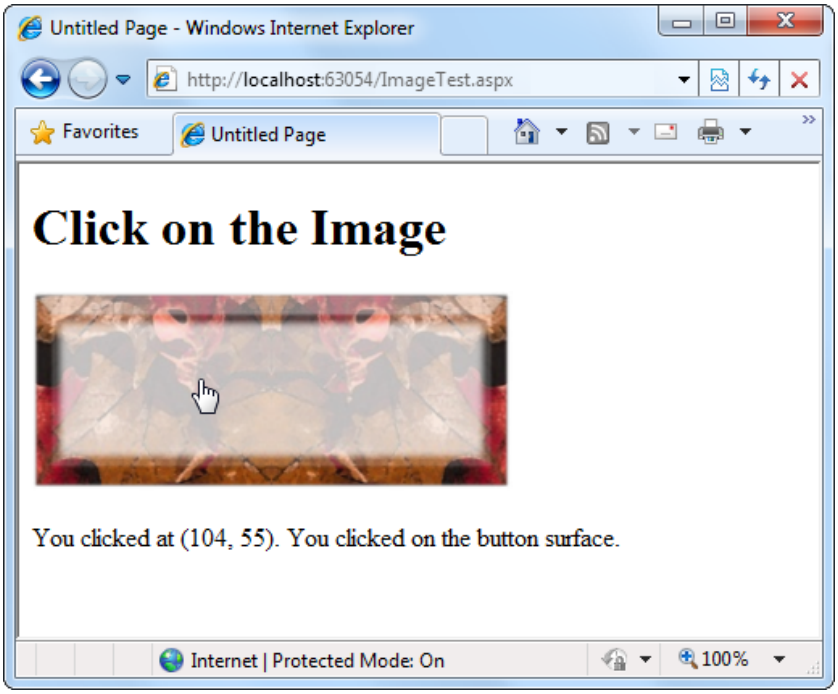

Figure 5-8. *Using an HtmlInputImage control*

The page code examines the click coordinates provided by the ImageClickEventArgs object and displays them in another control. Here's the page code you need:

```
public partial class ImageTest : System.Web.UI.Page
{
    protected void ImgButton_ServerClick(Object sender,
      ImageClickEventArgs e)
    {
        Result.InnerText = "You clicked at (" + e.X.ToString() +
          ", "+e.Y.ToString() + "). ";

        if ((e.Y<100) && (e.Y>20) && (e.X>20) && (e.X<275))
        {
            Result.InnerText += "You clicked on the button surface.";
        }
        else
        {
            Result.InnerText += "You clicked the button border.";
        }
    }
}
```

The HtmlControl Base Class

Every HTML control inherits from the base class HtmlControl. This relationship means that every HTML control will support a basic set of properties and features. Table 5-6 shows these properties.

Table 5-6. *HtmlControl Properties*

Property	Description
Attributes	Provides a collection of all the attributes that are set in the control tag, and their values. Rather than reading or setting an attribute through the Attributes, it's better to use the corresponding property in the control class. However, the Attributes collection is useful if you need to add or configure a custom attribute or an attribute that doesn't have a corresponding property.
Controls	Provides a collection of all the controls contained inside the current control. (For example, a server control could contain an <input> server control.) Each object is provided as a generic System.Web.UI.Control object so that you may need to cast the reference to access control-specific properties.
Disabled	Disables the control when set to true, thereby ensuring that the user cannot interact with it, and its events will not be fired.
EnableViewState	Disables the automatic state management for this control when set to false. In this case, the control will be reset to the properties and formatting specified in the control tag every time the page is posted back. If this is set to true (the default), the control stores its state in a hidden input field on the page, thereby ensuring that any changes you make in code are remembered. (For more information, see the "View State" section earlier in this chapter.)
Page	Provides a reference to the web page that contains this control as a System.Web.UI.Page object.
Parent	Provides a reference to the control that contains this control. If the control is placed directly on the page (rather than inside another control), it will return a reference to the page object.
Style	Provides a collection of CSS style properties that can be used to format the control.
TagName	Indicates the name of the underlying HTML element (for example, img or div).
Visible	Hides the control when set to false and will not be rendered to the final HTML page that is sent to the client.

To set the initial value of a property, you can configure the control in the Page.Load event handler, or you can adjust the control tag in the .aspx file by adding attributes. Note that the Page.Load event occurs after the page is initialized with the default values and the tag settings. This means your code can override the properties set in the tag (but not vice versa).

The HtmlContainerControl Class

Any HTML control that requires a closing tag inherits from the HtmlContainer class (which in turn inherits from the more basic HtmlControl class). For example, elements such as <a>, <form>, and always use a closing tag, because they can contain other HTML elements. On the other hand, elements such as <img> and <input> are used only as stand-alone tags. Thus, the HtmlAnchor, HtmlForm, and HtmlGenericControl classes inherit from HtmlContainerControl, while HtmlInputImage and HtmlInputButton do not.

The HtmlContainer control adds two properties to those defined in HtmlControl, as described in Table 5-7.

Table 5-7. *HtmlContainerControl Properties*

Property	Description
InnerHtml	The HTML content between the opening and closing tags of the control. Special characters that are set through this property will not be converted to the equivalent HTML entities. This means you can use this property to apply formatting with nested tags such as <b>, <i>, and <h1>.
InnerText	The text content between the opening and closing tags of the control. Special characters will be automatically converted to HTML entities and displayed as text (for example, the less-than character (<) will be converted to `<` and will be displayed as <in the web page). This means you can't use HTML tags to apply additional formatting with this property. The simple currency converter page uses the InnerText property to enter results into a <p> element.

The HtmlInputControl Class

The HtmlInputControl class inherits from HtmlControl and adds some properties (shown in Table 5-8) that are used for the <input> element. As you've already learned, the <input> element can represent different controls, depending on the type attribute. The <input type = "text"> element is a text box, and <input type = "submit"> is a button.

Table 5-8. *HtmlInputControl Properties*

Property	Description
Type	Provides the type of input control. For example, a control based on <input type = "file"> would return *file* for the type property.
Value	Returns the contents of the control as a string. In the simple currency converter, this property allowed the code to retrieve the information entered in the text input control.

Using the Page Class

One control you haven't considered in detail yet is the Page class. As explained in the previous chapter, every web page is a custom class that inherits from System.Web.UI.Page. By inheriting from this class, your web page class acquires a number of properties and methods that your code can use. These include properties for enabling caching, validation, and tracing, which are discussed throughout this book. Table 5-9 provides an overview of some of the more fundamental Page class properties, which you'll use throughout this book.

Table 5-9. *Basic Page Properties*

Property	Description
IsPostBack	This Boolean property indicates whether this is the first time the page is being run (false) or whether the page is being resubmitted in response to a control event, typically with stored view state information (true). You'll usually check this property in the Page.Load event handler to ensure that your initial web page initialization is performed only once.
EnableViewState	When set to false, this overrides the EnableViewState property of the contained controls, thereby ensuring that no controls will maintain state information.
Application	This collection holds information that's shared between all users in your website. For example, you can use the Application collection to count the number of times a page has been visited. You'll learn more in Chapter 8.
Session	This collection holds information for a single user, so it can be used in different pages. For example, you can use the Session collection to store the items in the current user's shopping basket on an e-commerce website. You'll learn more in Chapter 8.
Cache	This collection allows you to store objects that are time-consuming to create so they can be reused in other pages or for other clients. This technique, when implemented properly, can improve performance of your web pages. Chapter 23 discusses caching in detail.
Request	This refers to an HttpRequest object that contains information about the current web request. You can use the HttpRequest object to get information about the user's browser, although you'll probably prefer to leave these details to ASP.NET. You'll use the HttpRequest object to transmit information from one page to another with the query string in Chapter 8.
Response	This refers to an HttpResponse object that represents the response ASP.NET will send to the user's browser. You'll use the HttpResponse object to create cookies in Chapter 8, and you'll see how it allows you to redirect the user to a different web page later in this chapter.
Server	This refers to an HttpServerUtility object that allows you to perform a few miscellaneous tasks. For example, it allows you to encode text so that it's safe to place it in a URL or in the HTML markup of your page. You'll learn more about these features in this chapter.
User	If the user has been authenticated, this property will be initialized with user information. Chapter 19 describes this property in more detail.

Many of the Page properties provide complete objects. For example, the Page.Response property allows you to access the Response object anywhere in your page, and the Page.Server property allows you to access the Server object. These two objects are fundamentally important for two basic tasks that are explained in the following sections. First, you can use the Response and Server objects to send the user from one page to another, which is a key part of any ASP.NET application. Second, you can use the Server object to encode text that may contain special characters so it can be inserted into web page HTML.

Sending the User to a New Page

In the currency converter example, everything took place in a single page. In a more typical website, the user will need to surf from one page to another to perform different tasks or complete a single operation.

There are several ways to transfer a user from one page to another. One of the simplest is to use an ordinary <a> anchor element, which turns a portion of text into a hyperlink. In this example, the word *here* is a link to another page:

```
Click <a href="newpage.aspx">here</a> to go to newpage.aspx.
```

Another option is to send the user to a new page by using code. This approach is useful if you want to use your code to perform some other work before you redirect the user. It's also handy if you need to use code to decide where to send the user. For example, if you create a sequence of pages for placing an order, you might send existing customers straight to the checkout, while new visitors are redirected to a registration page.

To perform redirection in code, you first need a control that causes the page to be posted back. In other words, you need an event handler that reacts to the ServerClick event of a control such as HtmlInputButton or HtmlAnchor. When the page is posted back and your event handler runs, you can use the HttpResponse.Redirect() method to send the user to the new page.

Remember, you can get access to the current HttpResponse object through the Page.Response property. Here's an example that sends the user to a different page in the same website directory:

```
Response.Redirect("newpage.aspx");
```

When you use the Redirect() method, ASP.NET immediately stops processing the page and sends a redirect message back to the browser. Any code that occurs after the Redirect() call won't be executed. When the browser receives the redirect message, it sends a request for the new page.

You can use the Redirect() method to send the user to any type of page. You can even send the user to another website by using an absolute URL (a URL that starts with `http://`), as shown here:

```
Response.Redirect("http://www.prosetech.com");
```

ASP.NET gives you one other option for sending the user to a new page. You can use the HttpServerUtility.Transfer() method instead of Response.Redirect(). An HttpServerUtility object is provided through the Page.Server property, so your redirection code would look like this:

```
Server.Transfer("newpage.aspx");
```

The advantage of using the Transfer() method is that it doesn't involve the browser. Instead of sending a redirect message back to the browser, ASP.NET simply starts processing the new page as though the user had originally requested that page. This behavior saves a bit of time, but it also introduces some significant limitations. You can't use Transfer() to send the user to another website or to a non-ASP.NET page (such as an HTML page). The Transfer() method allows you to jump only from one ASP.NET page to another, in the same web application. Furthermore, when you use Transfer(), the user won't have any idea that another page has taken over, because the browser will still show the original URL. This can cause a problem if you want to support browser bookmarks. On the whole, it's much more common to use HttpResponse.Redirect() than HttpServerUtility.Transfer().

Working with HTML Encoding

As you already know, certain characters have a special meaning in HTML. For example, angle brackets (< >) are always used to create tags. This can cause problems if you want to use these characters as part of the content of your web page.

For example, imagine you want to display this text on a web page:

```
Enter a word <here>
```

If you try to write this information to a page or place it inside a control, you end up with this instead:

```
Enter a word
```

The problem is that the browser has tried to interpret the <here> as an HTML tag. A similar problem occurs if you actually use valid HTML tags. For example, consider this text:

```
To bold text use the <b> tag.
```

Not only will the text <b> not appear, but the browser will interpret it as an instruction to make the text that follows bold. To circumvent this automatic behavior, you need to convert potential problematic values to their HTML equivalents. For example, < becomes `<` in your final HTML page, which the browser displays as the <character. Table 5-10 lists some special characters that need to be encoded.

Table 5-10. *Common HTML Special Characters*

Result	Description	Encoded Entity
	Nonbreaking space	` `
<	Less-than symbol	`<`
>	Greater-than symbol	`>`
&	Ampersand	`&`
"	Quotation mark	`&quot;`

You can perform this transformation on your own, or you can circumvent the problem by using the InnerText property of an HTML server control. When you set the contents of a control by using InnerText, any illegal characters are automatically converted into their HTML equivalents. However, this won't help if you want to set a tag that contains a mix of embedded HTML tags and encoded characters. It also won't be of any use for controls that don't provide an InnerText property, such as the Label web control you'll examine in the next chapter. In these cases, you can use the HttpServerUtility.HtmlEncode() method to replace the special characters. (Remember, an HttpServerUtility object is provided through the Page.Server property.)

Here's an example:

```
// Will output as "Enter a word &lt;here&gt;" in the HTML file, but the
// browser will display it as "Enter a word<here>".
ctrl.InnerHtml=Server.HtmlEncode("Enter a word<here>");
```

Or consider this example, which mingles real HTML tags with text that needs to be encoded:

```
ctrl.InnerHtml="To<b>bold</b>text use the ";
ctrl.InnerHtml+= Server.HtmlEncode("<b>")+" tag.";
```

Figure 5-9 shows the results of successfully and incorrectly encoding special HTML characters. You can refer to the HtmlEncodeTest.aspx page included with the examples for this chapter.

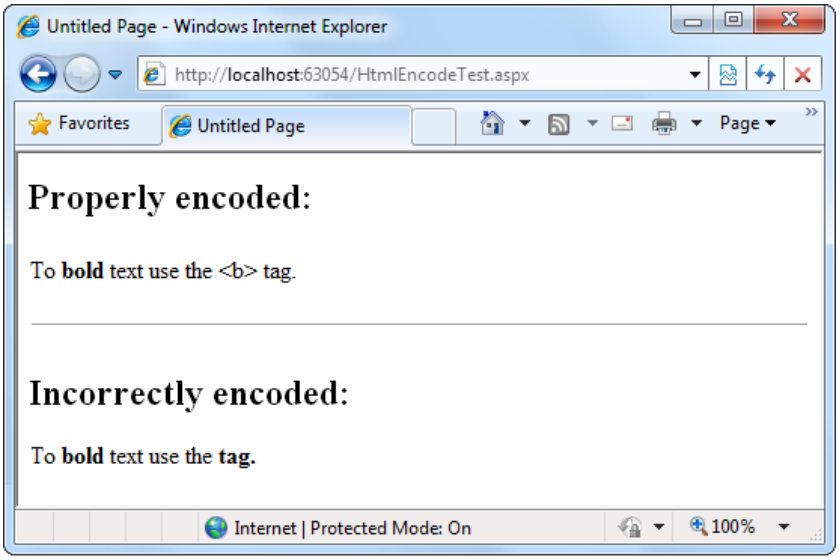

Figure 5-9. *Encoding special HTML characters*

The HtmlEncode() method is particularly useful if you're retrieving values from a database and you aren't sure whether the text is valid HTML. You can use the HtmlDecode() method to revert the text to its normal form if you need to perform additional operations or comparisons with it in your code.

Along with the HtmlEncode() and HtmlDecode() methods, the HttpServerUtility class also includes UrlEncode() and UrlDecode() methods. Much as HtmlEncode() allows you to convert text to valid HTML with no special characters, UrlEncode() allows you to convert text into a form that can be used in a URL. This technique is particularly useful if you want to pass information from one page to another by tacking it onto the end of the URL. You'll see this technique demonstrated in Chapter 8.

Using Application Events

In this chapter, you've seen how ASP.NET controls fire events that you can handle in your code. Although server controls are the most common source of events, there's another type of event that you'll occasionally encounter: *application events*. Application events aren't nearly as important in an ASP.NET application as the events fired by server controls, but you might use them to perform additional processing tasks. For example, by using application events, you can write logging code that runs every time a request is received, no matter what page is being requested. Basic ASP.NET features such as session state and authentication use application events to plug into the ASP.NET processing pipeline.

You can't handle application events in the code-behind for a web form. Instead, you need the help of another ingredient: the global.asax file.

The global.asax File

The global.asax file allows you to write code that responds to global application events. These events fire at various points during the lifetime of a web application, including when the application domain is first created (when the first request is received for a page in your website folder).

To add a global.asax file to an application in Visual Studio, choose Website ä Add New Item, and select the Global Application Class file type. Then click OK. (If you already have a global.asax file in your project, you won't see the Global Application Class file type, because Visual Studio is smart enough to realize that a single website can't have two.)

The global.asax file looks similar to a normal .aspx file, except that it can't contain any HTML or ASP.NET tags. Instead, it contains event handlers that respond to application events. When you add the global.asax file, Visual Studio inserts several ready-made application event handlers. You simply need to fill in some code.

For example, the following global.asax file reacts to the EndRequest event, which happens just before the page is sent to the user:

```
<%@ Application Language="C#" %>

<script language="c#" runat="server">
    protected void Application_EndRequest(object sender, EventArgs e)
    {
          Response.Write("<hr />This page was served at " +
            DateTime.Now.ToString());
    }
</script>
```

This event handler uses the Write() method of the built-in Response object to write a footer at the bottom of the page with the date and time that the page was created (see Figure 5-10).

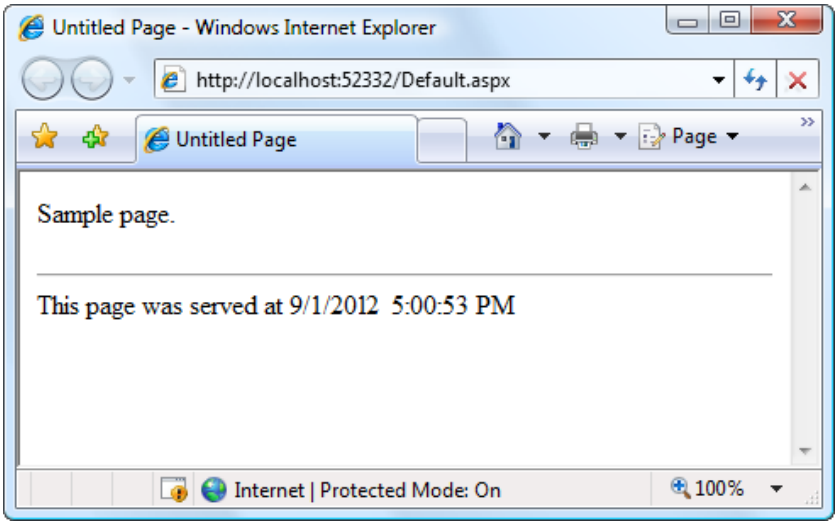

Figure 5-10. *A sample page with an automatic footer*

Each ASP.NET application can have one global.asax file. After you place it in the appropriate website directory, ASP.NET recognizes it and uses it automatically. For example, if you add the global.asax file shown previously to a web application, every web page in that application will include a footer.

■ **Note** This technique—responding to application events and using the Response.Write() method—isn't the best way to add a footer to the pages in your website. A better approach is to add a user control that creates the footer (Chapter 11) or define a master page template that includes a footer (Chapter 12).

Additional Application Events

Application.EndRequest is only one of more than a dozen events you can respond to in your code. To create a different event handler, you simply need to create a subroutine with the defined name. Table 5-11 lists some of the most common application events that you'll use.

Table 5-11. *Basic Application Events*

Event-Handling Method	Description
Application_Start()	Occurs when the application starts, which is the first time it receives a request from any user. It doesn't occur on subsequent requests. This event is commonly used to create or cache some initial information that will be reused later.
Application_End()	Occurs when the application is shutting down, generally because the web server is being restarted. You can insert cleanup code here.
Application_BeginRequest()	Occurs with each request the application receives, just before the page code is executed.
Application_EndRequest()	Occurs with each request the application receives, just after the page code is executed.
Session_Start()	Occurs whenever a new user request is received and a session is started. Sessions are discussed in detail in Chapter 8.
Session_End()	Occurs when a session times out or is programmatically ended. This event is raised only if you are using in-process session-state storage (the InProc mode, not the StateServer or SQLServer modes).
Application_Error()	Occurs in response to an unhandled error. You can find more information about error handling in Chapter 7.

Configuring an ASP.NET Application

The last topic you'll consider in this chapter is the ASP.NET configuration file system.

Every web application includes a web.config file that configures fundamental settings—everything from the way error messages are shown to the security settings that lock out unwanted visitors. You'll consider the settings in the web.config file throughout this book. (And there are many more settings that you *won't* consider in this book, because they're used much more rarely.)

The ASP.NET configuration files have several key advantages:

> *They are never locked*: You can update web.config settings at any point, even while your application is running. If there are any requests currently under way, they'll continue to use the old settings, while new requests will get the changed settings right away.

They are easily accessed and replicated: Provided you have the appropriate network rights, you can change a web.config file from a remote computer. You can also copy the web.config file and use it to apply identical settings to another application or another web server that runs the same application in a web farm scenario.

The settings are easy to edit and understand: The settings in the web.config file are human-readable, which means they can be edited and understood without needing a special configuration tool.

In the following sections, you'll get a high-level overview of the web.config file and learn how ASP.NET's configuration system works.

Working with the web.config File

The web.config file uses a predefined XML format. The entire content of the file is nested in a root <configuration> element. Inside this element are several more subsections, some of which you'll never change, and others which are more important.

Here's the basic skeletal structure of the web.config file:

```
<?xml version="1.0" ?>
<configuration>
    <appSettings>...</appSettings>
    <connectionStrings>...</connectionStrings>
    <system.web>...</system.web>
</configuration>
```

Note that the web.config file is case-sensitive, like all XML documents, and starts every setting with a lowercase letter. This means you cannot write <AppSettings> instead of <appSettings>.

■ **Tip** To learn more about XML, the format used for the web.config file, you can refer to Chapter 18.

As a web developer, there are three sections in the web.config file that you'll work with. The <appSettings> section allows you to add your own miscellaneous pieces of information. You'll learn how to use it in the next section. The <connectionStrings> section allows you to define the connection information for accessing a database. You'll learn about this section in Chapter 14. Finally, the <system.web> section holds every ASP.NET setting you'll need to configure.

Inside the <system.web> element are separate elements for each aspect of website configuration. You can include as few or as many of these as you want. For example, if you want to control how ASP.NET's security works, you'd add the <authentication> and <authorization> sections.

When you create a new website, it starts with no web.config file. But when you run that website in Visual Studio and choose to enable debugging, Visual Studio creates a basic web.config file with a small number of settings. It looks like this:

```
<?xml version="1.0"?>
<configuration>
  <appSettings>
    <add key="aspnet:UseTaskFriendlySynchronizationContext" value="true"/>
    <add key="ValidationSettings:UnobtrusiveValidationMode" value="WebForms"/>
  </appSettings>
  <system.web>
    <compilation debug="true" targetFramework="4.5"/>
    <httpRuntime requestValidationMode="4.5" targetFramework="4.5"
     encoderType="…"/>
```

```
    <pages controlRenderingCompatibilityVersion="4.5"/>
    <machineKey compatibilityMode="Framework45"/>
  </system.web>
</configuration>
```

This configuration file includes several details. First, there are two hard-coded values (in the <appSettings> section) that your code, or other parts of ASP.NET, can pay attention to. You'll learn how that works in a moment.

Most important is the <compilation> element, which uses two key attributes to specify two key details:

debug: This attribute tells ASP.NET whether to compile the application in debug mode so you can use Visual Studio's debugging tools. As you saw in Chapter 4, Visual Studio asks whether you want to switch on debugging mode the first time you run a new application (and you should always choose yes).

■ **Note** The disadvantage of debugging support is that it slows down your application ever so slightly, and small slowdowns can add up to big bottlenecks in a heavily trafficked web application in the real world. For that reason, you should remove the `debug="true"` attribute when you're ready to deploy your application to a live web server.

---

targetFramework: This attribute tells Visual Studio what version of .NET you're planning to use on your web server (which determines the features you'll have access to in your web application). As you learned in Chapter 4, you set the target framework when you create your website, and you can change it at any time after.The rest of the configuration details—the <httpRuntime>, <pages,> and <machineKey> elements—are simply there for backward compatibility. In the autogenerated web.config file, shown earlier in this section, all these details simply tell ASP.NET to use the latest behavior introduced with .NET 4.5. (Although the changes from .NET 4 to .NET 4.5 are small, minor differences could trip up certain applications. These configuration settings are a sort of safety valve that allows developers to get back to the way things were if a new tweak or fix changes the behavior of the application when they move it to ASP.NET 4.5.)

The <system.web> section is also home for many more important settings that aren't used in the autogenerated web.config file. You'll consider the different elements that you can add to the <system.web> section throughout this book.

Understanding Nested Configuration

ASP.NET uses a multilayered configuration system that allows you to set settings at different levels.

Every web server starts with some basic settings that are defined in a directory such as c:\Windows\Microsoft.NET\Framework\[Version]\Config. The [Version] part of the directory name depends on the version of .NET that you have installed. At the time of this writing, the latest build was v4.0.30319, which makes the full folder name c:\Windows\Microsoft.NET\Framework64**v4.0.30319**\Config.

■ **Note** Don't be confused by the fact that the version number in the folder seems to indicate .NET 4 rather than .NET 4.5. This harmless quirk is a side-effect of the way .NET 4.5 was released. Technically, .NET 4.5 is an *in-place update* that extends .NET 4 without replacing it. Thus, .NET 4.5 keeps using the same configuration folder as .NET 4.

The Config folder contains two files, named machine.config and web.config. Generally, you won't edit either of these files by hand, because they affect the entire computer. Instead, you'll configure the web.config file in your

web application folder. Using that file, you can set additional settings or override the defaults that are configured in the two system files.

More interestingly, you can use different settings for different parts of your application. To use this technique, you need to create additional subdirectories inside your virtual directory. These subdirectories can contain their own web.config files with additional settings.

Subdirectories inherit web.config settings from the parent directory. For example, imagine you create a website in the directory c:\ASP.NET\TestWeb. Inside this directory, you create a folder named Secure. Pages in the c:\ASP.NET\TestWeb\Secure directory can acquire settings from three files, as shown in Figure 5-11.

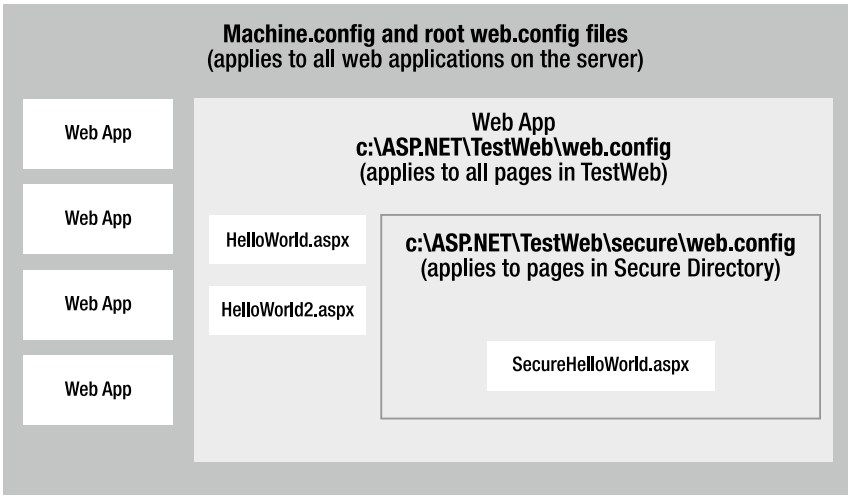

Figure 5-11. *Configuration inheritance*

Any machine.config or web.config settings that aren't explicitly overridden in the c:\ASP.NET\TestWeb\Secure\web.config file will still apply to the SecureHelloWorld.aspx page. In this way, subdirectories can specify just a small set of settings that differ from the rest of the web application. One reason you might want to use multiple directories in an application is to apply different security settings. Files that need to be secured would then be placed in a dedicated directory with a web.config file that defines more-stringent security settings.

Storing Custom Settings in the web.config File

ASP.NET also allows you to store your own settings in the web.config file, in an element called <appSettings>. Note that the <appSettings> element is nested in the root <configuration> element. Here's the basic structure:

```
<?xml version="1.0" ?>
<configuration>
  <appSettings>
    <!-- Custom application settings go here. -->
  </appSettings>
    <system.web>
         <!-- ASP.NET Configuration sections go here. -->
    </system.web>
</configuration>
```

■ **Note** This example adds a comment in the place where you'd normally find additional settings. XML comments are bracketed with the <!-- and --> character sequences. You can also use XML comments to temporarily disable a setting in a configuration file.

The custom settings that you add are written as simple string variables. You might want to use a special web.config setting for several reasons:

To centralize an important setting that needs to be used in many different pages: For example, you could create a variable that stores a database query. Any page that needs to use this query can then retrieve this value and use it.

To make it easy to quickly switch between different modes of operation: For example, you might create a special debugging variable. Your web pages could check for this variable and, if it's set to a specified value, output additional information to help you test the application.

To set some initial values: Depending on the operation, the user might be able to modify these values, but the web.config file could supply the defaults.

You can enter custom settings by using an <add> element that identifies a unique variable name (key) and the variable contents (value). The following example adds a variable that defines a file path where important information is stored:

```
<appSettings>
  <add key="DataFilePath"
   value="e:\NetworkShare\Documents\WebApp\Shared" />
</appSettings>
```

You can add as many application settings as you want, although this example defines just one.

You can create a simple test page to query this information and display the results, as shown in the following example (which is provided with the sample code as ShowSettings.aspx and ShowSettings.aspx.cs). You retrieve custom application settings from web.config by key name, using the WebConfigurationManager class, which is found in the System.Web.Configuration namespace. This class provides a static property called AppSettings with a collection of application settings.

```
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.Configuration;
public partial class ShowSettings : System.Web.UI.Page
{
     protected void Page_Load()
     {
          Results.InnerHtml="This app will look for data in the directory:<br />";
          Results.InnerHtml+= "<b>"
          Results.InnerHtml+= WebConfigurationManager.AppSettings["DataFilePath"];
          Results.InnerHtml+= "</b>";
     }
}
```

■ **Tip** Notice that this code formats the text by inserting HTML tags into the label alongside the text content, including bold tags (<b>) to emphasize certain words, and a line break (<br />) to split the output over multiple lines. This is a common technique.

In Chapter 17, you'll learn how to get file and directory information and read and write files. For now, the simple application just displays the custom web.config setting, as shown in Figure 5-12.

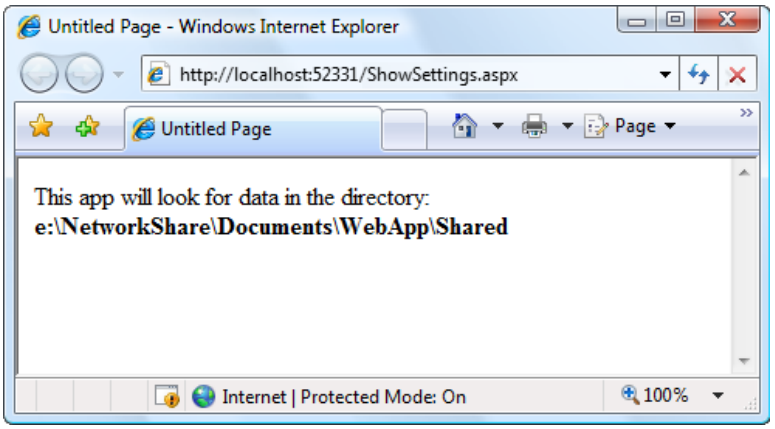

Figure 5-12. *Displaying custom application settings*

ASP.NET web servers are configured, by default, to deny any requests for .config files. This means remote users will not be able to access the file. Instead, they'll receive the error message shown in Figure 5-13.

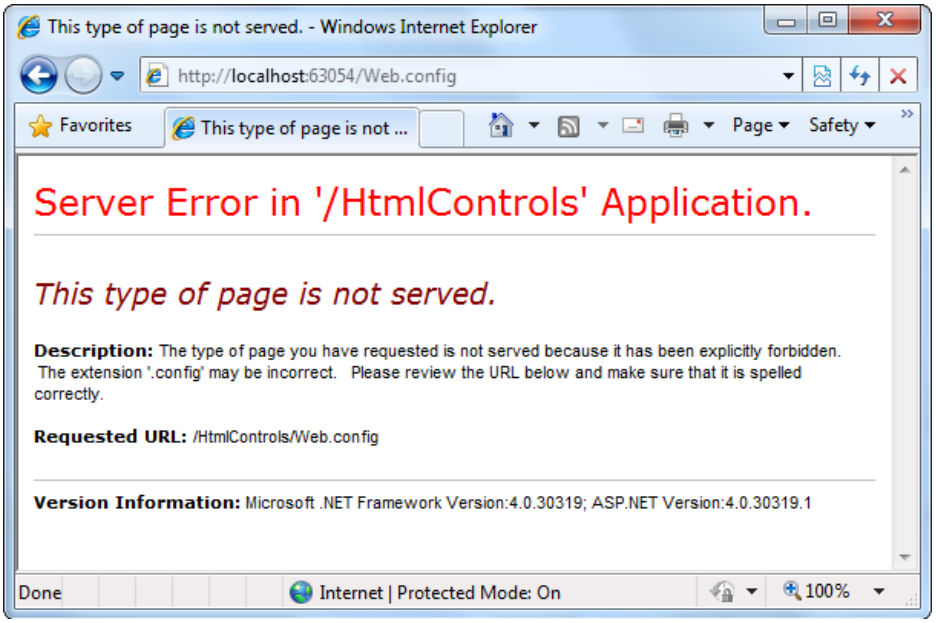

Figure 5-13. *Requests for web.config are denied.*

Using the Website Administration Tool

Editing the web.config file by hand is refreshingly straightforward, but it can be a bit tedious. To help alleviate the drudgery, ASP.NET includes a graphical configuration tool called the *Website Administration Tool (WAT)*, which lets you configure various parts of the web.config file by using a web page interface. To run the WAT to configure the current web project in Visual Studio, select Website ä ASP.NET Configuration (or click the ASP.NET Configuration icon that's at the top of the Solution Explorer). A web browser window will appear (see Figure 5-14). Internet Explorer will automatically log you on under the current Windows user account, allowing you to make changes.

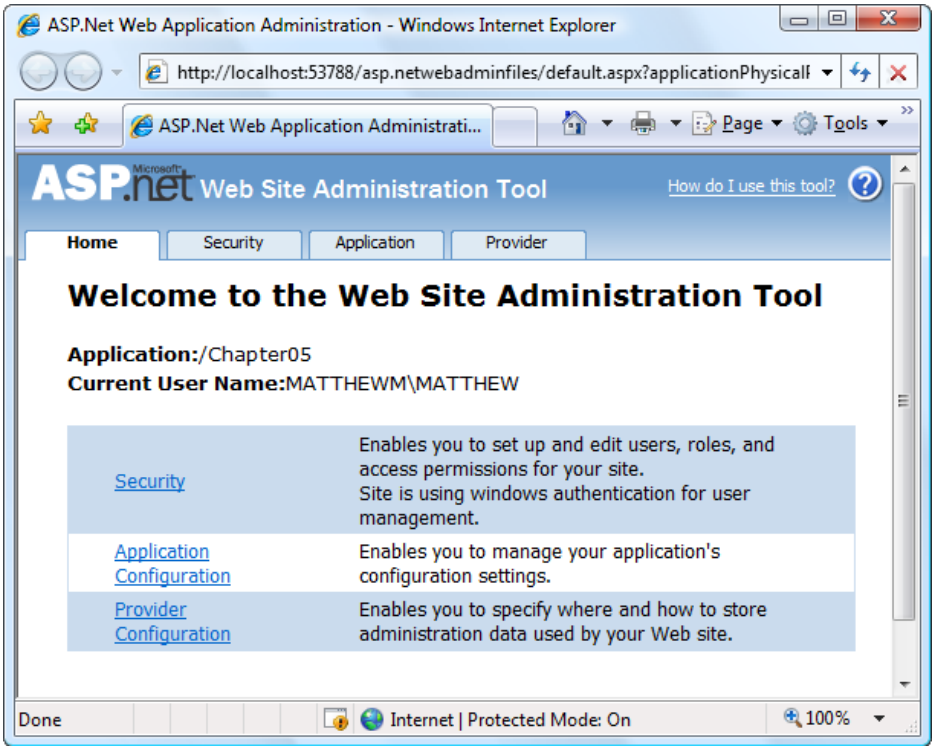

Figure 5-14. *Running the WAT*

You can use the WAT to automate the web.config changes you made in the previous example. To try this, click the Application tab. Using this tab, you can create a new setting (click the Create Application Settings link). If you click Manage Application Settings, you'll see a list with all the settings that are defined in your application (Figure 5-15). You can then choose to remove or edit any one of them.

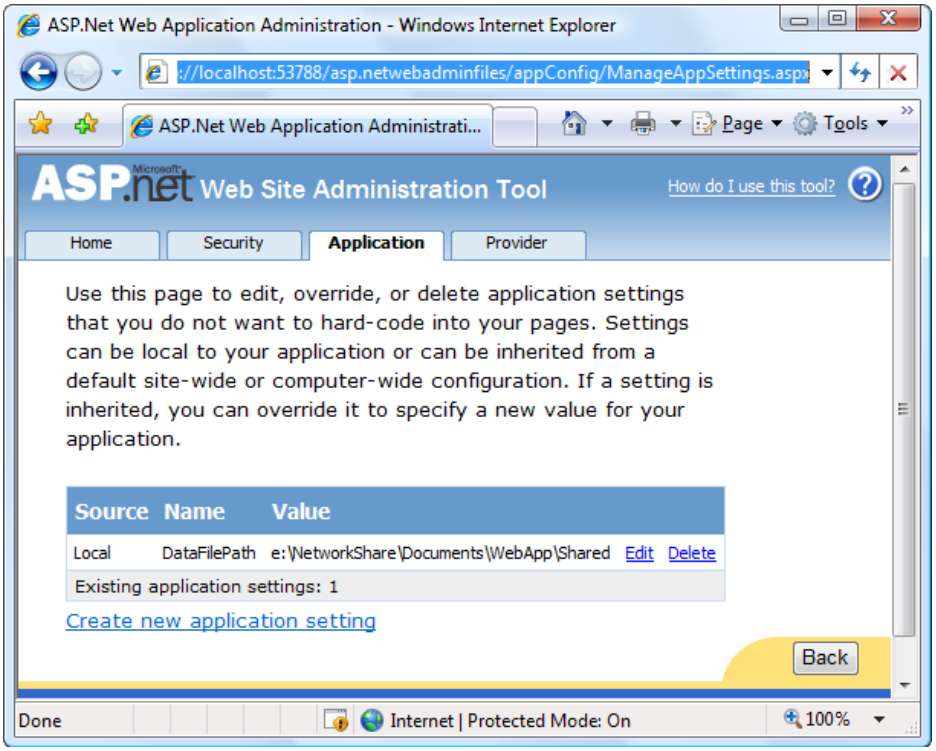

Figure 5-15. *Editing an application setting with the WAT*

This is the essential idea behind the WAT. You make your changes using a graphical interface (a web page), and the WAT generates the settings you need and adds them to the web.config file for your application behind the scenes. Of course, the WAT has a number of settings for configuring more-complex ASP.NET settings, and you'll use it throughout this book.

■ **Note** The WAT works only while you're developing a web application. In other words, you can't deploy a website to a live web server and then attempt to use the WAT. However, if you need to reconfigure an already-deployed application, you can use the graphical IIS Manager tool, which provides some of the same features as the WAT (and many additional ones). You'll learn more about IIS configuration in Chapter 26.

The Last Word

This chapter presented you with your first look at three core ASP.NET topics: web applications, server controls, and configuration. You should now understand how to create a very simple ASP.NET page, react to its events, and manipulate its content in code.

HTML controls are a compromise between ASP.NET web controls and traditional HTML. They use the familiar HTML elements but provide a limited object-oriented interface. Essentially, HTML controls are designed to be straightforward, predictable, and automatically compatible with existing programs. With HTML controls, the final HTML page that is sent to the client closely resembles the markup in the original .aspx page.

In the next chapter, you'll learn about web controls, which provide a more sophisticated object interface over the top of the underlying HTML. If you're starting a new project or need to add some of ASP.NET's most powerful controls, web controls are the best option.

CHAPTER 6

Web Controls

The previous chapter introduced the event-driven and control-based programming model of ASP.NET. This model allows you to create programs for the Web by using the same object-oriented techniques you would use to write an ordinary desktop application.

However, HTML server controls really show only a glimpse of what is possible with ASP.NET's server control model. To see some of the real advantages, you need to use the richer and more extensible *web controls*. In this chapter, you'll explore the basic web controls and their class hierarchy. You'll also delve deeper into ASP.NET's event handling and learn the details of the web page life cycle. Finally, you'll put your knowledge to work by creating a web page that lets the user design a greeting card.

Stepping Up to Web Controls

Now that you've seen the new model of server controls, you might wonder why you need additional web controls. But in fact, HTML controls are much more limited than server controls need to be. For example, every HTML control corresponds directly to a single HTML element. Web controls, on the other hand, have no such restriction—they can switch from one element to another depending on how you're using them, or they can render themselves by using a complex combination of multiple elements.

These are some of the reasons you should switch to web controls:

They provide a rich user interface: A web control is programmed as an object but doesn't necessarily correspond to a single element in the final HTML page. For example, you might create a single Calendar or GridView control, which will be rendered as dozens of HTML elements in the final page. When using ASP.NET programs, you don't need to know the lower-level HTML details. The control creates the required HTML tags for you.

They provide a consistent object model: HTML is full of quirks and idiosyncrasies. For example, a simple text box can appear as one of three elements, including <textarea>, <input type="text">, and <input type="password">. With web controls, these three elements are consolidated as a single TextBox control. Depending on the properties you set, the underlying HTML element that ASP.NET renders may differ. Similarly, the names of properties don't follow the HTML attribute names. For example, controls that display text, whether it's a caption or a text box that can be edited by the user, expose a Text property.

They tailor their output automatically: ASP.NET server controls can detect the type of browser and automatically adjust the HTML code they write to take advantage of features such as support for JavaScript. You don't need to know about the client because ASP.NET handles that layer and automatically uses the best possible set of features. This feature is known as *adaptive rendering*.

They provide high-level features: You'll see that web controls allow you to access additional events, properties, and methods that don't correspond directly to typical HTML controls. ASP.NET implements these features by using a combination of tricks.

Throughout this book, you'll see examples that use the full set of web controls. To master ASP.NET development, you need to become comfortable with these user-interface ingredients and understand their abilities. HTML server controls, on the other hand, are less important for web forms development. You'll use them only if you're migrating an existing HTML page to the ASP.NET world, or if you need to have fine-grained control over the HTML code that will be generated and sent to the client.

Basic Web Control Classes

If you've ever created a Windows application before, you're probably familiar with the basic set of standard controls, including labels, buttons, and text boxes. ASP.NET provides web controls for all these standbys. (And if you've created .NET Windows applications, you'll notice that the class names and properties have many striking similarities, which are designed to make it easy to transfer the experience you acquire in one type of application to another.)

Table 6-1 lists the basic control classes and the HTML elements they generate. Some controls (such as Button and TextBox) can be rendered as different HTML elements. In this case, ASP.NET uses the element that matches the properties you've set. Also, some controls have no real HTML equivalent. For example, when the CheckBoxList and RadioButtonList controls render themselves, they may output a <table> that contains multiple HTML check boxes or radio buttons. ASP.NET exposes them as a single object on the server side for convenient programming, thus illustrating one of the primary strengths of web controls.

Table 6-1. *Basic Web Controls*

Control Class	Underlying HTML Element
Label	
Button	<input type="submit"> or <input type="button">
TextBox	<input type="text">, <input type="password">, or <textarea>
CheckBox	<input type="checkbox">
RadioButton	<input type="radio">
Hyperlink	<a>
LinkButton	<a> with a contained <img> tag
ImageButton	<input type="image">
Image	<img>
ListBox	<select size="X"> where X is the number of rows that are visible at once
DropDownList	<select>
CheckBoxList	A list or <table> with multiple <input type="checkbox"> tags
RadioButtonList	A list or <table> with multiple <input type="radio"> tags
BulletedList	An ordered list (numbered) or unordered list (bulleted)
Panel	
Table, TableRow, and TableCell	<table>, <tr>, and <td> or <th>

This table omits some of the more specialized controls used for data, navigation, security, and web portals. You'll see these controls as you learn about their features throughout this book.

The Web Control Tags

ASP.NET tags have a special format. They always begin with the prefix `asp:` followed by the class name. If there is no closing tag, the tag must end with `/>`. (This syntax convention is borrowed from XML, which you'll learn about in much more detail in Chapter 18.) Each attribute in the tag corresponds to a control property, except for the `runat="server"` attribute, which signals that the control should be processed on the server.

The following, for example, is an ASP.NET TextBox:

```
<asp:TextBox ID="txt" runat="server" />
```

When a client requests this .aspx page, the following HTML is returned. The name is a special attribute that ASP.NET uses to track the control.

```
<input type="text" ID="txt" name="txt" />
```

Alternatively, you could place some text in the TextBox, set its size, make it read-only, and change the background color. All these actions have defined properties. For example, the TextBox.TextMode property allows you to specify SingleLine (the default), MultiLine (for a <textarea> type of control), or Password (for an input control that displays bullets to hide the true value). You can adjust the color by using the BackColor and ForeColor properties. And you can tweak the size of the TextBox in one of two ways—either using the Rows and Columns properties (for the pure HTML approach) or using the Height and Width properties (for the style-based approach). Both have the same result.

Here's one example of a customized TextBox:

```
<asp:TextBox ID="txt" BackColor="Yellow" Text="Hello World"
 ReadOnly="True" TextMode="MultiLine" Rows="5" runat="server" />
```

The resulting HTML uses the <textarea> element and sets all the required attributes (such as rows and readonly) and the style attribute (with the background color). It also sets the cols attribute with the default width of 20 columns, even though you didn't explicitly set the TextBox.Columns property:

```
<textarea name="txt" rows="5" cols="20" readonly="readonly" ID="txt"
 style="background-color:Yellow;">Hello World</textarea>
```

Figure 6-1 shows the <textarea> element in the browser.

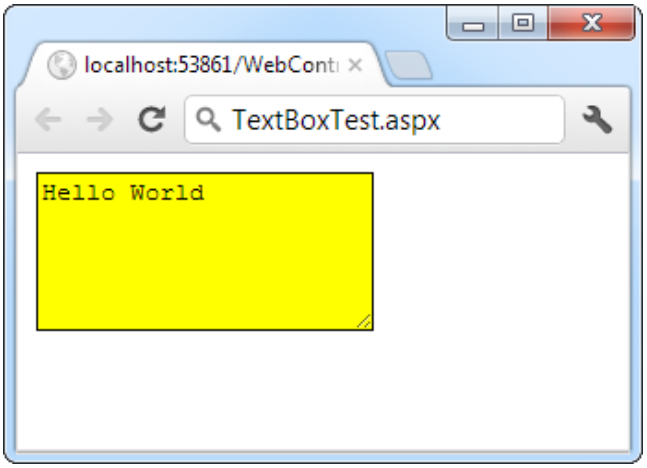

Figure 6-1. *A customized text box*

Clearly, it's easy to create a web control tag. It doesn't require any understanding of HTML. However, you *will* need to understand the control class and the properties that are available to you.

CASE-SENSITIVITY IN ASP.NET FORMS

The .aspx layout portion of a web page tolerates different capitalization for tag names, property names, and enumeration values. For example, the following two tags are equivalent, and both will be interpreted correctly by the ASP.NET engine, even though their case differs:

```
<asp:Button ID="Button1" runat="server"
  Enabled="False" Text="Button" Font-Size="XX-Small" />
<asp:button ID="Button2" runat="server"
  Enabled="false" tExT="Button" Font-Size="xx-small" />
```

This design was adopted to make .aspx pages behave more like ordinary HTML web pages, which ignore case completely. However, you can't use the same looseness in the tags that apply settings in the web.config file or the machine.config file. Here, case must match *exactly*.

Web Control Classes

Web control classes are defined in the System.Web.UI.WebControls namespace. They follow a slightly more tangled object hierarchy than HTML server controls. Figure 6-2 shows most, but not all, of the web controls that ASP.NET provides.

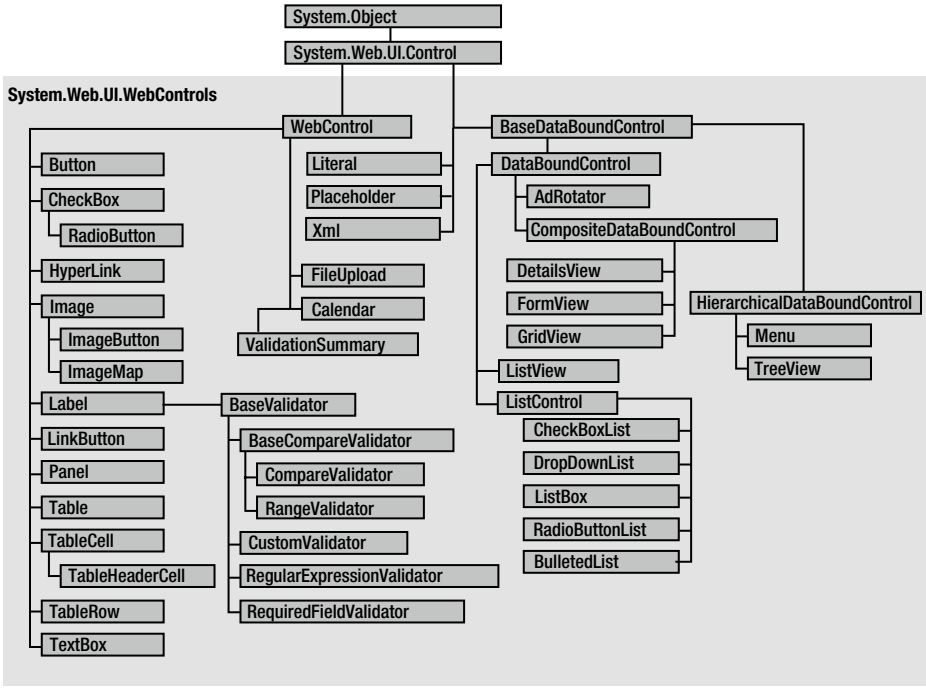

Figure 6-2. *The web control hierarchy*

This inheritance diagram includes some controls that you won't study in this chapter, including the data controls (such as the GridView, DetailsView, and FormView) and the validation controls. You'll explore these controls in later chapters.

The WebControl Base Class

Most web controls begin by inheriting from the WebControl base class. This class defines the essential functionality for tasks such as data binding and includes some basic properties that you can use with almost any web control, as described in Table 6-2.

Table 6-2. *WebControl Properties*

Property	Description
AccessKey	Specifies the keyboard shortcut as one letter. For example, if you set this to Y, the Alt+Y keyboard combination will automatically change focus to this web control (assuming the browser supports this feature).
BackColor, ForeColor, and BorderColor	Sets the colors used for the background, foreground, and border of the control. In most controls, the foreground color sets the text color.
BorderWidth	Specifies the size of the control border.
BorderStyle	One of the values from the BorderStyle enumeration, including Dashed, Dotted, Double, Groove, Ridge, Inset, Outset, Solid, and None.
Controls	Provides a collection of all the controls contained inside the current control. Each object is provided as a generic System.Web.UI.Control object, so you will need to cast the reference to access control-specific properties.
Enabled	When set to false, the control will be visible, but it will not be able to receive user input or focus.
EnableViewState	Set this to false to disable the automatic state management for this control. In this case, the control will be reset to the properties and formatting specified in the control tag (in the .aspx page) every time the page is posted back. If this is set to true (the default), the control uses the hidden input field to store information about its properties, ensuring that any changes you make in code are remembered.
Font	Specifies the font used to render any text in the control as a System.Web.UI.WebControls.FontInfo object.
Height and Width	Specifies the width and height of the control. For some controls, these properties will be ignored when used with older browsers.
ID	Specifies the name that you use to interact with the control in your code (and also serves as the basis for the ID that's used to name the top-level element in the rendered HTML).
Page	Provides a reference to the web page that contains this control as a System.Web.UI.Page object.
Parent	Provides a reference to the control that contains this control. If the control is placed directly on the page (rather than inside another control), it will return a reference to the page object.
TabIndex	A number that allows you to control the tab order. The control with a TabIndex of 0 has the focus when the page first loads. Pressing Tab moves the user to the control with the next lowest TabIndex, provided it is enabled. This property is supported only in Internet Explorer.
ToolTip	Displays a text message when the user hovers the mouse above the control. Many older browsers don't support this property.
Visible	When set to false, the control will be hidden and will not be rendered to the final HTML page that is sent to the client.

The next few sections describe some of the common concepts you'll use with almost any web control, including how to set properties that use units and enumerations and how to use colors and fonts.

Units

All the properties that use measurements, including BorderWidth, Height, and Width, require the Unit structure, which combines a numeric value with a type of measurement (pixels, percentage, and so on). This means when you set these properties in a control tag, you must make sure to append px (pixel) or % (for percentage) to the number to indicate the type of unit.

Here's an example with a Panel control that is 300 pixels high and has a width equal to 50 percent of the current browser window:

```
<asp:Panel Height="300px" Width="50%" ID="pnl" runat="server" />
```

If you're assigning a unit-based property through code, you need to use one of the static methods of the Unit type. Use Pixel() to supply a value in pixels, and use Percentage() to supply a percentage value:

```
// Convert the number 300 to a Unit object
// representing pixels, and assign it.
pnl.Height = Unit.Pixel(300);

// Convert the number 50 to a Unit object
// representing percent, and assign it.
pnl.Width = Unit.Percentage(50);
```

You could also manually create a Unit object and initialize it by using one of the supplied constructors and the UnitType enumeration. This requires a few more steps but allows you to easily assign the same unit to several controls:

```
// Create a Unit object.
Unit myUnit = new Unit(300, UnitType.Pixel);

// Assign the Unit object to several controls or properties.
pnl.Height = myUnit;
pnl.Width = myUnit;
```

Enumerations

Enumerations are used heavily in the .NET class library to group a set of related constants. For example, when you set a control's BorderStyle property, you can choose one of several predefined values from the BorderStyle enumeration. In code, you set an enumeration by using the dot syntax:

```
ctrl.BorderStyle = BorderStyle.Dashed;
```

In the .aspx file, you set an enumeration by specifying one of the allowed values as a string. You don't include the name of the enumeration type, which is assumed automatically.

```
<asp:Label BorderStyle="Dashed" Text="Border Test" ID="lbl"
 runat="server" />
```

Figure 6-3 shows the label with the altered border.

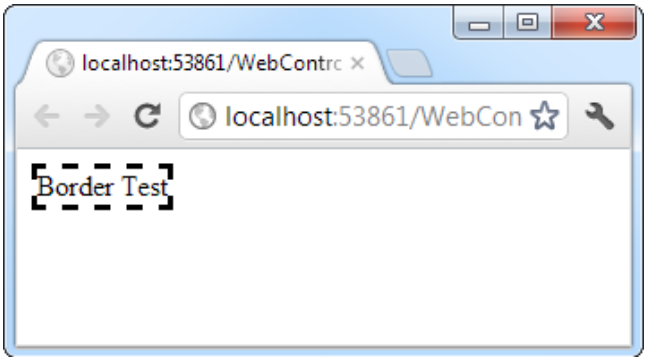

Figure 6-3. *Modifying the border style*

Colors

The Color property refers to a Color object from the System.Drawing namespace. You can create color objects in several ways:

Using an ARGB (alpha, red, green, blue) color value: You specify each value as an integer from 0 to 255. The alpha component represents the transparency of a color, and usually you'll use 255 to make the color completely opaque.

Using a predefined .NET color name: You choose the correspondingly named read-only property from the Color structure. These properties include the 140 HTML color names.

Using an HTML color name: You specify this value as a string by using the ColorTranslator class.

To use any of these techniques, you'll probably want to start by importing the System.Drawing namespace, as follows:

```
using System.Drawing;
```

The following code shows several ways to specify a color in code:

```
// Create a color from an ARGB value
int alpha = 255, red = 0, green = 255, blue = 0;
ctrl.ForeColor = Color.FromArgb(alpha, red, green, blue);

// Create a color using a .NET name
ctrl.ForeColor = Color.Crimson;

// Create a color from an HTML code
ctrl.ForeColor = ColorTranslator.FromHtml("Blue");
```

When defining a color in the .aspx file, you can use any one of the known color names:

```
<asp:TextBox ForeColor="Red" Text="Test" ID="txt" runat="server" />
```

The HTML color names that you can use are listed in the MSDN reference website (see http://msdn.microsoft.com/library/system.drawing.color.aspx). Alternatively, you can use a hexadecimal color number (in the format # <red> <green> <blue>), as shown here:

```
<asp:TextBox ForeColor="#ff50ff" Text="Test"
     ID="txt" runat="server" />
```

Fonts

The Font property actually references a full FontInfo object, which is defined in the System.Web.UI.WebControls namespace. Every FontInfo object has several properties that define its name, size, and style (see Table 6-3).

Table 6-3. *FontInfo Properties*

Property	Description
Name	A string indicating the font name (such as Verdana).
Names	An array of strings with font names, in the order of preference. The browser will use the first matching font that's installed on the user's computer.
Size	The size of the font as a FontUnit object. This can represent an absolute or relative size.
Bold, Italic, Strikeout, Underline, and Overline	Boolean properties that apply the given style attribute.

In code, you can assign a font by using the familiar dot syntax to set the various font properties:

```
ctrl.Font.Name = "Verdana";
ctrl.Font.Bold = true;
```

You can also set the size by using the FontUnit type:

```
// Specifies a relative size.
ctrl.Font.Size = FontUnit.Small;

// Specifies an absolute size of 14 pixels.
ctrl.Font.Size = FontUnit.Point(14);
```

In the .aspx file, you need to use a special "object walker" syntax to specify object properties such as Font. The object walker syntax uses a hyphen (-) to separate properties. For example, you could set a control with a specific font (Tahoma) and font size (40 point) like this:

```
<asp:TextBox Font-Name="Tahoma" Font-Size="40" Text="Size Test" ID="txt"
 runat="server" />
```

Or you could set a relative size like this:

```
<asp:TextBox Font-Name="Tahoma" Font-Size="Large" Text="Size Test"
 ID="txt" runat="server" />
```

Figure 6-4 shows the altered TextBox in this example.

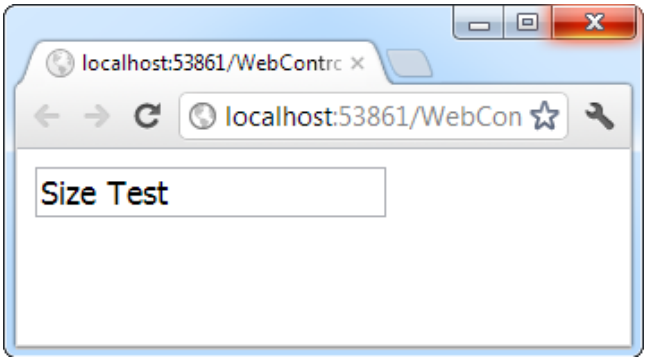

Figure 6-4. *Modifying a control's font*

A font setting is really just a recommendation. If the client computer doesn't have the font you request, it reverts to a standard font. To deal with this problem, it's common to specify a list of fonts, in order of preference. To do so, you use the Font.Names property instead of Font.Name, as shown here:

```
<asp:TextBox Font-Names="Verdana,Tahoma,Arial"
 Text="Size Test" ID="txt" runat="server" />
```

Here, the browser will use the Verdana font (if it has it). If not, it will fall back on Tahoma or, if that isn't present, Arial.

When specifying fonts, it's a good idea to end with one of the following fonts, which are supported on all browsers:

- Times
- Arial and Helvetica
- Courier

The following fonts are found on almost all Windows and Mac computers, but not necessarily on other operating systems:

- Verdana
- Georgia
- Tahoma
- Comic Sans
- Arial Black
- Impact

EMBEDDED FONTS

Recently, browser support has been growing for a feature called *embedded fonts* or *web fonts*. The basic idea is to allow you to format text in your web page with a fancy, nonstandard web font. When someone views the page, the browser downloads the corresponding font file automatically, and uses it (temporarily) to display the text in the page. The embedded font feature allows your page to use a virtually unlimited range of typefaces, without worrying about what clients have installed on their computers.

Because embedded fonts aren't universally supported (particularly in older browsers), they're generally used as an optional way to *enhance* the appearance of headings. That way, the web page can always fall back on one of the standard fonts described in the previous section if the web font doesn't work or it can't be downloaded. Usually, web designers don't use embedded fonts to format entire paragraphs of text, because of the risk that the text won't display properly on some browsers.

Technically, embedded fonts are a feature of CSS3. There is no ASP.NET-specific way to use embedded fonts. If you decide to use embedded fonts in an ASP.NET page, you will probably use them to format an ordinary HTML element, like a <p> paragraph, a , or a numbered heading (<h1>, <h2>, <h3>, and so on). You won't typically use them to format a web control (although you could, by setting the control's CssClass property to the CSS style that applies the embedded font). But in any case, it's up to you to add the style settings to your style sheet.

Using embedded fonts is a somewhat awkward procedure with several steps. First, you need the right font files, which you must upload to your web server. The challenge is that different browsers support different web font formats, with no single universal standard, so you'll need to include multiple copies of the same font. Furthermore, before you begin converting and uploading a font file you own, you need to make sure it's licensed for web use (most aren't). You can get a quick summary of the process at www.html5rocks.com/en/tutorials/webfonts/quick. But by far, the easiest strategy is to simply use a web font service such as Google Web Fonts. There you can pick from a huge gallery of slick web fonts. When you choose a font, Google will generate the corresponding CSS style—all you need to do is just paste it into the appropriate part of your style sheet. Google also makes all its web fonts available through its own web servers, so there are no extra files to upload to your website. To learn more, visit www.google.com/webfonts.

Focus

Unlike HTML server controls, every web control provides a Focus() method. The Focus() method affects only input controls (controls that can accept keystrokes from the user). When the page is rendered in the client browser, the user starts in the focused control.

For example, if you have a form that allows the user to edit customer information, you might call the Focus() method on the first text box in that form. That way, the cursor appears in this text box immediately when the page first loads in the browser. If the text box is partway down the form, the page even scrolls down to it automatically. The user can then move from control to control by using the time-honored Tab key.

If you're a seasoned HTML developer, you know there isn't any built-in way to give focus to an input control. Instead, you need to rely on JavaScript. This is the secret to ASP.NET's implementation. When your code is finished processing and the page is rendered, ASP.NET adds an extra block of JavaScript code to the end of your page. This JavaScript code simply sets the focus to the last control that used the Focus() method. If you haven't called Focus() at all, this code isn't added to the page.

Rather than call the Focus() method programmatically, you can set a control that should always be focused by setting the DefaultFocus property of the <form> tag:

```
<form DefaultFocus="TextBox2" runat="server">
```

You can override the default focus by calling the Focus() method in your code.

Another way to manage focus is by using access keys. For example, if you set the AccessKey property of a TextBox to A, pressing Alt+A will switch focus to the TextBox. Labels can also get into the game, even though they can't accept focus. The trick is to set the Label.AssociatedControlID property to specify a linked input control. That way, the label transfers focus to the associated control.

For example, the following label gives focus to TextBox2 when the keyboard combination Alt+2 is pressed:

```
<asp:Label AccessKey="2" AssociatedControlID="TextBox2" runat="server"
 Text="TextBox2:" />
<asp:TextBox runat="server" ID="TextBox2" />
```

Focusing and access keys are also supported in non-Microsoft browsers, including Firefox.

The Default Button

Along with control focusing, ASP.NET also allows you to designate a default button on a web page. The default button is the button that is "clicked" when the user presses the Enter key. For example, if your web page includes a form, you might want to make the submit button into a default button. That way, if the user hits Enter at any time, the page is posted back and the Button.Click event is fired for that button.

To designate a default button, you must set the HtmlForm.DefaultButton property with the ID of the respective control, as shown here:

```
<form DefaultButton="cmdSubmit" runat="server">
```

The default button must be a control that implements the IButtonControl interface. The interface is implemented by the Button, LinkButton, and ImageButton web controls but not by any of the HTML server controls.

In some cases, it makes sense to have more than one default button. For example, you might create a web page with two groups of input controls. Both groups may need a different default button. You can handle this by placing the groups into separate panels. The Panel control also exposes the DefaultButton property, which works when any input control it contains gets the focus.

CONTROL PREFIXES

When working with web controls, it's often useful to use a three-letter lowercase prefix to identify the type of control. The preceding example (and those in the rest of this book) follows this convention to make user interface code as clear as possible. Some recommended control prefixes are as follows:

- Button: cmd (or btn)
- CheckBox: chk
- Image: img
- Label: lbl
- List control: lst
- Panel: pnl
- RadioButton: opt
- TextBox: txt

If you're a veteran programmer, you'll also notice that this book doesn't use prefixes to identify data types. This is in keeping with the philosophy of .NET, which recognizes that data types can often change freely and without consequence and that variables often point to full-featured objects instead of simple data variables.

List Controls

The list controls include the ListBox, DropDownList, CheckBoxList, RadioButtonList, and BulletedList. They all work in essentially the same way but are rendered differently in the browser. The ListBox, for example, is a rectangular list that displays several entries, while the DropDownList shows only the selected item. The CheckBoxList and RadioButtonList are similar to the ListBox, but every item is rendered as a check box or option button, respectively. Finally, the BulletedList is the odd one out—it's the only list control that isn't selectable. Instead, it renders itself as a sequence of numbered or bulleted items.

All the selectable list controls provide a SelectedIndex property that indicates the selected row as a zero-based index (just like the HtmlSelect control you used in the previous chapter). For example, if the first item in the list is selected, the SelectedIndex will be 0. Selectable list controls also provide an additional SelectedItem property, which allows your code to retrieve the ListItem object that represents the selected item. The ListItem object provides three important properties: Text (the displayed content), Value (the hidden value from the HTML markup), and Selected (true or false depending on whether the item is selected).

In the previous chapter, you used code like this to retrieve the selected ListItem object from an HtmlSelect control called Currency, as follows:

```
ListItem item;
item = Currency.Items[Currency.SelectedIndex];
```

If you used the ListBox web control, you can simplify this code with a clearer syntax:

```
ListItem item;
item = Currency.SelectedItem;
```

Multiple-Select List Controls

Some list controls can allow multiple selections. This isn't allowed for the DropDownList or RadioButtonList, but it is supported for a ListBox, provided you have set the SelectionMode property to the enumerated value ListSelectionMode.Multiple. The user can then select multiple items by holding down the Ctrl key while clicking the items in the list. With the CheckBoxList, multiple selections are always possible.

The SelectedIndex and SelectedItem properties aren't much help with a list that supports multiple selection, because they will simply return the first selected item. Instead, you can find all the selected items by iterating through the Items collection of the list control and checking the ListItem.Selected property of each item. (If it's true, that item is one of the selected items.) Figure 6-5 shows a simple web page example that provides a list of computer languages. After the user clicks the OK button, the page indicates which selections the user made.

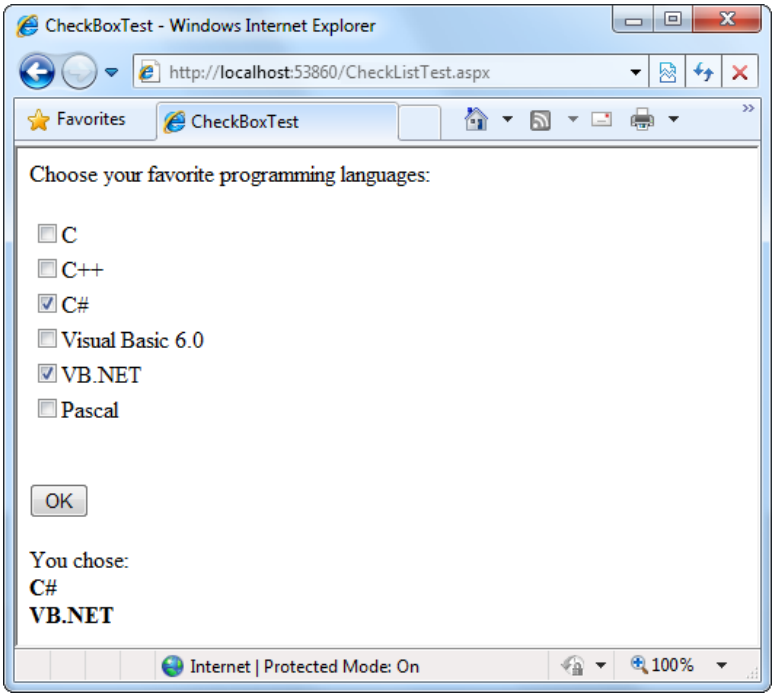

Figure 6-5. *A simple CheckListBox test*

The .aspx file for this page defines CheckListBox, Button, and Label controls, as shown here:

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="CheckListTest.aspx.cs" Inherits="CheckListTest" %>
<!DOCTYPE html>
<html>
<head runat="server">
  <title>CheckBoxTest</title>
</head>
<body>
  <form runat="server">
    <div>
      Choose your favorite programming languages:<br /><br />
      <asp:CheckBoxList ID="chklst" runat="server" /><br /><br />
      <asp:Button ID="cmdOK" Text="OK" OnClick="cmdOK_Click" runat="server" />
      <br /><br />
      <asp:Label ID="lblResult" runat="server" />
    </div>
  </form>
</body>
</html>
```

The code adds items to the CheckListBox at startup and iterates through the collection when the button is clicked:

```
public partial class CheckListTest : System.Web.UI.Page
{

    protected void Page_Load(object sender, EventArgs e)
    {
        if (!this.IsPostBack)
        {
            chklst.Items.Add("C");
            chklst.Items.Add("C++");
            chklst.Items.Add("C#");
            chklst.Items.Add("Visual Basic 6.0");
            chklst.Items.Add("VB.NET");
            chklst.Items.Add("Pascal");
        }
    }

    protected void cmdOK_Click(object sender, EventArgs e)
    {
        lblResult.Text = "You chose:<b>";

        foreach (ListItem lstItem in chklst.Items)
        {
            if (lstItem.Selected == true)
            {
                // Add text to label.
                lblResult.Text += "<br />" + lstItem.Text;
            }
        }
        lblResult.Text += "</b>";
    }
}
```

The BulletedList Control

The BulletedList control is a server-side equivalent of the <ul> (unordered list) and <ol> (ordered list) elements. As with all list controls, you set the collection of items that should be displayed through the Items property. Additionally, you can use the properties in Table 6-4 to configure how the items are displayed.

Table 6-4. *Added BulletedList Properties*

Property	Description
BulletStyle	Determines the type of list. Choose from Numbered (1, 2, 3, . . .); LowerAlpha (a, b, c, . . .) and UpperAlpha (A, B, C, . . .); LowerRoman (i, ii, iii, ; . . .) and UpperRoman (I, II, III, . . .); and the bullet symbols Disc, Circle, Square, or CustomImage (in which case you must set the BulletImageUrl property).
BulletImageUrl	If the BulletStyle is set to CustomImage, this points to the image that is placed to the left of each item as a bullet.
FirstBulletNumber	In an ordered list (using the Numbered, LowerAlpha, UpperAlpha, LowerRoman, and UpperRoman styles), this sets the first value. For example, if you set FirstBulletNumber to 3, the list might read 3, 4, 5 (for Numbered) or C, D, E (for UpperAlpha).
DisplayMode	Determines whether the text of each item is rendered as text (use Text, the default) or a hyperlink (use LinkButton or HyperLink). The difference between LinkButton and HyperLink is how they treat clicks. When you use LinkButton, the BulletedList fires a Click event that you can react to on the server to perform the navigation. When you use HyperLink, the BulletedList doesn't fire the Click event—instead, it treats the text of each list item as a relative or absolute URL, and renders them as ordinary HTML hyperlinks. When the user clicks an item, the browser attempts to navigate to that URL.

If you set the DisplayMode to LinkButton, you can react to the Button.Click event to determine which item was clicked. For example, say you have a BulletedList like this:

```
<asp:BulletedList BulletStyle="Numbered" DisplayMode="LinkButton"
  ID="BulletedList1" OnClick="BulletedList1_Click" runat="server">
</asp:BulletedList>
```

You can use the following code to intercept its clicks:

```
protected void BulletedList1_Click(object sender, BulletedListEventArgs e)
{
    // Find the clicked item in the list.
    // (You can't use the SelectedIndex property here because static lists
    // don't support selection.)
    string itemText = BulletedList1.Items[e.Index].Text;
    Label1.Text = "You choose item" + itemText;
}
```

Figure 6-6 shows all the BulletStyle values that the BulletList supports. When you click one of the items, the list changes to use that BulletStyle. You can try this example page with the sample WebControls project for this chapter.

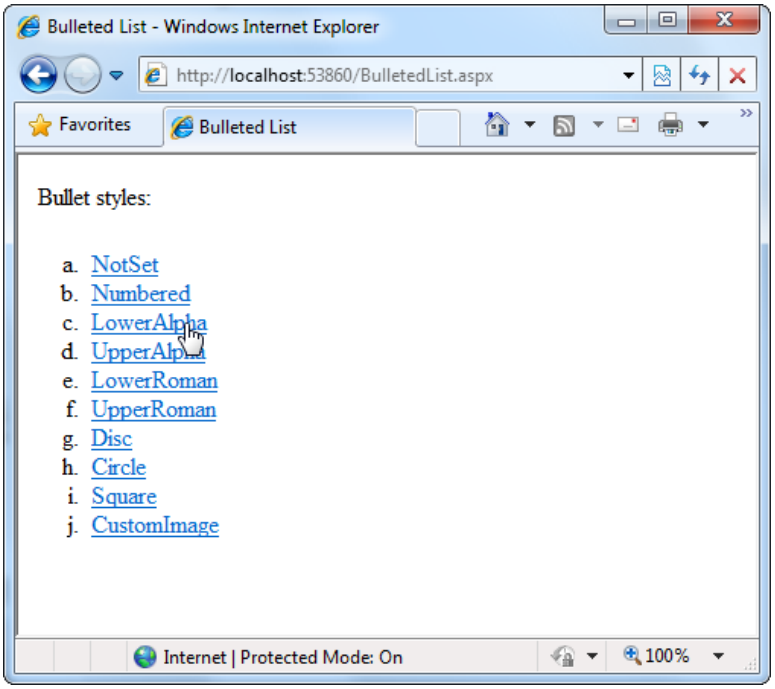

Figure 6-6. *Various BulletedList styles*

Table Controls

Essentially, the Table control is built out of a hierarchy of objects. Each Table object contains one or more TableRow objects. In turn, each TableRow object contains one or more TableCell objects. Each TableCell object contains other ASP.NET controls or HTML content that displays information. If you're familiar with the HTML table tags, this relationship (shown in Figure 6-7) will seem fairly logical.

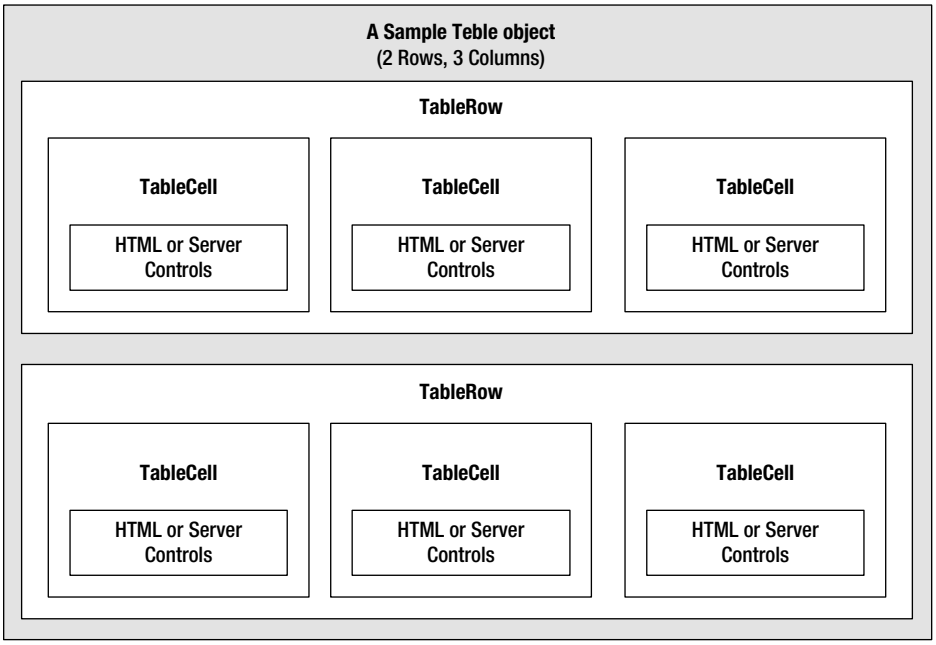

Figure 6-7. *Table control containment*

To create a table dynamically, you follow the same philosophy as you would for any other web control. First you create and configure the necessary ASP.NET objects. Then ASP.NET converts these objects to their final HTML representation before the page is sent to the client.

Consider the example shown in Figure 6-8. It allows the user to specify a number of rows and columns as well as whether cells should have borders.

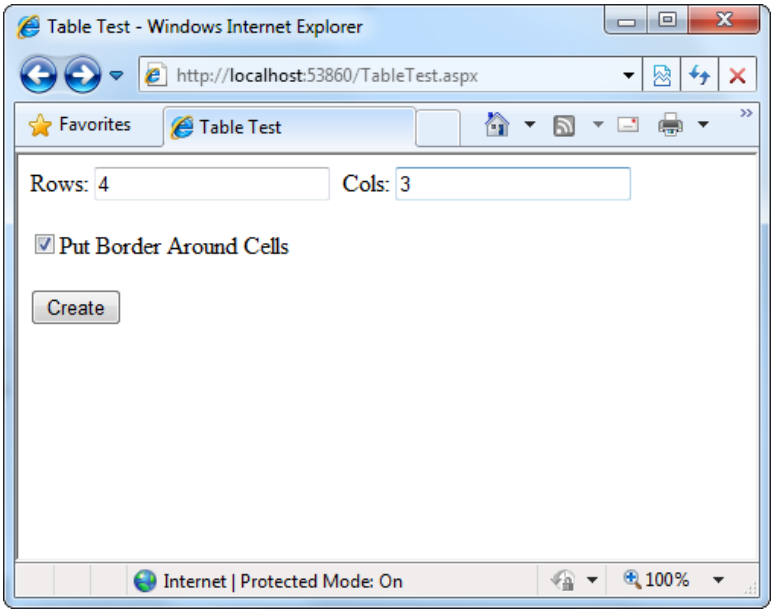

Figure 6-8. *The table test options*

When the user clicks the Create button, the table is filled dynamically with sample data according to the selected options, as shown in Figure 6-9.

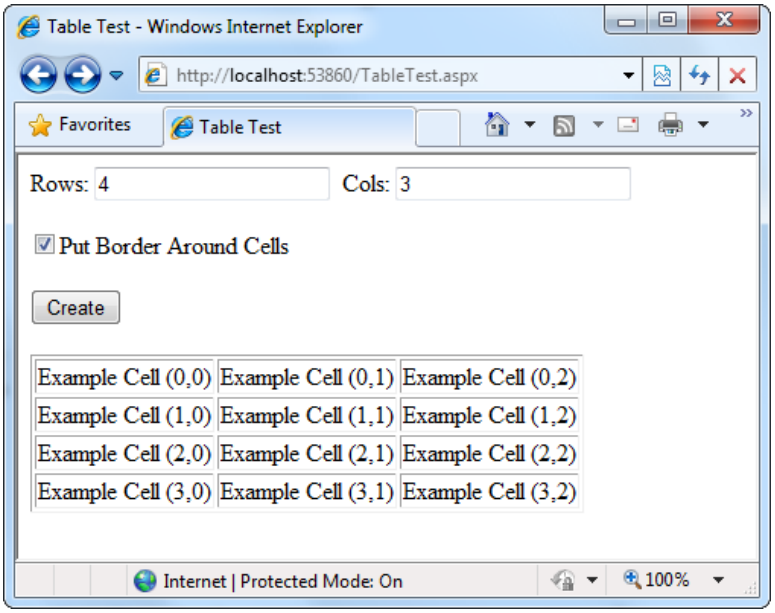

Figure 6-9. *A dynamically generated table*

The .aspx code creates the TextBox, CheckBox, Button, and Table controls:

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="TableTest.aspx.cs" Inherits="TableTest" %>
<!DOCTYPE html>
<html>
<head runat="server">
  <title>Table Test</title>
</head>
<body>
  <form runat="server">
    <div>
      Rows:
      <asp:TextBox ID="txtRows" runat="server" />
       Cols:
      <asp:TextBox ID="txtCols" runat="server" />
      <br /><br />
      <asp:CheckBox ID="chkBorder" runat="server"
       Text="Put Border Around Cells" />
      <br /><br />
      <asp:Button ID="cmdCreate" OnClick="cmdCreate_Click" runat="server"
       Text="Create" />
      <br /><br />
      <asp:Table ID="tbl" runat="server" />
    </div>
  </form>
</body>
</html>
```

You'll notice that the Table control doesn't contain any actual rows or cells. To make a valid table, you would need to nest several layers of tags. The following example creates a table with a single cell that contains the text *A Test Row*:

```
<asp:Table ID="tbl" runat="server">
  <asp:TableRow ID="row" runat="server">
    <asp:TableCell ID="cell" runat="server">A Sample Value</asp:TableCell>
  </asp:TableRow>
</asp:Table>
```

The table test web page doesn't have any nested elements. This means the table will be created as a server-side control object, but unless the code adds rows and cells, the table will not be rendered in the final HTML page.

The TableTest class uses two event handlers. When the page is first loaded, it adds a border around the table. When the button is clicked, it dynamically creates the required TableRow and TableCell objects in a loop.

```
public partial class TableTest : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        // Configure the table's appearance.
        // This could also be performed in the .aspx file
        // or in the cmdCreate_Click event handler.
        tbl.BorderStyle = BorderStyle.Inset;
        tbl.BorderWidth = Unit.Pixel(1);
    }
```

```
    protected void cmdCreate_Click(object sender, EventArgs e)
    {
        // Remove all the current rows and cells.
        // This is not necessary if EnableViewState is set to false.
        tbl.Controls.Clear();

        int rows = Int32.Parse(txtRows.Text);
        int cols = Int32.Parse(txtCols.Text);

        for (int row = 0; row < rows; row++)
        {
            // Create a new TableRow object.
            TableRow rowNew = new TableRow();

            // Put the TableRow in the Table.
            tbl.Controls.Add(rowNew);

            for (int col = 0; col < cols; col++)
            {
                // Create a new TableCell object.
                TableCell cellNew = new TableCell();

                cellNew.Text = "Example Cell (" + row.ToString() + ",";
                cellNew.Text += col.ToString() + ")";

                if (chkBorder.Checked)
                {
                    cellNew.BorderStyle = BorderStyle.Inset;
                    cellNew.BorderWidth = Unit.Pixel(1);
                }

                // Put the TableCell in the TableRow.
                rowNew.Controls.Add(cellNew);
            }
        }
    }
}
```

This code uses the Controls collection to add child controls. Every container control provides this property. You could also use the TableCell.Controls collection to add web controls to each TableCell. For example, you could place an Image control and a Label control in each cell. In this case, you can't set the TableCell.Text property. The following code snippet uses this technique, and Figure 6-10 displays the results:

```
// Create a new TableCell object.
cellNew = new TableCell();

// Create a new Label object.
Label lblNew = new Label();
lblNew.Text = "Example Cell (" + row.ToString() + ",";
lblNew.Text += col.ToString() + ")<br />";
```

```
System.Web.UI.WebControls.Image imgNew = new System.Web.UI.WebControls.Image();
imgNew.ImageUrl = "cellpic.png";

// Put the label and picture in the cell.
cellNew.Controls.Add(lblNew);
cellNew.Controls.Add(imgNew);

// Put the TableCell in the TableRow.
rowNew.Controls.Add(cellNew);
```

The real flexibility of the table test page is that each Table, TableRow, and TableCell is a full-featured object. If you want, you can give each cell a different border style, border color, and text color by setting the corresponding properties.

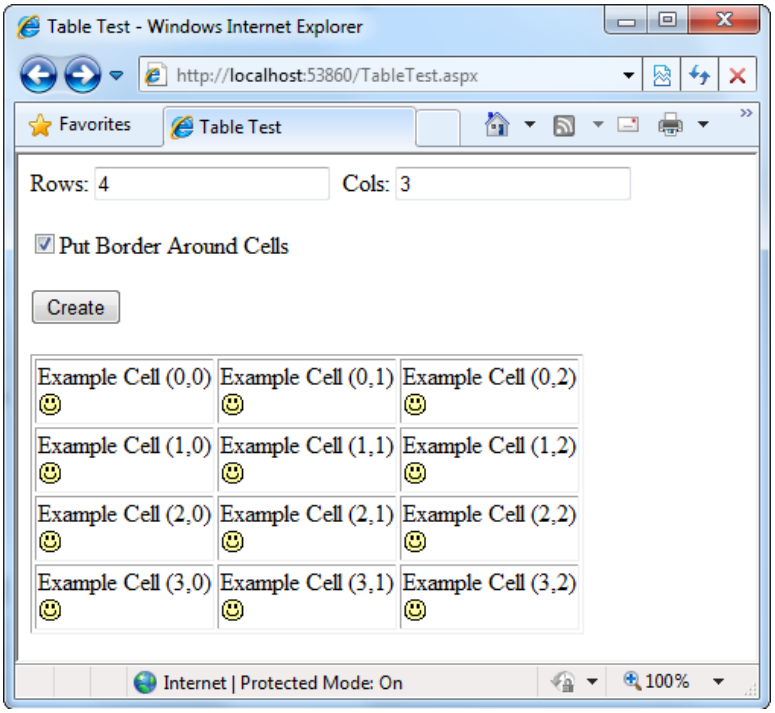

Figure 6-10. *A table with contained controls*

Web Control Events and AutoPostBack

The previous chapter explained that one of the main limitations of HTML server controls is their limited set of useful events—they have exactly two. HTML controls that trigger a postback, such as buttons, raise a ServerClick event. Input controls provide a ServerChange event that doesn't actually fire until the page is posted back.

ASP.NET server controls are really an ingenious illusion. You'll recall that the code in an ASP.NET page is processed on the server. It's then sent to the user as ordinary HTML. Figure 6-11 illustrates the order of events in page processing.

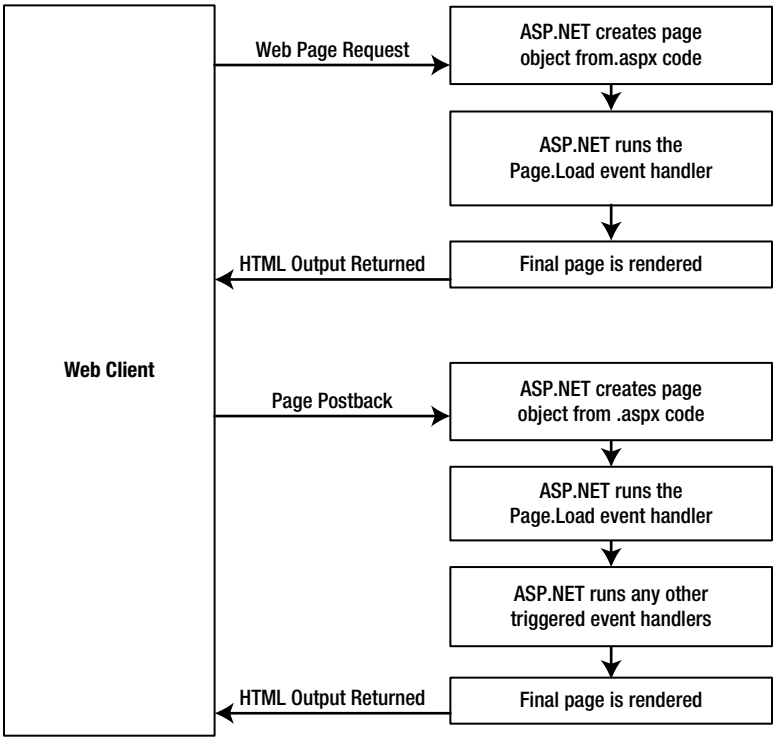

Figure 6-11. *The page-processing sequence*

This is the same in ASP.NET as it was in traditional ASP programming. The question is, how can you write server code that will react *immediately* to an event that occurs on the client?

Some events, such as the Click event of a button, do occur immediately. That's because when clicked, the button posts back the page. This is a basic convention of HTML forms. However, other actions *do* cause events but *don't* trigger a postback—for example, when the user changes the text in a text box (which triggers the TextChanged event) or chooses a new item in a list (the SelectedIndexChanged event). You might want to respond to these events, but without a postback, your code has no way to run.

ASP.NET handles this by giving you two options:

- You can wait until the next postback to react to the event. For example, imagine you want to react to the SelectedIndexChanged event in a list. If the user selects an item in a list, nothing happens immediately. However, if the user then clicks a button to post back the page, *two* events fire: Button.Click followed by ListBox.SelectedIndexChanged. And if you have several controls, it's quite possible for a single postback to result in several change events, which fire one after the other, in an undetermined order.
- You can use the *automatic postback* feature to force a control to post back the page immediately when it detects a specific user action. In this scenario, when the user clicks a new item in the list, the page is posted back, your code executes, and a new version of the page is returned.

The option you choose depends on the result you want. If you need to react immediately (for example, you want to update another control when a specific action takes place), you need to use automatic postbacks. On the other hand, automatic postbacks can sometimes make the page less responsive, because each postback and page refresh adds a short, but noticeable, delay and page refresh. (You'll learn how to create pages that update themselves without a noticeable page refresh when you consider ASP.NET AJAX in Chapter 25.)

All input web controls support automatic postbacks. Table 6-5 provides a basic list of web controls and their events.

Table 6-5. *Web Control Events*

Event	Web Controls That Provide It	Always Posts Back
Click	Button, ImageButton	True
TextChanged	TextBox (fires only after the user changes the focus to another control)	False
CheckedChanged	CheckBox, RadioButton	False
SelectedIndexChanged	DropDownList, ListBox, CheckBoxList, RadioButtonList	False

If you want to capture a change event (such as TextChanged, CheckedChanged, or SelectedIndexChanged) immediately, you need to set the control's AutoPostBack property to true. This way, the page will be submitted automatically when the user interacts with the control (for example, picks a selection in the list, clicks a radio button or a check box, or changes the text in a text box and then moves to a new control).

When the page is posted back, ASP.NET will examine the page, load all the current information, and then allow your code to perform some extra processing before returning the page back to the user (see Figure 6-12). Depending on the result you want, you could have a page that has some controls that post back automatically and others that don't.

This postback system isn't ideal for all events. For example, some events that you may be familiar with from Windows programs, such as mouse movement events or key press events, aren't practical in an ASP.NET application. Resubmitting the page every time a key is pressed or the mouse is moved would make the application unbearably slow and unresponsive.

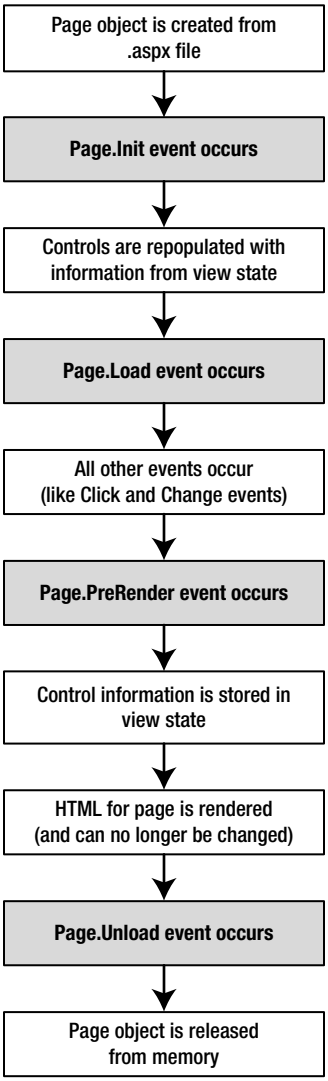

Figure 6-12. *The postback processing sequence*

How Postback Events Work

Chapter 1 explained that not all types of web programming use server-side code like ASP.NET. One common example of client-side web programming is JavaScript, which uses script code that's executed by the browser. ASP.NET uses the client-side abilities of JavaScript to bridge the gap between client-side and server-side code.

(Another scripting language is VBScript, but JavaScript is the only one that works on all modern browsers, including Internet Explorer, Chrome, Firefox, Safari, and Opera.)

Here's how it works: If you create a web page that includes one or more web controls that are configured to use AutoPostBack, ASP.NET adds a special JavaScript function to the rendered HTML page. This function is named __doPostBack(). When called, it triggers a postback, sending data back to the web server.

ASP.NET also adds two hidden input fields that are used to pass information back to the server. This information consists of the ID of the control that raised the event and any additional information that might be relevant. These fields are initially empty, as shown here:

```
<input type="hidden" name="__EVENTTARGET" ID="__EVENTTARGET" value="" />
<input type="hidden" name="__EVENTARGUMENT" ID="__EVENTARGUMENT" value="" />
```

The __doPostBack() function has the responsibility of setting these values with the appropriate information about the event and then submitting the form. The __doPostBack() function is shown here:

```
<script language="text/javascript">
function __doPostBack(eventTarget, eventArgument) {
    if (!theForm.onsubmit || (theForm.onsubmit() != false)) {
        theForm.__EVENTTARGET.value = eventTarget;
        theForm.__EVENTARGUMENT.value = eventArgument;
        theForm.submit();
    }
  ...
}
</script>
```

Remember, ASP.NET generates the __doPostBack() function automatically, provided at least one control on the page uses automatic postbacks.

Finally, any control that has its AutoPostBack property set to true is connected to the __doPostBack() function by using the onclick or onchange attributes. These attributes indicate what action the browser should take in response to the client-side JavaScript events onclick and onchange.

The following example shows the tag for a list control named lstBackColor, which posts back automatically. Whenever the user changes the selection in the list, the client-side onchange event fires. The browser then calls the __doPostBack() function, which sends the page back to the server.

```
<select ID="lstBackColor" onchange="__doPostBack('lstBackColor',")"
 language="javascript">
```

In other words, ASP.NET automatically changes a client-side JavaScript event into a server-side ASP.NET event, using the __doPostBack() function as an intermediary. Figure 6-13 shows this process.

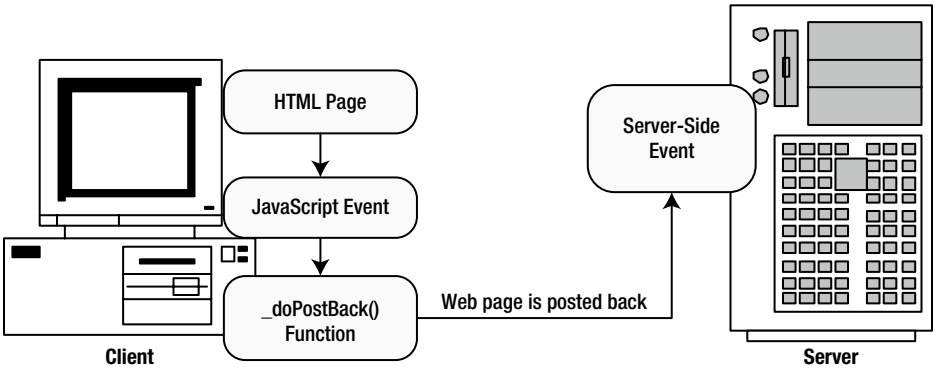

Figure 6-13. *An automatic postback*

The Page Life Cycle

To understand how web control events work, you need to have a solid understanding of the page life cycle. Consider what happens when a user changes a control that has the AutoPostBack property set to true:

1. On the client side, the JavaScript __doPostBack function is invoked, and the page is resubmitted to the server.
2. ASP.NET re-creates the Page object by using the .aspx file.
3. ASP.NET retrieves state information from the hidden view state field and updates the controls accordingly.
4. The Page.Load event is fired.
5. The appropriate change event is fired for the control. (If more than one control has been changed, the order of change events is undetermined.)
6. The Page.PreRender event fires, and the page is rendered (transformed from a set of objects to an HTML page).
7. Finally, the Page.Unload event is fired.
8. The new page is sent to the client.

To watch these events in action, it helps to create a simple event tracker application. All this application does is write a new entry to a list control every time one of the events it's monitoring occurs. This allows you to see the order in which events are triggered. Figure 6-14 shows what the window looks like after it's been loaded once, posted back (when the text box content was changed), and posted back again (when the check box state was changed).

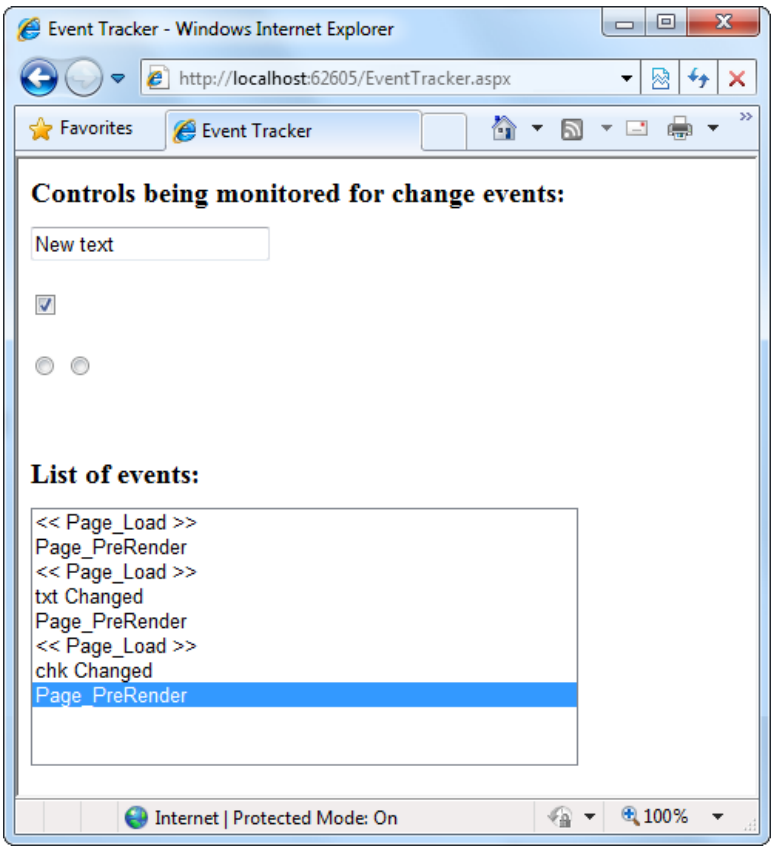

Figure 6-14. *The event tracker*

Listing 6-1 shows the markup code for the event tracker, and Listing 6-2 shows the code-behind class that makes it work.

Listing 6-1. EventTracker.aspx

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="EventTracker.aspx.cs" Inherits="EventTracker" %>
<!DOCTYPE html>
<html>
<head runat="server">
  <title>Event Tracker</title>
</head>
<body>
  <form runat="server">
    <div>
      <h1>Controls being monitored for change events:</h1>
      <asp:TextBox ID="txt" runat="server" AutoPostBack="true"
       OnTextChanged="CtrlChanged" />
      <br /><br />
```

```
      <asp:CheckBox ID="chk" runat="server" AutoPostBack="true"
       OnCheckedChanged="CtrlChanged"/>
      <br /><br />
      <asp:RadioButton ID="opt1" runat="server" GroupName="Sample"
       AutoPostBack="True" OnCheckedChanged="CtrlChanged"/>
      <asp:RadioButton ID="opt2" runat="server" GroupName="Sample"
       AutoPostBack="True" OnCheckedChanged="CtrlChanged"/>

       <h1>List of events:</h1>
      <asp:ListBox ID="lstEvents" runat="server" Width="355px"
       Height="150px" /><br />
      <br /><br /><br />
    </div>
  </form>
</body>
</html>
```

Listing 6-2. EventTracker.aspx.cs

```
public partial class EventTracker : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Log("<< Page_Load >>");
    }

    protected void Page_PreRender(object sender, EventArgs e)
    {
        // When the Page.PreRender event occurs, it is too late
        // to change the list.
        Log("Page_PreRender");
    }

    protected void CtrlChanged(Object sender, EventArgs e)
    {
        // Find the control ID of the sender.
        // This requires converting the Object type into a Control class.
        string ctrlName = ((Control)sender).ID;
        Log(ctrlName + " Changed");
    }

    private void Log(string entry)
    {
        lstEvents.Items.Add(entry);

        // Select the last item to scroll the list so the most recent
        // entries are visible.
        lstEvents.SelectedIndex = lstEvents.Items.Count - 1;
    }
}
```

Dissecting the Code . . .

The following points are worth noting about this code:

- The code writes to the ListBox by using a private Log() method. The Log() method adds the text and automatically scrolls to the bottom of the list each time a new entry is added, thereby ensuring that the most recent entries remain visible.
- All the change events are handled by the same method, CtrlChanged(). If you look carefully at the .aspx file, you'll notice that each input control connects its monitored event to the CtrlChanged() method. The event-handling code in the CtrlChanged() method uses the source parameter to find out what control sent the event, and it incorporates that information in the log string.
- The page includes event handlers for the Page.Load and Page.PreRender events. As with all page events, these event handlers are connected by method name. That means to add the event handler for the Page.PreRender event, you simply need to add a method named Page_PreRender(), like the one shown here.

An Interactive Web Page

Now that you've had a whirlwind tour of the basic web control model, it's time to put it to work with the second single-page utility. In this case, it's a simple example for a dynamic e-card generator. You could extend this sample (for example, allowing users to store e-cards to the database), but even on its own, this example demonstrates basic control manipulation with ASP.NET.

The web page is divided into two regions. On the left is an ordinary tag containing a set of web controls for specifying card options. On the right is a Panel control (named pnlCard), which contains two other controls (lblGreeting and imgDefault) that are used to display user-configurable text and a picture. This text and picture represents the greeting card. When the page first loads, the card hasn't yet been generated, and the right portion is blank (as shown in Figure 6-15).

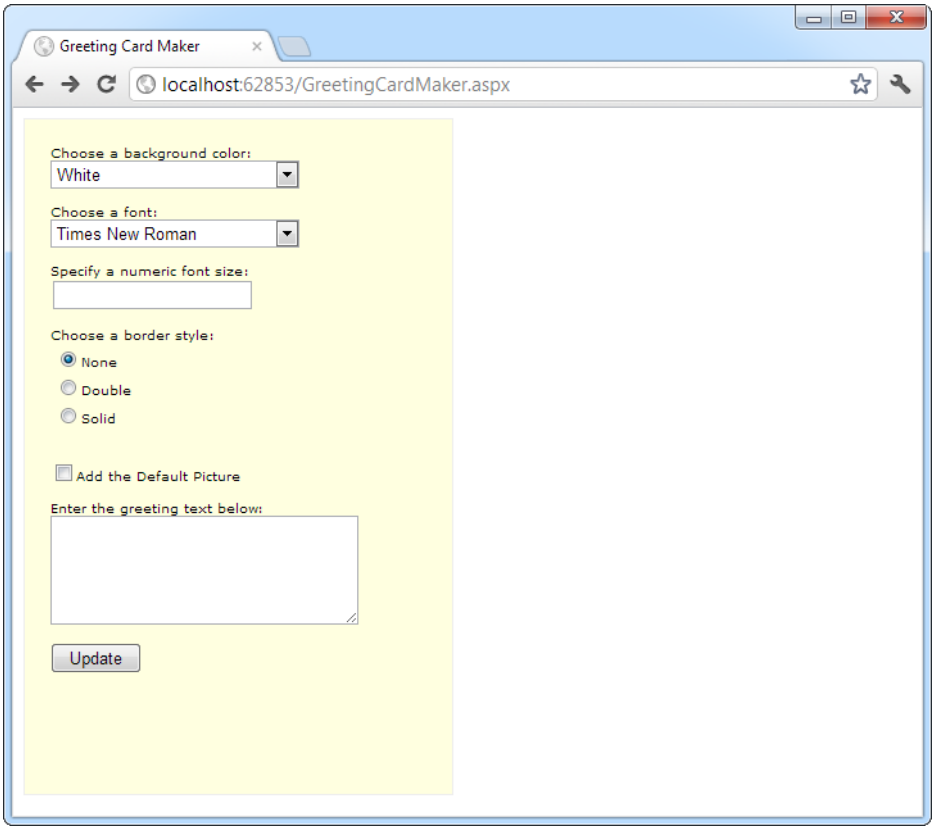

Figure 6-15. *The e-card generator*

■ **Tip** The element is useful when you want to group text and controls and apply a set of formatting properties (such as a color or font) to all of them. The element is also an essential tool for positioning blocks of content in a page. For these reasons, the element is used in many of the examples in this book. You'll learn more about using for layout and formatting in Chapter 12.

Whenever the user clicks the Update button, the page is posted back and the "card" is updated (see Figure 6-16).

Figure 6-16. *A user-configured greeting card*

The .aspx layout code is straightforward. Of course, the sheer length of it makes it difficult to work with efficiently. Here's the markup, with the formatting details trimmed down to the bare essentials:

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="GreetingCardMaker.aspx.cs" Inherits="GreetingCardMaker" %>
<!DOCTYPE html>
<html>
<head runat="server">
    <title>Greeting Card Maker</title>
</head>
<body>
  <form runat="server">
    <div>
      <!-- Here are the controls: -->
      Choose a background color:<br />
      <asp:DropDownList ID="lstBackColor" runat="server" Width="194px"
        Height="22px"/><br /><br />
      Choose a font:<br />
      <asp:DropDownList ID="lstFontName" runat="server" Width="194px"
        Height="22px" /><br /><br />
```

```
      Specify a numeric font size:<br />
      <asp:TextBox ID="txtFontSize" runat="server" /><br /><br />
      Choose a border style:<br />
      <asp:RadioButtonList ID="lstBorder" runat="server" Width="177px"
        Height="59px" /><br /><br />
      <asp:CheckBox ID="chkPicture" runat="server"
        Text="Add the Default Picture"></asp:CheckBox><br /><br />
      Enter the greeting text below:<br />
      <asp:TextBox ID="txtGreeting" runat="server" Width="240px" Height="85px"
        TextMode="MultiLine" /><br /><br />
      <asp:Button ID="cmdUpdate" OnClick="cmdUpdate_Click"
        runat="server" Width="71px" Height="24px" Text="Update" />
    </div>

    <!-- Here is the card: -->
    <asp:Panel ID="pnlCard" runat="server"
      Width="339px" Height="481px" HorizontalAlign="Center"
      style="POSITION: absolute; TOP: 16px; LEFT: 313px;">
    <br /> 
    <asp:Label ID="lblGreeting" runat="server" Width="256px"
      Height="150px" /><br /><br /><br />
    <asp:Image ID="imgDefault" runat="server" Width="212px"
      Height="160px" />
    </asp:Panel>
  </form>
</body>
</html>
```

To get the two-column layout in this example, you have two choices. You can use HTML tables (which are a somewhat old-fashioned technique), or you can use absolute positioning with CSS styles (as in this example). The essence of absolute positioning is easy to grasp. Just look at the style attribute in the Panel control, which specifies a fixed top and left coordinate on the web page. When the panel is rendered to HTML, this point becomes its top-left corner.

■ **Note** Absolute positioning is a feature of CSS, the Cascading Style Sheets standard. As such, it works in any XHTML element, not just ASP.NET controls. Absolute positioning is described in detail in Chapter 12.

The code follows the familiar pattern with an emphasis on two events: the Page.Load event, where initial values are set, and the Button.Click event, where the card is generated.

```
using System.Drawing;

public partial class GreetingCardMaker : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!this.IsPostBack)
        {
            // Set color options.
            lstBackColor.Items.Add("White");
```

```
            lstBackColor.Items.Add("Red");
            lstBackColor.Items.Add("Green");
            lstBackColor.Items.Add("Blue");
            lstBackColor.Items.Add("Yellow");

            // Set font options.
            lstFontName.Items.Add("Times New Roman");
            lstFontName.Items.Add("Arial");
            lstFontName.Items.Add("Verdana");
            lstFontName.Items.Add("Tahoma");

            // Set border style options by adding a series of
            // ListItem objects.
            ListItem item = new ListItem();

            // The item text indicates the name of the option.
            item.Text = BorderStyle.None.ToString();

            // The item value records the corresponding integer
            // from the enumeration. To obtain this value, you
            // must cast the enumeration value to an integer,
            // and then convert the number to a string so it
            // can be placed in the HTML page.
            item.Value = ((int)BorderStyle.None).ToString();

            // Add the item.
            lstBorder.Items.Add(item);

            // Now repeat the process for two other border styles.
            item = new ListItem();
            item.Text = BorderStyle.Double.ToString();
            item.Value = ((int)BorderStyle.Double).ToString();
            lstBorder.Items.Add(item);

            item = new ListItem();
            item.Text = BorderStyle.Solid.ToString();
            item.Value = ((int)BorderStyle.Solid).ToString();
            lstBorder.Items.Add(item);

            // Select the first border option.
            lstBorder.SelectedIndex = 0;

            // Set the picture.
            imgDefault.ImageUrl = "defaultpic.png";
        }
    }

    protected void cmdUpdate_Click(object sender, EventArgs e)
    {
        // Update the color.
        pnlCard.BackColor = Color.FromName(lstBackColor.SelectedItem.Text);
```

```
        // Update the font.
        lblGreeting.Font.Name = lstFontName.SelectedItem.Text;

        if (Int32.Parse(txtFontSize.Text) > 0)
        {
            lblGreeting.Font.Size =
              FontUnit.Point(Int32.Parse(txtFontSize.Text));
        }

        // Update the border style. This requires two conversion steps.
        // First, the value of the list item is converted from a string
        // into an integer. Next, the integer is converted to a value in
        // the BorderStyle enumeration.
        int borderValue = Int32.Parse(lstBorder.SelectedItem.Value);
        pnlCard.BorderStyle = (BorderStyle)borderValue;

        // Update the picture.
        if (chkPicture.Checked)
        {
            imgDefault.Visible = true;
        }
        else
        {
            imgDefault.Visible = false;
        }

        // Set the text.
        lblGreeting.Text = txtGreeting.Text;
    }
}
```

As you can see, this example limits the user to a few preset font and color choices. The code for the BorderStyle option is particularly interesting. The lstBorder control has a list that displays the text name of one of the BorderStyle enumerated values. You'll remember from the introductory chapters that every enumerated value is really an integer with a name assigned to it. The lstBorder also secretly stores the corresponding number so that the code can retrieve the number and set the enumeration easily when the user makes a selection and the cmdUpdate_Click event handler fires.

Improving the Greeting Card Generator

ASP.NET pages have access to the full .NET class library. With a little exploration, you'll find classes that might help the greeting-card maker, such as tools that let you retrieve all the known color names and all the fonts installed on the web server.

For example, you can fill the lstFontName control with a list of fonts by using the InstalledFontCollection class. To access it, you need to import the System.Drawing.Text namespace:

```
using System.Drawing.Text;
```

Here's the code that gets the list of fonts and uses it to fill the list:

```
InstalledFontCollection fonts = new InstalledFontCollection();
foreach (FontFamily family in fonts.Families)
```

```
{
    lstFontName.Items.Add(family.Name);
}
```

Figure 6-17 shows the resulting font list.

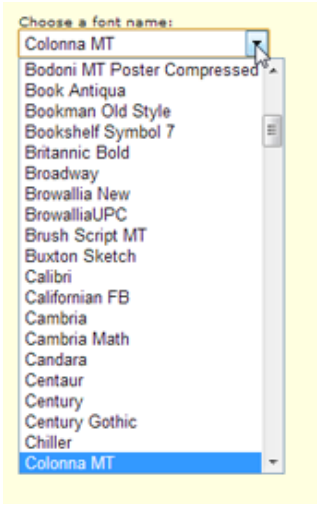

Figure 6-17. *The font list*

To get a list of the color names, you need to resort to a more advanced trick. Although you could hard-code a list of common colors, .NET provides a long list of color names in the System.Drawing.KnownColor enumeration. However, actually *extracting* the names from this enumeration takes some work.

The trick is to use a basic feature of all enumerations: the static Enum.GetNames() method, which inspects an enumeration and provides an array of strings, with one string for each value in the enumeration. The web page can then use data binding to automatically fill the list control with the information in the ColorArray. (You'll explore data binding in much more detail in Chapter 15.)

■ **Note** Don't worry if this example introduces a few features that look entirely alien! These features are more advanced (and aren't tied specifically to ASP.NET). However, they show you some of the flavor that the full .NET class library can provide for a mature application.
