The SitePoint Forums

If you’d like to communicate with us or anyone else on the SitePoint publishing team about this book, you should join SitePoint’s online community. The .NET forum, in particular, can offer an abundance of information above and beyond the solutions in this book.

In fact, you should join that community even if you don’t want to talk to us, because a lot of fun and experienced web designers and developers hang out there. It’s a good way to learn new stuff, get questions answered in a hurry, and just have fun.

The SitePoint Newsletters

In addition to books like this one, SitePoint publishes free email newsletters including The SitePoint Tribune and The SitePoint Tech Times. In them, you’ll read about the latest news, product releases, trends, tips, and techniques for all aspects of web development. If nothing else, you’ll find useful ASP.NET articles and tips, but if you’re interested in learning other technologies, you’ll find them especially valuable. Sign up to one or more SitePoint newsletters at http://www.sitepoint.com/newsletter/.

Your Feedback

If you can’t find your answer through the forums, or if you wish to contact us for any other reason, the best place to write is books@sitepoint.com. We have a well-manned email support system set up to track your inquiries, and if our support staff members are unable to answer your question, they will send it straight to us. Suggestions for improvements, as well as notices of any mistakes you may find, are especially welcome.

Acknowledgements

First and foremost, I’d like to thank the SitePoint team for doing such a great job in making this book possible, for being understanding as deadlines inevitably slipped past, and for the team’s personal touch, which made it a pleasure to work on this project.

---

2 http://www.sitepoint.com/forums/
won’t be the case with the ASP.NET scripts you’ll see through the rest of this book.

Once your new virtual directory has been created, you can see and configure it through the Internet Information Services management console shown in Figure 1.8. You can see the folder’s contents in the right-hand panel.

As index.htm is one of the default document names, you can access that page just by entering http://localhost/Learning/ into your browser’s address bar. To see and edit the default document names for a virtual directory (or any directory, for that matter), you can right-click the directory’s name in the IIS management console, click Properties, and select the Documents tab. You’ll see the dialog displayed in Figure 1.10.

**Figure 1.10. Default document types for the Learning virtual directory**

![Default Document Types Dialog](image)

By default, when we request a directory without specifying a filename, IIS looks for a page with the name of one of the default documents, such as index.htm or default.htm. If there is no index page, IIS assumes we want to see the contents of the requested location. This operation is allowed only if the Directory Browsing
**Custom Errors**

This option allows you to define your own custom error pages. Rather than presenting the standard error messages that appear within Internet Explorer, you can customize error messages with your company’s logo and messages of your choice.

**ASP.NET**

This tab allows you to configure the options for the ASP.NET applications stored in that folder.

One thing to note at this point is that we can set properties for the Default Web Site node, and choose to have them “propagate” down to all the virtual directories we’ve created.

**Using Cassini**

If you’re stuck using a version of Windows that doesn’t support IIS, you’ll need to make use of Cassini to get your simple ASP.NET web applications up and running. Cassini does not support virtual directories, security settings, or any of IIS’s other fancy features; it’s just a very simple web server that gives you the basics you need to get up and running.

To get started using Cassini:

1. Create a directory called `C:\WebDocs\Learning`, just like the one we created in the section called “Virtual Directories”.

2. Copy `index.htm` into this folder. We first saw `index.htm` in the section called “Using localhost”.

3. Start Cassini by opening `C:\Cassini` (or, if you chose to install Cassini somewhere else, open that folder), then double-click on the file `CassiniWebServer.exe`.

4. Cassini has just three configuration options:

   **Application Directory**
   
   It’s here that your application’s files are stored. Enter `C:\WebDocs\Learning` into this field.

   **Server Port**
   
   Web servers almost always operate on port 80, so we won’t touch this setting.
Server Management Studio Express is a free tool provided by Microsoft to allow you to manage your installation of SQL Server 2005.

To install SQL Server Management Studio Express, follow these steps:


2. This time, download the SQL Server Management Studio Express edition that corresponds to the SQL Server 2005 version that you installed previously.

3. After the download completes, execute the file and follow the steps to install the product.

Once it’s installed, SQL Server Manager Express can be accessed from Start > All Programs > Microsoft SQL Server 2005 > SQL Server Management Studio Express. When executed, it will first ask for your credentials, as Figure 1.12 illustrates.

By default, when installed, SQL Server 2005 Express Edition will only accept connections that use Windows Authentication, which means that you’ll use your Windows user account to log in to the SQL Server. Because you’re the user that installed SQL Server 2005, you’ll already have full privileges to the SQL Server. Click Connect to connect to your SQL Server 2005 instance.
Figure 1.14. Changing server settings with SQL Server Management Studio

database server, you must specify both the name of the computer and the name of the SQL Server instance in the form ComputerName/Instance-Name. You can see this specification back in Figure 1.12 and Figure 1.13, where we’re connecting to an instance called SQLEXPRESS on a computer called VM2.

Installing Visual Web Developer 2005

Visual Web Developer automates many of the tasks that you’d need to complete yourself in other environments, and includes many powerful features. For the first exercises in this book, we’ll recommend you use a simple text editor such as
The `runat="server"` attribute identifies the tag as something that needs to be handled on the server. In other words, the web browser will never see the `<asp:Label/>` tag; when the page is requested by the client, ASP.NET sees it and converts it to regular HTML tags before the page is sent to the browser. It’s up to us to write the code that will tell ASP.NET to replace this particular tag with the current time.

To do this, we must add some script to our page. ASP.NET gives you the choice of a number of different languages to use in your scripts. The two most common languages are VB and C#. Let’s take a look at examples using both. Here’s a version of the page in VB:

```xml
<asp:Label runat="server" id="timeLabel" />
```

Here’s the same page written in C#:

```xml
protected void Page_Load(object sender, EventArgs e) {
    timeLabel.Text = DateTime.Now.ToString();
}
```
So far, you’ve learned what ASP.NET is, and what it can do. You’ve installed the software you need to get going, and, having been introduced to some very simple form processing techniques, you even know how to create a simple ASP.NET page. Don’t worry if it all seems a little bewildering right now, because, as this book progresses, you’ll learn how to use ASP.NET at more advanced levels.

As the next few chapters unfold, we’ll explore some more advanced topics, including the use of controls, and various programming techniques. But before you can begin to develop applications with ASP.NET, you’ll need to understand the inner workings of a typical ASP.NET page—with this knowledge, you’ll be able to identify the parts of the ASP.NET page referenced in the examples we’ll discuss throughout this book. So, in this chapter, we’ll talk about some key mechanisms of an ASP.NET page, specifically:

- page structure
- view state
- namespaces
- directives
Figure 2.1. The life cycle of the ASP.NET page

Figure 2.2. The parts of an ASP.NET page
In VB code, a single quote or apostrophe (') indicates that the remainder of the line is to be ignored as a comment.

In C# code, two slashes (//) achieve the same end. C# code also lets us span a comment over multiple lines if we begin it with /* and end it with */, as in this example:

```csharp
C#
<script runat="server">
    void mySub()
    {
    /* Multi-line
         comment */
    }
</script>
```

Before .NET emerged, ASP also supported such script tags using a runat="server" attribute. However, they could only ever contain VBScript and, for a variety of reasons, they failed to gain favor among developers.

Code declaration blocks are generally placed inside the head of your ASP.NET page. The sample ASP.NET page shown in Figure 2.2, for instance, contains the following code declaration block:

```xml
Visual Basic

<script runat="server">
    Sub Page_Load()
        messageLabel.Text = "Hello World"
    End Sub
</script>
```

Perhaps you can work out what the equivalent C# code would be:

```csharp
C#

<script runat="server">
    void Page_Load()
    {
        messageLabel.Text = "Hello World";
    }
</script>
```

The <script runat="server"> tag also accepts two other attributes. We can set the language that’s used in this code declaration block via the language attribute:
These code blocks simply declare a String variable called `Title`, and assign it the value This is generated by a code render block.

Inline expression render blocks can be compared to `Response.Write` in classic ASP. They start with `<%=` and end with `%>`, and are used to display the values of variables and methods on a page. In our example, an inline expression appears immediately after our inline code block:

```
<%= Title %>
```

If you’re familiar with classic ASP, you’ll know what this code does: it simply outputs the value of the variable `Title` that we declared in the previous inline code block.

**ASP.NET Server Controls**

At the heart of any ASP.NET page lie server controls, which represent dynamic elements with which your users can interact. There are three basic types of server control: ASP.NET controls, HTML controls, and web user controls.

Usually, an ASP.NET control must reside within a `<form runat="server">` tag in order to function correctly. Controls offer the following advantages to ASP.NET developers:

- They give us the ability to access HTML elements easily from within our code: we can change these elements’ characteristics, check their values, or even update them dynamically from our server-side programming language of choice.

- ASP.NET controls retain their properties thanks to a mechanism called **view state**. We’ll be covering view state later in this chapter. For now, you need to know that view state prevents users from losing the data they’ve entered into a form once that form has been sent to the server for processing. When the response comes back to the client, text box entries, drop-down list selections, and so on, are all retained through view state.

- With ASP.NET controls, developers are able to separate a page’s presentational elements (everything the user sees) from its application logic (the dynamic portions of the ASP.NET page), so that each can be considered separately.

- Many ASP.NET controls can be “bound” to the data sources from which they will extract data for display with minimal (if any) coding effort.
As you learned at the end of the last chapter, one of the great things about using ASP.NET is that we can pick and choose which of the various .NET languages we like. In this chapter, we’ll look at the key programming principles that will underpin our use of Visual Basic and C#. We’ll start by discussing some basic concepts of programming ASP.NET web applications using these two languages. We’ll explore programming fundamentals such as variables, arrays, functions, operators, conditionals, loops, and events, and work through a quick introduction to object oriented programming (OOP). Next, we’ll dive into namespaces and address the topic of classes—seeing how they’re exposed through namespaces, and which ones you’ll use most often.

The final sections of the chapter cover some of the ideas underlying modern, effective ASP.NET design, including code-behind and the value it provides by helping us separate code from presentation. We finish with an examination of how object oriented programming techniques impact the ASP.NET developer.

Programming Basics

One of the building blocks of an ASP.NET page is the application logic: the actual programming code that allows the page to function. To get anywhere with ASP.NET, you need to grasp the concept of events. All ASP.NET pages will contain controls such as text boxes, checkboxes, and lists. Each of these controls
We can also declare and/or initialize a group of variables of the same type simultaneously. This practice isn’t recommended, though, as it makes the code more difficult to read.

```
Visual Basic
Dim carType As String, carColor As String = "blue"

C#
string carType, carColor = "blue";
```

Table 3.1 lists the most useful data types available in VB and C#.

### Table 3.1. A list of commonly used data types

<table>
<thead>
<tr>
<th>VB</th>
<th>C#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>int</td>
<td>whole numbers in the range -2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>Decimal</td>
<td>decimal</td>
<td>numbers up to 28 decimal places; this command is used most often when dealing with costs of items</td>
</tr>
<tr>
<td>String</td>
<td>string</td>
<td>any text value</td>
</tr>
<tr>
<td>Char</td>
<td>char</td>
<td>a single character (letter, number, or symbol)</td>
</tr>
<tr>
<td>Boolean</td>
<td>bool</td>
<td>true or false</td>
</tr>
<tr>
<td>Object</td>
<td>object</td>
<td>a generic type that can be used to refer to objects of any type</td>
</tr>
</tbody>
</table>

You’ll encounter many other data types as you progress, but this list provides an overview of the ones you’ll use most often.

**Many Aliases are Available**

These data types are the VB- and C#-specific aliases for types of the .NET Framework. For example, instead of Integer or int, you could use `System.Int32` in any .NET language; likewise, instead of Boolean or bool, you could use `System.Boolean`, and so on.
string getName()
{
  return "Zak Ruvalcaba";
}
// And now we'll use it in the Page_Load handler
void Page_Load()
{
  messageLabel.Text = getName();
}
</script>
</head>
<body>
<form runat="server">
  <asp:Label id="messageLabel" runat="server" />
</form>
</body>
</html>

When the page above is loaded in the browser, a Load event will be raised which will cause the Page_Load event handler to be called, which in turn will call the getName function. Figure 3.4 shows the result in the browser.

**Figure 3.4. Executing an ASP.NET function**

![ASP.NET Functions - Windows Internet Explorer](image)

Here’s what’s happening: the line in our Page_Load subroutine calls our function, which returns a simple string that we can assign to our label. In this simple example, we’re merely returning a fixed string, but the function could just as easily retrieve the name from a database (or somewhere else). The point is that, regardless of how the function gets its data, we call it in just the same way.

When we’re declaring our function, we must remember to specify the correct return type. Take a look at the following code:

Visual Basic

' Here's our function
Function addUp(x As Integer, y As Integer) As Integer
  Return x + y

call which will return an integer during execution. Converting numbers to strings is a very common task in ASP.NET, so it’s good to get a handle on it early.

**Converting Numbers to Strings**

There are more ways to convert numbers to strings in .NET, as the following lines of VB code illustrate:

```vbnet
messageLabel.Text = addUp(5, 2).ToString()
messageLabel.Text = Convert.ToString(addUp(5, 2))
```

If you prefer C#, these lines of code perform the same operations as the VB code above:

```csharp
messageLabel.Text = addUp(5, 2).ToString();
messageLabel.Text = Convert.ToString(addUp(5, 2));
```

Don’t be concerned if you’re a little confused by how these conversions work, though—the syntax will become clear once we discuss object oriented concepts later in this chapter.

**Operators**

Throwing around values with variables and functions isn’t of much use unless you can use them in some meaningful way, and to do so, we need operators. An operator is a symbol that has a certain meaning when it’s applied to a value. Don’t worry—operators are nowhere near as scary as they sound! In fact, in the last example, where our function added two numbers, we were using an operator: the addition operator, or + symbol. Most of the other operators are just as well known, although there are one or two that will probably be new to you. Table 3.2 outlines the operators that you’ll use most often in your ASP.NET development.

**Operators Abound!**

The list of operators in Table 3.2 is far from complete. You can find detailed (though poorly written) lists of the differences between VB and C# operators on the Code Project web site.³

---
This demonstrates that the loop repeats until the condition is no longer met. Try changing the code so that the counter variable is initialized to 20 instead of 0. When you open the page now, you won’t see anything on the screen, because the loop condition was never met.

The other form of the While loop, called a Do While loop, checks whether or not the condition has been met at the end of the code block, rather than at the beginning:

Visual Basic

Sub Page_Load(s As Object, e As EventArgs)
    ' Initialize counter
    Dim counter As Integer = 0
    ' Loop
    Do
        ' Update the label
        messageLabel.Text = counter.ToString()
        ' We use the += operator to increase variable by 1
        counter += 1
    Loop While counter <= 10
End Sub

C# File: Loops.aspx (excerpt)

void Page_Load()
{
    // initialize counter
    int counter = 0;
    // loop
    do
    {
        // Update the label
        messageLabel.Text = counter.ToString();
        // C# has the operator ++ to increase a variable by 1
        counter++;
    } while (counter <= 10);
}

If you run this code, you’ll see it provides the exact same output we saw when we tested the condition before the code block. However, we can see the crucial difference if we change the code so that the counter variable is initialized to 20. In this case, we will, in fact, see 20 displayed, because the loop code is executed once before the condition is even checked! There are some instances when this
is just what we want, so being able to place the condition at the end of the loop can be very handy.

A For loop is similar to a While loop, but we typically use it when we know beforehand how many times we need it to execute. The following example displays the count of items within a DropDownList control called productList:

Visual Basic

```vbnet
Dim i As Integer
For i = 1 To productList.Items.Count
    messageLabel.Text = i.ToString()
Next
```

C#

```csharp
int i;
for (i = 1; i <= productList.Items.Count; i++)
{
    messageLabel.Text = i.ToString();
}
```

In VB, the loop syntax specifies the starting and ending values for our counter variable within the For statement itself.

In C#, we assign a starting value (i = 1) along with a condition that will be tested each time we move through the loop (i <= productList.Items.Count), and identify how the counter variable should be incremented after each loop (i++). While this allows for some powerful variations on the theme in our C# code, it can be confusing at first. In VB, the syntax is considerably simpler, but it can be a bit limiting in exceptional cases.

The other type of For loop is For Each, which loops through every item within a collection. The following example loops through an array called arrayName:

Visual Basic

```vbnet
For Each item In arrayName
    messageLabel.Text = item
Next
```

C#

```csharp
foreach (string item in arrayName)
{
    messageLabel.Text = item;
}
```
This is just a simple example to help you visualize what OOP is all about. In the next few sections, we'll cover properties and methods in greater detail, and talk about classes and class instances, scope, events, and inheritance.

**Properties**

As we’ve seen, properties are characteristics shared by all objects of a particular class. In the case of our example, the following properties might be used to describe any given dog:

- color
- height
- length

In the same way, the more well known ASP.NET Button class exposes properties including:

- Width
- Height
- ID
- Text
- ForeColor
- BackColor

Unfortunately for me, if I get sick of Rayne’s color, I can’t change it in real life. However, if Rayne was a .NET object, we could change any of his properties in the same way that we set variables (although a property can be read-only or write-only). For instance, we could make him brown very easily:

**Visual Basic**

```vbnet
rayne.Color = "Brown"
```

**C#**

```csharp
rayne.Color = "Brown";
```
You’ll remember from the last section that we said our hypothetical **AustralianShepherd** class would inherit from the more general **Dog** class, which, in turn, would inherit from the even more general **Animal** class. This is exactly the kind of relationship that’s being shown in Figure 3.7—**Page** inherits methods and properties from the **TemplateControl** class, which in turn inherits from a more general class called **Control**. In the same way that we say that an Australian Shepherd is an Animal, we say that a **Page** is a **Control**. **Control**, like all .NET classes, inherits from **Object**.

Since **Object** is so important that every other class derives from it, either directly or indirectly, it deserves a closer look. **Object** contains the basic functionality that the designers of .NET felt should be available in any object. The **Object** class contains these public members:

- **Equals**
- **ReferenceEquals**
- **GetHashCode**
- **GetType**
If you save this page in the Learning folder and load it, you'd get the output shown in Figure 4.4.

Figure 4.4. Displaying the default calendar

The Calendar control contains a wide range of properties, methods, and events, including those listed in Table 4.3.
Table 4.3. Some of the Calendar control’s properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DayNameFormat</td>
<td>This property sets the format of the day names. Its possible values are FirstLetter, FirstTwoLetters, Full, and Short. The default is Short, which displays the three-letter abbreviation.</td>
</tr>
<tr>
<td>FirstDayOfWeek</td>
<td>This property sets the day of the week that begins each week in the calendar. By default, the value of this property is determined by your server’s region settings, but you can set this to Sunday or Monday if you want to control it.</td>
</tr>
<tr>
<td>NextPrevFormat</td>
<td>Set to CustomText by default, this property can be set to ShortMonth or FullMonth to control the format of the next and previous month links.</td>
</tr>
<tr>
<td>SelectedDate</td>
<td>This property contains a DateTime value that specifies the highlighted day. You’ll use this property a lot to determine which day the user has selected.</td>
</tr>
<tr>
<td>SelectionMode</td>
<td>This property determines whether days, weeks, or months can be selected; its possible values are Day, DayWeek, DayWeekMonth, and None, and the default is Day. When Day is selected, a user can only select a day; when DayWeek is selected, a user can select a day or an entire week; and so on.</td>
</tr>
<tr>
<td>SelectMonthText</td>
<td>This property controls the text of the link that’s displayed to allow users to select an entire month from the calendar.</td>
</tr>
<tr>
<td>SelectWeekText</td>
<td>This property controls the text of the link that’s displayed to allow users to select an entire week from the calendar.</td>
</tr>
<tr>
<td>ShowDayHeader</td>
<td>If True, this property displays the names of the days of the week. The default is True.</td>
</tr>
<tr>
<td>ShowGridLines</td>
<td>If True, this property renders the calendar with grid lines. The default is True.</td>
</tr>
<tr>
<td>ShowNextPrevMonth</td>
<td>If True, this property displays next/previous month links. The default is True.</td>
</tr>
<tr>
<td>ShowTitle</td>
<td>If True, this property displays the calendar’s title. The default is False.</td>
</tr>
</tbody>
</table>
As you’ve probably noticed by now, the .xml file enables you to specify properties for each banner advertisement by inserting appropriate elements inside each of the Ad elements. These elements include:

**ImageURL**  
the URL of the image to display for the banner ad

**NavigateURL**  
the web page to which your users will navigate when they click the banner ad

**AlternateText**  
the alternative text to display for browsers that do not support images

**Keyword**  
the keyword to use to categorize your banner ad

If you use the **KeywordFilter** property of the AdRotator control, you can specify the categories of banner ad to display.

**Impressions**  
the relative frequency that a particular banner ad should be shown in relation to other banner advertisements

The higher this number, the more frequently that specific banner will display in the browser. The number provided for this element can be as low as one, but cannot exceed 2,048,000,000; if it does, the page throws an exception.

Except for ImageURL, all these elements are optional. Also, if you specify an Ad without a NavigateURL, the banner ad will display without a hyperlink.

To make use of this Ads.xml file, create a new ASP.NET page, called AdRotator.aspx, with the following code:

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"  
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head>
  <title>AdRotator Control</title>
</head>
<body>
  <form runat="server">
    <asp:AdRotator ID="adRotator" runat="server"
...
Figure 4.8. A breadcrumb created using the SiteMapPath control

If you run the example now, you’ll see the breadcrumb appear exactly as it’s shown in Figure 4.8.

Note that the SiteMapPath control shows only the nodes that correspond to existing pages of your site, so if you don’t have a file named Default.aspx, the root node link won’t show up. Similarly, if the page you’re loading isn’t named TreeViewDemo.aspx, the SiteMapPath control won’t generate any output.

Menu

The Menu control is similar to TreeView in that it displays hierarchical data from a data source; the ways in which we work with both controls are also very similar. The most important differences between the two lie in their appearances, and the fact that Menu supports templates for better customization and displays only two levels of items (menu and submenu items).

MultiView

The MultiView control is similar to Panel in that it doesn’t generate interface elements itself, but contains other controls. A MultiView can store more pages of data (called views), and lets you show one page at a time. You can change the active view (the one being presented to the visitor) by setting the value of the
Figure 4.10. A simple form

includes a Label of the specified width, and a TextBox that accepts 20 characters; you’ll then be able to reuse the web user control wherever it’s needed in your project.

In your Learning folder, create a new file named SmartBox.ascx. Then, add the control’s constituent controls—a Label control and a TextBox control—as shown below:

```xml
<p>
<asp:Label ID="myLabel" runat="server" Text="" Width="100" />
<asp:TextBox ID="myTextBox" runat="server" Text="" Width="200"
MaxLength="20" />
</p>
```

Label Widths in Firefox

Unfortunately, setting the Width property of the Label control doesn’t guarantee that the label will appear at that width in all browsers. The current version of Firefox, for example, will not display the above label in the way it appears in Internet Explorer.

To get around this, you should use a CSS style sheet and the CssClass property, which we’ll take a look at later in this chapter.

In Chapter 3 we discussed properties briefly, but we didn’t explain how you could create your own properties within your own classes. So far, you’ve worked with
Just like web forms, web user controls can work with code-behind files, but, in an effort to keep our examples simple, we aren’t using them here. You’ll meet more complex web user controls in the chapters that follow.

When you use the SmartBox control in a form, you can set its label and have the text entered by the user, like this:

**Visual Basic**

```vbnet
mySmartBox.LabelText = "Address:"
userAddress = mySmartBox.Text
```

**C#**

```csharp
mySmartBox.LabelText = "Address:"
userAddress = mySmartBox.Text;
```

Let’s see how we implemented this functionality. In .NET, properties can be read-only, write-only, or read-write. In many cases, you’ll want to have properties that can be both read and write, but in this case, we want to be able to set the text of the inner Label, and to read the text from the TextBox.

To define a write-only property in VB, you need to use the `WriteOnly` modifier. Write-only properties need only define a special block of code that starts with the keyword `Set`. This block of code, called an accessor, is just like a subroutine that takes as a parameter the value that needs to be set. The block of code uses this value to perform the desired action—in the case of the `LabelText` property, that action sets the `Text` property of our `Label` control, as shown below:

**Visual Basic**

```
Public WriteOnly Property LabelText() As String
  Set(ByVal value As String)
    myLabel.Text = value
  End Set
End Property
```

Assuming that a form uses a SmartBox object called `mySmartBox`, we could set the `Text` property of the `Label` like this:

**Visual Basic**

```
mySmartBox.LabelText = "Address:"
```
Loading this page will produce the output we saw in Figure 4.10.

Now, this is a very simple example indeed, but we can easily extend it for other purposes. You can see in the code snippet that we set the LabelText property directly in the control’s tag; we could have accessed the properties from our code instead. Here’s an example:

Master Pages

Master pages are a new feature of ASP.NET 2.0 that can make an important difference in the way we compose web forms. Master pages are similar to web user controls in that they are also composed of HTML and other controls; they can be extended with the addition of events, methods, or properties; and they can’t be loaded directly by users—instead, they’re used as building blocks to design the structure of your web forms.

A master page is a page template that can be applied to give many web forms a consistent appearance. For example, a master page can set out a standard structure
If all the pages in the site have the same header, footer, and navigation menu, it makes sense to include these components in a master page, and to build several web forms that customize only the content areas on each page. We’ll begin to create such a site in Chapter 5, but let’s work through a quick example here.

To keep this example simple, we won’t include a menu here: we’ll include just the header, the footer, and the content placeholder. In your Learning folder, create a new file named FrontPages.master, and write the following code into it:

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head>
<title>Front Page</title>
</head>
<body>
<form id="myForm" runat="server">
<h1>Welcome to SuperSite Inc!</h1>
<asp:ContentPlaceHolder id="FrontPageContent" runat="server" />
<p>Copyright 2006</p>
</form>
</body>
</html>
```

The master page looks almost like a web form, except for one important detail: it has an empty `ContentPlaceHolder` control. If you want to build a web form based on this master page, you just need to reference the master page using the `Page` directive in the web form, and add a `Content` control that includes the content you want to insert.

Let's try it. Create a web form called FrontPage.aspx, and add this code to it:

```html
<%@ Page MasterPageFile="FrontPages.master" %>
<asp:Content id="myContent" runat="server"
  ContentPlaceHolderID="FrontPageContent">
<p>
  Welcome to our web site! We hope you'll enjoy your visit.
</p>
</asp:Content>
```
server controls we discussed in Chapter 4. In the other tabs, you’ll find other controls, including the validation controls we’ll discuss in Chapter 6, which can be found in the Validation tab. Figure 5.13 shows the toolbox with all its tabs in the collapsed state.

Figure 5.13. The collapsed Toolbox tabs

The Properties Window

When you select a control in the web forms designer, its properties are displayed automatically in the Properties window. For example, if you select the TextBox control we added to the form earlier, the properties of that TextBox will display in the Properties window. If it’s not visible, you can make it appear by selecting View > Properties Window.

The Properties window doesn’t just allow you to see the properties—it also lets you set them. Many properties—such as the colors that can be chosen from a palette—can be set visually, but in other cases, complex dialogs are available to
While you’re here, it’s a good idea to check that Default.aspx is included as a default file. If it is, then requesting http://localhost/Dorknozzle will load http://localhost/Dorknozzle/Default.aspx by default. To check this, click the Documents tab. If Default.aspx isn’t in the list, add it by clicking the Add... button and entering the filename, as shown in Figure 5.22.

Finally, click OK to close the Dorknozzle Properties window.

If no default document exists in the Dorknozzle folder, the web server will attempt to return a list of the files and folders inside the Dorknozzle folder—an operation that will only succeed if the Directory Browsing option shown in Figure 5.21 is enabled. If this option is left in its default, disabled state, this operation will result in an error.

Now, if you load http://localhost/Dorknozzle/ using any web browser, you should see a little magic (as Figure 5.23 reveals)!
The project will open. This time, the root entry in Solution Explorer will be http://localhost/Dorknozzle/ instead of c:\WebDocs\Dorknozzle\, as Figure 5.25 indicates.

**Figure 5.25. Solution Explorer displaying an HTTP location**

Visual Web Developer knows how to investigate your IIS location and display its contents automatically in the Solution Explorer. If the folder contents are changed outside of Visual Web Developer, you’ll need to right-click the root node and select “Refresh Folder” in order to refresh Visual Web Developer’s display of the directory’s contents.

**Core Web Application Features**

Let’s continue our exploration of the key topics related to developing ASP.NET web applications. We’ll put them into practice as we move through the book, but in this quick introduction, we’ll discuss:

- `Web.config`
- `Global.asax`
- `user sessions`
- `caching`
- `cookies`

**Web.config**

Almost every ASP.NET web application contains a file named `Web.config`, which stores various application settings. By default, all ASP.NET web applications are
configured in the `Machine.config` file, which contains machine-wide settings, and lives in the `C:\WINDOWS\Microsoft.NET\Framework\version\CONFIG` directory.

For the most part, you won’t want to make any modifications to this file. However, you can override certain settings of the `Machine.config` file by adding a `Web.config` file to the root directory of your application. You may already have this file in your project; if you don’t, you can add one by accessing File > New File..., then selecting Web Configuration File from the dialog that appears.

The `Web.config` file is an XML file that can hold configuration settings for the application in which the file resides. One of the most useful settings that `Web.config` controls is ASP.NET’s debug mode. If you’re using VB, you can enable debug mode by opening `Web.config` and editing the compilation element, which looks like this:

```xml
<compilation debug="false" strict="false" explicit="true" />
```

Enabling debug mode is as simple as changing the value of the `debug` attribute to `true`. The other attributes listed here were added by Visual Web Developer to offer a helping hand to VB developers migrating from older versions. For example, `strict="false"` makes the compiler forgive some of the mistakes we might make, such as using the wrong case in variable names.

If you’re using C#, you’ll need to create the `Web.config` file yourself. Go to File > New File..., then select Web Configuration File from the dialog that appears, and click Add. This will create the default `Web.config` file, which will contain the following section:

```xml
<compilation debug="true" strict="false" explicit="true" />
```
We can use classes from these namespaces in our code without needing to reference them in every file in which they’re used. As you can see, Visual Web Developer tries to offer an extra level of assistance for VB developers, but users of C# (or any other language) could also add these namespace references to Web.config.

You’ll learn more about working with Web.config as you progress through this book, so if you wish, you can skip the rest of these details for now and come back to them later as you need them.

The Web.config file’s root element is always configuration, which can contain three different types of elements:

configuration section groups
As ASP.NET and the .NET framework are so configurable, configuration files could easily become jumbled if we didn’t have a way to break the files into groups of related settings. A number of predefined section grouping tags let you do just that. For example, settings specific to ASP.NET must be placed inside a system.web section grouping element, while settings that are relevant to .NET’s networking classes belong inside a system.net element.

General settings, like the appSettings element we saw above, stand on their own, outside the section grouping tags. In this book, though, our configuration files will also contain a number of ASP.NET-specific settings, which live inside the system.web element.

configuration sections
These are the actual setting tags in our configuration file. Since a single element can contain a number of settings (e.g. the appSettings element we saw earlier could contain a number of different strings for use by the application), Microsoft calls each of these tags a “configuration section.” ASP.NET provides a wide range of built-in configuration sections to control the various aspects of your web applications.

The following list outlines some of the commonly used ASP.NET configuration sections, all of which must appear within the system.web section grouping element:
authentication
outlines configuration settings for user authentication, and is covered in
detail in Chapter 14

authorization
specifies users and roles, and controls their access to particular files
within an application; discussed more in Chapter 14.

compilation
contains settings that are related to page compilation, and lets you specify
the default language that’s used to compile pages

customErrors
used to customize the way errors display

globalization
used to customize character encoding for requests and responses

pages
handles the configuration of options for specific ASP.NET pages; allows
you to disable session state, buffering, and view state, for example

sessionState
contains configuration information for modifying session state (i.e. vari-
ables associated with a particular user’s visit to your site)

trace
contains information related to page and application tracing

configuration section handler declarations
ASP.NET’s configuration file system is so flexible that it allows you to define
your own configuration sections. For most purposes, the built-in configuration
sections will do nicely, but if we wanted to include some custom configuration
sections, we’d need to tell ASP.NET how to handle them. To do so, we’d
declare a configuration section handler for each custom configuration section
we wanted to create. This is pretty advanced stuff, so we won’t worry about
it in this book.

Global.asax

Global.asax is another special file that can be added to the root of an application.
It defines subroutines that are executed in response to application-wide events.
C#
Application.Remove("SiteName");

If you find you have multiple objects and application variables lingering in application state, you can remove them all at once using the `RemoveAll` method:

Visual Basic
Application.RemoveAll()

C#
Application.RemoveAll();

It’s important to be cautious when using application variables. Objects remain in application state until you remove them using the `Remove` or `RemoveAll` methods, or shut down the application in IIS. If you continue to save objects into the application state without removing them, you can place a heavy demand on server resources and dramatically decrease the performance of your applications.

Let’s take a look at application state in action. Application state is very commonly used to maintain hit counters, so our first task in this example will be to build one! Let’s modify the Default.aspx page that Visual Web Developer created for us. Double-click Default.aspx in Solution Explorer, and add a `Label` control inside the `<form>` element. You could drag the control from the Toolbox (in either Design View or Source View) and modify the generated code, or you could simply enter the new code by hand. We’ll also add a bit of text to the page, and change the `Label`’s ID to `myLabel`, as shown below:

In Design View, you should see your label appear inside the text, as shown in Figure 5.27.

Now, let’s modify the code-behind file to use an application variable that will keep track of the number of hits our page receives. Double-click in any empty space on your form; Visual Web Developer will create a `Page_Load` subroutine automatically, and display it in the code editor.
Before analyzing the code, press F5 to run the site and ensure that everything works properly. Every time you refresh the page, the hit counter should increase by one until it reaches ten, when it starts over. Now, shut down your browser altogether, and open the page in another browser. We’ve stored the value within application state, so when you restart the application, the page hit counter will remember the value it reached in the original browser, as Figure 5.28 shows.

If you play with the page, reloading it over and over again, you’ll see that the code increments PageCounter every time the page is loaded. First, though, the code verifies that the counter hasn’t reached or exceeded ten requests. If it has, the counter variable is removed from the application state:
void Session_Start(Object sender, EventArgs e)
{
  Session.Timeout = 1560;
}

Using the Cache Object

In traditional ASP, developers used application state to cache data. Although there’s nothing to prevent you from doing the same thing here, ASP.NET provides a new object, Cache, specifically for that purpose. Cache is also a collection, and we access its contents similarly to the way we accessed the contents of Application. Another similarity is that both have application-wide visibility, being shared between all users who access a web application.

Let’s assume that there’s a list of employees that you’d normally read from the database. To spare the database server’s resources, after you read the table from the database the first time, you might save it into the cache using a command like this:

Visual Basic
Cache("Employees") = employeesTable

C#
Cache["Employees"] = employeesTable;

By default, objects stay in the cache until we remove them, or server resources become low, at which point objects begin to be removed from the cache in the order in which they were added. The Cache object also lets us control expiration—if, for example, we want to add an object to the cache for a period of ten minutes, we can use the Insert method. Here’s an example:

Visual Basic
Cache.Insert("Employees", employeesTable, Nothing, DateTime.MaxValue, TimeSpan.FromMinutes(10))

C#
Cache.Insert("Employees", employeesTable, null, DateTime.MaxValue, TimeSpan.FromMinutes(10));

The third parameter, which in this case is Nothing or null, can be used to add cache dependencies. We could use such dependencies to invalidate cached items.
We’ll keep all the files related to the default appearance of Dorknozzle in this Blue folder.

Creating a New Style Sheet

We’ll start by adding a new CSS file to the Blue theme. CSS files can be created independently of themes, but it’s easier in the long term to save them to themes—this way, your solution becomes more manageable, and you can save different versions of your CSS files under different themes. Any files with the .css extension in a theme’s folder will be automatically linked to any web form that uses that theme.

Right-click the Blue folder, and select Add New Item... Select a New Style Sheet template to create a new file named Dorknozzle.css, and click Add. By default, Dorknozzle.css will be almost empty:

```css
body {
}
```

Let’s make this file more useful by adding more styles to it. We’ll use these styles soon, when we build the first page of Dorknozzle.

```css
body {
  font-family: Tahoma, Helvetica, Arial, sans-serif;
  font-size: 12px;
}

h1 {
  font-size: 25px;
}

a:link, a:visited {
  text-decoration: none;
  color: Blue;
}

a:hover {
  color: Red;
}

.Header {
```
Building the Master Page

This is where the real fun begins! All of the pages in Dorknozzle have a common structure, with the same header on the top, and the same menu on the left, so it makes sense to build a master page. With this master page in place, we’ll be able to create pages for the site by writing only the content that makes them different, rather than writing the header and the menu afresh for each page.

Figure 5.38. Creating a new master page

Right-click again on the root node in Solution Explorer and select Add New Item.... There, select the Master Page template from the list of available templates, and name it Dorknozzle.master. Choose the language you want to program the master page in from the Language drop-down list, and check the Place code in a
Extending Dorknozzle

We’ll extend the Dorknozzle site by adding an employee help desk request web form. This form will allow our fictitious employees to report hardware, software,
In more complex scenarios, if you enter the name of an object, the Watch window will let you explore its members as we just saw.

If you switch to the Locals window (Debug > Windows > Locals) shown in Figure 5.50, you can see the variables or objects that are visible from the line of code at which the execution was paused.

Figure 5.50. The Locals window

Another nice feature of Visual Web Developer is that when you hover your cursor over a variable, the editing window shows you at-a-glance information about that variable.

Sometimes, you’ll want to debug your application even if it doesn’t generate an exception. For example, you may find that your code isn’t generating the output you expected. In such cases, it makes sense to execute pieces of code line by line, and see in detail what happens at each step.

The most common way to get started with this kind of debugging is to set a breakpoint in the code. In Visual Web Developer, we do this by clicking on the gray bar on the left-hand side of the editing window. When we click there, a red bullet appears, and the line is highlighted with red to indicate that it’s a breakpoint, as Figure 5.51 illustrates.

Once the breakpoint is set, we execute the code. When the execution pointer reaches the line you selected, execution of the page will be paused and Visual Web Developer will open your page in debug mode. In debug mode, you can perform a number of tasks:

- View the values of your variables or objects.
- Step into any line of code by selecting Debug > Step Into. This executes the currently highlighted line, then pauses. If the selected line executes another local method, the execution pointer is moved to that method so that you can execute it line by line, too.
the `OnClick` property to the `Button` control, and give it the value `submitButton_Click`. This mimics what Visual Web Developer would do if you double-clicked the button in Design View.

```xml
<!-- Submit Button -->
<p>
    <asp:Button id="submitButton" runat="server" Text="Submit"
        OnClick="submitButton_Click" />
</p>
```

Next, create the `submitButton_Click` subroutine. You can add this between `<script runat="server">` and `</script>` tags in the head of the web form, or place it in a code-behind file. If Visual Web Developer generates these stubs for you, they may appear a little differently than they’re presented here:

Visual Basic

```vbnet
Protected Sub submitButton_Click(s As Object, e As EventArgs)
    submitButton.Text = "Clicked"
End Sub
```

C#

```csharp
protected void submitButton_Click(object sender, EventArgs e)
{
    submitButton.Text = "Clicked";
}
```

Now, if you’re trying to submit invalid data using a browser that has JavaScript enabled, this code will never be executed. However, if you disable your browser’s JavaScript, you’ll see the label on the `Button` control change to `Clicked`! Obviously, this is not an ideal situation—we’ll need to do a little more work to get validation working on the server side.

### Disabling JavaScript in Firefox

To disable JavaScript in Firefox, go to `Tools > Options...`, click the Content tab and uncheck the Enable JavaScript checkbox.

### Disabling JavaScript in Opera

To disable JavaScript in Opera, go to `Tools > Preferences...`, click the Advanced tab, select Content in the list on the left, and uncheck the Enable JavaScript checkbox.
**Disabling JavaScript in Internet Explorer**

To disable JavaScript in Internet Explorer, go to Tools > Internet Options... and click the Security tab. There, select the zone for which you’re changing the settings (the zone will be shown on the right-hand side of the browser’s status bar—it will likely be Local Intranet Zone if you’re developing on the local machine) and press Custom Level.... Scroll down to the Scripting section, and check the Disable radio button for Active Scripting.

ASP.NET makes it easy to verify on the server side if the submitted data complies to the validator rules without our having to write very much C# or VB code at all. All we need to do is to check the Page object’s IsValid property, which only returns True if all the validators on the page are happy with the data in the controls they’re validating. This approach will always work regardless of which web browser the user has, or the settings he or she has chosen.

Let’s add this property to our Click event handler:

**Visual Basic File:** Login.aspx (excerpt)

```vbnet
Protected Sub submitButton_Click(s As Object, e As EventArgs)
    If Page.IsValid Then
        submitButton.Text = "Valid"
    Else
        submitButton.Text = "Invalid!"
    End If
End Sub
```

**C# File:** Login.aspx (excerpt)

```csharp
protected void submitButton_Click(object s, EventArgs e)
{
    if (Page.IsValid)
    {
        submitButton.Text = "Valid";
    }
    else
    {
        submitButton.Text = "Invalid!";
    }
}
```

Load the page again after disabling JavaScript, and press the Submit button without entering any data in the text boxes. The text label on the button should change, as shown in Figure 6.2.
As you've probably noticed, the CompareValidator control differs very little from the RequiredFieldValidator control:

```xml
<asp:RequiredFieldValidator id="confirmPasswordReq" runat="server"
    ControlToValidate="confirmPasswordTextBox"
    ErrorMessage="Password confirmation is required!"
    SetFocusOnError="True" Display="Dynamic" />
<asp:CompareValidator id="comparePasswords" runat="server"
    ControlToCompare="passwordTextBox"
    ControlToValidate="confirmPasswordTextBox"
    ErrorMessage="Your passwords do not match up!"
    Display="Dynamic" />
```

The only difference is that in addition to a ControlToValidate property, the CompareValidator has a ControlToCompare property. We set these two properties to the IDs of the controls we want to compare. So, in our example, the ControlToValidate property is set to the confirmPasswordTextBox, and the ControlToCompare property is set to the passwordTextBox.

The CompareValidator can be used to compare the value of a control to a fixed value, too. CompareValidator can check whether the entered value is equal to, less than, or greater than, any given value. As an example, let's add an age field to our login form:

```xml
<!-- Age -->
<p>
    Age:<br />
    <asp:TextBox id="ageTextBox" runat="server" />
    <asp:RequiredFieldValidator id="ageReq" runat="server"
        ControlToValidate="ageTextBox"
        ErrorMessage="Age is required!"
        SetFocusOnError="True" Display="Dynamic" />
    <asp:CompareValidator id="ageCheck" runat="server"
        Operator="GreaterThan" Type="Integer"
        ControlToValidate="ageTextBox" ValueToCompare="15"
        ErrorMessage="You must be 16 years or older to log in" />
</p>
```

In this case, the CompareValidator control is used to check that the user is old enough to log in to our fictitious web application. Here, we set the Operator property of the CompareValidator to GreaterThan. This property can take on any of the values Equal, NotEqual, GreaterThan, GreaterThanEqual, LessThan,
Regular Expressions in JavaScript\(^4\)

another great article, this time on the use of regular expressions with JavaScript

Table 6.2. Common regular expression components and their descriptions

<table>
<thead>
<tr>
<th>Special Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>any character</td>
</tr>
<tr>
<td>^</td>
<td>beginning of string</td>
</tr>
<tr>
<td>$</td>
<td>end of string</td>
</tr>
<tr>
<td>\d</td>
<td>numeric digit</td>
</tr>
<tr>
<td>\s</td>
<td>whitespace character</td>
</tr>
<tr>
<td>\S</td>
<td>non-whitespace character</td>
</tr>
<tr>
<td>(abc)</td>
<td>string abc as a group of characters</td>
</tr>
<tr>
<td>?</td>
<td>preceding character or group is optional</td>
</tr>
<tr>
<td>+</td>
<td>one or more of the preceding character or group</td>
</tr>
<tr>
<td>*</td>
<td>zero or more of the preceding character or group</td>
</tr>
<tr>
<td>{n}</td>
<td>exactly (n) of the preceding character or group</td>
</tr>
<tr>
<td>{n,m}</td>
<td>(n) to (m) of the preceding character or group</td>
</tr>
<tr>
<td>(a</td>
<td>b)</td>
</tr>
<tr>
<td>$</td>
<td>a dollar sign (as opposed to the end of a string); we can ‘escape’ any of the special characters listed above by preceding it with a backslash. For example, . matches a period character, ? matches a question mark, and so on</td>
</tr>
</tbody>
</table>

You’ll find a complete guide and reference to regular expressions and their components in the .NET Framework SDK Documentation.

CustomValidator

The validation controls included with ASP.NET allow you to handle many kinds of validation, yet certain types of validation cannot be performed with these built-in controls. For instance, imagine that you needed to ensure that a new

\(^4\) http://www.sitepoint.com/article/expressions-javascript
user's login details were unique by checking them against a list of existing user-names on the server. The CustomValidator control can be helpful in this situation, and others like it. Let's see how:

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
<title>CustomValidator Control Sample</title>
<script runat="server" language="VB">
  Sub CheckUniqueUserName(s As Object, _
    e As ServerValidateEventArgs)
    Dim username As String = e.Value.ToLower
    If (username = "zak" Or username = "cristian") Then
      e.IsValid = False
    End If
  End Sub

  Sub submitButton_Click(s As Object, e As EventArgs)
    If Page.IsValid Then
      submitButton.Text = "Valid"
    Else
      submitButton.Text = "Invalid!"
    End If
  End Sub
</script>
</head>
<body>
<form runat="server">
  <p>New Username:<br />
    <asp:TextBox ID="usernameTextBox" runat="server" />
    <asp:CustomValidator ID="usernameUnique" runat="server" ControlToValidate="usernameTextBox"
      OnServerValidate="CheckUniqueUserName"
      ErrorMessage="This username already taken!" />
  </p>
  <p>
    <asp:Button ID="submitButton" runat="server" OnClick="submitButton_Click" Text="Submit" />
  </p>
</form>
</body>
</html>
```
IDENTITY
Identity columns are numbered automatically. If you set a column as an IDENTITY column, SQL Server will generate numbers automatically for that column as you add new rows to it. The first number in the column is called the identity seed. To generate subsequent numbers, the identity column adds a given value to the seed; the value that’s added is called the identity increment. By default, both the seed and increment have a value of 1, in which case the generated values are 1, 2, 3, and so on. If the identity seed were 5 and the identity increment were 10, the generated numbers would be 5, 15, 25, and so on.

IDENTITY is useful for ID columns, such as Department ID, for which you don’t care what the values are, as long as they’re unique. When you use IDENTITY, the generated values will always be unique. By default, you can’t specify values for an IDENTITY column. Note also that the column can never contain NULL.

Understanding NULL
Be sure not to see NULL as equivalent to 0 (in numerical columns), or an empty string (in the case of string columns). Both 0 and an empty string are values; NULL defines the lack of a value.

NULL and Default Values
I’ve often heard people say that when we set a default value for a column, it doesn’t matter whether or not we set it to accept NULLs. Many people seem to believe that columns with default values won’t store NULL.

That’s incorrect. You can modify a record after it was created, and change any field that will allow it to NULL. Your columns’ ability to store NULL is important for the integrity of your data, and it should reflect the purpose of that data. A default value does make things easier when we create new rows, but it’s not as vital as is correctly allowing (or disallowing) NULL in columns.

Primary Keys

Primary keys are the last fundamental concept that you need to understand before you can create your first data table. In the world of relational databases, each row in a table must be identified uniquely by a column called a key, on which all database operations are based.
The tables in your databases could contain hundreds or even thousands of rows of similar data—you could have several hundred employees in your Employees table alone. Imagine that your program needs to update or delete the record for John Smith, and there are several people with that name in your organization. You couldn’t rely on the database to find the record for the particular John Smith that you were trying to work with—it might end up updating or deleting the wrong record.

We can avoid these kinds of problems only by using a system that uniquely identifies each row in the table. The first step toward achieving this goal is to add to the table an ID column that provides a unique for each employee, as did the Employee ID column that we saw in Figure 7.1.

Remember that when we discussed this Employees table, we noted that you may be tempted to use each employee’s username to uniquely identify each employee. After all, that’s what the network administrator uses them for, so why shouldn’t you? It’s true that this column uniquely identifies each row in the table, and we call such a column a candidate key. However, it wouldn’t be a good idea to use this column in our database operations for a number of reasons. Firstly, network usernames have been known to change, and such a change would wreak havoc on any database of more than a couple of tables. As we’ll see later, keys are fundamental to establishing relationships between tables, and these relationships rely on the fact that keys will never change. Secondly, non-numeric keys require much more processing power than simple numeric ones. Using an nvarchar field to uniquely identify rows in your table will bring your SQL Server to a grinding halt much, much quicker than if you chose a simple, numeric key.

The column that we choose to uniquely identify a row in a table in practice is called the primary key. In the case of our Employee table, the Employee ID will always be unique, so it would be a suitable primary key.

**Multi-column Keys**

To make the concept of keys easier to understand, we kept the definition simple, although it’s not 100% technically correct. A key isn’t necessarily formed by a single column—it can be formed by two or more columns. If the key is made up of multiple columns, the set of values in those columns must be unique for any given record. We’ll see an example of such a key in a moment.

Although we usually refer to primary keys as if they were columns, technically they are constraints that we apply to the existing columns of a table. Constraints impose restrictions on the data we can enter into our tables, and the primary key
Creating the Employees Table

Figure 7.10. Specifying column properties

Figure 7.11. The Employees table
Populating the Data Tables

If tables represent drawers in a filing cabinet, rows represent individual paper records in those drawers. Suppose that our intranet web application was a real application. As people begin to register and interact with the application, rows are created within the various tables, and are filled up with the information about those people.

Once the data structures are in place, adding rows of data is as easy as typing information into the cells in the **Datasheet View** of a table, which looks a bit like a spreadsheet. To access it, right-click on the table and select **Show Table Data** in Visual Web Developer or **Open Table** in SQL Server Management Studio. You can use the window that opens to start entering data. Let’s add some sample data to the table you’ve just created so that we can test the Dorknozzle database as we develop the application. Table 7.7 to Table 7.11 represent the tables and data you should add.

### Inserting Data and Identity Columns

If you correctly set the ID column as an identity column, you won’t be allowed to specify the values manually—the ID values will be generated for you automatically. You need to be careful, because an ID value will never be generated twice on the same table. So even if you delete all the rows in a table, the database will not generate an ID with the value of 1; instead, it will continue creating new values from the last value that was generated for you.

Keep in mind that a new row is saved to the database at the moment that you move on to the next row. It’s very important that you remember this when you reach the last row, as you’ll need to move to an empty row even if you aren’t adding any more records.
The Employees table contains a few more columns than those outlined here, but, due to the size constraints of this page, I’ve left them out. Feel free to add your own data to the rest of the cells, or you could leave the remaining cells empty, as they’re marked to accept NULL.

**Table 7.9. The HelpDeskCategories table**

<table>
<thead>
<tr>
<th>CategoryID (Primary Key)</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Hardware</td>
</tr>
<tr>
<td>2</td>
<td>Software</td>
</tr>
<tr>
<td>3</td>
<td>Workstation</td>
</tr>
<tr>
<td>4</td>
<td>Other/Don't Know</td>
</tr>
</tbody>
</table>

**Table 7.10. The HelpDeskStatus table**

<table>
<thead>
<tr>
<th>StatusID (Primary Key)</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Open</td>
</tr>
<tr>
<td>2</td>
<td>Closed</td>
</tr>
</tbody>
</table>

**Table 7.11. The HelpDeskSubjects table**

<table>
<thead>
<tr>
<th>SubjectID (Primary Key)</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Computer won't start</td>
</tr>
<tr>
<td>2</td>
<td>Monitor won't turn on</td>
</tr>
<tr>
<td>3</td>
<td>Chair is broken</td>
</tr>
<tr>
<td>4</td>
<td>Office won't work</td>
</tr>
<tr>
<td>5</td>
<td>Windows won't work</td>
</tr>
<tr>
<td>6</td>
<td>Computer crashes</td>
</tr>
<tr>
<td>7</td>
<td>Other</td>
</tr>
</tbody>
</table>

**What IDENTITY Columns are not For**

In our examples, as in many real-world scenarios, the ID values are sequences that start with 1 and increment by 1. This makes many beginners assume that they can use the ID column as a record-counter of sorts, but this is a mistake. The ID is really an arbitrary number that we know to be unique; no other information should be discerned from it.
isting departments in the Department table. However, as with primary keys, just having the correct fields in place doesn’t mean that our data is guaranteed to be correct.

For example, try setting the DepartmentID field for one of the employees to 123. SQL Server won’t mind making the change for you, so if you tried this in practice, you’d end up storing invalid data. However, after we set the foreign keys correctly, SQL Server will be able to ensure the integrity of our data—specifically, it will forbid us to assign employees to nonexistent departments, or to delete departments with which employees are associated.

The easiest way to create foreign keys using Visual Web Developer or SQL Server Management Studio is through database diagrams, so let’s learn about them.

Using Database Diagrams

To keep the data consistent, the Dorknozzle database really should contain quite a few foreign keys. The good news is that you have access to a great feature called database diagrams, which makes it a cinch to create foreign keys. You can define the table relationships visually using the database diagrams tool in Visual Web Developer or SQL Server Management Studio, and have the foreign keys generated for you.

Database diagrams weren’t created specifically for the purpose of adding foreign keys. The primary use of diagrams is to offer a visual representation of the tables in your database and the relationships that exist between them, to help you to design the structure of your database. However, the diagrams editor included in Visual Web Developer and SQL Server Management Studio is very powerful, so you can use the diagrams to create new tables, modify the structure of existing tables, or add foreign keys.

Let’s start by creating a diagram for the Dorknozzle database. To create a database diagram in Visual Web Developer, right-click the Database Diagrams node, and select Add New Diagram, as shown in Figure 7.15.

The process is similar in SQL Server Management Studio, which, as Figure 7.16 illustrates, has a similar menu.

The first time you try to create a diagram, you’ll be asked to confirm the creation of the database structures that support diagrams. Select Yes from the dialog, which should look like the one shown in Figure 7.17.
There are three types of relationships that can occur between the tables in your database:

- one-to-one relationships
- one-to-many relationships
- many-to-many relationships

**One-to-one Relationships**

A one-to-one relationship means that for each record in one table, only one other related record can exist in another table.

One-to-one relationships are rarely used, since it is usually more efficient to just combine the two records and store them together as columns in a single table. For example, every employee in our database will have a phone number stored in the HomePhone column of the Employees table. In theory, we could store the phone number in a separate table and link to them via a foreign key in the Employees table, but this would be inefficient for our application, since we assume that one phone number can belong to only one employee. As such, we can leave this one-to-one relationship (along with any others) out of our database design.

**One-to-many Relationships**

The one-to-many relationship is by far the most common relationship type. Within a one-to-many relationship, each record in a table can be associated with multiple records from a second table. These records are usually related on the basis of the primary key from the first table. In the employees/departments example, a one-to-many relationship exists between the Employees and Departments tables, as one department can be associated with many employees.

When a foreign key is used to link two tables, the table that contains the foreign key is on the “many” side of the relationship, and the table that contains the primary key is on the “one” side of the relationship. In database diagrams, one-to-many relationships are signified by a line between the two tables; a golden key symbol appears next to the table on the “one” side of the relationship, and an infinity sign (∞) is displayed next to the table that could have many items related to each of its records. In Figure 7.27, those icons appear next to the Employees and Departments tables.
As you can see, one-to-many relationships are easy to spot if you have a diagram at hand—just look for the icons next to the tables. Note that the symbols don’t show the exact columns that form the relationship; they simply identify the tables involved.

Select the line that appears between two related tables to view the properties of the foreign key that defines that relationship. The properties display in the Properties window (you can open this by selecting View > Properties Window). As Figure 7.28 illustrates, they’re the same options we saw earlier in Figure 7.24.
In the query window, type your first command:

```
SELECT Name
FROM Employees
```

Click the Execute button or press F5. If everything works as planned, the result will appear similar to Figure 8.3.

Nice work! Now that we’ve taken our first look at SQL, let’s talk more about SQL queries.
Selecting Certain Fields

If you didn’t want to select all the fields from the database table, you’d include the names of the specific fields that you wanted in place of the * in your query. For example, if you’re interested in the department names—not their IDs—you could execute the following:

```
SELECT Department
FROM Departments
```

This statement would retrieve data from the Department field only. Rather than specifying the *, which would return all the fields within the database table, we specify only the fields that we need.

Selecting All Columns Using *

To improve performance in real-world development scenarios, it’s better to ask only for the columns that are of interest, rather than using *. Moreover, even when you need all the columns in a table, it’s better to specify them by name, to safeguard against the possibility that future changes, which cause more columns to be added to the table, affecting the queries you’re writing now.

It’s important to note that the order of the fields in a table determines the order in which the data will be retrieved. Take this query, for example:

```
SELECT DepartmentID, Department
FROM Departments
```

You could reverse the order in which the columns are returned with this query:
In this example, all rows whose DepartmentIDs are less than 2 or greater than 5 are returned.

**Matching Patterns with LIKE**

As we’ve just seen, the WHERE clause allows us to filter results based on criteria that we specify. The example we discussed earlier filtered rows by comparing two numbers, but SQL also knows how to handle strings. For example, if we needed to search the company’s Employees table for all employees named Zak Ruvalcaba, we’d use the following SQL statement:

```
SELECT EmployeeID, Username
FROM Employees
WHERE Name = 'Zak Ruvalcaba'
```

However, we won’t see many such queries in reality. In real-world scenarios, most record matching is done by matching the primary key of the table to some specific value. When an arbitrary string such as a name is used (as in the example above), it’s likely that we’re searching for data based on partially complete information.

A more realistic example is one in which we want to find all employees with the surname Ruvalcaba. The LIKE keyword allows us to perform pattern matching with the help of **wildcard characters**. The wildcard characters supported by SQL Server are the percentage symbol (%), which matches any sequence of zero or more characters, and the underscore symbol (_), which matches exactly one character.

If we wanted to find all names within our Employees table with the surname of Ruvalcaba, we could modify the SQL query using a wildcard, as follows:

```
SELECT EmployeeID, Name
FROM Employees
WHERE Name LIKE '%Ruvalcaba'
```

With this query, all records in which the Name column ends with Ruvalcaba are returned, as shown below.

<table>
<thead>
<tr>
<th>EmployeeID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Zak Ruvalcaba</td>
</tr>
</tbody>
</table>
Note that we’re using the IN operator instead of the equality operator (=). We do so because our subquery could return a list of values. For example, if we added another department with the name “Product Engineering,” or accidentally added another Engineering record to the Departments table, our subquery would return two IDs. So, whenever we’re dealing with subqueries like this, we should use the IN operator unless we’re absolutely certain that the subquery will return only one record.

Querying Multiple Tables

When using queries that involve multiple tables, it’s useful to take a look at the database diagram you created in Chapter 7 to see what columns exist in each table, and to get an idea of the relationships between the tables.

Table Joins

An inner join allows you to read and combine data from two tables between which a relationship is established. In Chapter 7, we created such a relationship between the Employees table and the Departments table using a foreign key.

Let’s make use of this relationship now, to obtain a list of all employees in the engineering department:

```sql
SELECT Employees.Name 
FROM Departments 
INNER JOIN Employees ON Departments.DepartmentID = Employees.DepartmentID 
WHERE Departments.Department LIKE '%Engineering'
```

The first thing to notice here is that we qualify our column names by preceding them with the name of the table to which they belong, and a period character (.). We use Employees.Name rather than Name, and Departments.DepartmentID instead of DepartmentID. We need to specify the name of the table whenever the column name exists in more than one table (as is the case with DepartmentID); in other cases (such as with Employees.Name), adding the name of the table is optional.

As an analogy, imagine that you have two colleagues at work named John. John Smith works in the same department as you, and his desk is just across the aisle. John Thomas, on the other hand, works in a different department on a different floor. When addressing a large group of colleagues, you would use John Smith’s full name, otherwise people could become confused. However, it would quickly become tiresome if you always used John Smith’s full name when dealing with
MOD returns the remainder of one value divided by another. The following query would return the value 2:

```
SELECT MOD(8, 3)
```

SIGN
This function returns -1, 0, or 1, to indicate the sign of the argument.

POWER
This function returns the result of one value raised to the power of another. The following query returns the result of $2^3$:

```
SELECT POWER(2, 3)
```

SQRT
SQRT returns the non-negative square root of a value.

Many, many more mathematical functions are available—check SQL Server Books Online for a full list.

### String Functions

String functions work with literal text values rather than numeric values.

**UPPER, LOWER**
This function returns the value passed in as all uppercase or all lowercase, respectively. Take the following query as an example:

```
SELECT LOWER(Username), UPPER(State)
FROM Employees
```

The query above will return a list of usernames in lowercase, and a list of states in uppercase.

**LTRIM, RTRIM**
This function trims whitespace characters, such as spaces, from the left- or right-hand side of the string, respectively.

**REPLACE**
Use the REPLACE function to change a portion of a string to a new sequence of characters that you specify.

```
SELECT REPLACE('I like chocolate', 'like', 'love')
```
DATEADD
adds an interval to an existing date (a number of days, weeks, etc.) in order to obtain a new date

DATEDIFF
calculates the difference between two specified dates

DATEPART
returns a part of a date (such as the day, month, or year)

DAY
returns the day number from a date

MONTH
returns the month number from a date

YEAR
returns the year from a date

We won’t be working with these functions in our example application, but it’s good to keep them in mind. Here’s a quick example that displays the current year:

SELECT YEAR(GETDATE())

The result (assuming it’s still 2006, of course) is shown below:

<table>
<thead>
<tr>
<th>CurrentYear</th>
</tr>
</thead>
<tbody>
<tr>
<td>2006</td>
</tr>
</tbody>
</table>

(1 row(s) affected)

Working with Groups of Values

Transact-SQL includes two very useful clauses that handle the grouping of records, and the filtering of these groups: GROUP BY and HAVING. These clauses can help you find answers to questions like, “Which are the departments in my company that have at least three employees?” and “What is the average salary in each department?”

2 Assuming, of course, that your Employees table has a Salary column, or some other way of keeping track of salaries.
Try the above SQL statement. Then, to read the new list of records, execute the following:

```sql
SELECT DepartmentID, Department
FROM Departments
```

All records in the `Departments` table will be displayed, along with our Cool New Department and its automatically-generated `DepartmentID`.

**Identity Values**

To obtain programatically the identity value that we just generated, we can use the `scope_identity` function like this:

```sql
SELECT scope_identity()
```

**The UPDATE Statement**

We use the UPDATE statement to make changes to existing records within our database tables. The UPDATE statement requires certain keywords, and usually a WHERE clause, in order to modify particular records. Consider this code:

```sql
UPDATE Employees
SET Name = 'Zak Christian Ruvalcaba'
WHERE EmployeeID = 1
```

This statement would change the name of the employee whose `EmployeeID` is 1. Let’s break down the UPDATE statement's syntax:

**UPDATE**

This clause identifies the statement as one that modifies the named table in the database.

**table name**

We give the name of the table we’re updating.

**SET**

The SET clause specifies the columns we want to modify, and gives their new values.

**column names and values**

We provide a list of column names and values, separated by commas.
The command above would execute successfully because there aren’t any employees linked to the new department.

**Deleting Records**

Like the `UPDATE` command, the `WHERE` clause is best used together with `DELETE`; otherwise, you can end up deleting all the records in the table inadvertently!

**Stored Procedures**

Stored procedures are database objects that group one or more T-SQL statements. Much like VB or C# functions, stored procedures can take parameters and return values.

Stored procedures are used to group SQL commands that form a single, logical action. For example, let’s say you want to add a new web site functionality that allows departments to be deleted. Now, as you know, you must delete all of the department’s employees before you can delete the department itself.

To help with such management issues, you could have a stored procedure that copies the employees of that department to another table (called `Employees-Backup`), deletes those employees from the main `Employees` table, then removes the department from the `Department` table. As you can imagine, having all this logic saved as a stored procedure can make working with databases much easier.

We’ll see a more realistic example of a stored procedure in the next chapter, when we start to add more features to the Dorknozzle project, but until then, let’s learn how to create a stored procedure in SQL Server, and how to execute it.

The basic form of a stored procedure is as follows:

```sql
CREATE PROCEDURE ProcedureName
(
  @Parameter1 DataType,
  @Parameter2 DataType,
  ...
)
AS
-- SQL Commands here
```
If you get sick of typing quotes, ampersands, and underscores, you can combine the three bold strings in the above code into a single string. However, I’ll continue to present connection strings as above throughout this book—not only are they more readable that way, but they fit on the page, too!

If you’re using C#, your code should look like this:

```csharp
protected void Page_Load(object sender, EventArgs e)
{
    // Define database connection
    SqlConnection conn = new SqlConnection(
        "Server=localhost\SqlExpress;Database=Dorknozzle;" +
        "Integrated Security=True");
}
```

Be aware that, in C#, the backslash (\) character has a special meaning when it appears inside a string, so, when we wish to use one, we have to use the double backslash (\\) shown here.

### Preparing the Command

Now we’re at step three, in which we create a `SqlCommand` object and pass in our SQL statement. The `SqlCommand` object accepts two parameters: the first is the SQL statement, and the second is the connection object that we created in the previous step.

```csharp
protected void Page_Load(object sender, EventArgs e)
{
    // Define database connection
    SqlConnection conn = new SqlConnection(
        "Server=localhost\SqlExpress;Database=Dorknozzle;" +
        "Integrated Security=True");
}
```
Executing the Command

When we’re ready to run the query, we open the connection and execute the command. The SqlCommand class has three methods that we can use to execute a command; we simply choose between them depending on the specifics of the query. The three methods are as follows:

**ExecuteReader**

ExecuteReader is used for queries or stored procedures that return one or more rows of data. ExecuteReader returns an SqlDataReader object that can be used to read the results of the query one by one, in a forward-only, read-only order. Using the SqlDataReader object is the fastest way to retrieve records from the database, but it can’t be used to update the data or to access the results in random order.

The SqlDataReader keeps the database connection open until all the records have been read. This can be a problem, as the database server will usually have a limited number of connections—people who are using your application simultaneously may start to see errors if you leave these connections open. To alleviate this problem, we can read all the results from the SqlDataReader object into an object such as a DataTable, which stores the data locally without needing a database connection. You’ll learn more about the DataTable object in Chapter 12.

**ExecuteScalar**

ExecuteScalar is used to execute SQL queries or stored procedures that return a single value, such as a query that counts the number of employees in a company. This method returns an Object, which you can convert to specific data types depending on the kinds of data you expect to receive.

**ExecuteNonQuery**

ExecuteNonQuery is an oddly-named method that’s used to execute stored procedures and SQL queries that insert, modify, or update data. The return value will be the number of affected rows.
We already know that the **SqlDataReader** class reads the data row by row, in a forward-only fashion. Only one row can be read at any moment. When we call `reader.Read`, our `SqlDataReader` reads the next row of data from the database. If there’s data to be read, it returns `True`; otherwise—if we’ve already read the last record returned by the query—the `Read` method returns `False`. If we view this page in the browser, we’ll see something like Figure 9.4.

**Using Parameters with Queries**

What if the user doesn’t want to view information for all employees, but instead, wants to see details for one specific employee?

To get this information from our `Employees` table, we’d run the following query, replacing `EmployeeID` with the ID of the employee in which the user was interested.

```
SELECT EmployeeID, Name, Username, Password
FROM Employees
WHERE EmployeeID = EmployeeID
```

Let’s build a page like the one shown in Figure 9.5 to display this information.

**Figure 9.5. Retrieving details of a specific employee**

Create a new web form called `QueryParameters.aspx` and alter it to reflect the code shown here:

```
<%@ Page Language="VB" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```
Let’s go ahead and add the necessary code to `Page_Load` in `HelpDesk.aspx` to populate the `DropDownList` controls from the database. After the changes are made, the lists will be populated with the data you added to your database in Chapter 7, as illustrated in Figure 9.10.

**Figure 9.10. A drop-down list created with data binding**

Open `HelpDesk.aspx` in Design View and double-click an empty space on the form to have the signature of the `Page_Load` method generated for you. Then, add the following code:

```vbnet
Imports System.Data.SqlClient
Imports System.Configuration

Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
    If Not IsPostBack Then
        ' Define data objects
        Dim conn As SqlConnection
        Dim categoryComm As SqlCommand
        Dim subjectComm As SqlCommand
        Dim reader As SqlDataReader
        ' Read the connection string from Web.config
```

Figure 9.11. Displaying an error message in the catch block

C# File: HelpDesk.aspx.cs (excerpt)

protected void submitButton_Click(object sender, EventArgs e)
{
    if (Page.IsValid)
    {
        // Code that uses the data entered by the user
    }
}

If Page.IsValid Then
    ' Code that uses the data entered by the user
End If
End Sub
Make Sure you’ve Set the Identity Property!

Note that when we’re inserting a new record into the HelpDesk table, we rely on the ID column, RequestID, to be generated automatically for us by the database. If we forget to set RequestID as an identity column, we’ll receive an exception every time we try to add a new help desk request!

Did you notice the use of the ExecuteNonQuery method? As you know, we use this method when we’re executing any SQL query that doesn’t return a set of results, such as INSERT, UPDATE, and DELETE queries.

You’ll remember that, in order to make the example simpler, we hard-coded the EmployeeID (to the value of 5), and the Status (to the value of 1). To make the application complete, you could add another drop-down list from which employees
Once the SqlCommand object has been created using this UPDATE statement, we simply pass in the necessary parameters, as we did with the INSERT statement. The important thing to remember when updating records is that you must take care to perform the UPDATE on the correct record. To do this, you must include a WHERE clause that specifies the correct record using a value from a suitable unique column (usually the primary key), as shown above.

### Handle Updates with Care!

When updating a table with some new data, if you don’t specify a WHERE clause, every record in the table will be updated with the new data, and (usually) there’s no way to undo the action!

Let’s put all this theory into practice as we build the Admin Tools page. The database doesn’t contain a table that’s dedicated to this page; however, we’ll use the Admin Tools page as a centralized location for a number of tables associated with other pages, including the Employees and Departments tables. For instance, in this section, we’ll allow an administrator to change the details of a specific employee.

Create a new web form named AdminTools.aspx in the same way you created the other web forms we’ve built so far. Use the Dorknozzle.master master page and a code-behind file. Then, add the following code to the content placeholder, and modify the page title as shown below.
Deleting Records

Just as we can insert and update records within the database, we can also delete them. Again, most of the code for deleting records resembles that which we’ve already seen. The only major part that changes is the SQL statement within the command:

Visual Basic

```vbc
comm = New SqlCommand("DELETE FROM Table " &
    "WHERE UniqueField=@UniqueFieldParameter", conn)
```

C#

```csharp
comm = new SqlCommand("DELETE FROM Table " +
    "WHERE UniqueField=@UniqueFieldParameter", conn);
```

Once we’ve created the DELETE query’s SqlCommand object, we can simply pass in the necessary parameter:

Visual Basic

```vbc
comm.Parameters.Add("@UniqueFieldParameter", _
    System.Data.SqlDbType.Type)
```

```vbc
comm.Parameters("@UniqueFieldParameter").Value = UniqueValue
```

C#

```csharp
comm.Parameters.Add("@UniqueFieldParameter",
    System.Data.SqlDbType.Type);
```

```csharp
comm.Parameters["@UniqueFieldParameter"].Value = UniqueValue;
```

To demonstrate the process of deleting an item from a database table, we’ll expand on the Admin Tools page. Since we’re allowing administrators to update information within the Employees table, let’s also give them the ability to delete an employee’s record from the database. To do this, we’ll place a new Button control for deleting the selected record next to our Update Employee button.

Start by adding the new control at the end of AdminTools.aspx:

```xml
<br />
<asp:Button ID="deleteButton" Text="Delete Employee"
    Enabled="False" runat="server" />
```

Handling DataList Events

One problem you may encounter when working with container controls such as the DataList or the Repeater is that you can’t access the controls inside their templates directly from your code. For example, consider the following ItemTemplate, which contains a Button control:

```xml
<asp:DataList ID="employeesList" runat="server">
    <ItemTemplate>
        Employee ID: <strong><%#Eval("EmployeeID")%></strong>
        <asp:Button runat="server" ID="myButton" Text="Select" />
    </ItemTemplate>
</asp:DataList>
```

Although it may not be obvious at the first glance, you can’t access the Button easily through your code. The following code would generate an error:

```vbnet
' Don't try this at home
myButton.Enabled = False
```

Things get even more complicated if you want to handle the Button’s Click event, because—you guessed it—you can’t do so without jumping through some pretty complicated hoops.

So, if we can’t handle events raised by the buttons and links inside a template, how can we interact with the data in each template? We’ll improve our employee directory by making a simpler, basic view of the items, and add a “View More” link that users can click in order to access more details about the employee. To keep things simple, for now, we’ll hide only the employee ID from the standard view; we’ll show it when the visitor clicks the View More link.

After we implement this feature, our list will appear as shown in Figure 10.2. You’ll be able to view more details about any employee by clicking on the appropriate link.

Open EmployeeDirectory.aspx, and modify the ItemTemplate of the DataList as shown below:

```xml
<asp:DataList id="employeesList" runat="server">
    <ItemTemplate>
        <asp:Literal ID="extraDetailsLiteral" runat="server">
        </asp:Literal>
    </ItemTemplate>
</asp:DataList>
```
erty, and the employee’s new name and username from the TextBox control. The techniques used in this code are the ones we used earlier, but be sure to read the code carefully to ensure that you understand how it works.

Visual Basic

ElseIf e.CommandName = "CancelEditing" Then
    ' Cancel edit mode
    employeesList.EditItemIndex = -1
    ' Refresh the DataList
    BindList()
ElseIf e.CommandName = "UpdateItem" Then
    ' Get the employee ID
    Dim employeeId As Integer = e.CommandArgument
    ' Get the new username
    Dim nameTextBox As TextBox = e.Item.FindControl("nameTextBox")
    Dim newName As String = nameTextBox.Text
    ' Get the new name
    Dim usernameTextBox As TextBox = e.Item.FindControl("usernameTextBox")
    Dim newUsername As String = usernameTextBox.Text
    ' Update the item
    UpdateItem(employeeId, newName, newUsername)
    ' Cancel edit mode
    employeesList.EditItemIndex = -1
    ' Refresh the DataList
    BindList()
End If
End Sub

C#

else if (e.CommandName == "CancelEditing")
{
    // Cancel edit mode
    employeesList.EditItemIndex = -1;
    // Refresh the DataList
    BindList();
}
else if (e.CommandName == "UpdateItem")
{
    // Get the employee ID
    int employeeId = Convert.ToInt32(e.CommandArgument);
    // Get the new username
    TextBox nameTextBox = (TextBox)e.Item.FindControl("nameTextBox");
    string newName = nameTextBox.Text;
    string newUsername = nameTextBox.Text;
Finally
  
  ' Close the connection
  conn.Close()
End Try

End Sub

C# File: EmployeeDirectory.aspx.cs (excerpt)

protected void UpdateItem(int employeeId, string newName, string newUsername)
{
    // Declare data objects
    SqlConnection conn;
    SqlCommand comm;
    // Read the connection string from Web.config
    string connectionString =
        ConfigurationManager.ConnectionStrings["Dorknozzle"].ConnectionString;
    // Initialize connection
    conn = new SqlConnection(connectionString);
    // Create command
    comm = new SqlCommand("UpdateEmployee", conn);
    // Specify we're calling a stored procedure
    comm.CommandType = System.Data.CommandType.StoredProcedure;
    // Add command parameters
    comm.Parameters.Add("@EmployeeID", SqlDbType.Int);
    comm.Parameters["@EmployeeID"].Value = employeeId;
    comm.Parameters.Add("@NewName", SqlDbType.NVarChar, 50);
    comm.Parameters["@NewName"].Value = newName;
    comm.Parameters.Add("@NewUsername", SqlDbType.NVarChar, 50);
    comm.Parameters["@NewUsername"].Value = newUsername;
    // Enclose database code in Try-Catch-Finally
    try
    {
        // Open the connection
        conn.Open();
        // Execute the command
        comm.ExecuteNonQuery();
    }
    finally
    {
        // Close the connection
        conn.Close();
    }
}
The significance of these new elements is as follows:

**HeaderStyle**
- customizes the appearance of the DataList’s heading

**ItemStyle**
- customizes the appearance of each item displayed within the DataList

**AlternatingItemStyle**
- customizes the appearance of every other item displayed within the DataList
using System;
using System.Data;
using System.Configuration;
using System.Collections;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Data.SqlClient;

public partial class AddressBook : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            BindGrid();
        }
    }

    private void BindGrid()
    {
        // Define data objects
        SqlConnection conn;
        SqlCommand comm;
        SqlDataReader reader;
        // Read the connection string from Web.config
        string connectionString = ConfigurationManager.ConnectionStrings[
            "Dorknozzle"].ConnectionString;
        // Initialize connection
        conn = new SqlConnection(connectionString);
        // Create command
        comm = new SqlCommand(
            "SELECT EmployeeID, Name, City, State, MobilePhone " +
            "FROM Employees", conn);
        // Enclose database code in Try-Catch-Finally
        try
        {
            // Open the connection
            conn.Open();
            // Execute the command
            reader = comm.ExecuteReader();
            // Fill the grid with data
grid.DataSource = reader;
grid.DataBind();
// Close the reader
reader.Close();
}
finally
{
    // Close the connection
    conn.Close();
}
}

What's going on here? If you disregard the fact that you're binding the SqlDataReader to a GridView instead of a Repeater or DataList, the code is almost identical to that which we saw in the previous chapter.

Now save your work and open the page in the browser. Figure 11.2 shows how the GridView presents all of the data within the Employees table in a cleanly formatted structure.

**Figure 11.2. Displaying the address book in GridView**
that you want displayed. To do so, list the columns inside the `<asp:GridView>` and `</asp:GridView>` tags, as shown below:

```
<asp:GridView ID="grid" runat="server"
    AutoGenerateColumns="False">
    <Columns>
        <asp:BoundField DataField="Name" HeaderText="Name" />
        <asp:BoundField DataField="City" HeaderText="City" />
        <asp:BoundField DataField="MobilePhone"
            HeaderText="Mobile Phone" />
    </Columns>
</asp:GridView>
```

Notice that each column that we want to display is created using a `BoundField` control inside a set of `<Columns>` and `</Columns>` tags. Each `BoundField` control has a `DataField` property, which specifies the name of the column, and a `HeaderText` property, which sets the name of the column as you want it displayed to the user.

Now, save your work and view it in the browser. This time, only the columns that you specified to be bound are displayed in the `GridView`. The results should appear as shown in Figure 11.3.

Note that if you don’t include the `HeaderText` property for any of the bound columns, those columns will not have a header.

We’ve now succeeded in displaying only the information we want to display, but the `GridView` still looks plain. In the next section, we’ll use styles to customize the look of our `GridView`.

**Styling the GridView with Templates, Skins, and CSS**

The `GridView` control offers a number of design-time features that are tightly integrated with the Visual Web Developer designer. As with the `DataList` class, when you click the grid’s smart tag, you get quick access to a number of very useful features, as Figure 11.4 illustrates.
If you click the **Auto Format**… link from the smart tag menu and choose one of the predefined styles, Visual Web Developer generates a number of template styles for you, like this:
Figure 11.6. Adding a new GridView column

If you’re using Visual Web Developer, you can quickly and easily add a new column to your table in Design View. Click the GridView’s smart tag, and click the Add New Column... item, as shown in Figure 11.6.

In the dialog that appears, change the field type to ButtonField, change the command name to Select, and set the Text field to Select, so the dialog appears as it does in Figure 11.7.

Figure 11.7. Adding a new field

After clicking OK, your brand new column shows up in Design View. If you switch to Source View, you can see it there, too:

File: AddressBook.aspx (excerpt)

```xml
<asp:GridView ID="grid" runat="server"
    AutoGenerateColumns="false">
    <Columns>
        <asp:BoundField DataField="Name" HeaderText="Name" />
        <asp:BoundField DataField="City" HeaderText="City" />
    </Columns>
</asp:GridView>
```
contains many fields—so many, in fact, that the main grid can’t display all of them.

A common use of the DetailsView control is to create a page that shows a list of items, and allows you to drill down to view the details of each item. For instance, an ecommerce site might initially present users with only a little information about all available products, to reduce download time and make the information more readable. Users could then select a product to see a more detailed view of that product.

Let’s see how this works by using a GridView and a DetailsView in our Address Book web form.

Replace detailsLabel with a DetailsView control, as shown in the following code snippet:

```html
</asp:GridView>
<br />
<asp:DetailsView id="employeeDetails" runat="server" />
</asp:Content>
```

Next, we’ll modify the BindGrid method to specify the grid’s data key. The data key feature of the GridView control basically allows us to store a piece of data about each row without actually displaying that data. We’ll use it to store the EmployeeID of each record. Later, when we need to retrieve additional data about the selected employee, we’ll be able to read the employee’s ID from the data key, and use it in our SELECT query.

Add this row to your code-behind file:

```vbnet
' Open the connection
cconn.Open()
' Execute the command
reader = comm.ExecuteReader()
' Fill the grid with data
grid.DataSource = reader
grid.DataKeyNames = New String() {"EmployeeID"}
grid.DataBind()
' Close the reader
reader.Close()```
employeeDetails.FindControl("editAddressTextBox")
Dim newCityTextBox As TextBox =
  employeeDetails.FindControl("editCityTextBox")
' Extract the updated data from the TextBoxes
Dim newAddress As String = newAddressTextBox.Text
Dim newCity As String = newCityTextBox.Text
' Declare data objects
Dim conn As SqlConnection
Dim comm As SqlCommand
' Read the connection string from Web.config
Dim connectionString As String =
  ConfigurationManager.ConnectionStrings("
    "Dorknizzle").ConnectionString
' Initialize connection
conn = New SqlConnection(connectionString)
' Create command
comm = New SqlCommand("UpdateEmployeeDetails", conn)
comm.CommandType = Data.CommandType.StoredProcedure
' Add command parameters
comm.Parameters.Add("@EmployeeID", Data.SqlDbType.Int)
comm.Parameters("@EmployeeID").Value = employeeId
comm.Parameters("@NewAddress").Value = newAddress
comm.Parameters("@NewCity").Value = newCity
' Enclose database code in Try-Catch-Finally
Try
  ' Open the connection
  conn.Open()
  ' Execute the command
  comm.ExecuteNonQuery()
Finally
  ' Close the connection
  conn.Close()
End Try
' Exit edit mode
employeeDetails.ChangeMode(DetailsViewMode.ReadOnly)
' Reload the employees grid
BindGrid()
' Reload the details view
BindDetails()
End Sub

c# File: AddressBook.aspx.cs (excerpt)
protected void employeeDetails_ItemUpdating(object sender,
  DetailsViewUpdateEventArgs e)
Next, we call a stored procedure to take care of the database update. To create this stored procedure, run the following script in SQL Server Management Studio:

```sql
CREATE PROCEDURE UpdateEmployeeDetails
(
    @EmployeeID Int,
    @NewAddress nvarchar(50),
    @NewCity nvarchar(50)
)
AS
UPDATE Employees
SET Address = @NewAddress, City = @NewCity
WHERE EmployeeID = @EmployeeID
```
Binding the DetailsView to a SqlDataSource

Here, our aim is to replicate the functionality the DetailsView gave us in Chapter 11, and to add functionality that will allow users to add and delete employees’ records.

Let’s start by adding another SqlDataSource control, either next to or below the existing one, in AddressBook.aspx. Give the new SqlDataSource the name employeeDataSource. Click its smart tag, and select Configure Data Source. The Configure Data Source wizard will appear again.

In the first screen, choose the Dorknozzle connection string. Click Next and you’ll be taken to the second screen, where there’s a bit more work to do. Start by specifying the Employees table and checking all of its columns, as shown in Figure 12.9.

Figure 12.9. Choosing fields
Next, click the WHERE... button. In the dialog that opens, select the EmployeeID column, specify the = operator, and select Control in the Source field. For the Control ID select grid, and leave the default value empty, as Figure 12.10 shows.

Finally, click Add, and the expression will be added to the WHERE clause list. The SQL expression that’s generated will filter the results on the basis of the value selected in the GridView control. Click OK to close the dialog, then click the Advanced... button. Check the Generate INSERT, UPDATE, and DELETE statements checkbox, as shown in Figure 12.11.

Click OK to exit the Advanced SQL Generation Options dialog, then click Next. In the next screen, feel free to click on Test Query to ensure everything’s working as expected. If you click Test Query, you’ll be asked for the Employee ID’s type and value. Enter 1 for the value, leave the type as Int32, then click OK. The row should display as shown in Figure 12.12.

Click Finish.

Congratulations! Your new SqlDataSource is ready to fill your DetailsView. Next, we need to tie this SqlDataSource to the DetailsView and specify how we want the DetailsView to behave. Open AddressBooks.aspx, locate the DetailsView control and set the properties as outlined in Table 12.2.
Figure 12.11. Generating INSERT, UPDATE, and DELETE statements

Figure 12.12. Testing the query generated for our data source
Table 12.2. Properties to set for the DetailsView control

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoGenerateDeleteButton</td>
<td>True</td>
</tr>
<tr>
<td>AutoGenerateEditButton</td>
<td>True</td>
</tr>
<tr>
<td>AutoGenerateInsertButton</td>
<td>True</td>
</tr>
<tr>
<td>AllowPaging</td>
<td>False</td>
</tr>
<tr>
<td>DataSourceID</td>
<td>employeeDataSource</td>
</tr>
<tr>
<td>DataKeyNames</td>
<td>EmployeeID</td>
</tr>
</tbody>
</table>

Recreating the Columns

If you’re using Design View, make sure you choose Yes when you’re asked about recreating the DetailsView rows and data keys. If you’re not using Design View, set the columns as shown here:

```xml
<Fields>
  <asp:BoundField DataField="EmployeeID"
    HeaderText="EmployeeID" InsertVisible="False"
    ReadOnly="True" SortExpression="EmployeeID" />
  <asp:BoundField DataField="DepartmentID"
    HeaderText="DepartmentID"
    SortExpression="DepartmentID" />
  <asp:BoundField DataField="Name"
    HeaderText="Name"
    SortExpression="Name" />
  <asp:BoundField DataField="Username"
    HeaderText="Username"
    SortExpression="Username" />
  <asp:BoundField DataField="Password"
    HeaderText="Password"
    SortExpression="Password"/>
  :
  <asp:BoundField DataField="MobilePhone"
    HeaderText="MobilePhone"
    SortExpression="MobilePhone" />
</Fields>
```

You’re ready! Execute the project, and enjoy the new functionality that you implemented without writing a single line of code! Take it for a quick spin to ensure that the features for editing and deleting users are perfectly functional!
the name of a department than a department ID when they’re updating or inserting the details of an employee. Figure 12.16 shows how the page will look once we’ve created this functionality.

**Figure 12.16. Viewing the Department drop-down list in DetailsView**

Start by adding a new SqlDataSource control beside the two existing data source controls in AddressBook.aspx. Name the control departmentsDataSource, click its smart tag, and select Configure Data Source. In the first screen, select the Dorknozzle connection, then click Next. Specify the Departments table and select both of its columns, as shown in Figure 12.17.

Click Next, then Finish to save the data source configuration. The definition of your new data source control will look like this:

```xml
<asp:SqlDataSource id="departmentsDataSource" runat="server"
     ConnectionString="<%$ ConnectionStrings:Dorknozzle %>">
```

File: AddressBook.aspx (excerpt)
Modify this generated template as highlighted below:

```xml
<asp:TemplateField HeaderText="Department"
    SortExpression="DepartmentID">
    <EditItemTemplate>
        <asp:DropDownList id="didDdl" runat="server"
            DataSourceID="departmentsDataSource"
            DataTextField="Department" DataValueField="DepartmentID"
            SelectedValue='<%# Bind("DepartmentID") %>' />
    </EditItemTemplate>
    <InsertItemTemplate>
        <asp:DropDownList ID="didDdl" runat="server"
            DataSourceID="departmentsDataSource"
            DataTextField="Department" DataValueField="DepartmentID"
            SelectedValue='<%# Bind("DepartmentID") %>' />
    </InsertItemTemplate>
    <ItemTemplate>
        <asp:DropDownList ID="didDdl" runat="server"
            DataSourceID="departmentsDataSource"
            DataTextField="Department" DataValueField="DepartmentID"
            SelectedValue='<%# Bind("DepartmentID") %>'
            Enabled="False" />
    </ItemTemplate>
</asp:TemplateField>
```

When you reload your address book now, you’ll see that the departments are displayed in a drop-down list. You can use that list when you’re inserting and editing employee data—a feature that the intranet’s users are sure to find very helpful!

**More on SqlDataSource**

The SqlDataSource object can make programming easier when it’s used correctly and responsibly. However, the simplicity of the SqlDataSource control comes at the cost of flexibility and maintainability, and introduces the potential for performance problems.
the database—you simply retrieve the data from the data set again and again. Figure 12.19 illustrates this point.
ascending) or DESC (for descending). So, if you were sorting the DepartmentID column, the Sort property would need to be set to DepartmentID ASC or Department DESC.

This property must be set before the data binding is performed, as is shown in the following code, which will sort the data by DepartmentID in numeric order:

Visual Basic
dataTable.DefaultView.Sort = "DepartmentID DESC"
departmentsGridView.DataSource = dataTable.DefaultView
departmentsGridView.DataBind()

C#
dataTable.DefaultView.Sort = "Department DESC";
departmentsGridView.DataSource = dataTable.DefaultView;
departmentsGridView.DataBind();

It’s a pretty simple task to sort a DataView in code like this, but if we want to let users sort the data on the basis of any column, in any direction, things get a little bit more complicated. In this case, we need to remember the previous sort method between requests.

In order to be truly user-friendly, our grid should behave like this:

- The first time a column header is clicked, the grid should sort the data in ascending order, based on that column.
- When the same column header is clicked multiple times, the grid should alternate between sorting the data in that column in ascending and descending modes.

When a column heading is clicked, the grid’s Sorting event is fired. In our case, the Sorting event handler (which we’ll look at in a moment) saves the details of the sort column and direction in two properties:

- gridSortExpression retains the name of the column on which we’re sorting the data (such as Department)
- gridSortDirection can be either SortDirection.Ascending or SortDirection.Descending

We create a sorting expression using these properties in BindGrid:
In order to implement the sorting functionality as explained above, we need to remember between client requests which column is being sorted, and whether it’s being sorted in ascending or descending order. That’s what the properties `gridSortExpression` and `gridSortDirection` do:

Visual Basic

```vbnet
Private Property gridSortDirection() As String
    Get
        ' Initial state is Ascending
        If (ViewState("GridSortDirection") Is Nothing) Then
            ViewState("GridSortDirection") = SortDirection.Ascending
        End If
        ' Return the state
        Return ViewState("GridSortDirection")
    End Get
    Set(ByVal value)
        ViewState("GridSortDirection") = value
    End Set
End Property
```

C#

```csharp
// Initial state is Ascending
if (ViewState("GridSortDirection") == null) { ViewState("GridSortDirection") = SortDirection.Ascending; }
```

Private Property `gridSortExpression()`

```vbnet
Private Property gridSortExpression() As String
    Get
        ' Initial sort expression is DepartmentID
        Return gridSortExpression
    End Get
End Property
```
Here, we use the ViewState collection to store information about which column is being sorted, and the direction in which it's being sorted.

When the Sorting event handler fires, we set the gridSortExpression and gridSortDirection properties. The method starts by retrieving the name of the clicked column:

```
Visual Basic
Protected Sub departmentsGrid_Sorting(ByVal sender As Object, ByVal e As System.Web.UI.WebControls.GridViewSortEventArgs) Handles departmentsGrid.Sorting
    ' Retrieve the name of the clicked column (sort expression)
    Dim sortExpression As String = e.SortExpression

C#
protected void departmentsGrid_Sorting(object sender, GridViewSortEventArgs e)
{
    // Retrieve the name of the clicked column (sort expression)
    string sortExpression = e.SortExpression;
}
```

Next, we check whether the previously-clicked column is the same as the newly-clicked column. If it is, we need to toggle the sorting direction. Otherwise, we set the sort direction to ascending:

```
Visual Basic
' Decide and save the new sort direction
If (sortExpression = gridSortExpression) Then
    If gridSortDirection = SortDirection.Ascending Then
        gridSortDirection = SortDirection.Descending
    Else
        gridSortDirection = SortDirection.Ascending
    End If
Else
    gridSortDirection = WebControls.SortDirection.Ascending
End If

C#
// Decide and save the new sort direction
if (sortExpression == gridSortExpression) {
    if (gridSortDirection == SortDirection.Ascending) {
        gridSortDirection = SortDirection.Descending;
    }
```
Basic Security Guidelines

The primary and most important element of building secure applications is to consider and plan an application’s security from the early stages of its development. Of course, we must know the potential internal and external threats to which an application will be exposed before we can plan the security aspects of that system. Generally speaking, ASP.NET web application security involves—but is not limited to—the following considerations:

**Validate user input.**
Back in Chapter 6, you learned how to use validation controls to enable client-side validation of user input, and how to double-check that validation on the server side.

Since the input your application will receive from web browsers is ultimately under users’ control, there’s always a possibility that the submitted data will not be what you expect. The submission of bad or corrupted data can generate errors in your web application, and compromise its security.

**Protect your database.**
The database is quite often the most important asset we need to protect—after all, it’s here that most of the information our application relies upon is stored. **SQL injection attacks**, which target the database, are a common threat to web application security. If the app builds SQL commands by naively assembling text strings that include data received from user input, an attacker can alter the meaning of the commands the application produces simply by including malicious code in the user input.¹

You’ve already learned how to use ADO.NET to make use of command parameters, and parameterized stored procedures, in order to include user input in SQL queries. Fortunately, ADO.NET has built-in protection against injection attacks. Moreover, if you specify the data types of the parameters you add, ASP.NET will throw an exception in cases where the input parameter doesn’t match the expected data type.

¹ You’ll find a detailed article on SQL injection attacks at http://www.unixwiz.net/techtips/sql-injection.html.
In this case, the users with the login names of jruvalcaba and zruvalcaba are allowed access to the application, but all other users (whether they’re logged in or not) will be denied access.

Now that you have a basic understanding of the ways in which user access is configured within the Web.config file, let’s see how we can use Web.config to store a list of users for our application.

**Storing Users in Web.config**

The great thing about the Web.config file is that it is secure enough for us to store user names and passwords in it with confidence. The <credentials> tag, shown here within the forms extent of the Web.config file, defines login credentials for two users:

```xml
<authentication mode="Forms">
  <forms>
    <credentials passwordFormat="Clear">
      <user name="zak" password="zak" />
      <user name="jessica" password="jessica" />
    </credentials>
  </forms>
</authentication>
```

As we want to prevent users from browsing the site if they’re not logged in, we use the appropriate <deny> tag in our <authorization> tag. The names and passwords of the users we will permit can then simply be specified in the <credentials> tag. Change your Web.config file to match the one shown above, and we’ll try another example.

Let’s modify the code that lies within the <head> tag of the Login.aspx page to validate the user names and passwords based on the Web.config file. Here’s what this change looks like:
Securing your Web Application

Now we have two roles, and two users (admin and cristian), but we still need to secure the application. You should have restricted access earlier in this chapter by modifying Web.config like this:

```xml
<authorization>
  <deny users="?" />
</authorization>
```

If you haven’t already done so, you can add this code now, or use Visual Web Developer to add it for you. Open the ASP.NET Web Site Administration Tool, click the Security tab, and click Create access rules. Create a new access rule for the Dorknozzle directory, as shown in Figure 13.14, to Deny all Anonymous users.

Figure 13.14. No anonymous users in Dorknozzle
with the exception of the Admin Tools link. When you click Admin Tools, you should be sent back to the Login page. This time, log in with the admin user details, and voilà! You’ll gain access to the Admin Tools page as well.

Let’s take a few moments to customize the look of your login controls. Stop the execution of the project, and switch back to Login.aspx in Design View. Select the Login control and click its smart tag to see the three very useful options shown in Figure 13.16.

Figure 13.16. Options for the Login control

The Administer Website link launches the ASP.NET Web Site Administration Tool. The Convert to Template option transforms the current layout of your control into templates, which you can then customize down to the smallest detail. The Auto Format link lets you select a predefined style to apply to this control.

If you were working in a production scenario, I’d advise you to select Convert to Template and use CSS to fine-tune the appearance of your control, as we did with the GridView and DetailsView controls in Chapter 11. However, for the purposes of this exercise, let’s just set the BorderStyle property of the Login control to Solid, and the BorderWidth property to 1px.

It was simple to add login functionality—we even changed its appearance with just a few mouse clicks! There are just one or two more things that we need to take care of before we can continue to add features to our site. First, let’s deal with personalization.

Customizing User Display

The next feature we want to implement is functionality that gives the user a way to log out of the application. After you perform the changes that we’re about to implement, logged-in users will have the option to log out, as Figure 13.17 illustrates.

On the other hand, users that aren’t logged in won’t see the menu at all, as Figure 13.18 indicates.
Writing Content to a Text File

For the purposes of the next few exercises, let’s work again with our old friend, the Learning web application. Start Visual Web Developer, go to File > Open Web Site, and open the Learning application.

Right-click the project in Solution Explorer, and select Add New Item. Select the Web Form template, name it WriteFile.aspx, and make sure you aren’t using a code-behind file or a master page. Click Add, then enter the code shown here in bold:

```vbnet
<%@ Page Language="VB" %>
<%@ Import Namespace="System.IO" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">
</script>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Writing to Text Files</title>
</head>
<body>
    <form id="form1" runat="server">
        Write the following text within a text file:<br />
        <asp:TextBox ID="myText" runat="server" />
        <asp:Button ID="writeButton" Text="Write" runat="server"
            OnClick="WriteText" />
    </form>
</body>
</html>
```

As you can see, we import the System.IO namespace—the namespace that contains the classes for working with text files—first. Next, we add a TextBox control to handle collection of the user-entered text, and a Button control to send the information to the server for processing.

Next, in the <head> tag, we’ll create the WriteText method mentioned in the OnClick attribute of the Button. This method will write the contents of the TextBox to the text file:
Also note that, rather than specifying the full path to the text file, you can use the `MapPath` method to generate the full path to the text file automatically. All you need to do is give the method a path relative to the current directory, as follows:

```
Visual Basic

Using streamWriter As StreamWriter = File.AppendText( _
        MapPath("myText.txt"))
```

File: `WriteFile.aspx` (excerpt)
If you’re using C#, you should place the following code in the `<script runat="server">` section:

```csharp
C# File: FileUpload.aspx (excerpt)
<script runat="server">
    void UploadFile(Object s, EventArgs e) {
        // Did the user upload any file?
        if (fileUpload.HasFile) {
            // Get the name of the file
            string fileName = fileUpload.FileName;
            // Upload the file on the server
            fileUpload.SaveAs(MapPath(fileName));
            // Inform the user about the file upload success
            label.Text = "File " + fileName + " uploaded.";
        } else
            label.Text = "No file uploaded!";
    }
</script>
```

Load the script, and click the Upload! button without selecting a file. The message “No file uploaded!” is displayed, as shown in Figure 14.11.

**Figure 14.11. An error arising as a file has not been specified**
Appendix A: Web Control Reference

The following reference includes a list of important properties, methods, and events for most of the controls you’ll find in the Visual Web Developer Toolbox.

I’ve grouped the lists of controls on the basis of their locations within the Toolbox:

- standard controls
- validation controls
- navigation controls
- HTML server controls

As all the web controls listed are derived from the WebControl class, they inherit its properties and methods. First up, let’s review the more useful of these, which can be used with any of the web controls.

The WebControl Class

Properties

AccessKey: specifies a shortcut key that quickly selects a control without the user needing to use a mouse; the shortcut command is usually **Alt** plus a letter or number

Attributes: allows the accessing and manipulation of the attributes of the HTML code rendered by the control

BackColor: the control’s current background color

BorderColor: color for the border

BorderStyle: style of border drawn around the web control; default is **NotSet**; other values are None, Solid, Double, Groove, Ridge, Dotted, Dashed, Inset, and Outset
CellSpacing sets the number of pixels between individual CheckBoxes within the CheckBoxList.

DataMember represents the particular table within the data source.

DataSource represents the actual data source to use when binding to a CheckBoxList.

DataTextField represents the field within the data source to use with the CheckBoxList text label.

DataTextFieldString a format string that determines how the data is displayed.

DataValueField represents the field within the data source to use with the CheckBoxList's value.

Items the collection of items within the CheckBoxList.

RepeatColumns determines the number of columns to use when displaying the CheckBoxList.

RepeatDirection indicates the direction in which the CheckBoxes should repeat; possible values are Horizontal and Vertical.

RepeatLayout determines how the check boxes are formatted; possible values are Table and Flow; default is Table.

SelectedIndex represents the index selected within the CheckBoxList.

SelectedItem represents the item selected within the CheckBoxList.

Events

SelectedIndexChanged raised when a CheckBox within the CheckBoxList is selected.
SetActiveView  
sets the active view to the View received as parameter

Events

ActiveViewChanged  
fires when the active view of the MultiView changes

Panel

Properties

BackImageURL  
the URL of the background image to use within the Panel

HorizontalAlignment  
sets the horizontal alignment of the Panel; possible values are Center, Justify, Left, No Set, and Right

Wrap  
wraps the contents within the Panel when True; default value is True.

Visible  
controls the visibility of the Panel

PlaceHolder

Properties

Visible  
controls the visibility of the PlaceHolder

RadioButton

Properties

AutoPostBack  
automatically posts the form containing the RadioButton whenever checked or unchecked is True

Checked  
shows the RadioButton as checked if set to True

GroupName  
determines the name of the group to which the RadioButton belongs

Text  
specifies the text displayed next to the RadioButton
ControlToValidate specifies the ID of the control that you want to validate

Display shows how the error message within the validation control will be displayed; possible values are Static, Dynamic, and None; default is Static

EnableClientScript enables or disables client-side validation; by default, is set as Enabled

Enabled enables or disables client and server-side validation; by default, is set as Enabled

ErrorMessage specifies the error message that will be displayed to the user

IsValid has the value True when the validation check succeeds, and False otherwise

Text sets the error message displayed by the control when validation fails

Methods

Validate performs validation and modifies the IsValid property

Events

ServerValidate represents the function for performing server-side validation

RangeValidator

Properties

ControlToValidate specifies the ID of the control that you want to validate

Display shows how the error message within the validation control will be displayed; possible values are Static, Dynamic, and None; default is Static

EnableClientScript enables or disables client-side validation; set as Enabled by default
Navigation Web Controls

SiteMapPath

Properties

- **CurrentNodeStyle**: the style used to display the current node
- **CurrentNodeTemplate**: the template used to display the current node
- **NodeStyle**: the style used to display SiteMapPath nodes
- **NodeTemplate**: the template used to display nodes
- **ParentLevelsDisplayed**: the maximum number of parent nodes to display
- **PathDirection**: specifies the path direction to display; possible values are `PathDirection.CurrentToRoot` and `PathDirection.RootToCurrent`
- **PathSeparator**: the string used to separate path nodes
- **PathSeparatorStyle**: the styles used to display the path separator
- **PathSeparatorTemplate**: the template used to display the separator
- **Provider**: the SiteMapProvider object associated with the SiteMapPath; the default site map provider is `XmlSiteMapProvider`, which reads its data from the `Web.sitemap` file
- **RenderCurrentNodeAsLink**: when set to `True`, the current site map site will be displayed as a link; default value is `False`
- **RootNodeStyle**: the style used to display the root node
- **RootNodeTemplate**: the template used to display the root node
- **ShowToolTips**: specifies whether the node links should display tooltips when the cursor hovers over them
MenuItemDataBound fired when a menu item is bound to its data source

**TreeView**

**Properties**

**AutoGenerateDataBindings**
- a Boolean value specifying whether the `TreeView` should automatically generate tree node bindings; default is `True`

**CheckedNodes**
- a collection of `TreeNode` objects representing the checked `TreeView` nodes

**CollapseImageToolTip**
- the tooltip for the image displayed for the “collapse” node indicator

**CollapseImageUrl**
- a string representing the URL for a custom image to be used as the “collapse” node indicator

**EnableClientScript**
- a Boolean value that specifies whether or not the `TreeView` should generate client-side JavaScript that expands or collapses nodes; `True` by default
- When the value is `False`, a server postback needs to be performed every time the user expands or collapses a node.

**ExpandDepth**
- an integer representing the number of `TreeView` levels that are expanded when the control is displayed for the first time; default is `-1`, which displays all the nodes

**ExpandImageToolTip**
- the tooltip for the image displayed for the “expand” node indicator

**ExpandImageUrl**
- a string representing the URL for a custom image to be used as the “expand” node indicator

**HoverNodeStyle**
- a `TreeNodeStyle` object used to define the styles of a node when the cursor is hovered over it
Properties

Attributes a collection of the element’s attribute names and their values

CausesValidation if True, validation is performed when the button is clicked; default is True

Disabled if set to True, the control will be disabled

ID contains the control’s ID

Name the name of the button

Style contains the control’s CSS properties

TagName returns the element’s tag name

Type specifies the type of control displayed by this input element

Value equivalent to the value attribute of the HTML tag

Visible if set to False, the control won’t be visible

Events

ServerClick raised when the user clicks the button

HtmlInputCheckBox Control

The HtmlInputCheckBox control corresponds to an <input type="checkbox" runat="server"> tag.

Properties

Attributes a collection of the element’s attribute names and their values

Checked a Boolean value that specifies whether or not the element is to be checked; default is False
**HtmlInputText Control**

The HtmlInputText control corresponds to an `<input runat="server">` tag with a type attribute of text or password.

**Properties**

- **Attributes**: a collection of the element’s attribute names and their values
- **Disabled**: if set to True, the control will be disabled
- **ID**: contains the control’s ID
- **MaxLength**: sets the maximum number of characters allowed in the text box
- **Name**: the name of the text box
- **Size**: the width of the text box
- **Style**: contains the control’s CSS properties
- **TagName**: returns the element’s tag name
- **Type**: specifies the type of control displayed by this input element
- **Value**: equivalent to the value attribute of the HTML tag
- **Visible**: if set to False, the control won’t be visible

**Events**

- **ServerChange**: occurs when the text in the control has changed

**HtmlSelect Control**

The HtmlSelect control corresponds to an HTML `<select runat="server">` tag (which creates a drop-down list).
choosing, 475
customizing in GridView, 435–436
displaying selected, 437
properties, 264–265
read-only, 461
Combine method, 589
combining lines of code, 70
CommandField column, 441, 456
company newsletter page
creating, 601–610
CompareValidator control, 231–233, 628
difference from RequiredFieldValidator
control, 232
example, 231
for data type checks, 233
to compare value of control to a fixed
value, 233
values, 222
compilation errors, 210
computer name, 341
conditional logic, 70–71
configuration errors, 210
configuration file
elements, 169
configuration section groups, 169
configuration section handler declarations, 170
configuration sections, 169
configuration settings, 169
ConfigurationManager class, 364, 554
Configure Data Source, 473, 479, 490
configuring
Cassini, 12
Internet Information Service, 11
web server, 11–21
confirmPasswordTextBox control, 230
connection string, 334, 336, 363, 578
in Web.config, 364
specifying, 474, 479
constraints, 266
constructors, 81
Content control, 134
ContentLength, 590
ContentPlaceHolder, 200
ContentPlaceHolder control, 133–135
ContentType, 591
color control binding, 365
Control class, 85
color control events, 52–56
subroutines, 54–56
ControlToCompare property, 232
ControlToValidate property, 230, 232
Convert this field into a TemplateField
link, 460
Convert to Template option, 564
cookies, 183–186
creating, 185
CSS (see Cascading Style Sheets)
CssClass property, 139–141
current databases, 256
custom errors, 212–213
Custom Errors option, 20
customErrors, 212
CustomValidator control, 239–242, 629
D
data access code
bulletproofing, 351–354
data adapters, 497
data binding, 355, 365–371, 410
and sorting, 516
DataSet to GridView control, 509
DefaultView does not apply when
binding to a DataSet, 510
using stored procedures, 397–399
using the master page, 199–200
using validation controls, 245–250
view the logged-in user sees, 565
web application, 148
welcome page, 201
Dorknozzle Properties window, 161, 164
drop-down list
created with data binding, 366
selecting directories or files from, 584
DropdownList control, 101, 110, 203, 366, 386, 584, 619
Dynamic mode (validation), 228

E
e As EventArgs (VB), 56
Edit button, 452, 454, 459
Edit columns, 459
Edit employee button, 414
Edit Fields, 491
Edit link, 457
edit mode
DetailsView control, 453, 456–459
GridView control, 453, 456–459
Edit Templates, 423
editing
DataList items, 413–422
field's properties, 460
EditItemStyle, 426
<EditItemTemplate> template, 405, 414, 461
EditRoleGroups, 568
EditUpdate checkbox, 456
element type selectors, 137
Else statement, 70
e-mail
configuring the SMTP server, 595–597
sending a test email, 597–600
sending with ASP.NET, 593–610
e-mail address
invalid, 237
embedded style sheets, 137
employee database, 253, 258, 266
creating the employee table, 267–270
creating the remaining tables, 271–273
entities, 261
relational design concepts, 276–278
with Department ID field, 260
employee details, 261
employee directory, 145, 182, 404
completed page, 360
e-mailing, 360–399
data binding, 365–371
deleting records, 394–397
hiding employee details, 407
inserting records, 371–378
showing employee ID, 411
styled list, 425
updated using DataList, 403–404
updating records, 378–393
using stored procedures, 397–399
viewing an employee in edit mode, 421
employee help desk request web form, 201–204
employee ID, 344–345, 349, 376, 386, 411
invalid, 351
employee list, 343
in a drop-down, 386
employee table, 274
creating, 267–270
extracting information from, 294
referencing records from the Departments table, 279
structure, 270
using Repeater control, 356–360
Index

employeeDataSource, 488
employeesList control, 382
    populating with list of employees
from database, 382–385
encryption, 529
    asymmetric algorithms, 529
    symmetric algorithms, 529
End Sub, 30–31
Enforce Foreign Key Constraint, 290
entities (tables), 258
Equals, 85
error messages, 205–206, 210, 212–213
    invalid email address, 237
    validation errors, 236
Event button, 463
event handler, 52, 58
event receiver, 83
event sender, 83
EventArgs e (C#), 56
events, 51
(see also control events; page events)
naming using past and present tense, 454
    triggered by DetailsView action
types, 454
    triggered by GridView action types, 454
events (object), 77, 83
"everything is an object", 84
Exception class
    properties, 217
Exception object, 217
exceptions, 206
    handling locally, 213–218
Execute button, 296
ExecuteNonQuery method, 337, 371, 376
ExecuteReader method, 337, 339
ExecuteScalar method, 337
executing a page, 158
    with debugging, 158
    without debugging, 158
Exit (VB), 76
exiting loops prematurely, 76
expressions, 310
expressions (SQL), 310
external style sheets, 136

F
fields
    choosing, 479
fields (database), 253
fields (object), 77
field’s properties
    editing, 460
File class, 573
    File paths
    working with, 586–589
file sharing
    disabling, 574
FileBytes property, 590–591
FileContent property, 590–591
FileName property, 590
files, 584
    uploading, 590–593
FileStream class, 572
FileUpload control, 125, 590–591, 619
Fill method, 502
filtering data, 520
filtering groups, 321–322
Finally block, 214–215, 351, 364
FindControl method, 412, 466
FirstBulletNumber property, 112
float data type, 263–264
floating point numbers, 59
FLOOR function, 314
font, 138
FooterStyle, 426
<FooterTemplate> template, 356, 405
For Each loop, 72, 75
For loops, 72, 75–76, 205
For Next loop, 72
foreign keys, 278–280
HTML comments, 41
HTML control classes, 96
HTML documents, 607
HTML elements
access to, 40
HTML hidden form field, 46
HTML output
visitors’ browser interpretation of, 529
HTML pages, 94
HTML server controls, 95–97, 643–658
accessing properties of, 99–100
assigning IDs t, 101
essentially as HTML tags with run-at="server" attribute, 98
survey form example, 97–101
using, 97–101
HTML tags, 34, 42–43
in HTML server controls, 98
manipulation, 96
HtmlAnchor control, 644
HtmlButton control, 97–98, 109, 644
HtmlForm control, 97–98, 645
HtmlGeneric control, 646
htmlInputImage control, 651
HtmlImage control, 647
HtmlInputButton control, 647
HtmlInputCheckBox control, 648
HtmlInputFile control, 649
HtmlInputHidden control, 650
HtmlInputRadioButton control, 652
HtmlInputText control, 97–98, 653
HtmlSelect control, 97–98, 653
HtmlTable control, 655
HtmlTableCell control, 656
HtmlTableRow control, 657
HtmlTextArea control, 658
HttpResponseHeader option, 19
HttpContext class, 184
HTTPS (HTTP Secure) protocol, 529
HttpUtility.HtmlEncode, 529
HyperLink control, 107, 620
HyperLinkField column, 441

I
identity increment, 265
IDENTITY primary key, 329
IDENTITY property (columns), 265
and primary key, 267
what they are not for, 275
identity seed, 265
identity values, 324
IDENTITY_INSERT property, 267
If statement, 69, 72
combined With Else statement, 70
VB, 74
IIf (in VB), 484
IIS (Internet Information Services)
Image control, 108, 621
ImageButton control, 106, 621
ImageField column, 441
ImageMap control, 108–109, 622
Images folder, 198
ImageUrl, 119
ImageUrl attribute, 107
Import directive, 36, 47
Imports (VB), 86
Impressions value, 119
IN operator, 313
use in SELECT queries, 305–306
IndexOutOfRangeException class, 206
inheritance, 83
initialization (variables), 59
inline code, 39
inline expressions, 39–40
inline style rules, 137
inner join, 309
input element, 101
inputString, 581
Insert method, 182
INSERT query, 371, 480
INSERT statement, 323–324, 329
P
page
definition, 56
Page class, 95
documentation, 85
page counter, 181
page counters, 174–180
Page directive, 36, 47, 90, 134
page events, 56–58
  order of execution, 57
  subroutines, 56
page templates, 132–135
Page.IsValid property, 225–226, 242
  245
Page_Load event, 56
Page_ModeChanging event, 181, 335, 361,
  366, 369, 410, 430
Page_PreRender event, 56
Page_UnLoad event, 57
PageIndex property, 505
PageIndexChanging event, 504
PageIndexChanging event handler, 505
PagerStyle, 477
pages element, 194–195
PageSize, 477
paging, 478, 504–506
paging buttons, 477
Panel control, 109–110, 123, 625
parameters, 346
  in functions and subroutines, 67
  out, 349
  use with queries, 344–350
parent tag, 355
parser errors, 210
partial classes, 90–91
  usage, 91
Pascal Casing, 101
Passport accounts, 531
Passport authentication, 531
password confirmation text box, 226
operators, 68–70
  definition, 68
  to break long lines of code, 70
  to combine lines of code, 70
operators (SQL), 311–313
OR operator, 312
ORDER BY clause
  for sorting query results, 306–307
  specifying, 475
ORDER BY... button, 474
out parameters, 349
overwriting text, 578
Page.Init event, 56
Page_Load event, 56
Page_Load method, 181, 335, 361,
  366, 369, 410, 430
PageIndex property, 505
PageIndexChanging event, 504
PageIndexChanging event handler, 505
PagerStyle, 477
pages element, 194–195
PageSize, 477
paging, 478, 504–506
paging buttons, 477
Panel control, 109–110, 123, 625
parameters, 346
  in functions and subroutines, 67
  out, 349
  use with queries, 344–350
parent tag, 355
parser errors, 210
partial classes, 90–91
  usage, 91
Pascal Casing, 101
Passport accounts, 531
Passport authentication, 531
password confirmation text box, 226
operators, 68–70
  definition, 68
  to break long lines of code, 70
  to combine lines of code, 70
operators (SQL), 311–313
OR operator, 312
ORDER BY clause
  for sorting query results, 306–307
  specifying, 475
ORDER BY... button, 474
out parameters, 349
overwriting text, 578
syntax, 324
updateButton control, 382
UpdateCommand property, 521
UpdateEmployee stored procedure, 421
UpdateItem method, 419–421
updating database records, 378–393
use WHERE statement, 379
updating DetailsView records, 463–468
updating existing data, 322–326
Upload button, 592–593
uploading files, 590–593
uploading files from the client to the server, 572
UPPER function, 315
uppercase, 315
user access
denying/allowing, 539, 557
setting individual rules, 561
user accounts
hard-coded, 535–538
User instance databases, 546
user interaction
with web application, 3
username, 567
editing, 422
entering, 536
storage in authentication ticket, 537
verification, 537
usernameTextBox control, 230
users, 3
creating, 554–556
Users role
assigning, 558
using (C#), 86
Using construct, 577
using statements, 153

**V**

validating user input, 528
validation controls, 219–250, 528, 628–633
CompareValidator, 231–233
CustomValidator, 239–242
RangeValidator, 233–234
RegularExpressionValidator, 236–239
RequiredFieldValidator, 230
using, 229–242
ValidationSummary, 235–236
validation errors, 236
validation groups, 242–245
default, 245
Validation tab, 229
ValidationExpression, 236
ValidationGroup property, 242
ValidationSummary control, 235–236
Value property, 105
variable declarations, 59
variables, 59
data types, 59–60
initialization, 59
VB, 4, 28, 48
and arrays, 207
arrays, 62
as strongly-typed language, 61
case sensitivity, 71
Click event, 53
Do While loop, 74
default, 62
declaring an array, 63
enabling Debug Mode, 167
End Sub to mark end of script, 30–31
file upload, 591
For loop, 75
functions, 65
HTML server controls in, 99
If Else statement, 70
 operators, 69
While loop, 342, 580, 582
While loops, 72
    results of, 73
whitespace characters
    trimming, 315
wildcard characters, 304
Windows Authentication, 335
Windows authentication, 531
Wizard control, 125
Write button, 578
WriteOnly modifier, 129–130
write-only properties, 129–130
WriteText method, 576
writing to text files, 571, 576–580
    permissions, 573–575
wwwroot folder, 13
    web server access to files in, 13
WYSIWYG interface, 150
X
XML basics, 117
Xml control, 628
XmlDataSource object, 470
Y
YEAR function, 318
Z
zero-based arrays, 64
zooming, 282