<chapter> 4

ASP.NET Pages

Alive!!! It's alive!!…
The page is ALIVE!!!!!

I don't want to sound like a nag or like I'm harping on the same issue over and over, but ASP.NET is really about events and objects. The ASP.NET page is no exception. It is an object in the eyes of ASP.NET just like anything else. It has properties and methods and can be interacted with, similar to the other objects you've seen so far, and the plethora of things to come as you move through the remainder of this book.

Because events are a big part of ASP.NET and also a huge part of the power of the ASP.NET page object, it's important to look closely at these event and the order in which they occur so that you can utilize them to their fullest potential.

Understanding Page Events

Maybe this isn't the best way to explain this. Maybe my mother wouldn't be so proud of my openness. Maybe these examples will be too graphic for you, but it's the best way I can describe it, so please bear with me.

Every morning of every workday I start my day the same way. I get up, drag my carcass into the living room, and turn on the television. I stare at overpaid morning show hosts like a dribbling fool, with about as much cerebral activity as an earthworm trying to figure out what to eat. After the first flicker of lights goes off in my cranium, I drag my carcass to the shower to defrost my brain.

In the shower I have a set routine so as not to miss any vital, proper hygienic functions and to ensure that the other people in my office and our clients don't give me those strange looks anymore when I enter the room. I think if you took the time to analyze this portion of your life, you'd see that without conscious thought, you pick up the bar of soap and proceed to cleanse without ever putting a thought into what you are doing. You may be singing or thinking about the day you are about to face and before you know it…Boom! You're finished.

It's a routine that I go through every morning almost without thought. And that's a good thing; if I needed to be able to think before I could shower, I wouldn't arrive at work until 10:00 a.m., or until I've had several cups of joe. This practice is part of a subconscious, programmed routine that has been burned into my mind from years of habit.

When I then go to work, I respond to the world around me. I make decisions based on what stimulus comes my way. Maybe someone comes to the office for a meeting. They ask questions and I answer them. The phone rings and I pick it up. I get e-mail and I occasionally read it and less often answer it. I am affected by the world around me and what kind of input it gives me. I then go home and enjoy the solace (this isn't a joke and I'm not kidding) of my family.

At the end of the day I get into my pajamas and go to sleep.

ASP.NET pages are a lot like this. They go through a routine every time they are called. These routines are the ASP.NET pages' *events*. They are like the steps I go through every morning to prepare for my day, or like showering by the same routine everyday.

ASP.NET pages have a routine of events that happen, and they happen in the same way, in the same order, every time. You've seen one of these events, `Page_Load`, in some of the previous examples. Let's look into these events and the others that a page goes through when it is executed.

The three main events, although there are others, are as follows:

- `Page_Init`
- `Page_Load`
- `Page_Unload`

Page_Init

The `Page_Init` event is the first to occur when an ASP.NET page is executed. This is where you should perform any initialization steps that you need to set up or create instances of server controls. Server controls are discussed in later chapters, so just keep this event in mind.

You don't want to try to access controls in this event because there is no guarantee that they have been created yet. It is during this event that they are created, and you can control whether your attempt to use these objects will be thwarted by the server processing your request before the object has been created.

The following is an example of the structure of how to use the `Page_Init` event

Visual Basic .NET

```
Sub Page_Init()
     'Place your Page_Init code here
End Sub
```

C#

```
void Page_Init(){
    //Place your Page_Init code here
}
```

Note that the `Page_Init` event fires only the first time the page is loaded. When you use a web form and post back to this page again, the `Page_Init` event doesn't fire. But the `Page_Load` event fires each time the page loads.

Page_Load

This is the page event where you will be doing most of your work. This event occurs only when all the objects on the page have been created and are available for use. You will see—within this book and in other examples available in the .NET Framework SDK and ASP.NET-related web sites—that the lion's share of work on ASP.NET pages is done during this event. We've been using this event since the beginning of the book and will continue to use it in just about every example.

Although you've seen it a zillion times already in the book, for consistency's sake I'll show you the form here. It doesn't look a whole lot different from the `Page_Init` example, and for all intents and purposes the only thing that's different is that the word `Init` is substituted with the word `Load`.

Visual Basic .NET

```
Sub Page_Load()
     'Place your Page_Load code here
End Sub
```

C#

```
void Page_Load(){
    //Place your Page_Load code here
}
```

Page_Unload

`Page_Unload` is the counterpart to `Page_Init`. Just as `Page_Init` is an event that happens *before* anything else happens, `Page_Unload` happens *after* everything else happens. It is available for you to perform any operation you need to after you are completely finished with the page.

For instance, imagine that you temporarily needed to create a file on the server during the page's processing. You wouldn't want to leave it there for eternity, especially if the file was unique to each visitor of the web site. You could have loads and loads of files building on your server without any way to get rid of them. But if you were a good boy or girl, you could destroy the file during the page's `Page_Unload` event and make the server administrator a happy camper.

Just to be fair and impartial, I don't want to leave out showing you the structure of the `Page_Unload` event. Look familiar?

Visual Basic .NET

```
Sub Page_Unload()
     'Place your Page_Unload code here
End Sub
```

C#

```
void Page_Unload(){
    //Place your Page_Unload code here
}
```

Getting back to my morning routine, it looks like this:

1. `Peter_Init`. Roll carcass from bed to in front of the television.
2. `Peter_Load`. Take shower brainlessly, get dressed (make sure socks match and colors coordinate—check with wife for confirmation). Get into car and drive to the office.
3. Handle the day in all its glory and all the blessings that come with it.
4. `Peter_Unload`. Get into jammies and go to sleep.

It's that routine, and I behave just like the `Page` object does. When I run through these events, I am investigating and affecting all kinds of things. I'm finding out the condition of the world that morning by listening to news, changing the state of my brain to somewhat functional, changing the direction that my hair points from an erratic bird's nest to some semblance of a hairdo, and more.

I'm doing this through checking and setting properties and executing methods, so to speak. During `Peter_Init`, I execute the `RollCarcass()` method to change the `Peter.Sleeping` property from true to false.

During `Peter_Load` I'm checking the value of the `eye.bags` property and seeing what the value of the `hair.color` property is, which is generally grayer than the day before. I'm assuring that the `body.odor` property is set to zero by executing the `Shower()` method.

I then have the ability to respond to events and stimulus from the world around me throughout the day. Then during the `Peter_Unload` event, I execute the `CollapseFromExaustion()` method to set the `Peter.Sleeping` property to true.

Can you see how these different events at different times have a direct affect on my condition? ASP.NET pages can be affected just like this with their different events. Now are you beginning to see more clearly how events and objects interact in ASP.NET and how this is a totally different paradigm from any traditional way of web programming in HTML or Active Server Pages.

As I said in the beginning of the chapter, ASP.NET is all about events and objects, and the ASP.NET page is no exception. You know that objects are made up of their properties and methods, and now you know that objects can also have events, as well.

The `page` object has the three mentioned events, as well as others that execute without intervention from the designer, but other events also affect ASP.NET pages.

User-Initiated Events

Just as I am faced with input from the world around me after the `Peter_Onload` event has finished, a page can also deal with events initiated by the web page's visitor.

Let's look at an example of some events, both self executing and user initiated. Below is a page that shows the date and asks you to pick what mood you're in. In the code samples, you'll also be shown another neat server control called a `RadioButtonList` and a cool feature of the .NET Framework called Databinding. You will also see a property of the `Page` object called `IsPostBack`. We will discuss this later in this chapter, but again, don't get hung up on these things—just concentrate on the events in the page.

Visual Basic .NET

```
<%@ page language="vb" runat="server"%>
<script  runat=server>

Sub Page_Load()
    dim TodaysDate as Date
    TodaysDate = DateTime.Now.ToShortDateString
    OurTitle.Text = "<u><b>Today's Date is " + TodaysDate    + "</b></u>"

    If Not IsPostBack then
        dim MoodArray(3) as String
        MoodArray(0) = "Good Mood"
        MoodArray(1) = "Okay Mood"
        MoodArray(2) = "Bad Mood"
```

```
        MoodArray(3) = "Totally Melancholy "

        YourMood.DataSource = MoodArray
        YourMood.DataBind()
    End If

End Sub

Sub CheckMood(sender As Object, e As System.EventArgs)
    If YourMood.SelectedIndex > -1 then
        SelectedMood.Text = "The Mood that you selected is " +
        ➥YourMood.SelectedItem.Text
    Else
        SelectedMood.Text = "What? You don't feel anything?"
    End If
End Sub

</script>
<html>
<title>What's Your Mood?</title>
<body>
<form id="MoodForm" runat="server">
<asp:label id="OurTitle" runat="server"/><br>
<asp:RadioButtonList id="YourMood" runat="server"/>
<asp:Button id="MoodButton" text="What's Your Mood?" onClick="CheckMood"
➥runat="server"/>
<br><br>
<asp:label id="SelectedMood" runat="server"/><br>
</form>
</body>
</html>
```

C#

```
<%@ page language="c#" runat="server"%>
<script  runat=server>

void Page_Load(){
    String TodaysDate;
    string[] MoodArray = new string[4];

    TodaysDate = DateTime.Now.ToShortDateString();
    OurTitle.Text = "<u><b>Today's Date is " + TodaysDate    + "</b></u>";

    if (!IsPostBack){
        MoodArray[0] = "Good Mood";
        MoodArray[1] = "Okay Mood";
        MoodArray[2] = "Bad Mood";
        MoodArray[3] = "Totally Melancholy ";

    YourMood.DataSource = MoodArray;
```

continues

C# (continued)

```
    YourMood.DataBind();
    }

}

void CheckMood(object Source, System.EventArgs s){
    if (YourMood.SelectedIndex > -1) {
        SelectedMood.Text = "The Mood that you selected is " +
        ➥YourMood.SelectedItem.Text;
    }else{
        SelectedMood.Text = "What? You don't feel anything?";
    }
}

</script>
<html>
<title>What's Your Mood?</title>
<body>
<form runat="server">
<asp:label id="OurTitle" runat="server"/><br>
<asp:RadioButtonList id="YourMood" runat="server"/>
<asp:Button id="MoodButton" text="What's Your Mood?" onClick="CheckMood"
➥runat="server"/>
<br><br>
<asp:label id="SelectedMood" runat="server"/><br>
</form>
</body>
</html>
```

If you look at Figure 4.1, you can see the results of the initial load of the page. The `Page_Load` event fires, at which time the date is created. I then set the `Text` property of `OurTitle` and I build an array that will make up the radio buttons.

If you look back at the code samples again, you can see that attached to the button is an `onClick` event that calls a function called `"CheckMood"`. I know that this looks pretty similar to a client-side JavaScript function call, but remember that ASP.NET is a server-side technology. If you look at the code delivered to the browser, you see that there is no `onClick` event to be seen.

```
<input type="submit" name="MoodButton" value="What's Your Mood?" id="MoodButton" />
```

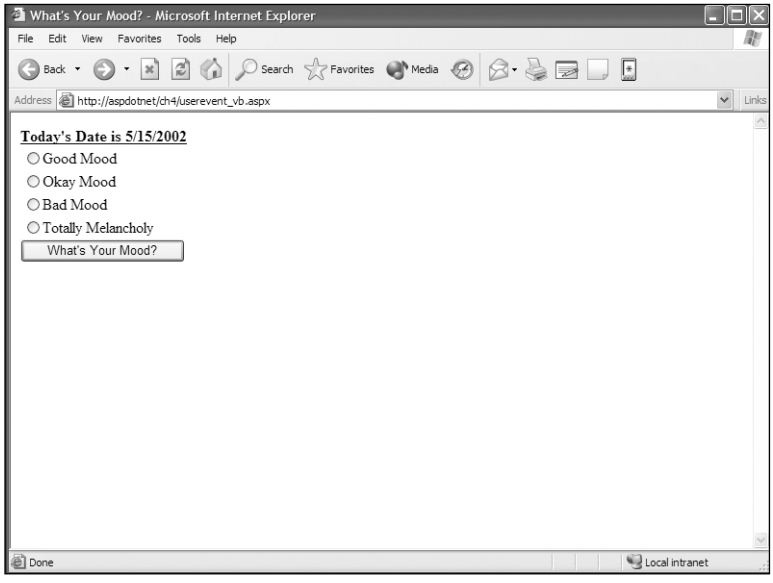

FIGURE 4.1

The `Page_Load` *event has built the page, but the* `onClick` *event hasn't had any effect because the button hasn't been pressed yet.*

ASP.NET knows whether you pressed this button—not by a typical client-side `onClick` event, but by inspecting the form that is posted and seeing whether this button was pressed. The terminology is similar to client-side JavaScript, but the function and method is totally different.

Now it's time to pick a mood and click the button. You can see in Figure 4.2 that the mood is now displayed because the `onClick` event that took place server-side executed the function called `CheckMood`, which sets the text of the label.

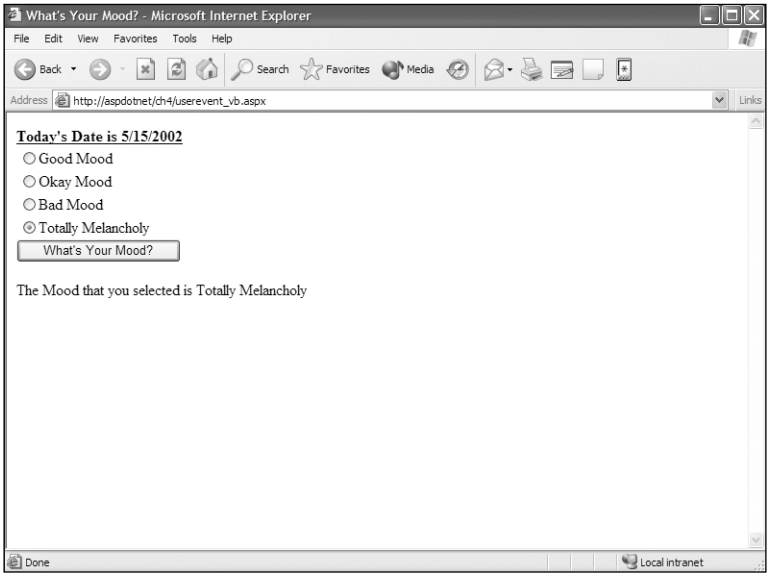

FIGURE 4.2

The `onClick` *event is fired by clicking the button.*

To reinforce the point that ASP.NET is smart about calling functions and that what is executed is determined by the `onClick` event of the button, I have put together a sample with two different buttons that call two different functions. Each button uses its own `onClick` event.

Visual Basic .NET

```
<%@ page language="vb" EnableViewState="false" runat="server"%>
<script  runat=server>

Sub CountDown(sender As Object, e As System.EventArgs)
        dim i as Integer
        for i = CDbl(Text1.Text) to 1    Step -1
            OurLabel.Text += "Countdown: " + i.ToString() + "<br>"
        next
End Sub

Sub StringLength(sender As Object, e As System.EventArgs)
    OurLabel.Text = "The length of this string is: " + Text1.Text.Length.toString
End Sub

</script>
<html>
<head>
<title>What do you want?</title>
```

```
</head>
<body>
<form runat="server">
Either enter a word to find its length or a number to count down from<br>
<asp:TextBox id="Text1" runat="server"/>
<asp:Button id="btnCountDown" text="Count Down" onClick="CountDown"
➥runat="server"/>
<asp:Button id="btnLength" text="Get Length" onClick="StringLength"
➥runat="server"/>
<br><br>
<asp:label id="OurLabel" runat="server"/><br>
</form>
</body>
</html>
```

C#

```
<%@ page language="cs" EnableViewState="false" runat="server"%>
<script  runat=server>
void CountDown(object Source, System.EventArgs s){
    int i;
    for (i = Convert.ToInt16(Text1.Text);i >= 1;—i){
        OurLabel.Text += "Countdown: " + i.ToString() + "<br>";
    }
}

void StringLength(object Source, System.EventArgs s){
    OurLabel.Text = "The length of this string is: " +
    ➥Text1.Text.Length.ToString();
}

</script>
<html>
<head>
<title>What do you want?</title>
</head>
<body>
<form runat="server">
Either enter a word to find its length or a number to count down from<br>
<asp:TextBox id="Text1" runat="server"/>
<asp:Button id="btnCountDown" text="Count Down" onClick="CountDown"
➥runat="server"/>
<asp:Button id="btnLength" text="Get Length" onClick="StringLength"
➥runat="server"/>
<br><br>
<asp:label id="OurLabel" runat="server"/><br>
</form>
</body>
</html>
```

If you look at Figure 4.3 you can see that after the Count Down button was clicked, with the value of 10 in the text box, the `CountDown` function was executed and the code generated and displayed properly.

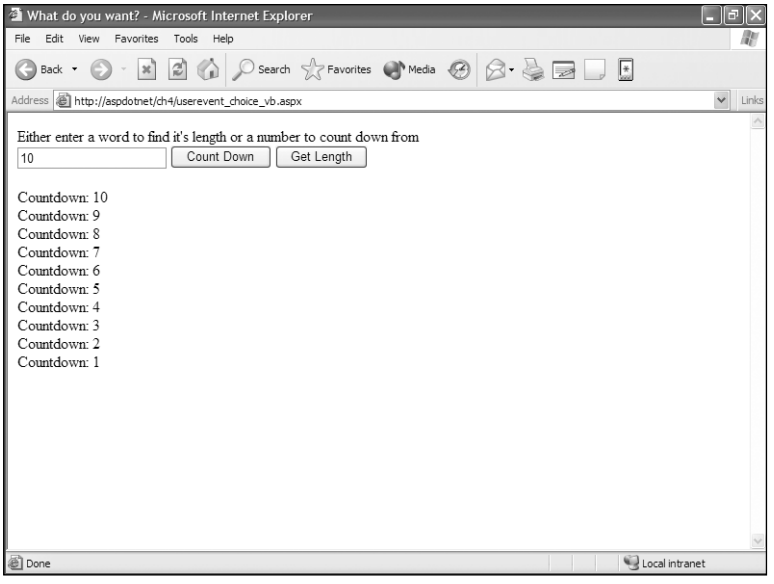

FIGURE 4.3

Clicking the Count Down button causes the `CountDown` function to execute.

Now if you put a string such as "What is the length?" in the text box and click the Get Length button, you are executing the `StringLength` function.

NOTE

This example has a bit of hidden danger in that if someone enters a string in the text box and clicks the Count Down button, ASP.NET will cause an error. This is because it can't convert a `String` type to an `Integer` in this circumstance. But you don't need to worry about this in your real-world applications because ASP.NET provides some really, REALLY cool answers to validating input data (that we will be devoting an entire chapter to later). The validators would totally solve any issues like this—and more.

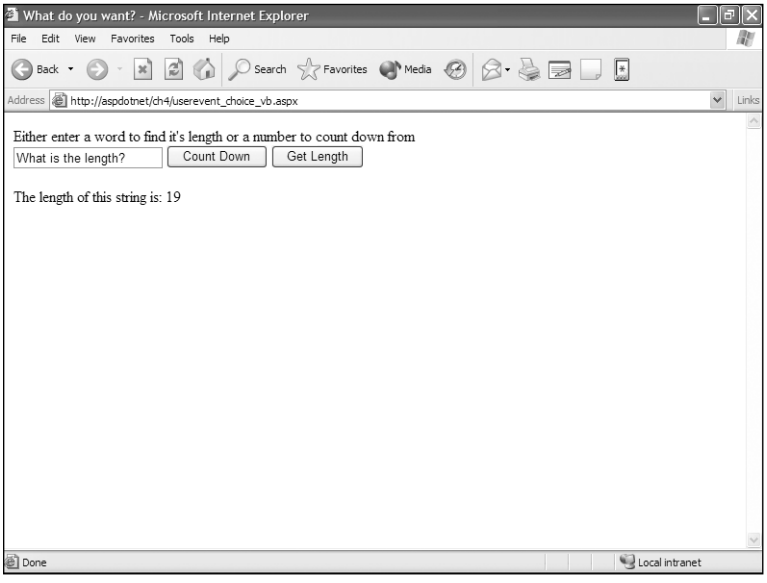

FIGURE 4.4

Clicking the Get Length button causes the `StringLength` *function to execute.*

As if all the objects and events covered earlier weren't enough, the concept of user-initiated events opens up a totally new way of thinking again about how you can handle and manipulate data, depending on how a user interacts with your web application.

Other events are also available that can help you manipulate data and objects, and I would encourage you again to go to the class browser located at the following link and look at what events you can use on each object:

```
http://samples.gotdotnet.com/quickstart/aspplus/
```

Now that we've touched on the different events, both default and user-initiated, let's move on to looking at some of the key properties of the `Page` object.

Understanding Page Properties

As I've said over and over and over and over again, ASP.NET is an object-oriented programming paradigm, and ASP.NET pages are no exception. They are objects, as well, and have properties and methods just like every other object.

Let's take some time to explore some of the key properties that the ASP.NET page object has. First, it is helpful to know where in the .NET Framework the `Page` object is. It is located in `System.Web.UI`. You can go there in the class browser that I mentioned in earlier chapters (`http://www.gotdotnet.com`) and investigate all the properties. In this book, however, we are going to cover the biggies, the heavyweights, the granddaddies. In other words, we're going to cover the ones that are used most commonly.

The `Page` object properties covered in this section are:

- `Application`
- `IsPostBack`
- `Request`
- `Response`

There are actually a few additional properties, such as `Session` and `Validation`, that are covered in other chapters and are pointed out as `Page` object properties then.

NOTE

If you come from a traditional ASP programming background, many of these probably look familiar to you. In traditional ASP, many of these ASP.NET Page properties were what made up the default objects in the language. You ask, "Why are they now properties of the page instead of remaining objects?" In reality they are still objects and can be found in the `System.Web` *section of the .NET Framework. To simplify the use of objects, many properties of objects are really just instances of other objects. You will see this throughout the framework. Actually, just about every property is an instance of another object. For instance, any property that is a string is actually an instance of the* `System.String` *object. So properties in the .NET Framework are generally just instances of other objects.*

Application

The `Application` property or `HttpApplicationState` object contains a collection of data that is available across the entire application during the life of the application. It is a shared variable among the users and is an easy-to-use place to store global information that will be needed across your application.

Setting an `Application` variable is a piece of cake:

Visual Basic .NET

```
Application("Publisher") = "New Riders"
OurLabel.Text = Application("Publisher")
```

C#

```
Application("Publisher") = "New Riders";
OurLabel.Text = Application("Publisher");
```

This would write the words "New Riders" to your browsers. You typically wouldn't go around setting an application variable all over the place every time a page loaded. Because it is an application-level variable, under most circumstances it really needs to be set only once during the life of an application. ASP.NET provides a way to set `Application` variables (in addition to many other things) within a file called the global.asax, which resides in the root folder of your application. On a web site, this is typically the folder that your domain points to, where your index or default home page is located. This example creates four application variables in the `Application_OnStart` event, which happens the first time a web page is requested from an application. This doesn't mean that web service on the server must be stopped and started if you make a change to anything in the `Application_OnStart`. ASP.NET is smart enough to recognize the change and reset the variable's value.

Visual Basic .NET—`global.asax`

```
<script language="vb" runat=server>
Sub Application_OnStart()
    Application("Publisher") = "New Riders"
    Application("BookTitle") = "ASP.NET for Web Designers"
    Application("Author") = "Peter"
    Application("Rating") = "5 Stars, WHAHOOO!!"
End Sub
</script>
```

C#—**global.asax**

```
<script language="c#" runat=server>
void Application_OnStart(){
    Application["Publisher"] = "New Riders";
    Application["BookTitle"] = "ASP.NET for Web Designers";
    Application["Author"] = "Peter";
    Application["Rating"] = "5 Stars, WHAHOOO!!";
}
</script>
```

Now, if you make a file that requests the application variables, you can see how they are retrieved.

Visual Basic .NET—**page_application_vb.aspx**

```
<%@ page language="vb" runat="server"%>
<script  runat=server>

Sub Page_Load()
    Title.Text = "<u>Title:</u> " + Application("BookTitle")
    Publisher.Text = "<u>Publisher:</u> " + Application("Publisher")
    Author.Text = "<u>Author:</u> " + Application("Author")
    BookRating.Text = "<u>Rating:</u> " + Application("Rating")
End Sub

</script>
<html>
<title>Application</title>
<body>
<asp:label id="Title" runat="server"/><br>
<asp:label id="Publisher" runat="server"/><br>
<asp:label id="Author" runat="server"/><br>
<asp:label id="BookRating" runat="server"/>
</body>
</html>
```

C#—**page_application_cs.aspx**

```
<%@ page language="cs" runat="server"%>
<script  runat=server>

void Page_Load(){
    Title.Text = "<u>Title:</u> " + Application["BookTitle"];
    Publisher.Text = "<u>Publisher:</u> " + Application["Publisher"];
    Author.Text = "<u>Author:</u> " + Application["Author"];
    BookRating.Text = "<u>Rating:</u> " + Application["Rating"];
}
```

```
</script>
<html>
<title>Application</title>
<body>
<asp:label id="Title" runat="server"/><br>
<asp:label id="Publisher" runat="server"/><br>
<asp:label id="Author" runat="server"/><br>
<asp:label id="BookRating" runat="server"/>
</body>
</html>
```

As you can see in Figure 4.5, the page requested these application variables and displayed them in the browser just as expected.

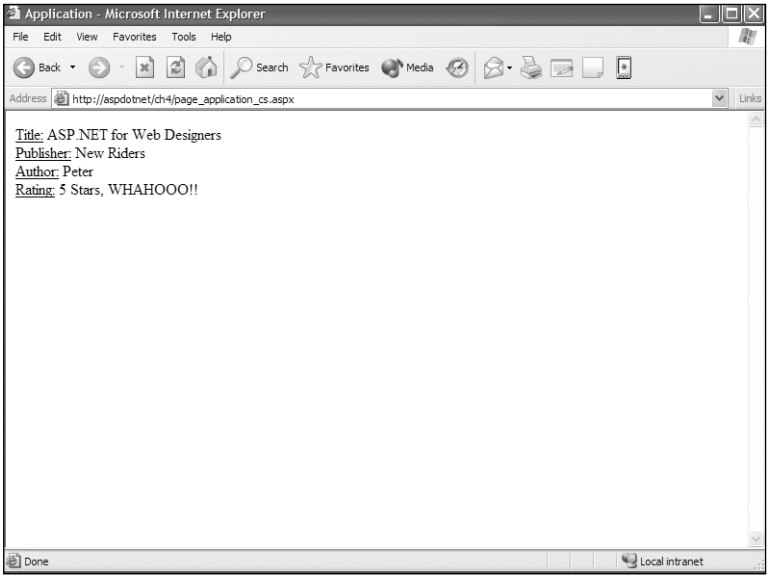

FIGURE 4.5

Application variables are a simple way to make routine and application-wide information available to you.

TIP

As I mentioned in Chapter 1, there is another file that can contain not only application variables but a ton of other information: the web.config file. This is an XML document that allows you to configure many elements of your web application and isn't limited to one file per application. You can add a web.config file to every folder in your application to set its configuration, rules, or variables. This file is a bit out of the scope of this book, but if you are looking for greater control over your application, the web.config file may be your answer. In addition, there are great performance advantages in using variables from within the web.config file over an application variable. You can learn more about configuring your web applications with the web.config file in the .NET Framework SDK at the following link:

```
ms-help://MS.NETFrameworkSDK/cpguidenf/html/
➥cpconaspnetconfiguration.htm
```

IsPostBack

You saw the `IsPostBack` property mentioned in one of the earlier examples. This is an incredibly useful property that you will find yourself using all the time when you are processing forms.

As a matter of fact, this property is at the core of some of the most powerful features in ASP.NET and its pages. ASP.NET web forms are built on the concept of making round trips to the server. It's a simple property whose value is either true or false. The property is set to true whenever you post a page to the server with a `runat="server"` directive in the `form` tag. `Form` tags with `runat="server"` directives always post back to themselves. If you put an `action` property in the tag, it will basically be ignored.

This property is basically used, as I've said, to determine whether a page is being loaded for the first time or is being posted back to itself. Take a look:

Visual Basic .NET—**page_postback_vb.aspx**

```
<%@ page language="vb" runat="server"%>
<script  runat=server>

Sub Page_Load()
    OurTitle.Text = "No this page wasn't posted back"

    If IsPostBack then
        OurTitle.Text = "Yes this page has been posted back"
    End If

End Sub
</script>
<html>
<title>Was this page posted back?</title>
<body>
<form runat="server">
<asp:label id="OurTitle" runat="server"/><br><br>
<asp:Button id="PostBack" text="Post this Form?"  runat="server"/>
</form>
</body>
</html>
```

C#—**page_postback_cs.aspx**

```
<%@ page language="cs" runat="server"%>
<script  runat=server>

void Page_Load(){
    OurTitle.Text = "No this page wasn't posted back";

    if (IsPostBack){
        OurTitle.Text = "Yes this page has been posted back";
    }
}

</script>
<html>
<title>Was this page posted back?</title>
<body>
<form runat="server">
<asp:label id="OurTitle" runat="server"/><br><br>
<asp:Button id="PostBack" text="Post this Form?"  runat="server"/>
</form>
</body>
</html>
```

You can see in Figure 4.6 that the first time you request this page, you are presented with the default message because the `IsPostBack` property is false on the initial load of that page, so the `if` branching statement doesn't execute its code.

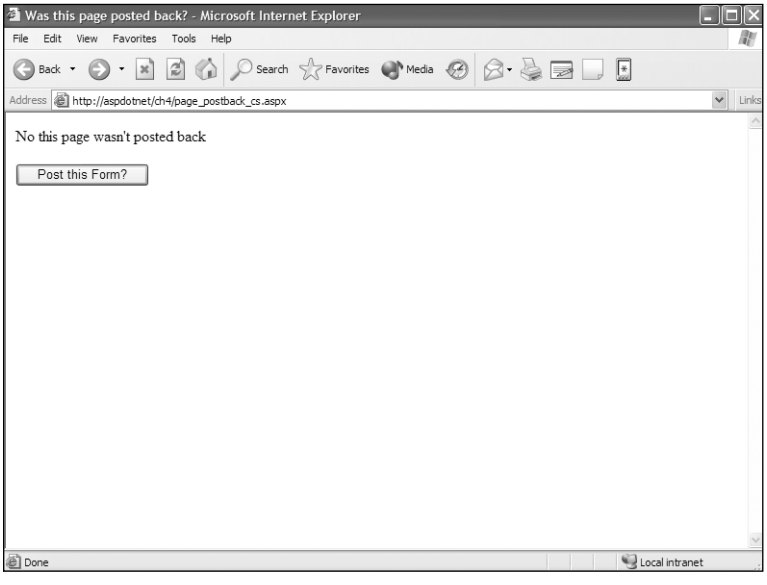

FIGURE 4.6

The first time the page is loaded, the page's `IsPostBack` *property is false.*

Now what happens if you click the button and post the form? In Figure 4.7 you can see that after you post the form, the label's text value is changed because `IsPostBack` is now true and the `if` block executes.

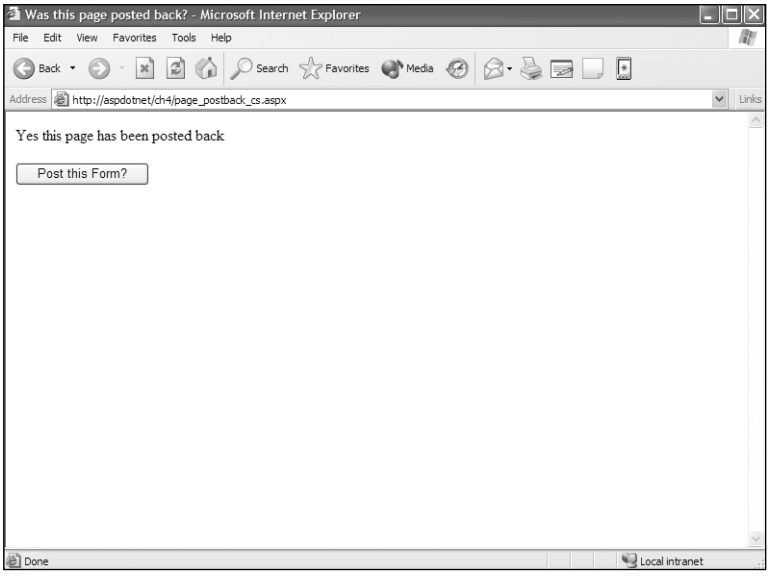

FIGURE 4.7

Now that you posted the page. the `IsPostBack` *property is true and the branching statement executes.*

This technique will be used and demonstrated in abundance in later chapters, where you will be able to see it under many different circumstances. So hold your horses and don't get your undies in a twist.

Request and *Response*

I've always told people who ask me about the Internet that it's not a big mystical thing that can't be understood or pinned down. It's really very simple. It's a communication tool, just like a phone or even more simply a piece of paper and a pen. It's a way to say something to others and, thankfully, as dynamic languages have become so prolific, they can say stuff back and we can respond to what they said programmatically.

Picture two grandmothers sitting at a kitchen table enjoying a cup of tea and good conversation. The air is filled with the pleasing aroma of fresh baked apple pie so thick you can almost taste it.

Ethel: "Bernice, do you want to see my latest masterpiece? It's a deep dish apple pie done to perfection."

Bernice: "Certainly Ethel. Let me have a look see."

Ethel gets up from the table and saunters over to the window sill where the golden brown pie sits. She carefully slides it off the sill onto her toweled hand and returns with the same grace to the table. She approaches Bernice and lowers it in front of her so the waves of succulent aroma waft up to her nose.

Bernice: "Um Um!!!!! Can I have a piece of that?"

Ethel: "You sure can."

Ethel cuts and serves a slice of that delicious pie to Bernice, who proceeds to ravage it mercilessly. Whipped cream, crust, and apples fly as Bernice devours her prize. In between forkfuls, Bernice lifts her apple pie covered face and says:

Bernice: "Can I have the recipe? This is delicious!"

Ethel, also now covered in the remains of a once beautiful piece of apple pie from Bernice's frenzy, pulls the recipe from her recipe file and gives it to her dear friend…Bernice.

—The End—

Okay, a little too dramatic for you? I'm sorry. Sometimes I get carried away. I have behaved quite professionally so far this chapter, and I though it was time to get silly again.

Anyway, the point of my dramatic depiction of Ethel and Bernice was to demonstrate communication and to give an example that shows a bunch of different types of requests and responses. Later we will try to programmatically reproduce this scene and help you see it in action. First, though, it's a good idea to look a bit more into the `Request` and `Response` properties of your ASP.NET pages.

Request

If you can think of the `Request` as a question mark, you will quickly and easily understand what it does in concept. It's similar to Bernice asking whether she can have a piece of pie or have a copy of the recipe. It's also good for finding out what the response was to a question we ask just like Ethel asked whether Bernice wanted to see the pie. We find out the condition of things being sent to the server with the `Request`.

As I said with the `Application` property, the `Request` and `Response` properties are both actually objects in their own right and can be explored in the class browser I redundantly mention.

You can get tons of data by using the `Request` object, and I covered a few of them here. There are a handful that you will find yourself using over and over—such as `Request.Form`, `Request.Querystring`, and others—and that you will find yourself using as a very pointed tools for specific tasks. You won't use all the properties of the `Request` all the time, but they are powerful nonetheless.

For instance, let's take the `Request.IsSecureConnection` property. This is a nifty little tool that checks to see whether you are communicating via an HTTPS secure protocol or not. It returns a Boolean value of true or false, depending on whether the connection is secure. It's a neat gadget that will definitely come in handy in certain situations, but it's not something you're gonna use every day. Table 4.1 provides a list of some that you *will* use just about every day.

TABLE 4.1 Request Object Properties

Request Property	Description
`Browser`	An instance of the `HTTPBrowserCapabilites` object that is chock full of information about the client's browser and its capabilities.
`Cookies`	Enables you to store information on the client's machine and retrieve it at a later time.
`Form`	Retrieve the values of `form` elements submitted during a `form` post.
`QueryString`	Retrieve the values of name/value pairs stored in the requesting URL. These can be inserted directly or can be the result of a form being submitted via the `Get` method.
`ServerVariables`	An ocean full of variables about the request.

Now take a look at a few of these in action. The following is a code example that incorporates a few of these `Request` properties.

Visual Basic .NET—**page_request_vb.aspx**

```
<%@ page language="vb" runat="server"%>
<script  runat=server>

Sub Page_Load()
        OurLabel.Text = "Here are some of our Request properties in action<br>"
        OurLabel.Text += "My Browser is: " + Request.Browser.Browser + "<br>"
        OurLabel.Text += "Our Querystring Color is: " +
        ➥Request.QueryString("color") + "<br>"
        OurLabel.Text += "This file is located at: " +
        ➥Request.ServerVariables("Path_Translated")  + "<br>"

End Sub

</script>
<html>
<head>
<title>Page Request</title>
</head>
<body>
<asp:label id="OurLabel" runat="server"/>
</body>
</html>
```

C#—**page_request_cs.aspx**

```
<%@ page language="c#" runat="server"%>
<script  runat=server>

void Page_Load(){
        OurLabel.Text = "Here are some of our Request properties in action<br>";
        OurLabel.Text += "My Browser is: " + Request.Browser.Browser + "<br>";
        OurLabel.Text += "Our Querystring Color is: " +
        ➥Request.QueryString["color"] + "<br>";
        OurLabel.Text += "This file is located at: " +
        ➥Request.ServerVariables["Path_Translated"]  + "<br>";
}

</script>
<html>
<head>
<title>Page Request</title>
</head>
<body>
<asp:label id="OurLabel" runat="server"/>
</body>
</html>
```

> **NOTE**
>
> *Please note that whenever you are specifying an item in a collection, such as the* `Request.Form` *or* `Request.Querystring`*, the bracket types around the item are different in the two languages. In Visual Basic .NET, you use parentheses and quotes around the item name, whereas in C# you use square brackets and quotes. If you come from a traditional ASP/VBScript background and decide to use C# as your language, you will need to watch out for this.*

If you look at Figure 4.8, you can see we are requesting the value of the browser being used to view the ASP.NET page, a value in the `QueryString`—or in other words, name/value pairs passed in the URL of the page—and a `ServerVariable` called `"Path_Translated"`.

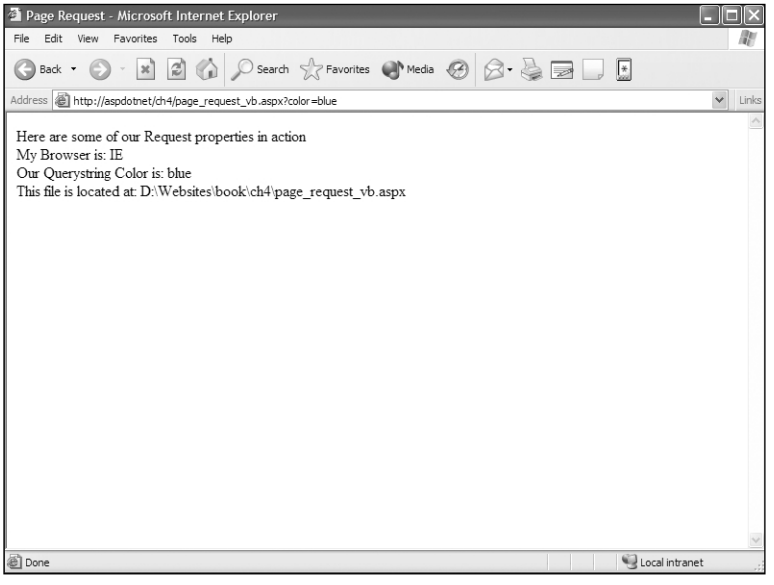

FIGURE 4.8

The `HTTPRequest` *object enables you to retrieve data either passed to a page or submitted by the user.*

There are times when data that you request is a property like the `Request.Browser.Browser` property. This may look redundant with `Browser.Browser` being in the `Request`, but remember that you are dealing with an addressing system here, or a namespace. The first "Browser" is a property of the page and

is an instance of the `HTTPBrowserProperties` object. That object has a property called `Browser`, and hence the `Browser.Browser` in the `Request`. It might have made more sense if I had written `Request.Browser.JavaScript`, which would have returned a Boolean true or false depending on whether a browser is JavaScript-enabled.

The second request type that you will see is called a *collection* and requires you to identify what you're looking for. With the `QueryString`, I knew I was looking for "color" and requested that from the `QueryString` collection. The `Request.Form` collection operates the same way. The contents of the `Form` and `QueryString` generally are determined by the programmers of the website, and you can pass whatever name/value pairs you would like through these methods.

With the `ServerVariables`, you need to specify the variable that you are requesting, but these aren't programmer-defined. The available variables are determined in ASP.NET and are a bit too numerous to list and describe in detail in this book. I would recommend that you investigate the .NET Framework SDK for more information on `ServerVariables`.

Now that you've seen how to get information from the user and how to request all types of different data, you need to be able to respond to those requests. ASP.NET has very generously provided an instance of the `HTTPResponse` object as a page property called `Response`.

Response

Every time that Bernice or Ethel asked a question or made a request, the other had a response. When Ethel asked:

> "Bernice, do you want to see my latest masterpiece? It's a deep dish apple pie done to perfection."

Bernice had a response:

> "Certainly Ethel. Let me have a look see."

There was a conversation. This is what the `Request` and its counterpart the `Response` allow us to do: carry on a conversation with the visitors of the web sites we create.

Table 4.2 lists a few of the common `Response` object's properties and methods.

TABLE 4.2 *Response* Object Properties and Methods

Response	Description
`Buffer`	This controls the flow of data from the server to the user's browser. When the buffer is set to true, which it is by default, data isn't sent to the browser until the server completely processes the page. When the buffer is set to false, data is sent to the browser as soon as it's processed and doesn't wait for the entire page to process.
`Clear()`	Use this when the buffer is set to true and you want to get rid of everything processed up to where the `Clear()` is.
`End()`	Use this when the buffer is set to true and you want to send to the browser what you've processed up to the point where the `End()` is and stop processing the page at that point.
`Flush()`	Use this when the buffer is set to true and you want to send to the browser what you've processed up to the point where the `Clear()` is.
`Redirect()`	Allows you to send the user to a new URL.
`Write()`	Allows you to write information into ASP.NET pages.
`WriteFile()`	Writes the specified file directly to an ASP.NET page.

I have this really great news; do you want to hear it? Nah. I'm not gonna tell you 'til I know everything. I want to wait 'til I know the whole story first before I let you know. Well…maybe I'll tell you what I know so far.

UGGHHHHHH!!! It's all over!! I don't want to talk about it anymore. I can't go on. I can't even face it anymore. I'm ruined…I'm ruined ……I'm ruined.

Dramatic, huh? In a feeble and yet to be determined successful manner I was trying to parallel how the `Buffer` and some of its methods would work in a practical application. If you want to hold back information from being delivered, you set the page's `Buffer` property to `True`. Then you can manipulate the page however you want as you progress. Look at an example of the `HttpResponse` object in action. You are using an object in the `System.Threading` namespace to pause the pages processing using a method called `Sleep`. You will hardly ever have need for this, but it helps me to demonstrate some of the `Buffer` features of the `HttpResponse` object.

Visual Basic .NET

```
<%@ page language="vb" runat="server" buffer="true"%>
<%@ import namespace="System.Threading"%>
<html>
<head>
<title>Hi</title>
</head>
<body>
<%
Response.Write(DateTime.Now + " The First Response.Write() has executed.<BR>" )
Response.Flush()
Thread.Sleep(5000)
Response.Write(DateTime.Now + " The Second Response.Write() has executed.<BR>" )
Response.Flush()
Thread.Sleep(5000)
Response.Write(DateTime.Now + " The Third Response.Write() has executed.<BR>" )
Response.Clear()
Thread.Sleep(5000)
Response.Write(DateTime.Now + " The Fourth Response.Write() has executed.<BR>" )
Response.End()
Response.Write("Where does this text go?")
%>
</body>
</html>
```

C#

```
<%@ page language="c#" runat="server" buffer="true"%>
<%@ import namespace="System.Threading"%>
<html>
<head>
<title>Hi</title>
</head>
<body>
<%
Response.Write(DateTime.Now + " The First Response.Write() has executed.<BR>" );
Response.Flush();
Thread.Sleep(5000);
Response.Write(DateTime.Now + " The Second Response.Write() has executed.<BR>" );
Response.Flush();
Thread.Sleep(5000);
Response.Write(DateTime.Now + " The Third Response.Write() has executed.<BR>" );
Response.Clear();
Thread.Sleep(5000);
Response.Write(DateTime.Now + " The Fourth Response.Write() has executed.<BR>" );
Response.End();
Response.Write("Where does this text go?");
%>
</body>
</html>
```

In Figure 4.9 you can see that three out of the five `Response.Write()` commands were rendered to the browser. Notice the times at which they were delivered to the browser. Can you see which two `Writes` are missing?

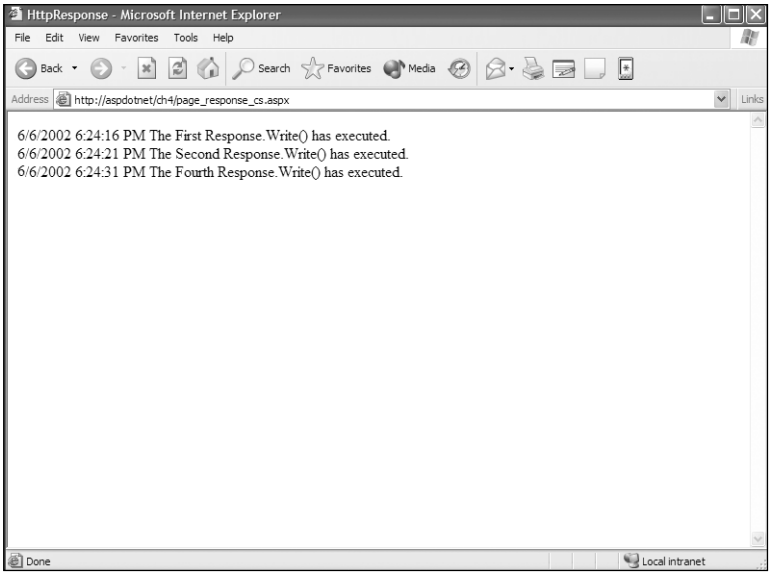

FIGURE 4.9

The `HTTPResponse` *object offers several ways to respond and to control what is output to ASP.NET pages.*

The first `Write` happens and is delivered to the browser when you execute the `Response.Flush()` method as is the second one with a lapse of five seconds between when the first and second `Write`. Notice that the third `Write` is missing from the browser window, but why? Simple. The `Response.Clear()` method disposed of everything that was still in the buffer. In other words, the buffer is like a container in which processed code is stored. If you use a flush, you dump the bucket to the browser. If you use a `Clear()`, you dump the bucket to the trash never to be seen again. Later, a `Response.End()` is used, which is just like a `clear()` except it completely stops page processing. Look at the generated HTML for this page. Do you notice anything peculiar about the tags that came after the `Response.End()`?

```
<html>
<head>
<title>HttpResponse</title>
</head>
<body>
2/6/2002 7:06:39 PM The First Response.Write() has executed.<BR>
2/6/2002 7:06:44 PM The Second Response.Write() has executed.<BR>
2/6/2002 7:06:54 PM The Fourth Response.Write() has executed.<BR>
```

The closing `</body>` and `</html>` tags are missing because the `Response.End()` isn't specific to stopping the processing of just ASP.NET but the entire HTTP stream of data. It basically dumps the bucket at that point and that's it. Nothing else gets processed.

What is a practical application for the buffer? One place we use it all the time and is a good example is on check-out pages of commerce applications we build. We have clients who send their credit card transactions to a third-party processing company and rely on this third-party processor's server to respond back to us indicating whether the card is accepted or declined. During this period of waiting, however, the visitor needs to be kept busy. Our technique is to build an HTML layer with a message that says "We're now processing your card." Then we flush the Buffer (dump the bucket) and send that message to the visitor's browser. Next, we send the credit card information to the third-party company and wait. When we get a reply from the transaction company we build code to hide the layer and do some fancy-schmancy stuff in the back end database depending on whether the card was accepted or declined. We either build a receipt page or a page with a message as to why the transaction was declined. Then we flush the buffer again and deliver the rest of what's in the bucket.

If we didn't have the buffer, the visitor would either be faced with a page that sat idle with no evidence of any activity and hope something was happening, or we'd be forced to jump through flaming hoops and redirect to different pages a bunch of times to accomplish the same thing. And visitors do not enjoy unexplained screen flashes and redirections. With the buffer, we can display a screen telling them what is happening and then deliver a response without any confusion.

I also want to add a thing or two about the `Response.Write` method. This was a highly used and abused method of writing data to traditional ASP pages. It was one of the only ways to dynamically write things into ASP pages.

This is not the case anymore, and with the proliferation of server controls such as the label used so far to display the example results, you will find yourself using `Response.Write` less and less. It is actually a bit out of place in the ASP.NET programming language because it really operates under the more traditional in-line, interpreted model than the event-oriented environment of ASP.NET. The `Response.Write` method doesn't allow you to affect other objects; it just does what it does, where it is, and that is that.

I must also mention the `Response.Redirect` method, as well. This is an oft-used method for transporting the user to another page or URL. The syntax for using this method is to simply place the desired location within the parentheses of the method call, enclosed in quotes if it is a string like the following example.

```
Response.Redirect("anotherpage.asp")
```

You can only use `Response.Redirect` if the HTML stream hasn't begun. In other words, if the buffer has begun to send HTML code to the browser, or if HTML code has been generated at all, using the `Response.Redirect` method causes an error.

As you can see, the `Request` and `Response` objects help you understand and carry on a conversation with the users of your web applications. These are critical tools in your toolbox and are objects you will be seeing over and over throughout the remainder of this book, as well as in your experience in programming in this wonderful language called ASP.NET.

Understanding Directives

ASP.NET directives can simply be described as instructions and settings used to describe how ASP.NET web form pages (.aspx) or User control (.ascx) pages will be processed by the .NET Framework. They are little do-dads that we insert in our ASP.NET pages to control how a page acts or the characteristics it has.

These directives aren't very different from a person's personality traits. Some people are shy, whereas others are outspoken. Some like loud, raucous music, whereas others enjoy the soft sounds of folk music. Some people can compute complicated math problems in their heads, and others can write wondrous stories that capture the imagination.

ASP.NET directives help you give your pages personality and capabilities. You can cause one page to behave in one fashion, whereas another page responds to the same stimulus in a totally different way. One page can have one set of skills or capabilities, whereas another has a totally different set. Directives give us this power.

Table 4.3 shows a list of directives that you will commonly use, but this isn't an exhaustive list. As is true with many aspects of the .NET Framework, the full scope of directives is beyond the scope of this book, but you can see the full list of directives, their functions and their players, in the .NET Framework SDK at the following link:

```
ms-help://MS.NETFrameworkSDK/cpgenref/html/cpconpagedirectives.htm
```

TABLE 4.3 Directives

Directive	Description
`@Page`	This defines page-specific attributes used in ASP.NET pages (.aspx).
`@Control`	This defines control-specific attributes used in ASP.NET User controls (.ascx).
`@Import`	Imports a namespace into a page or User control.
`@Register`	Creates an alias to User controls, allowing them to be rendered to the browser when they are included in a requested page.

@Page

`@Page` directives allow you to set attributes that directly affect your ASP.NET pages that end with the .aspx extension. There are 26 directives available, and you can explore the exhaustive list in the SDK at the link I provided a few paragraphs back, but Table 4.4 contains some of the more commonly used `@page` directives.

TABLE 4.4 *@Page* Directives

Directive	Description
`Buffer`	As seen in the previous section on the `Response` object, this sets whether the Buffer is enabled or not. This is true by default and would need to be set explicitly to false.
`EnableSessionState`	This defines Session-State requirements for the page. If this is set to true (default) then Session-State is enabled; if false Session-State is disabled; or if ReadOnly then Session state can be read but not changed.
`EnableViewState`	This direct controls whether `ViewState(Maintain GUI state)` is maintained across page requests. It is true by default.

continues

TABLE 4.4 Continued

Directive	Description
`Explicit`	Specifies whether all variables on a page must be declared using a `Dim`, `Private`, `Public`, or `ReDim` statement in Visual Basic .NET. The default is false.
`Inherits`	Defines a class from a code-behind page that a page should inherit. (Don't worry, this will be explained is a few short minutes.)
`Language`	Informs the .NET Framework of the language used for all inline(`<% %>`) and server-side script blocks within the ASP.NET page.
`Src`	This contains the name of the source file for an included code-behind page.
`Trace`	This indicates whether tracing is enabled or not. True is enabled; otherwise false. False is the default. Tracing is a debugging function in ASP.NET that allows you to see diagnostic and debug information about an ASP.NET page.

Some of these may seem a bit Greek to you, but you will be seeing many of them throughout the book, and they will be more clear to you then. For the time being, just keep this section in mind and look at it as a seed we're planting that will spring back to your memory later in the book.

The following is a small example of what an `@Page` directive looks like:

```
<%@ page language="c#" buffer="true" Explicit="true" runat="server" %>
```

@Control

The `@Control` directive operates in exactly the same way the `@Page` directive does, except that it is used specifically for User controls only. The next chapter discusses User controls in depth, but if you remember from Chapter 1, a User control is ASP.NET's answer to Server-Side Include files.

The `@Control` directive uses many of the same directives that the `@Page` directive uses. From the directives I included for `@Page`, `@Control` uses all of them except `Buffer`, `EnableSessionState`, and `Trace`. If you think about it, this makes perfect sense because these are attributes that affect an entire page and can't be applied to small sections of included documents, like a User control.

@Import

The `@Import` directive is used to allow you to import a namespace (group of class objects) into your ASP.NET page, making all its classes available for use on your page. You can use this function to import namespaces that are part of the .NET Framework, or you can also use it to import namespaces that are user-defined.

The following namespaces are automatically imported into your ASP.NET pages, so all the class objects within these namespaces are available without you having to explicitly import them.

- `System`
- `System.Collections`
- `System.Collections.Specialized`
- `System.Configuration`
- `System.IO`
- `System.Text`
- `System.Text.RegularExpressions`
- `System.Web`
- `System.Web.Caching`
- `System.Web.Security`
- `System.Web.SessionState`
- `System.Web.UI`
- `System.Web.UI.HtmlControls`
- `System.Web.UI.WebControls`

As I said before, the `@Import` directive can be used to import existing .NET Framework namespaces and user-defined namespaces as well. In an earlier chapter, you created a class (blueprint) for a ball. Also, I've outlined creating a user-defined class in Appendix B of this class in the namespace called `Peter.Toybox`. The following is an example of how to import this namespace, as well as of the .NET Framework namespace called `System.Data`.

```
<%@Import Namespace="Peter.Toybox" %>
<%@Import Namespace="System.Data" %>
```

@Register

The `@Register` directive is a powerful tool used for dynamically creating tags that represent your User controls.

WHAT? Can you repeat that in English?

First let me say that the next chapter shreds through User control in depth. We'll cover them from soup to nuts but for now try to remember that they are like Server-Side Includes except much cooler.

Let's imagine I create User controls for my site that will be used as my page's header and footer. These files will have an .ascx file extension to identify them as User controls. The `@Register` directive allows me to create a tag name and tag prefix with which my controls are associated, and it allows me to drop my User controls into my page with a single simple tag. I will call my header `header.ascx` and my footer `footer.ascx` for this example.

```
<%@Page Language="vb" runat="server"%>
<%@Register TagPrefix="OurTag" Tagname="Header" Src="header.ascx"%>
<%@Register TagPrefix="OurTag" Tagname="Footer" Src="footer.ascx"%>
<html>
<head>
<title>@Register</title>
</head>
<body>
<OurTag:Header id="OurHeader" runat="server"/>
Here is our pages content
<OurTag:Footer id="OurFooter" runat="server"/>
</body>
</html>
```

As you can see, I assign a `TagPrefix`, `Tagname`, and `Src` to define what file I'm attaching to `TagPrefix:Tagname`. As you can see, the `TagPrefix` and `Tagname` are separated by a colon. The `TagPrefix` can be used as many times as you want, and is set up this way so you can easily identify your own tags within your code. It's a simple yet powerful way to use User controls.

You can see how these different directives help you to shape and form your ASP.NET pages and control how they look, act and feel to your end user. We will be exploring and using in more abundance as time and this book progresses.

Code Separation and Using Code-Behind

As we've discussed before, ASP.NET has provided a really cool way to keep code separate by changing the paradigm from an inline, interpreted language to an object/event-oriented paradigm. But ASP.NET doesn't stop there. It provides another level of separation of code and content if you want to use it. It's called the *code-behind* technique.

This subject was mentioned briefly in Chapter 1 and also appeared in a little example, but here the code-behind technique is dissected so you can fully understand what's happening.

The following is an example of a typical ASP.NET page that doesn't utilize any code-behind techniques. It has an `ASP:Label` control that you're familiar with, a few server controls, and a button with a server-side function attached to its `onClick` event. It's nothing very complicated and its results are pretty predictable, as you can see in Figure 4.10.

Visual Basic .NET—**`codebehind_not_vb.aspx`**

```
<%@ page language="vb" runat="server"%>
<script runat="server">
Sub SubmitThis( s As Object, e As EventArgs )
    myLabel.Text = "You Selected: " + myList.SelectedItem.Text + " and " +
    ➥myDropdown.SelectedItem.Value
End Sub

Sub Page_Load
    myLabel.Text = ""
End Sub
</script>
<html>
<head><title>Not Codebehind</title></head>
<body>
<form Runat="Server">
<asp:Label ID="myLabel" forecolor="Red" font-bold="True" MaintainState="False"
➥Runat="Server" />
<br>
<asp:ListBox ID="myList" Runat="Server">
 <asp:ListItem Text="Red" />
 <asp:ListItem Text="Green" />
 <asp:ListItem Text="Blue" />
</asp:ListBox>
<br>
```

continues

Visual Basic .NET (continued)

```
<asp:DropDownList ID="myDropdown" Runat="Server">
 <asp:ListItem Text="One" />
 <asp:ListItem Text="Two" />
 <asp:ListItem Text="Three" />
</asp:DropDownList>
<br>
<asp:Button Text="Place Order" OnClick="SubmitThis" Runat="Server" />
</form>
</body>
</html>
```

C#—codebehind_not_cs.aspx

```
<%@ page language="c#" runat="server"%>
<script runat="server">

public void SubmitThis(object s, EventArgs e ){
        myLabel.Text = "You Selected: " + myList.SelectedItem.Text + " and " +
        ➥myDropdown.SelectedItem.Value;
    }
void Page_Load(){
        myLabel.Text = "";
    }
</script>

<html>
<head><title>Not Code Behind</title></head>
<body>
<form Runat="Server">

<asp:Label ID="myLabel" forecolor="Red" font-bold="True" MaintainState="False"
➥Runat="Server" />
<br>
<asp:ListBox ID="myList" Runat="Server">
 <asp:ListItem Text="Red" />
 <asp:ListItem Text="Green" />
 <asp:ListItem Text="Blue" />
</asp:ListBox>
<br>
<asp:DropDownList ID="myDropdown" Runat="Server">
 <asp:ListItem Text="One" />
 <asp:ListItem Text="Two" />
 <asp:ListItem Text="Three" />
</asp:DropDownList>
<br>
<asp:Button Text="Place Order" OnClick="SubmitThis" Runat="Server" />

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

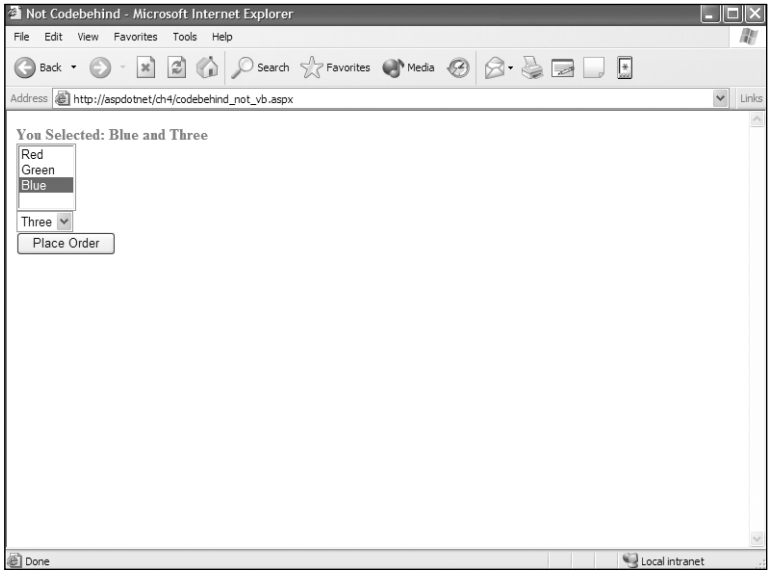

FIGURE 4.10

A predictable and typical ASP.NET page not utilizing any code-behind techniques.

The following is an example of the preceding page and function, but it uses the code-behind technique to separate logical or programming code from content or display code. The ASP.NET page ends with the .aspx extension. This is the content page. This simple example has the same label, a list box control, a drop-down list control, and a form button on the page. As you can see, there is no programming code on this page, with the exception of the `@page` directive that describes what class is inherited from the code-behind page and the name of the file that is the code-behind page.

Visual Basic .NET—`codebehind_vb.aspx`

```
<%@ page inherits="vbsamplecodebehind" src="codebehind_vb.vb" %>
<html>
<head><title>Code Behind</title></head>
<body>
<form Runat="Server">

<asp:Label ID="myLabel" forecolor="Red" font-bold="True" MaintainState="False"
➥Runat="Server" />

<asp:ListBox ID="myList" Runat="Server">
 <asp:ListItem Text="Red" />
 <asp:ListItem Text="Green" />
 <asp:ListItem Text="Blue" />
</asp:ListBox>
```

continues

Visual Basic .NET—(continued)

```
<br>
<asp:DropDownList ID="myDropdown" Runat="Server">
 <asp:ListItem Text="One" />
 <asp:ListItem Text="Two" />
 <asp:ListItem Text="Three" />
</asp:DropDownList>
<br>
<asp:Button Text="Place Order" OnClick="SubmitThis" Runat="Server" />
</form>
</body>
</html>
```

C#—**codebehind_cs.aspx**

```
<%@ page inherits="cssamplecodebehind" src="codebehind_cs.cs" %>
<html>
<head><title>Code Behind</title></head>
<body>
<form Runat="Server">

<asp:Label ID="myLabel" forecolor="Red" font-bold="True" MaintainState="False"
➥Runat="Server" />

<asp:ListBox ID="myList" Runat="Server">
 <asp:ListItem Text="Red" />
 <asp:ListItem Text="Green" />
 <asp:ListItem Text="Blue" />
</asp:ListBox>
<br>
<asp:DropDownList ID="myDropdown" Runat="Server">
 <asp:ListItem Text="One" />
 <asp:ListItem Text="Two" />
 <asp:ListItem Text="Three" />
</asp:DropDownList>
<br>

<asp:Button Text="Place Order" OnClick="SubmitThis" Runat="Server" />

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

The line highlighted in bold shows an example of the `Inherits` directive and `Src` directive in the `@Page` portion of the previously discussed directives. The source of the code-behind page is described in the `Src` value, and the value of `inherits` describes the class name that you will see in the following code-behind pages. Notice that the class name in the `@Page` directive is identical to the class name in the code-behind page.

Visual Basic .NET—**codebehind_vb.vb**

```
Imports System
Imports System.Web.UI
Imports System.Web.UI.WebControls
Imports System.Web.UI.HtmlControls


Public Class vbsamplecodebehind
    Inherits Page

    Protected WithEvents myLabel as Label
    Protected WithEvents myList as ListBox
    Protected WithEvents myDropdown as DropDownList

    Sub SubmitThis( s As Object, e As EventArgs )
        myLabel.Text = "You Selected: " + myList.SelectedItem.Text + " and " +
        ➥myDropdown.SelectedItem.Value

    End Sub

    Sub Page_Load
        myLabel.Text = ""
    End Sub
End Class
```

C#—**codebehind_cs.cs**

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


public class cssamplecodebehind  : System.Web.UI.Page
    {
    protected Label myLabel;
    protected ListBox myList;
    protected DropDownList myDropdown;

    public void SubmitThis(object s, EventArgs e )
    {
        myLabel.Text = "You Selected: " + myList.SelectedItem.Text + " and " +
        ➥myDropdown.SelectedItem.Value;
    }



    void Page_Load()
    {
        myLabel.Text = "";
    }
}
```

You can see in Figure 4.11 that you will get the exact results you'd expect, and to the user the functionality of the page is identical to that of the page that doesn't use code-behind.

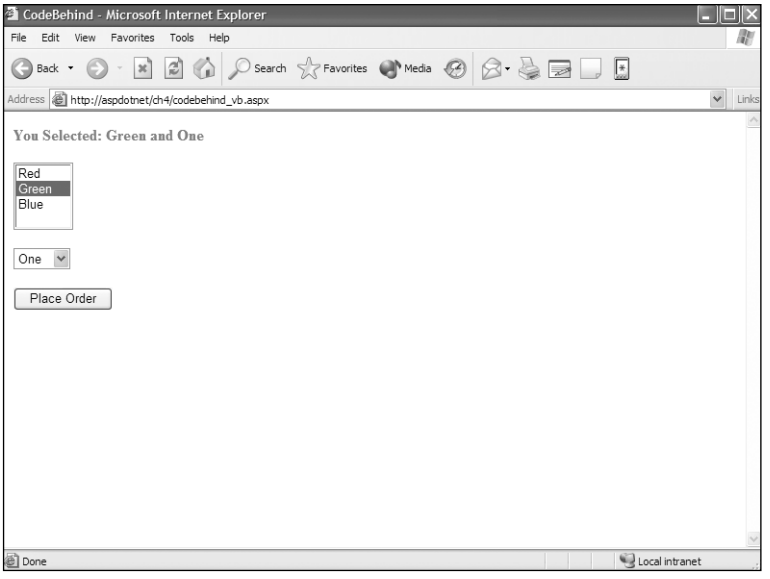

FIGURE 4.11

The end user won't know whether you are using code-behind pages or not. The result to them will be the same.

It is very cool that you can separate code out even further than just by having all the logic at the top of the page and the content or display aspects after that, but that doesn't come without a bit of a price.

Like every convenience, it comes at a cost, and with regard to code-behind the cost comes in the form of more typing. I explain as I go through the highlighted portions of the code-behind example.

First (though not part of the price) I'd like you to take notice of the classname called either `vbsamplecodebehind` or `cssamplecodebehind`, depending on the language you use. This is the class that is created in the code-behind page and what is inherited in the `@page` directive in the ASP.NET page with the .aspx extension.

Now on to the price you pay for the additional layer of separation provided by code-behind. First, code-behind pages don't automatically import any namespaces, as .aspx pages do, so you must explicitly include them into your code-behind pages. In Visual Basic .NET you use the `Imports` keyword to do this and in C# you use the `using` keyword.

Second, you must create a class to contain the logical code in your code-behind page so that you can inherit this class into your .aspx page.

And third, you must create instances of every object that is in your .aspx page that you will reference in your code-behind page. You do this in the same way that you would create any other object, but you must assure that the name of the object you create in your code-behind page is identical to the ID of your object on the .aspx page.

If your pages are very complex, the number of namespaces and objects you need to create in your code-behind pages can be extensive, and the extra work can outweigh the advantage of being able to separate out your code this way. You'll need to consider this, of course, as part of a personal or team decision on which method you'll use to create your ASP.NET pages, but it's nice to have the options.

Also note that the extension of the code-behind page is either .vb or .cs, depending on what language your code-behind page contains. This extension is what tells the .NET Framework what language your code-behind page contains. Another small nugget of code-behind is that you aren't forced to use the same language your .aspx file uses in your code-behind pages. In other words, you could have easily used a C# code-behind page in a Visual Basic .NET .aspx page or vice-versa. As long as the classname and source in the .aspx page's `@page` directive correctly match the class of our code-behind page, your page will work properly.

Summary

Now that you've explored ASP.NET pages and dug into some of their core functions, you can now move onto more of the tools provided by ASP.NET to simplify design features and aspects. (You didn't think this whole book was going to be about how to code in ASP.NET.) In the next chapter you're going to explore ASP.NET User controls and see how they open up a whole new world for you as a designer.