CHAPTER 1

Overview of the ASP.NET Framework

IN THIS CHAPTER

Let's start by building a simple ASP.NET page.

NOTE

For information on installing ASP.NET, see the last section of this chapter.

If you are using Visual Web Developer or Visual Studio, you first need to create a new website. Start Visual Web Developer and select the menu option File, New Web Site. The New Web Site dialog box appears (see Figure 1.1). Enter the folder where you want your new website to be created in the Location field and click the OK button.

NOTE

When you create a new website, you might receive an error message warning you that you need to enable script debugging in Internet Explorer. You'll want to enable script debugging to build Ajax applications. We discuss Ajax in Part IX of this book, "ASP.NET AJAX."

After you create a new website, you can add an ASP.NET page to it. Select the menu option Web Site, Add New Item. Select Web Form and enter the value `FirstPage.aspx` in the Name field. Make sure that both the `Place Code in Separate File` and `Select Master Page` check boxes are unchecked, and click the Add button to create the new ASP.NET page (see Figure 1.2).

The code for the first ASP.NET page is contained in Listing 1.1.

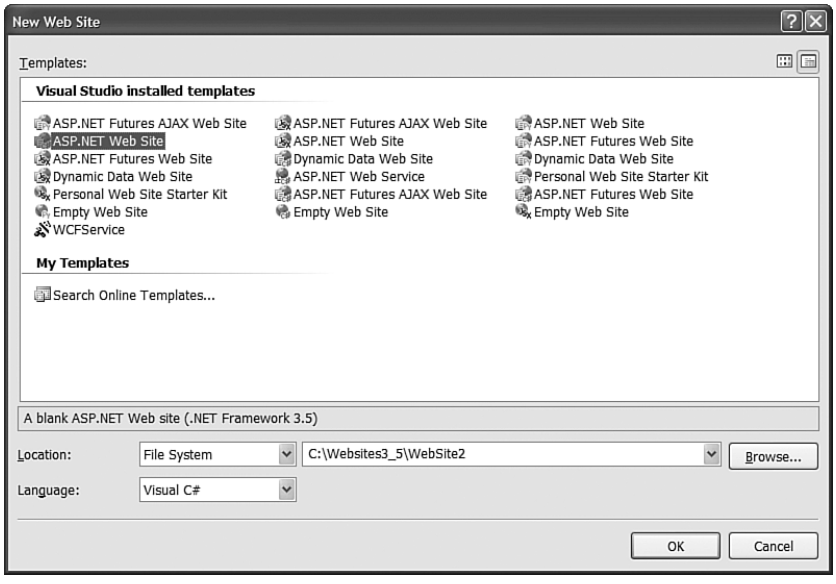

FIGURE 1.1 Creating a new website.

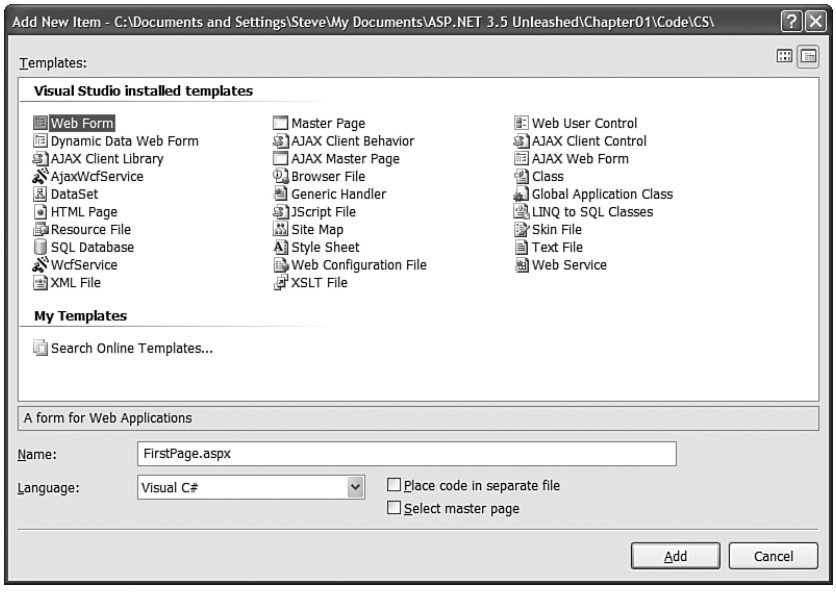

FIGURE 1.2 Adding a new ASP.NET page.

LISTING 1.1 FirstPage.aspx

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">

    void Page_Load()
    {
        lblServerTime.Text = DateTime.Now.ToString();
    }

</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
    <title>First Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

    Welcome to ASP.NET 3.5! The current date and time is:

    <asp:Label
        id="lblServerTime"
        Runat="server" />

    </div>
    </form>
</body>
</html>
```

NOTE

The CD that accompanies this book contains both C# and VB.NET versions of all the code samples. The code samples are also posted online at www.Superexpert.com. Go to the Books section of the website and you can view the listings for each chapter and try the listings "live."

The ASP.NET page in Listing 1.1 displays a brief message and the server's current date and time. You can view the page in Listing 1.1 in a browser by right-clicking the page and selecting View in Browser (see Figure 1.3).

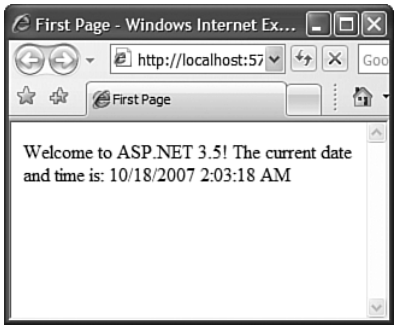

FIGURE 1.3 Viewing `FirstPage.aspx` in a browser.

The page in Listing 1.1 is an extremely simple page. However, it does illustrate the most common elements of an ASP.NET page. The page contains a directive, a code declaration block, and a page render block.

The first line, in Listing 1.1, contains a directive. It looks like this:

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

A directive always begins with the special characters `<%@` and ends with the characters `%>`. Directives are used primarily to provide the compiler with the information it needs to compile the page.

For example, the directive in Listing 1.1 indicates that the code contained in the page is C# code. The page is compiled by the C# compiler and not another compiler such as the Visual Basic .NET (VB.NET) compiler.

The next part of the page begins with the opening `<script runat="server">` tag and ends with the closing `</script>` tag. The `<script>` tag contains something called the *code declaration block*.

The code declaration block contains all the methods used in the page. It contains all the page's functions and subroutines. The code declaration block in Listing 1.1 includes a single method named `Page_Load()`, which looks like this:

```
void Page_Load()
{
    lblServerTime.Text = DateTime.Now.ToString();
}
```

This method assigns the current date and time to the `Text` property of a `Label` control contained in the body of the page named `lblServerTime`.

The `Page_Load()` method is an example of an event handler. This method handles the `Page Load` event. Each and every time the page loads, the method automatically executes and assigns the current date and time to the `Label` control.

The final part of the page is called the *page render block*. The page render block contains everything that is rendered to the browser. In Listing 1.1, the render block includes everything between the opening and closing `<html>` tags.

The majority of the page render block consists of everyday HTML. For example, the page contains the standard HTML `<head>` and `<body>` tags. In Listing 1.1, there are two special things contained in the page render block.

First, notice that the page contains a `<form>` tag that looks like this:

```
<form id="form1" runat="server">
```

This is an example of an ASP.NET control. Because the tag includes a `runat="server"` attribute, the tag represents an ASP.NET control that executes on the server.

ASP.NET pages are often called *web form* pages because they almost always contain a server-side form element.

The page render block also contains a `Label` control. The `Label` control is declared with the `<asp:Label>` tag. In Listing 1.1, the `Label` control is used to display the current date and time.

Controls are the heart of the ASP.NET framework. Most of the ink contained in this book is devoted to describing the properties and features of the ASP.NET controls.

Controls are discussed in more detail shortly. However, first you need to understand the .NET Framework.

NOTE

By default, ASP.NET pages are compatible with the XHTML 1.0 Transitional standard. You'll notice that the page in Listing 1.1 includes an XHTML 1.0 Transitional DOCTYPE. For details on how the ASP.NET framework complies with both XHTML and accessibility standards, see my article at the Microsoft MSDN website (msdn.Microsoft.com), entitled "Building ASP.NET 2.0 Web Sites Using Web Standards."

ASP.NET and the .NET Framework

ASP.NET is part of the Microsoft .NET Framework. To build ASP.NET pages, you need to take advantage of the features of the .NET Framework. The .NET Framework consists of two parts: the Framework Class Library and the Common Language Runtime.

Understanding the Framework Class Library

The .NET Framework contains thousands of classes that you can use when building an application. The Framework Class Library was designed to make it easier to perform the most common programming tasks. Here are just a few examples of the classes in the framework:

- **`File` class**—Enables you to represent a file on your hard drive. You can use the `File` class to check whether a file exists, create a new file, delete a file, and perform many other file-related tasks.
- **`Graphics` class**—Enables you to work with different types of images such as GIF, PNG, BMP, and JPEG images. You can use the `Graphics` class to draw rectangles, arcs, ellipsis, and other elements on an image.
- **`Random` class**—Enables you to generate a random number.
- **`SmtpClient` class**—Enables you to send email. You can use the `SmtpClient` class to send emails that contain attachments and HTML content.

These are only four examples of classes in the Framework. The .NET Framework contains almost 13,000 classes you can use when building applications.

You can view all the classes contained in the Framework by opening the Microsoft .NET Framework SDK documentation and expanding the Class Library node (see Figure 1.4). If you don't have the SDK documentation installed on your computer, then see the last section of this chapter.

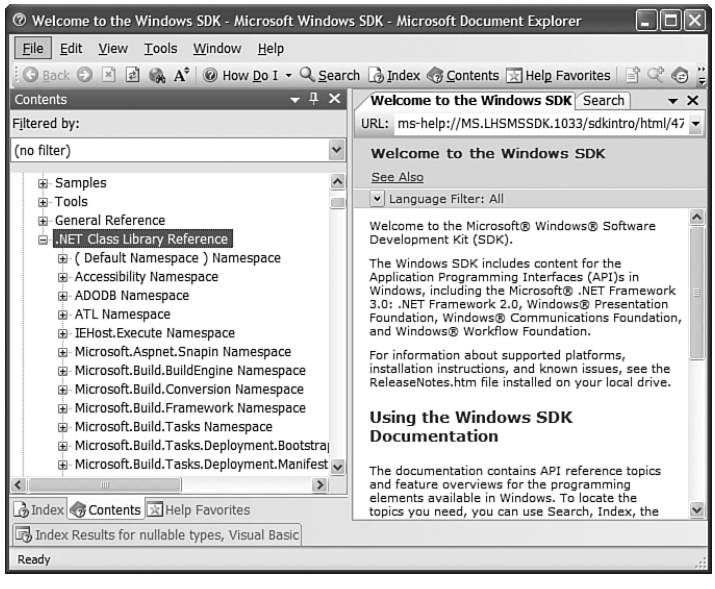

FIGURE 1.4 Opening the Microsoft .NET Framework SDK Documentation.

NOTE

The Microsoft .NET Framework 2.0 includes 18,619 types; 12,909 classes; 401,759 public methods; 93,105 public properties; and 30,546 public events. The .NET Framework 3.0 and 3.5 build on top of this base set of types with even more classes.

Each class in the Framework can include properties, methods, and events. The properties, methods, and events exposed by a class are the members of a class. For example, here is a partial list of the members of the `SmtpClient` class:

- Properties
  - **`Host`**—The name or IP address of your email server
  - **`Port`**—The number of the port to use when sending an email message
- Methods
  - **`Send`**—Enables you to send an email message synchronously
  - **`SendAsync`**—Enables you to send an email message asynchronously
- Events
  - **`SendCompleted`**—Raised when an asynchronous send operation completes

If you know the members of a class, then you know everything that you can do with a class. For example, the `SmtpClient` class includes two properties named `Host` and `Port`, which enable you to specify the email server and port to use when sending an email message.

The `SmtpClient` class also includes two methods you can use to send an email: `Send()` and `SendAsync()`. The `Send` method blocks further program execution until the send operation is completed. The `SendAsync()` method, on the other hand, sends the email asynchronously. Unlike the `Send()` method, the `SendAsync()` method does not wait to check whether the send operation was successful.

Finally, the `SmtpClient` class includes an event named `SendCompleted`, which is raised when an asynchronous send operation completes. You can create an event handler for the `SendCompleted` event that displays a message when the email has been successfully sent.

The page in Listing 1.2 sends an email by using the `SmtpClient` class and calling its `Send()` method.

LISTING 1.2 `SendMail.aspx`

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Net.Mail" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">

    void Page_Load()
    {
        SmtpClient client = new SmtpClient();
        client.Host = "localhost";
        client.Port = 25;
        client.Send("steve@somewhere", "steve@superexpert.com",
           "Let's eat lunch!", "Lunch at the Steak House?");
```

LISTING 1.2 Continued

```
    }

</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Send Mail</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

    Email sent!

    </div>
    </form>
</body>
</html>
```

The page in Listing 1.2 calls the `SmtpClient Send()` method to send the email. The first parameter is the from: address; the second parameter is the to: address; the third parameter is the subject; and the final parameter is the body of the email.

WARNING

The page in Listing 1.2 sends the email by using the local SMTP Server. If your SMTP Server is not enabled, then you'll receive the error `An existing connection was forcibly closed by the remote host.` You can enable your local SMTP Server by opening Internet Information Services, right-clicking Default SMTP Virtual Server, and selecting Start.

Understanding Namespaces There are almost 13,000 classes in the .NET Framework. This is an overwhelming number. If Microsoft simply jumbled all the classes together, then you would never find anything. Fortunately, Microsoft divided the classes in the Framework into separate namespaces.

A *namespace* is simply a category. For example, all the classes related to working with the file system are located in the `System.IO` namespace. All the classes for working a Microsoft SQL Server database are located in the `System.Data.SqlClient` namespace.

Before you can use a class in a page, you must indicate the namespace associated with the class. There are multiple ways of doing this.

First, you can fully qualify a class name with its namespace. For example, because the `File` class is contained in the `System.IO` namespace, you can use the following statement to check whether a file exists:

```
System.IO.File.Exists("SomeFile.txt")
```

Specifying a namespace each and every time you use a class can quickly become tedious (it involves a lot of typing). A second option is to import a namespace.

You can add an `<%@ Import %>` directive to a page to import a particular namespace. In Listing 1.2, we imported the `System.Net.Mail` namespace because the `SmtpClient` is part of this namespace. The page in Listing 1.2 includes the following directive near the very top of the page:

```
<%@ Import Namespace="System.Net.Mail" %>
```

After you import a particular namespace, you can use all the classes in that namespace without qualifying the class names.

Finally, if you discover that you are using a namespace in multiple pages in your application, then you can configure all the pages in your application to recognize the namespace.

NOTE

A web configuration file is a special type of file that you can add to your application to configure your application. Be aware that the file is an XML file and, therefore, all the tags contained in the file are case sensitive. You can add a web configuration file to your application by selecting Web Site, Add New Item and selecting Web Configuration File. Chapter 28, "Configuring Applications," discusses web configuration files in detail.

If you add the web configuration file in Listing 1.3 to your application, then you do not need to import the `System.Net.Mail` namespace in a page to use the classes from this namespace. For example, if you include the `Web.config` file in your project, you can remove the `<%@ Import %>` directive from the page in Listing 1.2.

LISTING 1.3 `Web.Config`

```
<configuration>
    <system.web>
      <pages>
        <namespaces>
          <add namespace="System.Net.Mail"/>
        </namespaces>
      </pages>
    </system.web>
</configuration>
```

You don't have to import every namespace. The ASP.NET Framework gives you the most commonly used namespaces for free. These namespaces are as follows:

- `System`
- `System.Collections`
- `System.Collections.Specialized`
- `System.Configuration`
- `System.Text`
- `System.Text.RegularExpressions`
- `System.Web`
- `System.Web.Caching`
- `System.Web.SessionState`
- `System.Web.Security`
- `System.Web.Profile`
- `System.Web.UI`
- `System.Web.UI.WebControls`
- `System.Web.UI.WebControls.WebParts`
- `System.Web.UI.HTMLControls`

The default namespaces are listed inside the `pages` element in the root web configuration file located at the following path:

```
\WINDOWS\Microsoft.NET\Framework\v2.0.50727\CONFIG\Web.Config
```

Understanding Assemblies An assembly is the actual `.dll` file on your hard drive where the classes in the .NET Framework are stored. For example, all the classes contained in the ASP.NET Framework are located in an assembly named `System.Web.dll`.

More accurately, an assembly is the primary unit of deployment, security, and version control in the .NET Framework. Because an assembly can span multiple files, an assembly is often referred to as a "logical" dll.

There are two types of assemblies: private and shared. A private assembly can be used by only a single application. A shared assembly, on the other hand, can be used by all applications located on the same server.

Shared assemblies are located in the Global Assembly Cache (GAC). For example, the `System.Web.dll` assembly and all the other assemblies included with the .NET Framework are located in the Global Assembly Cache.

NOTE

The Global Assembly Cache is located physically in your computer's \WINDOWS\Assembly folder.

Before you can use a class contained in an assembly in your application, you must add a reference to the assembly. By default, an ASP.NET application references the most common assemblies contained in the Global Assembly Cache:

- `mscorlib.dll`
- `System.dll`
- `System.Configuration.dll`
- `System.Web.dll`
- `System.Data.dll`
- `System.Web.Services.dll`
- `System.Xml.dll`
- `System.Drawing.dll`
- `System.EnterpriseServices.dll`
- `System.Web.Mobile.dll`

In addition, websites built to target the .NET Framework 3.5 also reference the following assemblies:

- System.Web.Extensions
- System.Xml.Linq
- System.Data.DataSetExtensions

NOTE

You can target a website to work with the .NET Framework 2.0, .NET Framework 3.0, or .NET Framework 3.5. Within Visual Web Developer, select the menu option Website, Start Options and select the Build tab. You can select the framework to target from a dropdown list.

To use any particular class in the .NET Framework, you must do two things. First, your application must reference the assembly that contains the class. Second, your application must import the namespace associated with the class.

In most cases, you won't worry about referencing the necessary assembly because the most common assemblies are referenced automatically. However, if you need to use a specialized assembly, you need to add a reference explicitly to the assembly. For example, if you

need to interact with Active Directory by using the classes in the System.DirectoryServices namespace, then you will need to add a reference to the System.DirectoryServices.dll assembly to your application.

Each class entry in the .NET Framework SDK documentation lists the assembly and namespace associated with the class. For example, if you look up the `MessageQueue` class in the documentation, you'll discover that this class is located in the `System.Messaging` namespace located in the `System.Messaging.dll` assembly.

If you are using Visual Web Developer, you can add a reference to an assembly explicitly by selecting the menu option Web Site, Add Reference, and selecting the name of the assembly that you need to reference. For example, adding a reference to the `System.Messaging.dll` assembly results in the web configuration file in Listing 1.4 being added to your application.

LISTING 1.4 `Web.Config`

```
<configuration>
<system.web>
  <compilation>
  <assemblies>
  <add
    assembly="System.Messaging, Version=2.0.0.0,
    Culture=neutral, PublicKeyToken=B03F5F7F11D50A3A"/>
  </assemblies>
  </compilation>
</system.web>
</configuration>
```

If you prefer not to use Visual Web Developer, then you can add the reference to the `System.Messaging.dll` assembly by creating the file in Listing 1.4 by hand.

Understanding the Common Language Runtime

The second part of the .NET Framework is the Common Language Runtime (CLR). The Common Language Runtime is responsible for executing your application code.

When you write an application for the .NET Framework with a language such as C# or Visual Basic .NET, your source code is never compiled directly into machine code. Instead, the C# or Visual Basic compiler converts your code into a special language named MSIL (Microsoft Intermediate Language).

MSIL looks very much like an object-oriented assembly language. However, unlike a typical assembly language, it is not CPU specific. MSIL is a low-level and platform-independent language.

When your application actually executes, the MSIL code is "just-in-time" compiled into machine code by the JITTER (the Just-In-Time compiler). Normally, your entire application

is not compiled from MSIL into machine code. Instead, only the methods that are actually called during execution are compiled.

In reality, the .NET Framework understands only one language: MSIL. However, you can write applications using languages such as Visual Basic .NET and C# for the .NET Framework because the .NET Framework includes compilers for these languages that enable you to compile your code into MSIL.

You can write code for the .NET Framework using any one of dozens of different languages, including the following:

- Ada
- Apl
- Caml
- COBOL
- Eiffel
- Forth
- Fortran
- JavaScript
- Oberon
- PERL
- Pascal
- PHP
- Python
- RPG
- Scheme
- Small Talk

The vast majority of developers building ASP.NET applications write the applications in either C# or Visual Basic .NET. Many of the other .NET languages in the preceding list are academic experiments.

Once upon a time, if you wanted to become a developer, you concentrated on becoming proficient at a particular language. For example, you became a C++ programmer, a COBOL programmer, or a Visual Basic Programmer.

When it comes to the .NET Framework, however, knowing a particular language is not particularly important. The choice of which language to use when building a .NET application is largely a preference choice. If you like case-sensitivity and curly braces, then you should use the C# programming language. If you want to be lazy about casing and you don't like semicolons, then write your code with Visual Basic .NET.

All the real action in the .NET Framework happens in the Framework Class Library. If you want to become a good programmer using Microsoft technologies, you need to learn how to use the methods, properties, and events of the 13,000 classes included in the Framework. From the point of view of the .NET Framework, it doesn't matter whether you are using these classes from a Visual Basic .NET or C# application.

NOTE

All the code samples in this book were written in both C# and Visual Basic .NET. The VB.NET code samples are included on the CD that accompanies this book and at the Superexpert website (www.Superexpert.com).

Understanding ASP.NET Controls

ASP.NET controls are the heart of the ASP.NET Framework. An ASP.NET control is a .NET class that executes on the server and renders certain content to the browser.

For example, in the first ASP.NET page created at the beginning of this chapter, a Label control was used to display the current date and time. The ASP.NET framework includes over 70 controls, which enable you to do everything from displaying a list of database records to displaying a randomly rotating banner advertisement.

In this section, you are provided with an overview of the controls included in the ASP.NET Framework. You also learn how to handle events that are raised by controls and how to take advantage of View State.

Overview of ASP.NET Controls

The ASP.NET Framework contains over 70 controls. These controls can be divided into eight groups:

- **Standard Controls**—The standard controls enable you to render standard form elements such as buttons, input fields, and labels. We examine these controls in detail in the following chapter, "Using the Standard Controls."
- **Validation Controls**—The validation controls enable you to validate form data before you submit the data to the server. For example, you can use a `RequiredFieldValidator` control to check whether a user entered a value for a required input field. These controls are discussed in Chapter 3, "Using the Validation Controls."
- **Rich Controls**—The rich controls enable you to render things such as calendars, file upload buttons, rotating banner advertisements, and multi-step wizards. These controls are discussed in Chapter 4, "Using the Rich Controls."
- **Data Controls**—The data controls enable you to work with data such as database data. For example, you can use these controls to submit new records to a database table or display a list of database records. These controls are discussed in detail in Part III of this book, "Performing Data Access."

- **Navigation Controls**—The navigation controls enable you to display standard navigation elements such as menus, tree views, and bread crumb trails. These controls are discussed in Chapter 19, "Using the Navigation Controls."
- **Login Controls**—The login controls enable you to display login, change password, and registration forms. These controls are discussed in Chapter 22, "Using the Login Controls."
- **HTML Controls**—The HTML controls enable you to convert any HTML tag into a server-side control. We discuss this group of controls in the next section of this chapter.

With the exception of the HTML controls, you declare and use all the ASP.NET controls in a page in exactly the same way. For example, if you want to display a text input field in a page, then you can declare a `TextBox` control like this:

```
<asp:TextBox id="TextBox1" runat="Server" />
```

This control declaration looks like the declaration for an HTML tag. Remember, however, unlike an HTML tag, a control is a .NET class that executes on the server and not in the web browser.

When the `TextBox` control is rendered to the browser, it renders the following content:

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

The first part of the control declaration, the `asp:` prefix, indicates the namespace for the control. All the standard ASP.NET controls are contained in the `System.Web.UI.WebControls` namespace. The prefix `asp:` represents this namespace.

Next, the declaration contains the name of the control being declared. In this case, a `TextBox` control is being declared.

This declaration also includes an ID attribute. You use the ID to refer to the control in the page within your code. Every control must have a unique ID.

NOTE

You should always assign an ID attribute to every control even when you don't need to program against it. If you don't provide an ID attribute, then certain features of the ASP.NET Framework (such as two-way databinding) won't work.

The declaration also includes a `runat="Server"` attribute. This attribute marks the tag as representing a server-side control. If you neglect to include this attribute, then the `TextBox` tag would be passed, without being executed, to the browser. The browser would simply ignore the tag.

Finally, notice that the tag ends with a forward slash. The forward slash is shorthand for creating a closing `</asp:TextBox>` tag. You can, if you prefer, declare the `TextBox` control like this:

```
<asp:TextBox id="TextBox1" runat="server"></asp:TextBox>
```

In this case, the opening tag does not contain a forward slash and an explicit closing tag is included.

Understanding HTML Controls

You declare HTML controls in a different way than you declare standard ASP.NET controls. The ASP.NET Framework enables you to take any HTML tag (real or imaginary) and add a `runat="server"` attribute to the tag. The `runat="server"` attribute converts the HTML tag into a server-side ASP.NET control.

For example, the page in Listing 1.5 contains a `<span>` tag, which has been converted into an ASP.NET control.

LISTING 1.5 `HtmlControls.aspx`

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">

    void Page_Load()
    {
        spanNow.InnerText = DateTime.Now.ToString("T");
    }

</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>HTML Controls</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

    At the tone, the time will be:
    <span id="spanNow" runat="server" />

    </div>
    </form>
</body>
</html>
```

Notice that the `<span>` tag in Listing 1.5 looks just like a normal HTML `<span>` tag except for the addition of the `runat="server"` attribute.

Because the tag in Listing 1.5 is a server-side HTML control, you can program against it. In Listing 1.5, the current date and time are assigned to the tag in the Page_Load() method.

The HTML controls are included in the ASP.NET Framework to make it easier to convert existing HTML pages to use the ASP.NET Framework. I rarely use the HTML controls in this book because, in general, the standard ASP.NET controls provide all the same functionality and more.

Understanding and Handling Control Events

The majority of the ASP.NET controls support one or more events. For example, the ASP.NET Button control supports the Click event. The Click event is raised on the server after you click the button rendered by the Button control in the browser.

The page in Listing 1.6 illustrates how you can write code that executes when a user clicks the button rendered by the Button control (in other words, it illustrates how you can create a Click event handler).

LISTING 1.6 ShowButtonClick.aspx

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">

    protected void btnSubmit_Click(object sender, EventArgs e)
    {
        Label1.Text = "Thanks!";
    }
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Show Button Click</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

    <asp:Button
        id="btnSubmit"
        Text="Click Here"
        OnClick="btnSubmit_Click"
        Runat="server" />

    <br /><br />
```

LISTING 1.6 Continued

```
    <asp:Label
        id="Label1"
        Runat="server" />

    </div>
    </form>
</body>
</html>
```

Notice that the `Button` control in Listing 1.6 includes an `OnClick` attribute. This attribute points to a subroutine named `btnSubmit_Click()`. The `btnSubmit_Click()` subroutine is the handler for the `Button Click` event. This subroutine executes whenever you click the button (see Figure 1.5).

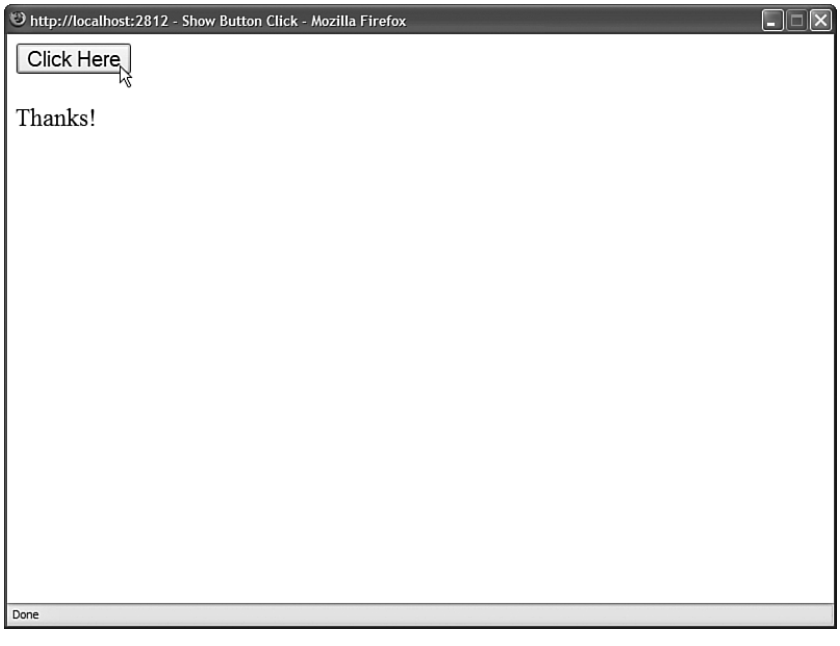

FIGURE 1.5 Raising a `Click` event.

You can add an event handler automatically to a control in multiple ways when using Visual Web Developer. In Source view, add a handler by selecting a control from the top-left drop-down list and selecting an event from the top-right drop-down list. The event handler code is added to the page automatically (see Figure 1.6).

In Design view, you can double-click a control to add a handler for the control's default event. Double-clicking a control switches you to Source view and adds the event handler.

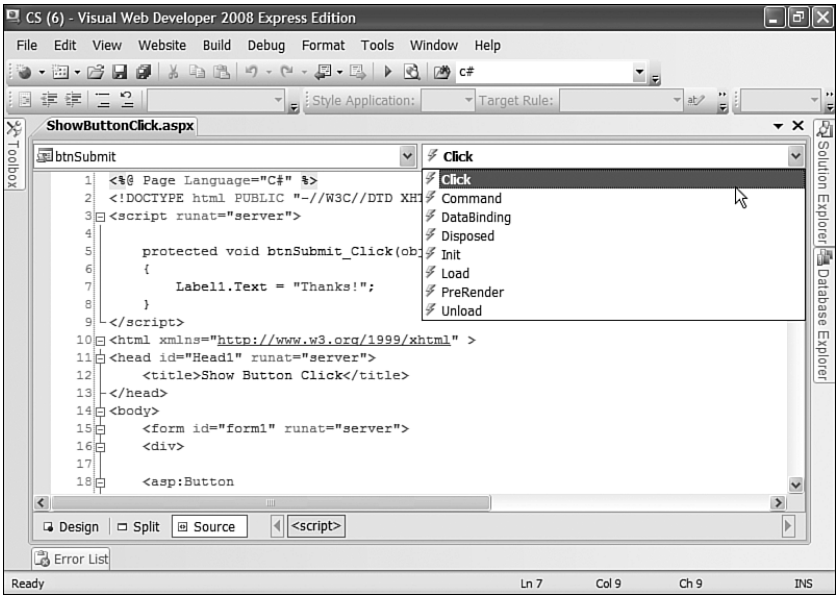

FIGURE 1.6 Adding an event handler from Source view.

Finally, from Design view, after selecting a control on the designer surface you can add an event handler from the Properties window by clicking the Events button (the lightning bolt) and double-clicking next to the name of any of the events (see Figure 1.7).

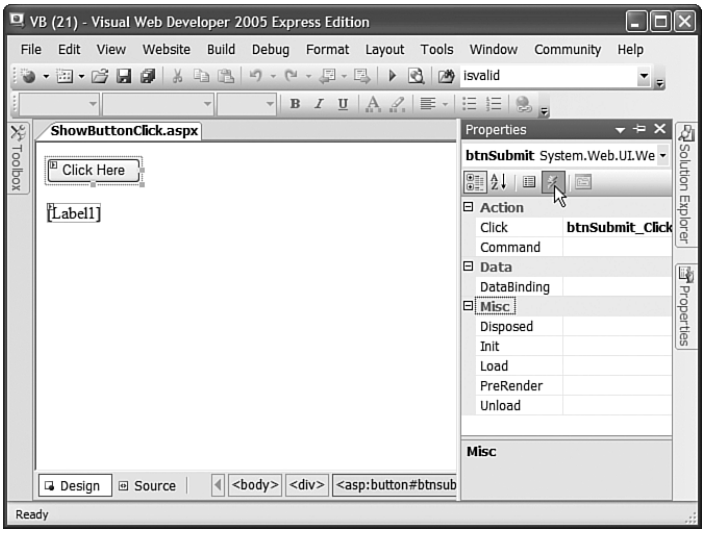

FIGURE 1.7 Adding an event handler from the Properties window.

It is important to understand that all ASP.NET control events happen on the server. For example, the `Click` event is not raised when you actually click a button. The `Click` event is not raised until the page containing the `Button` control is posted back to the server.

The ASP.NET Framework is a server-side web application framework. The .NET Framework code that you write executes on the server and not within the web browser. From the perspective of ASP.NET, nothing happens until the page is posted back to the server and can execute within the context of the .NET Framework.

Notice that two parameters are passed to the `btnSubmit_Click()` handler in Listing 1.6. All event handlers for ASP.NET controls have the same general signature.

The first parameter, the object parameter named `sender`, represents the control that raised the event. In other words, it represents the `Button` control which you clicked.

You can wire multiple controls in a page to the same event handler and use this first parameter to determine the particular control that raised the event. For example, the page in Listing 1.7 includes two `Button` controls. When you click either `Button` control, the text displayed by the `Button` control is updated (see Figure 1.8).

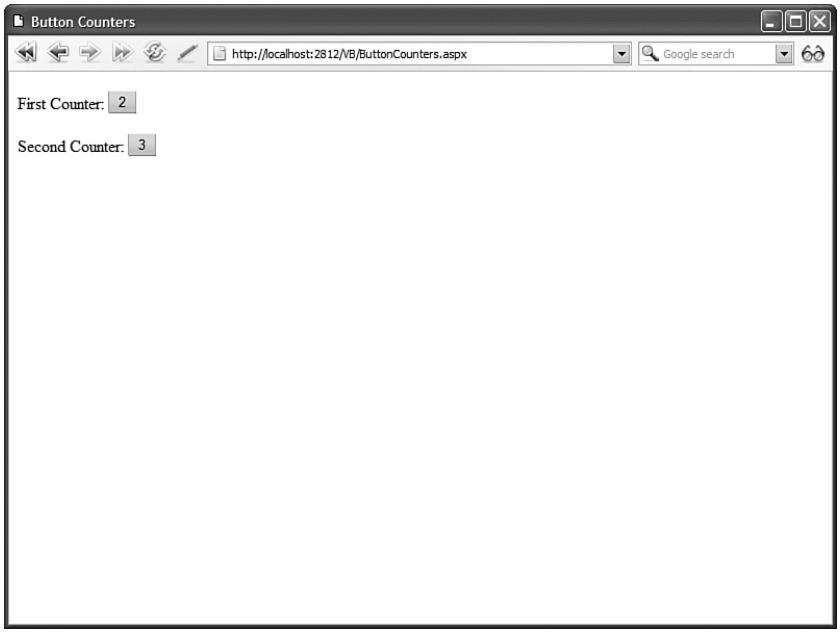

FIGURE 1.8 Handling two `Button` controls with one event handler.

LISTING 1.7 ButtonCounters.aspx

```
 <%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">

    protected void Button_Click(object sender, EventArgs e)
    {
        Button btn = (Button)sender;
        btn.Text = (Int32.Parse(btn.Text) + 1).ToString();
    }
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Button Counters</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

    First Counter:
    <asp:Button
        id="Button1"
        Text="0"
        OnClick="Button_Click"
        Runat="server" />

    <br /><br />

    Second Counter:
    <asp:Button
        id="Button2"
        Text="0"
        OnClick="Button_Click"
        Runat="server" />

    </div>
    </form>
</body>
</html>
```

The second parameter passed to the `Click` event handler, the `EventArgs` parameter named `e`, represents any additional event information associated with the event. No additional event information is associated with clicking a button, so this second parameter does not represent anything useful in either Listing 1.6 or Listing 1.7.

When you click an `ImageButton` control instead of a `Button` control, on the other hand, additional event information is passed to the event handler. When you click an `ImageButton` control, the X and Y coordinates of where you clicked are passed to the handler.

The page in Listing 1.8 contains an `ImageButton` control that displays a picture. When you click the picture, the X and Y coordinates of the spot you clicked are displayed in a `Label` control (see Figure 1.9).

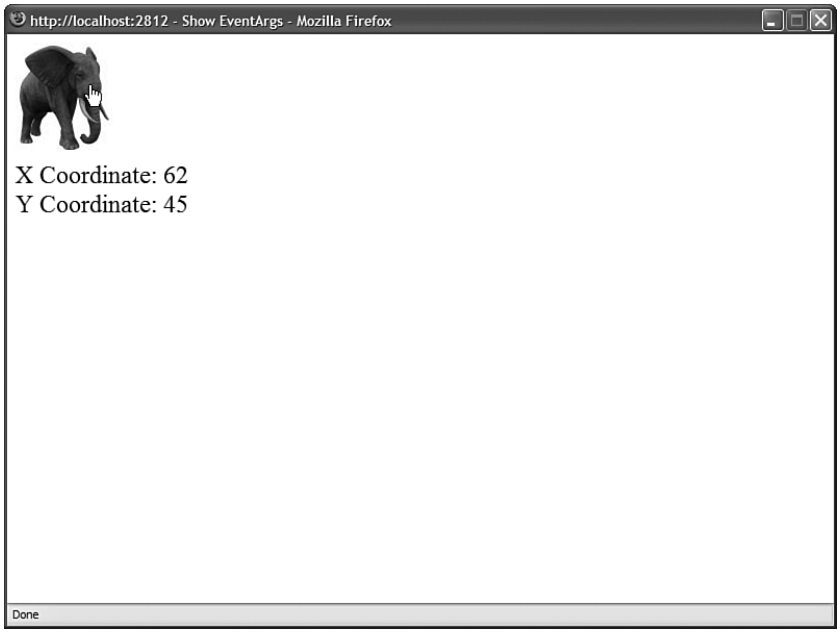

FIGURE 1.9 Clicking an `ImageButton`.

LISTING 1.8 `ShowEventArgs.aspx`

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">

    protected void btnElephant_Click(object sender, ImageClickEventArgs e)
    {
        lblX.Text = e.X.ToString();
        lblY.Text = e.Y.ToString();
    }
</script>
```

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Show EventArgs</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

    <asp:ImageButton
        id="btnElephant"
        ImageUrl="Elephant.jpg"
        Runat="server" OnClick="btnElephant_Click" />

    <br />
    X Coordinate:
    <asp:Label
        id="lblX"
        Runat="server" />
    <br />
    Y Coordinate:
    <asp:Label
        id="lblY"
        Runat="server" />

    </div>
    </form>
</body>
</html>
```

Notice that the second parameter passed to the `btnElephant_Click()` method is an `ImageClickEventArgs` parameter. Whenever the second parameter is not the default `EventArgs` parameter, you know that additional event information is being passed to the handler.

Understanding View State

The HTTP protocol, the fundamental protocol of the World Wide Web, is a stateless protocol. Each time you request a web page from a website, from the website's perspective, you are a completely new person.

The ASP.NET Framework, however, manages to transcend this limitation of the HTTP protocol. For example, if you assign a value to a `Label` control's `Text` property, the `Label` control retains this value across multiple page requests.

Consider the page in Listing 1.9. This page contains a `Button` control and a `Label` control. Each time you click the `Button` control, the value displayed by the `Label` control is incremented by 1 (see Figure 1.10). How does the `Label` control preserve its value across postbacks to the web server?

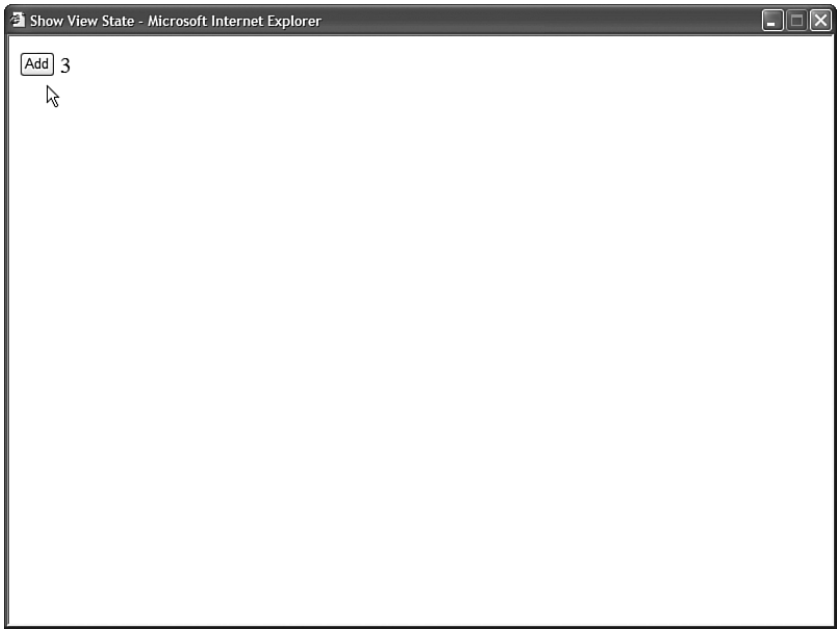

FIGURE 1.10 Preserving state between postbacks.

LISTING 1.9 `ShowViewState.aspx`

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">

    protected void btnAdd_Click(object sender, EventArgs e)
    {
        lblCounter.Text = (Int32.Parse(lblCounter.Text) + 1).ToString();
    }
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Show View State</title>
</head>
<body>
    <form id="form1" runat="server">
```

```
    <div>

    <asp:Button
        id="btnAdd"
        Text="Add"
        OnClick="btnAdd_Click"
        Runat="server" />

    <asp:Label
        id="lblCounter"
        Text="0"
        Runat="server" />

    </div>
    </form>
</body>
</html>
```

The ASP.NET Framework uses a trick called View State. If you open the page in Listing 1.9 in your browser and select View Source, you'll notice that the page includes a hidden form field named `__VIEWSTATE` that looks like this:

```
<input type="hidden" name="__VIEWSTATE" id="__
  VIEWSTATE" value="/wEPDwUKLTc2ODE1OTYxNw9kFgICBA9kFgIC
  Aw8PFgIeBFRleHQFATFkZGT3tMnThg9KZpGak55p367vfInj1w==" />
```

This hidden form field contains the value of the `Label` control's `Text` property (and the values of any other control properties that are stored in View State). When the page is posted back to the server, the ASP.NET Framework rips apart this string and re-creates the values of all the properties stored in View State. In this way, the ASP.NET Framework preserves the state of control properties across postbacks to the web server.

By default, View State is enabled for every control in the ASP.NET Framework. If you change the background color of a Calendar control, the new background color is remembered across postbacks. If you change the selected item in a `DropDownList`, the selected item is remembered across postbacks. The values of these properties are automatically stored in View State.

View State is a good thing, but sometimes it can be too much of a good thing. The `__VIEWSTATE` hidden form field can become very large. Stuffing too much data into View State can slow down the rendering of a page because the contents of the hidden field must be pushed back and forth between the web server and web browser.

You can determine how much View State each control contained in a page is consuming by enabling tracing for a page (see Figure 1.11). The page in Listing 1.10 includes a `Trace="true"` attribute in its `<%@ Page %>` directive, which enables tracing.

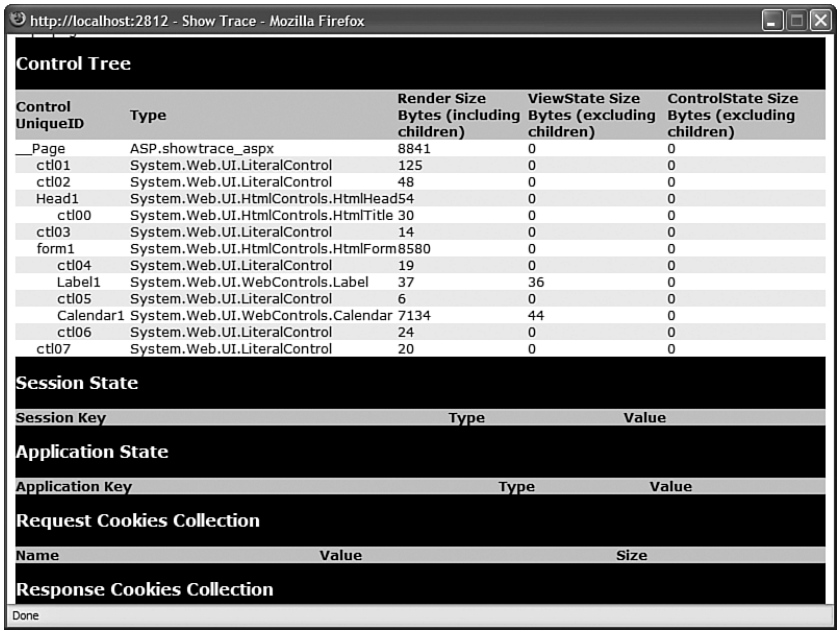

FIGURE 1.11 Viewing View State size for each control.

LISTING 1.10 ShowTrace.aspx

```
<%@ Page Language="C#" Trace="true" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">

    void Page_Load()
    {
        Label1.Text = "Hello World!";
        Calendar1.TodaysDate = DateTime.Now;
    }

</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Show Trace</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

    <asp:Label
```

```
        id="Label1"
        Runat="server" />
    <asp:Calendar
        id="Calendar1"
        TodayDayStyle-BackColor="Yellow"
        Runat="server" />

    </div>
    </form>
</body>
</html>
```

When you open the page in Listing 1.10, additional information about the page is appended to the bottom of the page. The Control Tree section displays the amount of View State used by each ASP.NET control contained in the page.

Every ASP.NET control includes a property named `EnableViewState`. If you set this property to the value `False`, then View State is disabled for the control. In that case, the values of the control properties are not remembered across postbacks to the server.

For example, the page in Listing 1.11 contains two `Label` controls and a `Button` control. The first `Label` has View State disabled and the second `Label` has View State enabled. When you click the button, only the value of the second `Label` control is incremented past 1.

LISTING 1.11 `DisableViewState.aspx`

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">

    protected void btnAdd_Click(object sender, EventArgs e)
    {
        Label1.Text = (Int32.Parse(Label1.Text) + 1).ToString();
        Label2.Text = (Int32.Parse(Label2.Text) + 1).ToString();
    }
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Disable View State</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
```

LISTING 1.11 Continued

```
    Label 1:
    <asp:Label
        id="Label1"
        EnableViewState="false"
        Text="0"
        Runat="server" />

    <br />

    Label 2:
    <asp:Label
        id="Label2"
        Text="0"
        Runat="server" />

    <br /><br />

    <asp:Button
        id="btnAdd"
        Text="Add"
        OnClick="btnAdd_Click"
        Runat="server" />

    </div>
    </form>
</body>
</html>
```

Sometimes, you might want to disable View State even when you aren't concerned with the size of the `__VIEWSTATE` hidden form field. For example, if you are using a `Label` control to display a form validation error message, you might want to start from scratch each time the page is submitted. In that case, simply disable View State for the `Label` control.

NOTE

The ASP.NET Framework version 2.0 introduced a new feature called Control State. Control State is similar to View State except that it is used to preserve only critical state information. For example, the `GridView` control uses Control State to store the selected row. Even if you disable View State, the `GridView` control remembers which row is selected.

Understanding ASP.NET Pages

This section examines ASP.NET pages in more detail. You learn about dynamic compilation and code-behind files. We also discuss the events supported by the `Page` class.

Understanding Dynamic Compilation

Strangely enough, when you create an ASP.NET page, you are actually creating the source code for a .NET class. You are creating a new instance of the `System.Web.UI.Page` class. The entire contents of an ASP.NET page, including all script and HTML content, are compiled into a .NET class.

When you request an ASP.NET page, the ASP.NET Framework checks for a .NET class that corresponds to the page. If a corresponding class does not exist, the Framework automatically compiles the page into a new class and stores the compiled class (the assembly) in the Temporary ASP.NET Files folder located at the following path:

```
\WINDOWS\Microsoft.NET\Framework\v2.0.50727\Temporary ASP.NET Files
```

The next time anyone requests the same page in the future, the page is not compiled again. The previously compiled class is executed and the results are returned to the browser.

Even if you unplug your web server, move to Borneo for three years, and start up your web server again, the next time someone requests the same page, the page does not need to be re-compiled. The compiled class is preserved in the Temporary ASP.NET Files folder until the source code for your application is modified.

When the class is added to the Temporary ASP.NET Files folder, a file dependency is created between the class and the original ASP.NET page. If the ASP.NET page is modified in any way, the corresponding .NET class is automatically deleted. The next time someone requests the page, the Framework automatically compiles the modified page source into a new .NET class.

This process is called *dynamic compilation*. Dynamic compilation enables ASP.NET applications to support thousands of simultaneous users. Unlike an ASP Classic page, for example, an ASP.NET page does not need to be parsed and compiled each and every time it is requested. An ASP.NET page is compiled only when an application is modified.

NOTE

You can precompile an entire ASP.NET application by using the `aspnet_compiler.exe` command-line tool. If you precompile an application, users don't experience the compilation delay resulting from the first page request.

In case you are curious, I've included the source code for the class that corresponds to the `FirstPage.aspx` page in Listing 1.12 (I've cleaned up the code and made it shorter to save space). I copied this file from the Temporary ASP.NET Files folder after enabling debugging for the application.

NOTE

You can disable dynamic compilation for a single page, the pages in a folder, or an entire website with the `CompilationMode` attribute. When the `CompilationMode` attribute is used with the `<%@ Page %>` directive, it enables you to disable dynamic compilation for a single page. When the `compilationMode` attribute is used with the `pages` element in a web configuration file, it enables you to disable dynamic compilation for an entire folder or application.

Disabling compilation is useful when you have thousands of pages in a website and you don't want to load too many assemblies into memory. When the `CompilationMode` attribute is set to the value `Never`, the page is never compiled and an assembly is never generated for the page. The page is interpreted at runtime.

You cannot disable compilation for pages that include server-side code. In particular, a no compile page cannot include a server-side `<script>...</script>` block. On the other hand, a no compile page can contain ASP.NET controls and databinding expressions.

LISTING 1.12 `FirstPage.aspx` Source

```
namespace ASP
{
    using System.Web.Profile;
    using System.Text.RegularExpressions;
    using System.Web.Caching;
    using System.Configuration;
    using System.Collections.Specialized;
    using System.Web.UI.WebControls.WebParts;
    using System.Web.UI.HtmlControls;
    using System.Web.UI.WebControls;
    using System.Web.UI;
    using System.Collections;
    using System;
    using System.Web.Security;
    using System.Web;
    using System.Web.SessionState;
    using System.Text;


    [System.Runtime.CompilerServices.CompilerGlobalScopeAttribute()]
    public class firstpage_aspx : global::System.Web.UI.Page,
    ➥System.Web.SessionState.IRequiresSessionState, System.Web.IHttpHandler
    {
        protected global::System.Web.UI.WebControls.Label lblServerTime;
        protected global::System.Web.UI.HtmlControls.HtmlForm form1;
```

```
private static bool @__initialized;
private static object @__fileDependencies;

void Page_Load()
{
    lblServerTime.Text = DateTime.Now.ToString();
}

public firstpage_aspx()
{
    string[] dependencies;
    ((global::System.Web.UI.Page)(this)).AppRelativeVirtualPath =
    ➥"~/FirstPage.aspx";
    if ((global::ASP.firstpage_aspx.@__initialized == false))
    {
        dependencies = new string[1];
        dependencies[0] = "~/FirstPage.aspx";
        global::ASP.firstpage_aspx.@__fileDependencies =
        ➥this.GetWrappedFileDependencies(dependencies);
        global::ASP.firstpage_aspx.@__initialized = true;
    }
    this.Server.ScriptTimeout = 30000000;
}

protected System.Web.Profile.DefaultProfile Profile
{
    get
    {
        return ((System.Web.Profile.DefaultProfile)(this.Context.Profile));
    }
}

protected System.Web.HttpApplication ApplicationInstance
{
    get
    {
        return ((System.Web.HttpApplication)(this.Context.
        ➥ApplicationInstance));
    }
}

private global::System.Web.UI.WebControls.Label
➥@__BuildControllblServerTime()
{
...code...
}
```

LISTING 1.12 Continued

```
        private global::System.Web.UI.HtmlControls.HtmlForm @__BuildControlform1()
        {
        ...code...
        }

        private void @__BuildControlTree(firstpage_aspx @__ctrl)
        {
        ...code...
        }

        protected override void FrameworkInitialize()
        {
            base.FrameworkInitialize();
            this.@__BuildControlTree(this);
            this.AddWrappedFileDependencies(global::ASP.firstpage_aspx.
           ➥@__fileDependencies);
            this.Request.ValidateInput();
        }

        public override int GetTypeHashCode()
        {
            return 579569163;
        }

        public override void ProcessRequest(System.Web.HttpContext context)
        {
            base.ProcessRequest(context);
        }
    }
}
```

The class in Listing 1.12 inherits from the `System.Web.UI.Page` class. The `ProcessRequest()` method is called by the ASP.NET Framework when the page is displayed. This method builds the page's control tree, which is the subject of the next section.

Understanding Control Trees

In the previous section, you learned that an ASP.NET page is really the source code for a .NET class. Alternatively, you can think of an ASP.NET page as a bag of controls. More accurately, because some controls might contain child controls, you can think of an ASP.NET page as a control tree.

For example, the page in Listing 1.13 contains a `DropDownList` control and a `Button` control. Furthermore, because the `<%@ Page %>` directive has the `Trace="true"` attribute, tracing is enabled for the page.

LISTING 1.13 ShowControlTree.aspx

```
<%@ Page Language="C#" Trace="true" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Show Control Tree</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

    <asp:DropDownList
        id="DropDownList1"
        Runat="server">
        <asp:ListItem Text="Oranges" />
        <asp:ListItem Text="Apples" />
    </asp:DropDownList>

    <asp:Button
        id="Button1"
        Text="Submit"
        Runat="server" />

    </div>
    </form>
</body>
</html>
```

When you open the page in Listing 1.12 in your browser, you can see the control tree for the page appended to the bottom of the page. It looks like this:

```
__Page ASP.showcontroltree_aspx
    ctl02 System.Web.UI.LiteralControl
    ctl00 System.Web.UI.HtmlControls.HtmlHead
        ctl01 System.Web.UI.HtmlControls.HtmlTitle
    ctl03 System.Web.UI.LiteralControl
    form1 System.Web.UI.HtmlControls.HtmlForm
        ctl04 System.Web.UI.LiteralControl
        DropDownList1 System.Web.UI.WebControls.DropDownList
        ctl05 System.Web.UI.LiteralControl
        Button1 System.Web.UI.WebControls.Button
        ctl06 System.Web.UI.LiteralControl
    ctl07
```

The root node in the control tree is the page itself. The page has an ID of `__Page`. The page class contains all the other controls in its child controls collection.

The control tree also contains an instance of the `HtmlForm` class named `form1`. This control is the server-side form tag contained in the page. It contains all the other form controls—the `DropDownList` and `Button` controls—as child controls.

Notice that there are several `LiteralControl` controls interspersed between the other controls in the control tree. What are these controls?

Remember that everything in an ASP.NET page is converted into a .NET class, including any HTML or plain text content in a page. The `LiteralControl` class represents the HTML content in the page (including any carriage returns between tags).

NOTE

Normally, you refer to a control in a page by its ID. However, there are situations in which this is not possible. In those cases, you can use the `FindControl()` method of the Control class to retrieve a control with a particular ID. The `FindControl()` method is similar to the JavaScript `getElementById()` method.

Using Code-Behind Pages

The ASP.NET Framework (and Visual Web Developer) enables you to create two different types of ASP.NET pages. You can create both single-file and two-file ASP.NET pages.

All the code samples in this book are written as single-file ASP.NET pages. In a single-file ASP.NET page, a single file contains both the page code and page controls. The page code is contained in a `<script runat="server">` tag.

As an alternative to a single-file ASP.NET page, you can create a two-file ASP.NET page. A two-file ASP.NET page is normally referred to as a *code-behind* page. In a code-behind page, the page code is contained in a separate file.

NOTE

Code-behind pages work in a different way after the ASP.NET 2.0 Framework than they did in the ASP.NET 1.x Framework. In ASP.NET 1.x, the two halves of a code-behind page were related by inheritance. After the ASP.NET 2.0 Framework, the two halves of a code-behind page are related by a combination of partial classes and inheritance.

For example, Listing 1.14 and Listing 1.15 contain the two halves of a code-behind page.

VISUAL WEB DEVELOPER NOTE

When using Visual Web Developer, you create a code-behind page by selecting Web Site, Add New Item, selecting the Web Form Item, and checking the Place Code in Separate File check box before adding the page.

LISTING 1.14 FirstPageCodeBehind.aspx

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="FirstPageCodeBehind.
➥aspx.cs" Inherits="FirstPageCodeBehind" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>First Page Code-Behind</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

    <asp:Button
        id="Button1"
        Text="Click Here"
        OnClick="Button1_Click"
        Runat="server" />

    <br /><br />

    <asp:Label
        id="Label1"
        Runat="server" />

    </div>
    </form>
</body>
</html>
```

LISTING 1.15 FirstPageCodeBehind.aspx.cs

```
using System;
using System.Data;
using System.Configuration;
using System.Collections;
using System.Web;
```

LISTING 1.15 Continued

```
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;

public partial class FirstPageCodeBehind : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Label1.Text = "Click the Button";
    }

    protected void Button1_Click(object sender, EventArgs e)
    {
        Label1.Text = "Thanks!";
    }
}
```

The page in Listing 1.14 is called the *presentation page*. It contains a `Button` control and a `Label` control. However, the page does not contain any code. All the code is contained in the code-behind file.

VISUAL WEB DEVELOPER NOTE

You can flip to the code-behind file for a page by right-clicking a page and selecting View Code.

The code-behind file in Listing 1.15 contains the `Page_Load()` and `Button1_Click()` handlers. The code-behind file in Listing 1.15 does not contain any controls.

Notice that the page in Listing 1.14 includes both a `CodeFile` and `Inherits` attribute in its `<%@ Page %>` directive. These attributes link the page to its code-behind file.

How Code-Behind Works: The Ugly Details In the previous version of the ASP.NET Framework (ASP.NET 1.x), two classes were generated by a code-behind page. One class corresponded to the presentation page and one class corresponded to the code-behind file. These classes were related to one another through class inheritance. The presentation page class inherited from the code-behind file class.

The problem with this method of associating presentation pages with their code-behind files was that it was very brittle. Inheritance is a one-way relationship. Anything that is true of the mother is true of the daughter, but not the other way around. Any control that

you declared in the presentation page was required to be declared in the code-behind file. Furthermore, the control had to be declared with exactly the same ID. Otherwise, the inheritance relationship would be broken and events raised by a control could not be handled in the code-behind file.

In the beta version of ASP.NET 2.0, a completely different method of associating presentation pages with their code-behind files was used. This new method was far less brittle. The two halves of a code-behind page were no longer related through inheritance, but through a new technology supported by the .NET 2.0 Framework called *partial classes*.

NOTE

Partial classes are discussed in Chapter 15, "Building Components."

Partial classes enable you to declare a class in more than one physical file. When the class gets compiled, one class is generated from all the partial classes. Any members of one partial class—including any private fields, methods, and properties—are accessible to any other partial classes of the same class. This makes sense because partial classes are combined eventually to create one final class.

The advantage of using partial classes is that you don't need to worry about declaring a control in both the presentation page and code-behind file. Anything that you declare in the presentation page is available automatically in the code-behind file, and anything you declare in the code-behind file is available automatically in the presentation page.

The beta version of the ASP.NET 2.0 Framework used partial classes to relate a presentation page with its code-behind file. However, certain advanced features of the ASP.NET 1.x Framework were not compatible with using partial classes. To support these advanced features, a more complex method of associating presentation pages with code-behind files is used in the final release of the ASP.NET 2.0 Framework.

NOTE

The ASP.NET 1.x Framework enabled you to create a custom base `Page` class and inherit every ASP.NET page in an application from the custom `Page` class. Relating pages and code-behind files with partial classes conflicted with inheriting from a custom base `Page` class. In the final release of the ASP.NET 2.0 Framework, you can once again create custom base `Page` classes. For a sample of a custom base `Page` class, see the final section of Chapter 5, "Designing Websites with Master Pages."

The final release of the ASP.NET 2.0 Framework uses a combination of inheritance and partial classes to relate presentation pages and code-behind files. The ASP.NET 2.0 Framework generates three classes whenever you create a code-behind page.

The first two classes correspond to the presentation page. For example, when you create the `FirstPageCodeBehind.aspx` page, the following two classes are generated automatically in the Temporary ASP.NET Files folder:

```
public partial class FirstPageCodeBehind
{
    protected System.Web.UI.WebControls.Button Button1;
    protected System.Web.UI.WebControls.Label Label1;

    ... additional code ...
}

public class firstpagecodebehind_aspx : FirstPageCodeBehind
{
    ... additional code ...
}
```

A third class is generated that corresponds to the code-behind file. Corresponding to the `FirstPageCodeBehind.aspx.cs` file, the following class is generated:

```
public partial class FirstPageCodeBehind : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Label1.Text = "Click the Button";
    }

    protected void Button1_Click(object sender, EventArgs e)
    {
        Label1.Text = "Thanks!";
    }
}
```

The `firstpagecodebehind_aspx` class is executed when the `FirstPageCodeBehind.aspx` page is requested from a browser. This class inherits from the `FirstPageCodeBehind` class. The `FirstPageCodeBehind` class is a partial class. It gets generated twice: once by the presentation page and once by the code-behind file.

The final release of the ASP.NET 2.0 Framework uses a combination of partial classes and inheritance to relate presentation pages and code-behind files. Because the page and code-behind classes are partial classes, unlike the previous version of ASP.NET, you no longer need to declare controls in both the presentation and code-behind page. Any control declared in the presentation page is accessible in the code-behind file automatically. Because the page class inherits from the code-behind class, the ASP.NET 2.0 Framework continues to support advanced features of the ASP.NET 1.x Framework such as custom base `Page` classes.

Deciding Between Single-File and Code-Behind Pages So, when should you use single-file ASP.NET pages and when should you use code-behind pages? This decision is a preference choice. There are intense arguments over this topic contained in blogs spread across the Internet.

I've heard it argued that code-behind pages are superior to single-file pages because code-behind pages enable you to more cleanly separate your user interface from your application logic. The problem with this argument is that the normal justification for separating your user interface from your application logic is code reuse. Building code-behind pages really doesn't promote code reuse. A better way to reuse application logic across multiple pages is to build separate component libraries. (Part IV of this book explores this topic.)

My personal preference is to build ASP.NET applications using single-file ASP.NET pages because this approach requires managing fewer files. However, I've built many applications using the code-behind model (such as some of the ASP.NET Starter Kits) without suffering dire consequences.

NOTE

The first versions of Visual Studio did not support building single-file ASP.NET pages. If you wanted to create single-file ASP.NET pages in the previous version of ASP.NET, you had to use an alternate development environment such as Web Matrix or Notepad.

Handling Page Events

Whenever you request an ASP.NET page, a particular set of events is raised in a particular sequence. This sequence of events is called the *page execution lifecycle*.

For example, we have already used the `Page Load` event in previous code samples in this chapter. You normally use the `Page Load` event to initialize the properties of controls contained in a page. However, the `Page Load` event is only one event supported by the `Page` class.

Here is the sequence of events that are raised whenever you request a page:

1. `PreInit`
2. `Init`
3. `InitComplete`
4. `PreLoad`
5. `Load`
6. `LoadComplete`
7. `PreRender`
8. `PreRenderComplete`
9. `SaveStateComplete`
10. `Unload`

Why so many events? Different things happen and different information is available at different stages in the page execution lifecycle.

For example, View State is not loaded until after the `InitComplete` event. Data posted to the server from a form control, such as a `TextBox` control, is also not available until after this event.

Ninety-nine percent of the time, you won't handle any of these events except for the `Load` and the `PreRender` events. The difference between these two events is that the `Load` event happens before any control events and the `PreRender` event happens after any control events.

The page in Listing 1.16 illustrates the difference between the `Load` and `PreRender` events. The page contains three event handlers: one for the `Load` event, one for the `Button Click` event, and one for the `PreRender` event. Each handler adds a message to a Label control (see Figure 1.12).

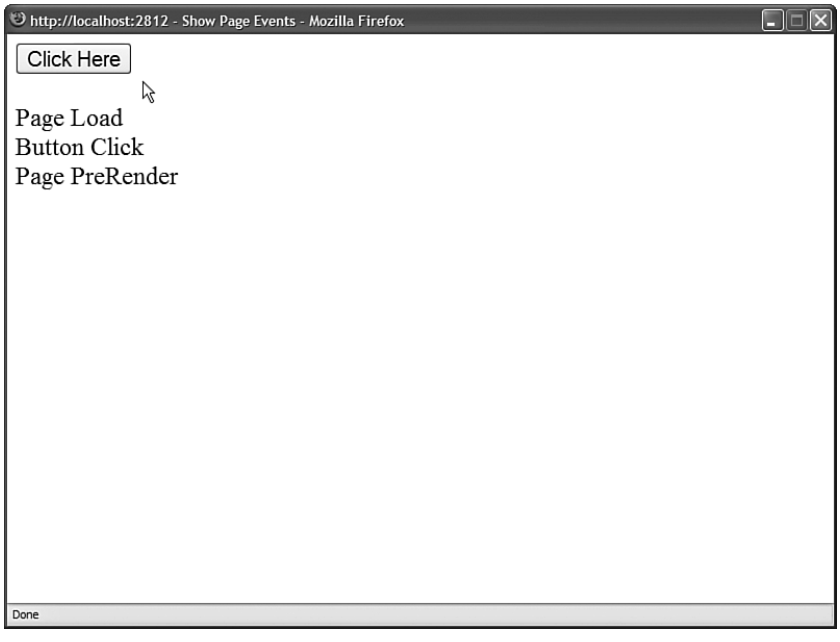

FIGURE 1.12 Viewing the sequence of page events.

LISTING 1.16 `ShowPageEvents.aspx`

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">

    void Page_Load(object sender, EventArgs e)
    {
        Label1.Text = "Page Load";
    }

    void Button1_Click(object sender, EventArgs e)
    {
```

```
        Label1.Text += "<br />Button Click";
    }

    void Page_PreRender()
    {
        Label1.Text += "<br />Page PreRender";
    }
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Show Page Events</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

    <asp:Button
        id="Button1"
        Text="Click Here"
        OnClick="Button1_Click"
        Runat="server" />

    <br /><br />

    <asp:Label
        id="Label1"
        Runat="server" />

    </div>
    </form>
</body>
</html>
```

When you click the `Button` control, the `Click` event does not happen on the server until after the `Load` event and before the `PreRender` event.

The other thing you should notice about the page in Listing 1.16 is the way the event handlers are wired to the `Page` events. ASP.NET pages support a feature named `AutoEventWireUp`, which is enabled by default. If you name a subroutine `Page_Load()`, the subroutine automatically handles the `Page Load` event; if you name a subroutine `Page_PreRender()`, the subroutine automatically handles the `Page PreRender` event, and so on.

WARNING

`AutoEventWireUp` does not work for every page event. For example, it does not work for the `Page_InitComplete()` event.

Using the `Page.IsPostBack` Property

The `Page` class includes a property called the `IsPostBack` property, which you can use to detect whether the page has already been posted back to the server.

Because of View State, when you initialize a control property, you do not want to initialize the property every time a page loads. Because View State saves the state of control properties across page posts, you typically initialize a control property only once, when the page first loads.

In fact, many controls don't work correctly if you re-initialize the properties of the control with each page load. In these cases, you must use the `IsPostBack` property to detect whether or not the page has been posted.

The page in Listing 1.17 illustrates how you can use the `Page.IsPostBack` property when adding items to a `DropDownList` control.

LISTING 1.17 `ShowIsPostBack.aspx`

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">

    void Page_Load()
    {
        if (!Page.IsPostBack)
        {
            // Create collection of items
            ArrayList items = new ArrayList();
            items.Add("Apples");
            items.Add("Oranges");

            // Bind to DropDownList
            DropDownList1.DataSource = items;
            DropDownList1.DataBind();
        }
    }

    protected void Button1_Click(object sender, EventArgs e)
    {
```

```
        Label1.Text = DropDownList1.SelectedItem.Text;
    }
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Show IsPostBack</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

    <asp:DropDownList
        id="DropDownList1"
        Runat="server" />

    <asp:Button
        id="Button1"
        Text="Select"
        OnClick="Button1_Click"
        Runat="server" />

    <br /><br />

    You selected:
    <asp:Label
        id="Label1"
        Runat="server" />

    </div>
    </form>
</body>
</html>
```

In Listing 1.17, the code in the `Page_Load()` event handler executes only once when the page first loads. When you post the page again, the `IsPostBack` property returns `True` and the code contained in the `Page_Load()` handler is skipped.

If you remove the `IsPostBack` check from the `Page_Load()` method, then you get a strange result. The `DropDownList` always displays its first item as the selected item. Binding the `DropDownList` to a collection of items re-initializes the `DropDownList` control. Therefore, you want to bind the `DropDownList` control only once, when the page first loads.

Debugging and Tracing ASP.NET Pages

The sad fact of life is that you spend the majority of your development time when building applications debugging the application.

In this section, you learn how to get detailed error messages when developing ASP.NET pages. You also learn how you can display custom trace messages that you can use when debugging a page.

Debugging ASP.NET Pages If you need to view detailed error messages when you execute a page, you need to enable debugging for either the page or your entire application. You can enable debugging for a page by adding a `Debug="true"` attribute to the `<%@ Page %>` directive. For example, the page in Listing 1.18 has debugging enabled.

LISTING 1.18 `ShowError.aspx`

```
 <%@ Page Language="C#" Debug="true" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">

    void Page_Load()
    {
        int zero = 0;
        Label1.Text = (1 / zero).ToString();
    }

</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Show Error</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

    <asp:Label
        id="Label1"
        Runat="server" />

    </div>
    </form>
</body>
</html>
```

WARNING

Make sure that you disable debugging before placing your application into production. When an application is compiled in debug mode, the compiler can't make certain performance optimizations.

When you open the page in Listing 1.18 in your web browser, a detailed error message is displayed (see Figure 1.13).

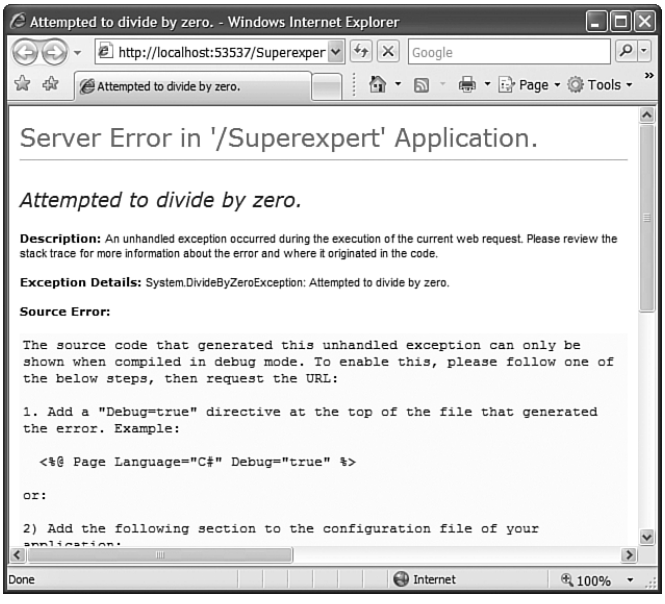

FIGURE 1.13 Viewing a detailed error message.

Rather than enable debugging for a single page, you can enable debugging for an entire application by adding the web configuration file in Listing 1.19 to your application.

LISTING 1.19 `Web.Config`

```
<configuration>
<system.web>
  <compilation debug="true" />
</system.web>
</configuration>
```

When debugging an ASP.NET application located on a remote web server, you need to disable custom errors. For security reasons, by default, the ASP.NET Framework doesn't display error messages when you request a page from a remote machine. When custom errors are enabled, you don't see errors on a remote machine. The modified web configuration file in Listing 1.20 disables custom errors.

LISTING 1.20 Modified `Web.Config`

```
<configuration>
<system.web>
  <compilation debug="true" />
```

LISTING 1.20 Continued

```
  <customErrors mode="Off" />
</system.web>
</configuration>
```

WARNING

For security and performance reasons, don't put websites into production with debug enabled, custom errors disabled, or trace enabled. On your production server, add the following element inside the system.web section of your machine.config file:

<deployment retail="true"/>

Adding this element disables debug mode, enables remote custom errors, and disables trace. You should add this element to the machine.config file located on all of your production servers.

Debugging Pages with Visual Web Developer If you are using Visual Web Developer, then you can display compilation error messages by performing a build on a page or an entire website. Select the menu option Build, Build Page or the menu option Build, Build Web Site. A list of compilation error messages and warnings appears in the Error List window (see Figure 1.14). You can double-click any of the errors to navigate directly to the code that caused the error.

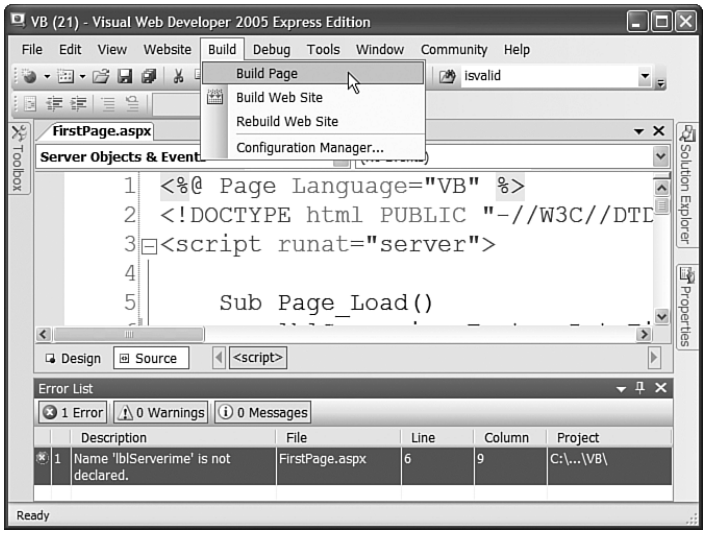

FIGURE 1.14 Performing a build in Visual Web Developer.

If you need to perform more advanced debugging, you can use the Visual Web Developer's debugger. The debugger enables you to set breakpoints and step line by line through your code.

You set a breakpoint by double-clicking the left-most column in Source view. When you add a breakpoint, a red circle appears (see Figure 1.15).

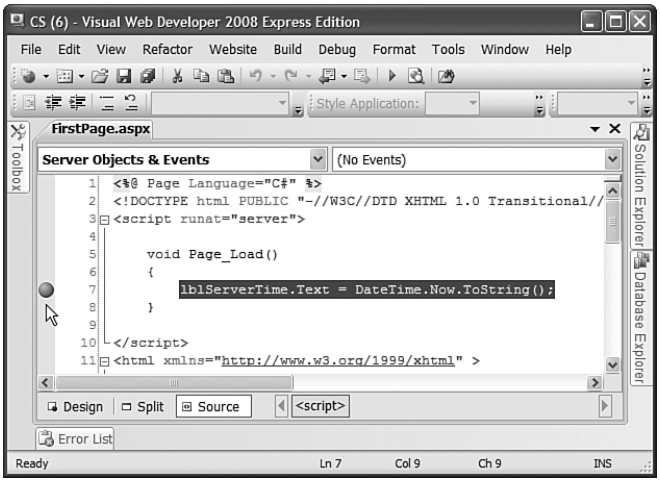

FIGURE 1.15 Setting a breakpoint.

After you set a breakpoint, run your application by selecting the menu option Debug, Start Debugging. Execution stops when the breakpoint is hit. At that point, you can hover your mouse over any variable or control property to view the current value of the variable or control property.

NOTE

You can designate one of the pages in your application as the Start Page. That way, whenever you run your application, the Start Page is executed regardless of the page that you have open. Set the Start Page by right-clicking a page in the Solution Explorer window and selecting the menu option Set As Start Page.

After you hit a breakpoint, you can continue execution by selecting Step Into, Step Over, or Step Out from the Debug menu or the toolbar. Here's an explanation of each of these options:

- **Step Into—**Executes the next line of code.
- **Step Over—**Executes the next line of code without leaving the current method.
- **Step Out—**Executes the next line of code and returns to the method that called the current method.

When you are finished debugging a page, you can continue, stop, or restart your application by selecting a particular option from the Debug menu or the toolbar.

Tracing Page Execution

If you want to output trace messages while a page executes, then you can enable tracing for a particular page or an entire application. The ASP.NET Framework supports both page-level tracing and application-level tracing.

The page in Listing 1.21 illustrates how you can take advantage of page-level tracing.

LISTING 1.21 PageTrace.aspx

```
 <%@ Page Language="C#" Trace="true" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">

    void Page_Load()
    {
        for (int counter = 0; counter < 10; counter++)
        {
            ListBox1.Items.Add("item " + counter.ToString());
            Trace.Warn("counter=" + counter.ToString());
        }
    }

</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Page Trace</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

    <asp:ListBox
        id="ListBox1"
        Runat="server" />

    </div>
    </form>
</body>
</html>
```

Notice that the `<%@ Page %>` directive in Listing 1.21 includes a `trace="true"` attribute. This attribute enables tracing and causes a Trace Information section to be appended to the bottom of the page (see Figure 1.16).

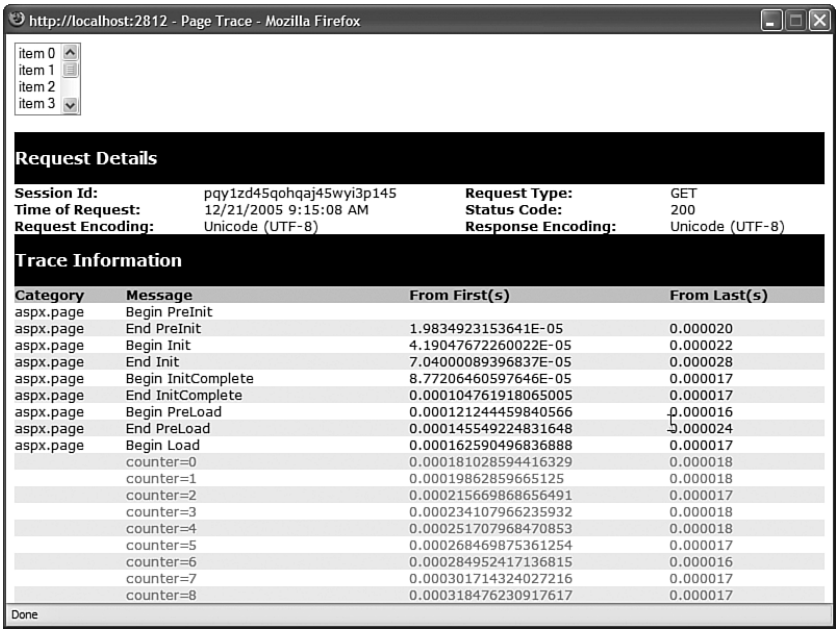

FIGURE 1.16 Viewing page trace information.

Notice, furthermore, that the `Page_Load()` handler uses the `Trace.Warn()` method to write messages to the Trace Information section. You can output any string to the Trace Information section that you please. In Listing 1.21, the current value of a variable named `counter` is displayed.

You'll want to take advantage of page tracing when you need to determine exactly what is happening when a page executes. You can call the `Trace.Warn()` method wherever you need in your code. Because the Trace Information section appears even when there is an error in your page, you can use tracing to diagnose the causes of any page errors.

One disadvantage of page tracing is that everyone in the world gets to see your trace information. You can get around this problem by taking advantage of application-level tracing. When application-level tracing is enabled, trace information appears only when you request a special page named `Trace.axd`.

To enable application-level tracing, you need to add the web configuration file in Listing 1.22 to your application.

LISTING 1.22 `Web.Config`

```
<configuration>
<system.web>
    <trace enabled="true" />
</system.web>
</configuration>
```

After you add the `Web.Config` file in Listing 1.22 to your application, you can request the `Trace.axd` page in your browser. The last 10 page requests made after application-level tracing is enabled are displayed (see Figure 1.17).

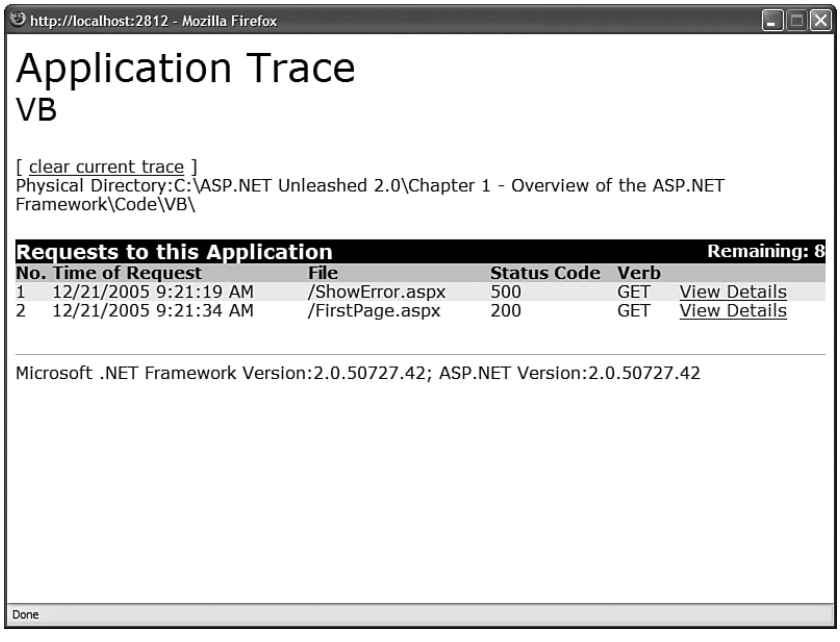

FIGURE 1.17 Viewing application trace information.

WARNING

By default, the `Trace.axd` page cannot be requested from a remote machine. If you need to access the `Trace.axd` page remotely, you need to add a `localOnly="false"` attribute to the trace element in the web configuration file.

If you click the View Details link next to any of the listed page requests, you can view all the trace messages outputted by the page. Messages written with the `Trace.Warn()` method are displayed by the `Trace.axd` page even when page-level tracing is disabled.

NOTE

You can use the new `writeToDiagnosticsTrace` attribute of the `trace` element to write all trace messages to the Output window of Visual Web Developer when you run an application. You can use the new `mostRecent` attribute to display the last 10 page requests rather than the 10 page requests after tracing was enabled.

WARNING

If you don't enable the `mostRecent` attribute when application level tracing is enabled, tracing will stop after 10 pages.

Installing the ASP.NET Framework

The easiest way to install the ASP.NET Framework is to install Visual Web Developer Express. You can download the latest version of Visual Web Developer from www.ASP.net, which is the official Microsoft ASP.NET website.

Installing Visual Web Developer Express also installs the following components:

- Microsoft .NET Framework version 3.5
- SQL Server Express

Visual Web Developer Express is compatible with the following operating systems:

- Windows 2000 Service Pack 4
- Windows XP Service Pack 2
- Windows Server 2003 Service Pack 1
- Windows x64 editions
- Windows Vista

I strongly recommend that you also download the .NET Framework SDK (Software Development Kit). The SDK includes additional documentation, sample code, and tools for building ASP.NET applications. You can download the SDK from the Microsoft MSDN website located at msdn.microsoft.com. The .NET Framework 3.5 SDK is included as part of the Microsoft Windows SDK for Windows Server 2008.

You can install Visual Web Developer Express on a computer that already has Visual Studio 2005 or Visual Web Developer 2005 installed. Different versions of the development environments can co-exist peacefully.

Furthermore, the same web server can serve ASP.NET 1.1 pages, ASP.NET 2.0 pages, ASP.NET 3.0 pages, and ASP.NET 3.5 pages. Each version of the .NET Framework is installed in the following folder:

```
C:\WINDOWS\Microsoft.NET\Framework
```

For example, on my computer, I have the following five versions of the .NET Framework installed (version 1.0, version 1.1, version 2.0, version 3.0, and version 3.5):

```
C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705
C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727
C:\WINDOWS\Microsoft.NET\Framework\v3.0
C:\WINDOWS\Microsoft.NET\Framework\v3.5
```

The first three folders include a command-line tool named `aspnet_regiis.exe`. You can use this tool to associate a particular virtual directory on your machine with a particular version of the .NET Framework.

For example, executing the following command from a command prompt located in the v1.0.3705, v1.1.4322, or v2.0.50727 folders enables the 1.0, 1.1, or 2.0 version of ASP.NET for a virtual directory named MyApplication:

```
aspnet_regiis -s W3SVC/1/ROOT/MyApplication
```

By executing the `aspnet_regiis.exe` tool located in the different .NET Framework version folders, you can map a particular virtual directory to any version of the ASP.NET Framework.

The .NET Frameworks 3.0 and 3.5 work differently than earlier versions. The 3.0 and 3.5 versions build on top of the existing .NET Framework 2.0. To use these versions of the .NET Framework, you need to add the correct assembly references to your website and use the correct versions of the C# or VB.NET compilers. You reference these assemblies and configure the compiler within your application's web.config file. When you create a new website in Visual Web Developer, the necessary configuration settings are included in your web.config file automatically.

You also have the option of targeting a particular version of the .NET Framework. To do this, select the menu option Website, Start Options and select the Build tab. You can choose to target the .NET Framework 2.0, .NET Framework 3.0, or .NET Framework 3.5 (see Figure 1.18).

NOTE

If you load an existing ASP.NET 2.0 website into Visual Web Developer 2008, Visual Web Developer will prompt you to upgrade the website to ASP.NET 3.5. When Visual Web Developer upgrades your website, it modifies your web.config file.

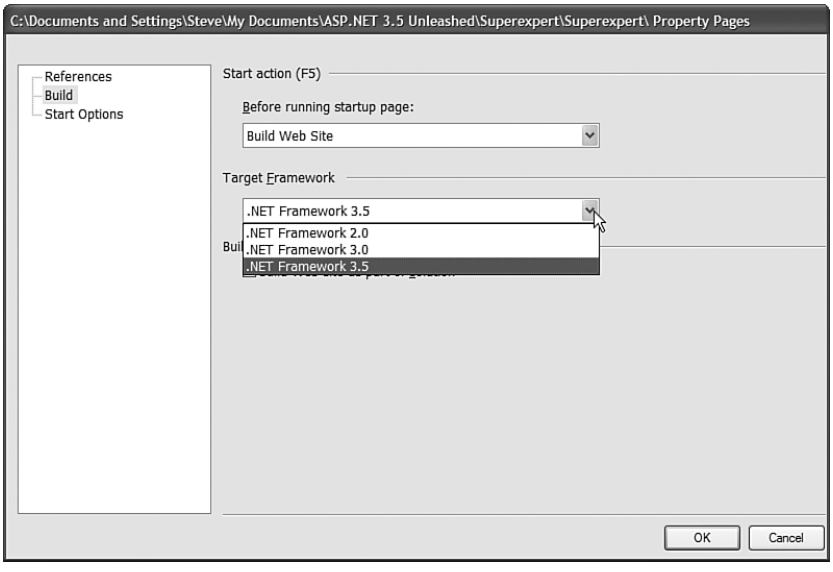

FIGURE 1.18 Targeting a particular version of the .NET Framework.

Summary

In this chapter, you were introduced to the ASP.NET 3.5 Framework. First, we built a simple ASP.NET page. You learned about the three main elements of an ASP.NET page: directives, code declaration blocks, and page render blocks.

Next, we discussed the .NET Framework. You learned about the 13,000 classes contained in the Framework Class Library and you learned about the features of the Common Language Runtime.

You also were provided with an overview of ASP.NET controls. You learned about the different groups of controls included in the .NET Framework. You also learned how to handle control events and take advantage of View State.

We also discussed ASP.NET pages. You learned how ASP.NET pages are dynamically compiled when they are first requested. We also examined how you can divide a single-file ASP.NET page into a code-behind page. You learned how to debug and trace the execution of an ASP.NET page.

At the end of the chapter, we covered installation issues in getting the ASP.NET Framework up and running. You learned how to map different Virtual Directories to different versions of the ASP.NET Framework. You also learned how to target different versions of the .NET Framework in your web configuration file.