Chapter 6

Customizing and Managing Your Site's Appearance

ASP.NET 2.0 provides a number of ways to customize the style of pages and controls in your Web application. This chapter covers these approaches. It examines the various appearance properties of Web server controls, illustrates how to use CSS with ASP.NET, moves on to themes and master pages, and then finishes with user controls.

Changing the Appearance of Server Controls

The previous chapters introduced many of the standard Web server controls. This section returns to the coverage of Web server controls by demonstrating how to more fully customize the appearance of these controls. The chapter does so in two ways. The first way uses common formatting properties of the Web server controls, whereas the second way uses cascading style sheets.

Using Common Appearance Properties

As mentioned back in Chapter 3, most of the standard Web server controls inherit from the `WebControl` class. This `WebControl` class in turn inherits from the `Control` class. Both of these base classes define properties that can be used to modify the appearance of any Web server control. Table 6.1 lists the principal appearance properties of the `WebControl` and `Control` classes.

Table 6.1 Appearance Properties of the WebControl and Control Classes

Property	Description
`BackColor`	The background color (using either a hexadecimal HTML color identifier or a standardized color name) of the control.
`BorderColor`	The color of the control's border.
`BorderWidth`	The thickness (in pixels) of the control's border.
`BorderStyle`	The style (e.g., dotted, dashed, solid, double, etc.) of the control's border. Possible values are described by the `BorderStyle` enumeration.
`CssClass`	The CSS class name assigned to the control.
`Enabled`	Toggles the functionality of the control; if set to `false`, the control is disabled.
`Font`	List of font names for the control.
`ForeColor`	The color of the text of the control.
`Height`	The height of the control.
`Style`	A collection of attributes that is rendered as an HTML `style` attribute.
`Visible`	Specifies whether the control is visible.
`Width`	The width of the control.

Any of the properties listed in Table 6.1 can be set declaratively or programmatically. For instance, the following markup demonstrates how to set the foreground and the background colors of a `Label` control.

```
<asp:Label id="labTest" runat="server"
   ForeColor="#CC33CC" BackColor="Blue"/>
```

To set the same properties programmatically, you could do so in a number of different ways, two of which are shown here.

```
labTest.ForeColor = Color.FromName("#CC33CC");
labTest.BackColor = Color.Blue;
```

`Color` is a C# `struct` that has fields for the predefined color names supported by the major browsers, as well as methods for creating a `Color` object from a name or from three numbers representing RGB values.

CORE NOTE

To use `Color`, you must also reference the `System.Drawing` namespace.

Most of the various appearance properties are rendered in the browser as inline CSS styles. For instance, the `Label` control from the preceding two examples would be rendered to the browser as

```
<span id="labTest" style="color:#CC33CC;
   background-color:Blue;"></span>
```

Using the Style Class

Setting the various formatting properties for your Web controls, whether through programming or declarative means, is acceptable when there are only one or two properties to set. However, when you have many properties that need to be changed in multiple controls, this approach is not very ideal. A better approach is to use the `Style` class.

The `Style` class is ideal for changing multiple properties to multiple controls all at once. It encapsulates most of the formatting properties listed in Table 6.1 and can be applied to multiple Web server controls to provide a common appearance. To use this class, you simply instantiate a `Style` object, set its various formatting properties, and then apply the style to any server control by using the control's `ApplyStyle` method. The following example illustrates this usage.

```
Style myStyle = new Style();
myStyle.ForeColor = Color.Green;
myStyle.Font.Name = "Arial";

// Now apply the styles to the controls
myLabel.ApplyStyle(myStyle);
myTextBox.ApplyStyle(myStyle);
```

The `Style` class is best for situations where you need to programmatically change the appearance of a set of controls all at once. If you simply need to set up a consistent appearance to a series of controls, it is almost always better to use Cascading Style Sheets (CSS) or ASP.NET themes, both of which are covered in this chapter.

Using CSS with Controls

The previous section demonstrated how to use some of the common appearance properties of Web server controls. Although these properties are very useful for customizing the appearance of your Web output, they do not contain the full formatting power of Cascading Style Sheets. Fortunately, you can also customize the appearance of Web server controls using CSS.

There are two ways to harness the power of CSS with Web server controls. The first way is to assign a CSS declaration to the `Style` attribute/property of a control. For instance, the following example sets the CSS letter-spacing property to increase the whitespace between each letter in the `Label` to 2 pixels, along with setting the font style to italic.

```
<asp:label id="labMsg" runat="server" Text="hello world"
   style="letter-spacing: 2px; font-style: italic" />
```

To achieve the same effect by programming, you would use

```
labMsg.Style["letter-spacing"] = "2px";
labMsg.Style["font-style"] = "italic";
```

Notice that from a programming perspective, the `Style` property is a ***named collection***. A named collection acts like an array, except that individual items can be retrieved either through a zero-based index or a unique name identifier.

All styles added to a control, whether declaratively or programmatically, are rendered in the browser as an inline CSS rule. For instance, either of the two previous two examples would be rendered to the browser as

```
<span id="labMsg" style="letter-spacing:2px;font-style: italic">
hello world
</span>
```

The other way to use CSS with Web server controls is to assign a CSS class to the `CssClass` property of a control. For instance, assume that you have the following embedded CSS class definition.

```
<style type="text/css">
   .pullQuote { background: silver;
                margin: 10px;
                font-family: Verdana, Arial,Helvetica,sans-serif;
                font-size: 10pt; }
</style>
```

You could assign this CSS class via

```
<asp:label id="labMsg2" runat="server" CssClass="pullQuote" />
```

To achieve the same effect by coding, you would use

```
labMsg2.CssClass = "pullQuote";
```

Listings 6.1 and 6.2 demonstrate how to style Web server controls with CSS by using both the `Style` and `CssClass` properties. The markup contains three `Label` controls, each of which contain a paragraph of text. The form also contains two `DropDownList` controls. The first changes the CSS `text-transform` property (which changes the text from uppercase to lowercase) for two of the `Label` controls. The second `DropDownList` control sets the other `Label` control's `CssClass` property to one of two predefined CSS classes, both of which float the paragraph so that it becomes a pull quote relative to the other text (see Figure 6.1).

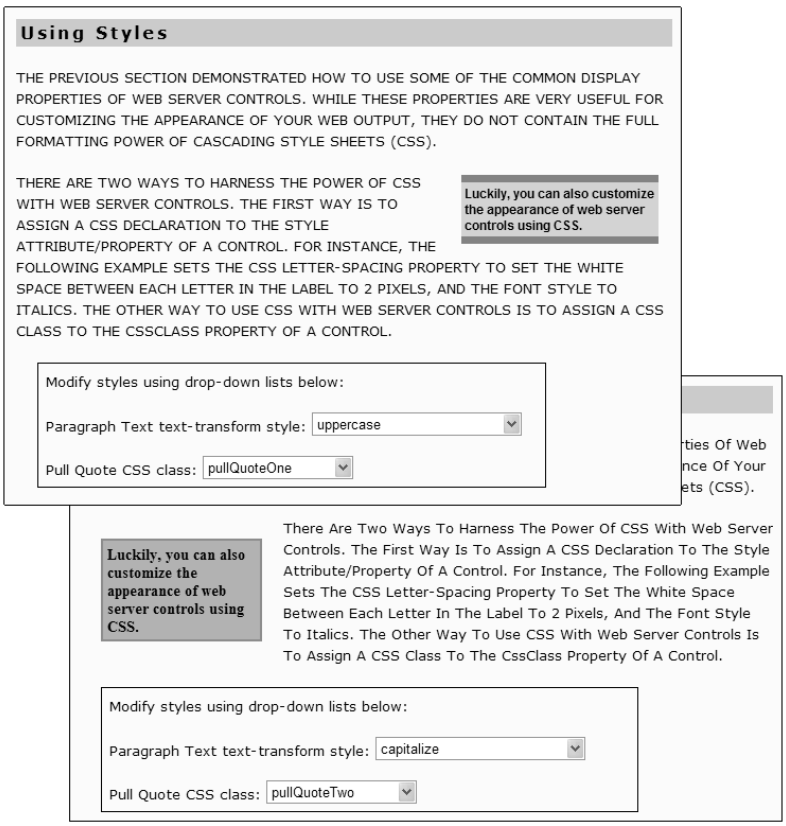

Figure 6.1 Using CSS.aspx

Although the example in Listing 6.1 uses embedded styles (that is, CSS rules within an HTML `<style>` element), you should generally locate a page's CSS in an external file and then link to it using the `<link>` element. By doing so, multiple pages in the site can use the same CSS file. As well, using an external CSS file generally reduces the page's bandwidth, because the browser can cache the CSS file.

CORE NOTE

You may notice that in Listing 6.1, the CSS styles use either `%` or `em` units rather than pixels. Doing so ensures that the relative font size and spacing work properly even if the user increases or decreases the browser's font size.

Listing 6.1 Using CSS.aspx

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title>Using CSS</title>
   <style type="text/css">
      body {
         background: #fffafa; margin: 1em;
         font: small/1.5em verdana, arial, sans-serif;
      }
      h1 {
         background: gold; color: black;
         font: bold 120% verdana, helvetica, sans-serif;
         letter-spacing: 0.25em; padding: 0.25em;
      }
      .controlPanel {
         padding: 0.5em; border: black 1px solid;
         background: #eee8a; margin: 1.5em; width: 75%;
      }
      .pullQuoteOne {
         padding: 0.25em; margin: 0.25em 1em;
         border: solid 7px #908070;
         border-right-width: 0; border-left-width: 0;
         background: lightgrey;
         float: right; width: 15em;
         font: bold 90% arial, helvetica, verdana, sans-serif;

      }
      .pullQuoteTwo {
         padding: 0.25em; margin: 1.5em;
         border: #82a91b 2px solid;
         background: #adc175;
```

```
         float: left; width: 10em;
         font: bold 105% times new roman, serif;
      }
   </style>
</head>
<body>
   <form id="form1" runat="server">
      <h1>Using Styles</h1>
      <p>
      <asp:Label ID="labOne" runat="server">
     The previous section demonstrated how to use some of the common
display properties of web server controls. While these properties are
very useful for customizing the appearance of your web output, they
do not contain the full formatting power of Cascading Style Sheets
(CSS).
      </asp:Label>
      </p>
      <p>
      <asp:Label ID="labPullQuote" runat="server">
      Luckily, you can also customize the appearance of web
      server controls using CSS.
      </asp:Label>
      </p>
      <p>
      <asp:Label ID="labTwo" runat="server">
      There are two ways to harness the power of CSS with web server
controls. The first way is to assign a CSS declaration to the Style
attribute/property of a control. For instance, the following example
sets the CSS letter-spacing property to set the white space between
each letter in the Label to 2 pixels, and the font style to italics.
The other way to use CSS with web server controls is to assign a CSS
class to the CssClass property of a control.
      </asp:Label>
      </p>

      <div class="controlPanel">
         <p>Modify styles using drop-down lists below:</p>
         <p>Paragraph Text text-transform style:
         <asp:DropDownList ID="drpParagraph" runat="server"
               AutoPostBack="True" OnSelectedIndexChanged=
               "drpParagraph_SelectedIndexChanged">
            <asp:ListItem Selected="True">
            Choose a text-transform style</asp:ListItem>
            <asp:ListItem Value="lowercase">
               lowercase</asp:ListItem>
            <asp:ListItem Value="uppercase">
               uppercase</asp:ListItem>
            <asp:ListItem Value="capitalize">
               capitalize</asp:ListItem>
```

```
            </asp:DropDownList>
            </p>
            <p>Pull Quote CSS class:
            <asp:DropDownList ID="drpPull" runat="server"
                  AutoPostBack="True" OnSelectedIndexChanged=
                  "drpPull_SelectedIndexChanged">
               <asp:ListItem Selected="True">
                  Choose a css class</asp:ListItem>
               <asp:ListItem Value="pullQuoteOne">
                  pullQuoteOne</asp:ListItem>
               <asp:ListItem Value="pullQuoteTwo">
                  pullQuoteTwo</asp:ListItem>
            </asp:DropDownList>
            </p>
         </div>
      </form>
</body>
</html>
```

The code-behind class (shown in Listing 6.2) for this example is quite straightforward. It contains selection event handlers for the two `DropDownList` controls. The first of these changes the `text-transform` CSS property of two `Label` controls based on the user's selection; the second sets the CSS class of the pull quote `Label` based on the user's selection.

Listing 6.2 Using CSS.aspx.cs

```
public partial class UsingCSS : System.Web.UI.Page
{
   /// <summary>
   /// Handler for text transform drop-down list
   /// </summary>
   protected void drpParagraph_SelectedIndexChanged(
         object sender, EventArgs e)
   {
      if (drpParagraph.SelectedIndex > 0)
      {
         labOne.Style["text-transform"] =
            drpParagraph.SelectedValue;
         labTwo.Style["text-transform"] =
            drpParagraph.SelectedValue;
      }
   }
   /// <summary>
   /// Handler for pull quote drop-down list
   /// </summary>
```

```
    protected void drpPull_SelectedIndexChanged(object sender,
          EventArgs e)
    {
       if (drpPull.SelectedIndex > 0)
          labPullQuote.CssClass = drpPull.SelectedValue;
    }
}
```

Appearance Properties, CSS, and ASP.NET

The intent of CSS is to separate the visual presentation details from the structured content of the HTML. Unfortunately, many ASP.NET authors do not fully take advantage of CSS, and instead litter their Web server controls with numerous appearance property settings (e.g., `BackColor`, `BorderColor`, etc.). Although it is true that these properties are rendered as inline CSS styles, the use of these properties still eliminates the principal benefit of CSS: the capability to centralize all appearance information for the Web page or Web site into one location, namely, an external CSS file. Also, because the appearance properties are rendered as inline CSS, this increases the size and the download time of the rendered page.

By limiting the use of appearance properties for Web server controls within your Web Forms, and using instead an external CSS file to contain all the site's styling, your Web Form's markup becomes simpler and easier to modify and maintain. For instance, rather than setting the `Font` property declaratively for a dozen Web server controls in a page to the identical value, it makes much more sense to do so via a single CSS rule. And if this CSS rule is contained in an external CSS file, it could be used throughout the site (that is, in multiple Web Forms), reducing the overall amount of markup in the site.

However, ASP.NET 2.0 does provide an additional mechanism for centralizing the setting the appearance of Web server controls on a site-wide basis, called themes and skins, which is our next topic.

CORE NOTE

There are many superb CSS resources available. Two of the best books are Charles Wyke-Smith's *Stylin' with CSS* (Pearson Education, 2005) and Eric Meyer's *Eric Meyer on CSS* (New Riders, 2002).

Using Themes and Skins

The previous section illustrates how you can customize your controls by setting the style properties of the controls themselves. ASP.NET 2.0 introduced the ***theme*** mechanism. This mechanism allows the developer to style the appearance of Web server controls on a site-wide basis. Like CSS, ASP.NET themes allow you to separate Web server control styling from the pages themselves, but have the additional benefit of having a complete object model that can be manipulated programmatically. Themes still allow you to use CSS for the majority of your visual formatting, and because theme support is built in to ASP.NET, you can dramatically alter the appearance of your Web site with a single line of programming (or a single line in the `Web.config` file). For instance, Figure 6.2 illustrates how a single Web Form's appearance can be radically transformed using three different themes.

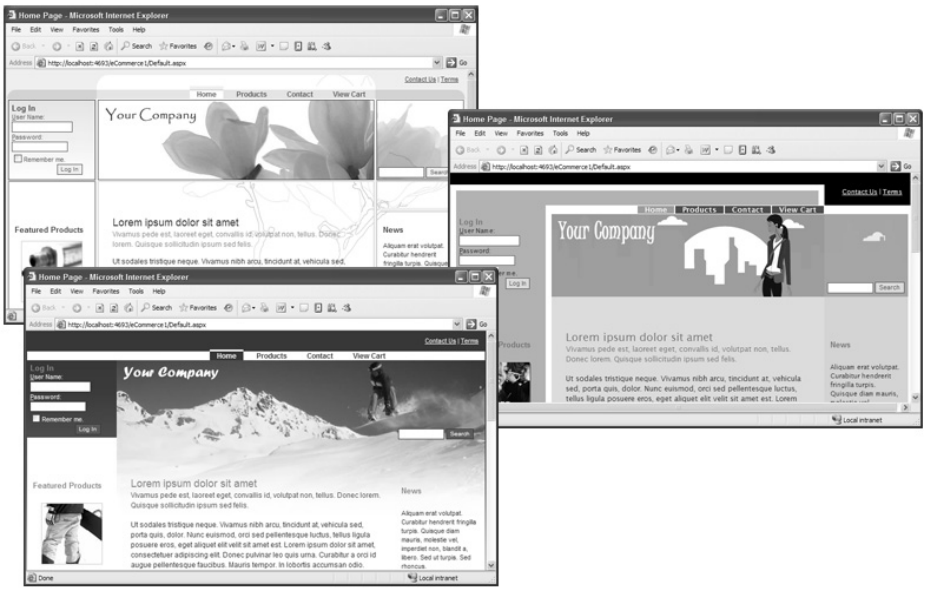

Figure 6.2 Same page—three different themes

An ASP.NET Web application can define multiple themes. Each theme resides in its own folder within the `App_Themes` folder in the root of your application. Within each theme folder, there are one or more ***skin*** files, as well as optional subfolders, CSS files, and image files (see Figure 6.3).

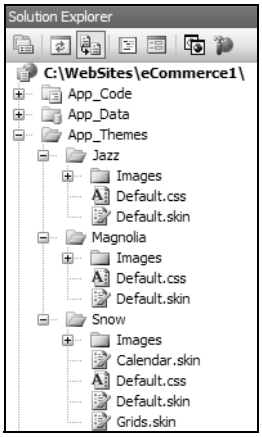

Figure 6.3 Theme file structure

Defining Skins

A skin describes the appearance of one or more control types. For example, a skin file might look like the following.

```
<asp:Label runat="server" ForeColor="Blue" Font-Size="10pt"
   Font-Name="Verdana" />
```

Notice that a skin simply contains a control definition *without* the `id` attribute. A given skin file can contain multiple control definitions. Alternately, many developers have a separate skin file for each control type (a theme can contain any number of skin files).

Not all properties can be skinned. Generally speaking, only properties relating to appearance (i.e., the properties in Table 6.1 plus additional properties depending upon the control) can be specified in a skin. Referencing a property that is not themeable in a skin file generates an error. As well, certain controls, such as the `Repeater`, are not themeable, generally because they do not inherit from the `WebControl` class.

CORE NOTE

It is important to note that property values specified by a skin override the property values set for the control in the `aspx` and `ascx` pages. This may seem counterintuitive from an object-oriented programming perspective, because you might expect the more specialized (the page) to override the general (the skin). However, you can make a control in a Web Form or user control ignore the settings in a skin by adding `EnableTheming="false"` to the control.

There is no Visual Studio designer support for creating skins. That is, the only way to create and modify a skin file is directly in Source view within Visual Studio. Even worse, Visual Studio's Intellisense is not available in Source view when modifying a skin. As an alternative, you could create a temporary Web Form, use the Design view or Source view as needed to add controls and set up their properties, copy and paste the markup to your skin file, and then remove the `id` attribute from each of the controls.

Creating Themes in Visual Studio

Themes reside in separate folders within the `App_Themes` folder within your site. You can create this folder yourself in Visual Studio by right-clicking the Web project in the Solution Explorer and choosing Add ASP.NET Folder → Theme option, as shown in Figure 6.4.

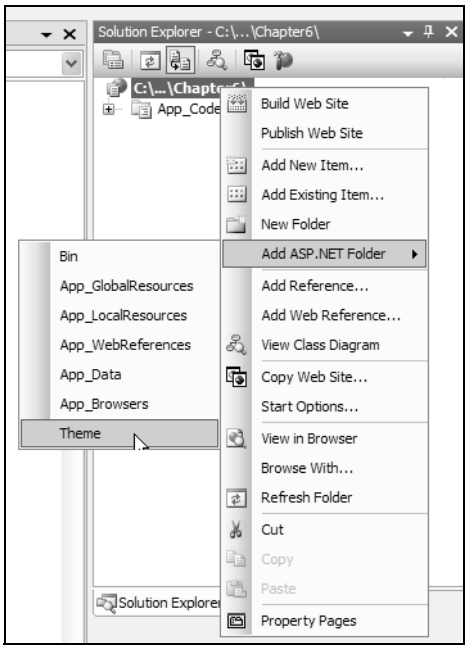

Figure 6.4 Adding a theme folder

Alternately, Visual Studio can automatically create a theme folder for you when you add a skin file via the Add New Item menu option (see Figure 6.5). In general, you probably want to avoid this approach because Visual Studio names the theme folder the same as the skin file.

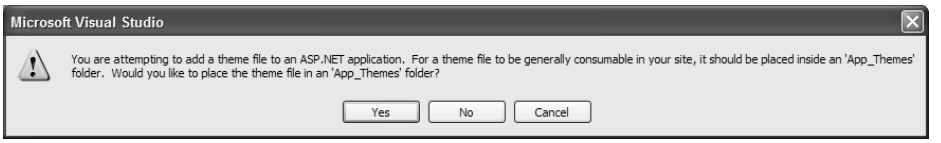

Figure 6.5 Automatically adding a theme folder

CORE NOTE

Microsoft provides several design templates that use a variety of themes and can be downloaded at `http://msdn.microsoft.com/asp.net/reference/design/templates`.

Walkthroughs 6.1 and 6.2 demonstrate how to create a theme and a skin.

Walkthrough 6.1 *Adding a Theme*

1. Use the Add ASP.NET Folder → Theme menu option in Visual Studio.
2. Name the folder `Cool`.
3. Use the Add ASP.NET Folder → Theme menu option in Visual Studio.
4. Name the folder `Professional`.

Walkthrough 6.2 *Creating a Skin*

1. Right-click the `Cool` theme and choose the Add New Item menu option in Visual Studio.
2. Choose the Skin template and name the file `Label.skin`.
3. Remove the commented example.
4. Add the following code:

```
<asp:Label runat="server" ForeColor="Green" />
```

5. Save and close the skin.

Applying a Theme

After a theme has been created (that is, after you've created one or more skin files), you can apply a theme to a page. This can be done in a few different ways. One way is to simply assign the theme using the `Theme=`*`themeName`* in the `Page` directive, for instance:

```
<%@ Page … Theme="Cool" %>
```

You can also set the theme for all pages in a site via the `Web.config` file. To do so, simply specify the theme via the `theme` attribute of the `pages` element within the `system.web` element, as shown in the following.

```
<system.web>
   …
   <pages theme="Cool" />
   …
</system.web>
```

CORE NOTE

Themes that are specified via the `Theme` attribute of the `Page` directive override the theme setting in the `Web.config` file.

A page's theme can also be set programmatically. To do so, you can set the `Theme` property (defined in the `Page` base class) for a form in its code-behind class. This is covered in more detail later in the chapter. Finally, another, less common way to set the theme is to set it for all sites on a machine via the `machine.config` file. This file is located at *`[windows]`*`\Microsoft.NET\Framework\`*`[version]`*`\CONFIG`. Just as with setting the theme for a site via the `Web.config` file shown earlier, you can set the global theme for the machine as a whole via the `Theme` attribute of the `pages` element within the `system.web` element of this `machine.config` file. The specified theme folder and its contents *must* be located in the global theme space for the machine, located at *`[windows]`*`\Microsoft.NET\Framework\`*`[version]`*`\ASP.NETClientFiles\Themes`.

How Themes Work

Now that you have seen how to create and use themes, let us peek under the hood and examine what ASP.NET does to make themes work. Like everything in ASP.NET, it all begins with a request. When a request arrives for a resource from an ASP.NET application that uses themes, the runtime parses and compiles all the skins

in each theme. Recall from Chapter 2 that the markup in an aspx page is parsed into a class and then compiled into an assembly; an analogous thing happens with the skin files—each theme is parsed and compiled into a separate assembly.

Each theme is realized as a concrete subclass of the PageTheme class. This class maintains a collection of ControlSkinDelegate objects. This delegate represents or "points to" the actual method that applies the correct skin to the control; this method exists in the class file generated for the skin.

When the runtime executes a page that has a theme, it iterates through the page's Controls collection, and if there is a delegate for the control type, it calls the delegated method (in the generated skin class) which then decorates (i.e., changes the properties specified in the skin) the specific control object.

Overriding Themes

As previously mentioned, skin definitions for a control type override any settings for that control type made within a given page. For instance, consider the following skin definition.

```
<asp:Label runat="server" ForeColor="Green" />
```

Now imagine that you use this skin in a page that contains the following markup.

```
<asp:Label runat="server" id="labOne" Text="Hello" />
<asp:Label runat="server" id="labTwo" ForeColor="Blue"
   Text="World" />
```

What text color will the content of these two Label controls have when rendered by the browser? In fact, both will be green, because *skin definitions override page definitions*. You can have a control ignore a skin setting via the EnableTheming property, as in the following.

```
<asp:Label id="labTwo" … EnableTheming="false" ForeColor="Blue"/>
```

There is another way to have properties defined within individual controls override skin settings. You can do so by changing the Page directive of the form and use the StyleSheetTheme attribute rather than the Theme attribute. Unlike those applied by the Theme attribute, the properties applied by the StyleSheetTheme are overridden by control properties defined within the page. As such, the StyleSheetTheme behaves in a manner more akin to the cascade within CSS. For instance, in the following example, the "World" text is blue.

```
<%@ Page … StyleSheetTheme="Cool" %>
…
<body>
  <asp:Label runat="server" id="labOne" Text="Hello" />
```

```
  <asp:Label runat="server" id="labTwo" ForeColor="Blue"
     Text="World" />
</body>
</html>
```

You can also set the `StyleSheetTheme` via the `Web.config` file. To set it in the `Web.config` file, you would use the `styleSheetTheme` attribute (rather than the `Theme` attribute), as in the following.

```
<system.web>
   …
   <pages styleSheetTheme="Cool" />
   …
</system.web>
```

Other than the fact that one allows skin properties to be overridden and the other does not, what else is different between the `Theme` and `StyleSheetTheme` attributes? A `Theme` is applied *after* the properties are applied to the server-side control, which is why the properties set by the `Theme` override those of the control. A `StyleSheetTheme` is applied *before* the properties from the server-side control, and are therefore overridden by the properties on the control. As well, the Visual Studio designer displays skins set by the `StyleSheetTheme`, but not by the `Theme`. If you specify both a `Theme` and `StyleSheetTheme`, the `Theme` takes precedence (i.e., control properties are overridden by the theme).

So which should you use? `StyleSheetTheme` is probably ideal during development because the Visual Studio designer displays the skins, and you can make quick changes to the page's Web server control appearance properties as part of your debugging and development. But because one of the primary benefits of themes is that you can change the entire appearance of a site through one simple theme change, it probably makes sense to use the `Theme` property when your site is ready for deployment; by then, you will no longer need the designer support and you will probably want to override any page properties by the formatting specified by the individual skins.

Named Skins

If you need to override the appearance of a skinned control, perhaps a better approach than using `StyleSheetTheme` is to use ***named skins***. For instance, you might not want all of your `Label` controls to have the same appearance. You can thus define alternate skin definitions for the `Label` control by giving the different skins separate `SkinID` values. You can then reference this `SkinID` in your Web Form's `Label` controls. For instance, let's define a skin file with the following content.

```
<asp:Label runat="server" ForeColor="Green" Font-Size="10pt" />
```

```
<asp:Label runat="server" ForeColor="Red" Font-Size="14pt"
   Font-Name="Verdana" Font-Bold="True" SkinID="Quote" />
```

To use the named skin in any of your Web Forms, you simply need to add the reference to the `SkinID` in the controls that will use the named skin, for instance:

```
<asp:Label ID="labQuote" runat="server" Text="Hello"
   SkinID="Quote"/>
<asp:Label runat="server" id="labMsg" ForeColor="Blue"
   Text="World" />
```

In this case, the word "Hello" appears in bold, 14pt red Verdana, whereas the word "World" appears as 10pt green text.

CORE NOTE

The `SkinId` does not have to be globally unique. It only needs to be unique for each control within the theme. For instance, each named `Label` control in a theme must have a unique `SkinID`, but a `TextBox` control could have the same `SkinId` as one of the `Label` controls.

Themes and Images

One of the more interesting features of themes is that a given theme folder can also contain images and CSS files. You can thus radically transform a Web page by substituting different images and different style sheets. For instance, different themes could use a different set of images for bullets, image buttons, or for the icons used by the `TreeView` control. The only requirement is that the skin must use a relative URL for the image. This means that the image files must exist somewhere inside the same themes folder as the skin file itself.

For instance, the following skin defines two controls. The first is a named skin that displays the masthead image for the site; the second defines the look for all `BulletedList` controls. The relative path for the images indicates that the files are contained in a subfolder named `images` within this particular theme folder.

```
<asp:Image runat="server" ImageUrl="images/logo.gif"
   AlternateText="Masthead Image" SkinID="logo" />

<asp:BulletedList runat="server" BulletStyle="CustomImage"
   BulletImageUrl="images/bullet.gif" DisplayMode="HyperLink" />
```

To use this skin, your Web Form might look like that shown here. Notice how the resulting code in the Web Form is quite simple, because the additional properties are contained in the skin rather than in the Web Form. Figure 6.6 illustrates how the

visual appearance of this form might vary simply by having different images for `logo.gif` and `bullet.gif` in two different themes containing the exact same skin.

```
<asp:Image runat="server" ID="imgLogo" SkinID="logo" />

<asp:BulletedList ID="blstSample" runat="server">
   <asp:ListItem Value="">Home</asp:ListItem>
   <asp:ListItem Value="">Browser</asp:ListItem>
   <asp:ListItem Value="">About</asp:ListItem>
</asp:BulletedList>
```

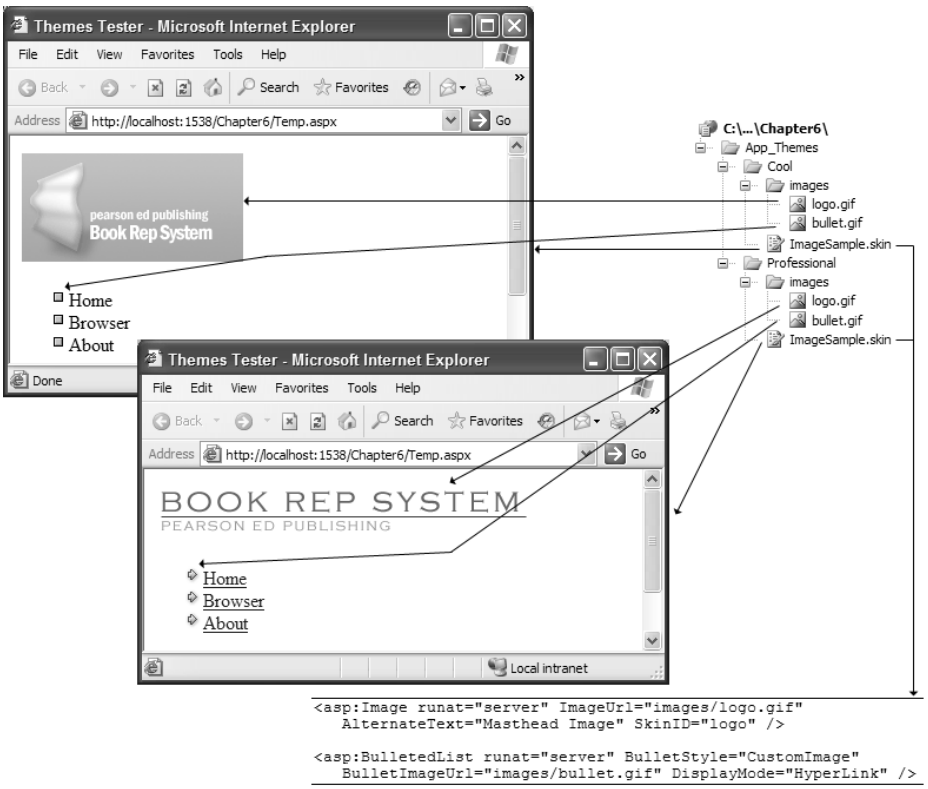

Figure 6.6 Using images in theme skins

Themes and CSS

There is a certain amount of overlap between CSS and ASP.NET skins. Both allow you to specify consistent site-wide formatting. For example, if you want all of your

text boxes to have a certain border style and color scheme, you could do so via a skin, as in the following.

```
<asp:TextBox runat="server" BackColor="Beige"
   ForeColor="DarkKhaki" BorderColor="LightGray"
   BorderWidth="1px" />
```

You could achieve the same effect by defining a CSS class in the theme's style sheet.

```
.myTextBoxes { border: 1px solid LightGray; color: DarkKhaki;
   background-color: Beige; }
```

You could then reference this class in the skin.

```
<asp:TextBox runat="server" CssClass="myTextBoxes" />
```

So which approach is better? In general, try to place as much appearance formatting as possible into a theme's CSS files. Many Web sites are created in conjunction with a Web designer, who undoubtedly is familiar already with CSS. A designer can easily modify and even swap the CSS files in any given theme without knowing anything about ASP.NET. If all of a site's formatting is contained within skin files, formatting changes will instead require the intervention of an ASP.NET developer (which is perhaps a good thing if your sole concern is to boost the employment of ASP.NET developers). As well, CSS provides more control over the appearance of a page than is possible with skins, which are limited to those appearance properties exposed by the Web server controls. If you use appearance properties to define the look of the controls in a theme's skin, the downloaded size of the resulting rendered markup will be larger than the CSS approach, because these properties will be emitted in each HTML element. Finally, even with skins, CSS probably is still necessary to format plain HTML text.

Ultimately, then, your site is much more maintainable if as much formatting as possible is contained within CSS. If your site design requires a change in the font or color, it is much more preferable to change just one file (the style sheet), rather than having to change both a style sheet and multiple skins. Finally, another benefit of CSS is that if your Web Forms use CSS rather than HTML tables for layout, different themes could completely change the layout of the site, as was illustrated back in Figure 6.2.

Thankfully, you do not have to choose between CSS and themes. Each theme can in fact contain multiple CSS files. ASP.NET automatically links all of a theme's CSS files into a page by adding the appropriate `<link>` element for each CSS file into the header of a page. Note that each page must have the `runat="server"` attribute in the `<head>` for this to occur, as shown here.

```
<head runat="server">
   <title>Some title here</title>
</head>
```

Despite the benefits of CSS, there are several things that skins can do, which CSS cannot. Some server control properties that are themeable have no CSS equivalent. For instance, you can specify the textual format to display the days of the week in a `Calendar` control or the locations for the hot spots in an `ImageMap` control. As well, because themes are applied on the server side, it is possible to dynamically specify a theme based on user input or configuration information without the bother of Javascript-based CSS file swapping.

In addition, the more complex templated controls, such as `Calendar`, `GridView`, `MultiView`, `Wizard`, and so on, are probably styled more easily through skins, because it is up to ASP.NET to determine exactly what markup to use when rendering these controls. Yet even here, there still is a role for CSS. For example, in the following `GridView` skin definition (we cover the `GridView` control later in Chapter 10), the three style templates contain appearance properties.

```
<asp:GridView runat="server" GridLines="Vertical" >
   <HeaderStyle BackColor="#FF9900" ForeColor="#FFFFFF" />
   <RowStyle BackColor="#FFFFFF" ForeColor="#333333" />
   <AlternatingRowStyle BackColor="#F2F2F2" ForeColor="#333333"/>
</asp:GridView>
```

A better approach is to define the template styling information via CSS classes.

```
.gridHeader
{
   background-color: #FF9900;
   color: #FFFFFF;
}
.gridRow
{
   background-color: #FFFFFF;
   color: #333333;
}
.gridAltRow
{
   background-color: #F2F2F2;
   color: #333333;
}
```

With the CSS classes defined, you could then simply reference these classes in the skin, as shown here.

```
<asp:GridView runat="server" GridLines="Vertical" >
   <HeaderStyle CssClass="gridHeader" />
   <RowStyle CssClass="gridRow" />
   <AlternatingRowStyle CssClass="gridAltRow" />
</asp:GridView>
```

Again, the benefit of this approach is that the typical Web designer, who probably does not know ASP.NET, can still make changes to the visual appearance of the ASP.NET application by modifying the appropriate CSS files.

Dynamically Setting the Theme

One of the key benefits that themes provide to the Web developer is the ability to programmatically change a page's theme. A page's theme can be set programmatically in the code-behind class. However, you may recall from Figure 2.3 in Chapter 2 that theme skins are applied before the controls and page are initialized. Thus, you must set the page's `Theme` property in the `PreInit` event handler for the page, as shown here.

```
protected void Page_PreInit(object o, EventArgs e)
{
    // Set theme for this page
    this.Theme = "Cool";
}
```

The `PreInit` event is new to ASP.NET 2.0. It is raised just before the `Init` event of the page, but after the controls for the page have been instantiated. Of course, with the simple example shown earlier, it makes much more sense to simply set the theme in the `Web.config` or in the `Page` directive. The real advantage of using the programmatic approach is that the page's theme can be set dynamically based on user preferences. This preference can even be persisted in a user profile (covered in Chapter 14), in a session (covered in Chapter 12), or in a database.

Like with the `Theme` property, you can also set the `StyleSheetTheme` property programmatically. However, unlike with setting the `Theme` property, the programmatic setting of the `StyleSheetTheme` property is not done in the `PreInit` method, but is achieved by overriding the property in your code-behind class, as in the following.

```
public override String StyleSheetTheme
{
  get { return "Cool"; }
}
```

Setting the theme dynamically based on user input is not quite so straightforward. The problem lies in the fact that the theme needs to be set in the `PreInit` event of the page, which is before any of the controls have had their values loaded from the view state or the form input data. For example, imagine a page with a drop-down list by which the user can select the theme for the page. The list would have an event handler that would process the user's theme selection, but unfortunately you cannot set the page theme in this event handler. It has to be set *before* the postback event handling in the page's `PreInit` event. Yet during the `PreInit` event, you cannot yet know what the user selected!

The solution to this conundrum is that you need some way to store the user's theme selection in a way that is available to the `PreInit` event, and then reload the page so that the `PreInit` event is invoked again. How then can you store the user's theme selection? Later in Chapter 14, you will learn about the user profile system, which could be used for this problem. An alternative we will use here is to use ASP.NET ***session state*** (refer to Chapter 12 for more detail).

Session state allows you to store and retrieve values for a browser session as the user moves from page to page in the site. Recall that HTTP is a stateless protocol. This means that a Web server treats each HTTP request for a page as an independent request and retains no knowledge of code-behind data members or form values used during previous requests. ASP.NET session state provides a way to identify requests received from the same browser during a limited period of time and provides the ability to persist values for the duration of that session.

ASP.NET session state can be accessed from an ASP.NET Web Form via the `Session` property of the `Page` base class. This property references an object collection that is indexed by name. When you save an object in the session collection, you identify it by using a name, which can be any text string that you want. You can thus use session state to save the user's theme choice in the list event handler, as shown here.

```
protected void drpThemes_SelectedIndexChanged(object o,
      EventArgs e)
{
   // Save theme name in session. You identify this value
   // using "theme" but you could use any name
   Session["theme"] = drpThemes.SelectedItem.Text;

   // Re-execute page so PreInit event can be re-triggered
   Server.Transfer(Request.Path);
}
```

The `PreInit` event handler can now retrieve the user's choice using the following.

```
protected void Page_PreInit(object sender, EventArgs e)
{
   // Retrieve theme from session
   string theme = (string)Session["theme"];

   // Must make sure that session exists
   if (theme != null)
   {
      // Set the page theme
      this.Page.Theme = theme;
   }
   else
   {
      // Set default theme if nothing in session because no theme
```

```
        // has been selected yet (or the session has timed-out)
        this.Page.Theme = "Cool";
    }
}
```

Notice that retrieving the theme name from the session collection requires a casting operation (from `object` to `string`). Also, the method cannot assume that the session collection actually contains a theme name. It might be the first time the page has been requested and thus the theme does not exist yet in the session. Alternately, the session may have timed out and thus is empty. For this reason, any time you retrieve an item from session state, you must always verify that the item does exist, typically by comparing it to `null`.

Creating a Sample Page with Two Themes

To finish our coverage of themes, let's examine an example that demonstrates the dynamic selection of themes. The example page contains a drop-down list that allows the user to select one of two different themes to be applied to that page. The markup for this page is shown in Listing 6.3. Notice that the page contains no appearance markup, only structured content. All formatting is contained in the theme skins and CSS files.

Listing 6.3 ThemeTester.aspx

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

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Themes Tester</title>
</head>
<body>
    <form id="form1" runat="server">
        <div id="container">
            <div id="header">
                <asp:Image runat="server" ID="imgLogo" SkinID="logo"/>
            </div>
            <div id="menu">
                <asp:BulletedList ID="blstSample" runat="server"  >
                    <asp:ListItem Value="">Home</asp:ListItem>
                    <asp:ListItem Value="">Products</asp:ListItem>
                    <asp:ListItem Value="">About</asp:ListItem>
                </asp:BulletedList>
            </div>
```

```
            <div id="content">
               <h1>Technical Books</h1>
               <p>
               Microsoft. Cisco. IBM. Hewlett Packard. Intel. Adobe.
               Macromedia. There's a reason the top technology
               companies choose Pearson Education as their publisher
               partners. Publishing over a thousand new titles each
               year, much of our content is also available online –
               so busy professionals and students can access it from
               any computer, day or night.
               </p>
               <div id="register">
                  <fieldset>
                      <legend>Contact Us</legend>
                      <asp:Label ID="labName" runat="server"
                         AssociatedControlID="txtName"
                         Text="<u>N</u>ame" />
                      <br />
                      <asp:TextBox ID="txtName" runat="server"
                        AccessKey="n" TabIndex="1" /><br />
                      <asp:Label ID="labEmail" runat="server"
                         AssociatedControlID="txtEmail"
                         Text="E<u>m</u>ail" /><br />
                      <asp:TextBox ID="txtEmail" runat="server"
                         AccessKey="m" TabIndex="2" /><br />
                      <asp:Label ID="labZip" runat="server"
                         AssociatedControlID="txtZip"
                         Text="<u>Z</u>ip" /><br />
                      <asp:TextBox ID="txtZip" runat="server"
                         AccessKey="z" TabIndex="3" /><br />
                      <asp:Button ID="btnSubmit" runat="server"
                         Text="Send" TabIndex="4" AccessKey="d" />
                  </fieldset>
               </div>
               <p>
               <asp:DropDownList ID="drpThemes" runat="server"
                     AutoPostBack="True"
                     OnSelectedIndexChanged=
                      "drpThemes_SelectedIndexChanged" >
                  <asp:ListItem>Pick a theme</asp:ListItem>
                  <asp:ListItem>Cool</asp:ListItem>
                  <asp:ListItem>Professional</asp:ListItem>
               </asp:DropDownList>
               </p>
            </div>
         </div>
      </form>
</body>
</html>
```

The markup contains a drop-down list that allows the user to select the theme. The code-behind for the page performs this processing, as shown in Listing 6.4.

Listing 6.4 ThemeTester.aspx.cs

```
public partial class ThemeTester : System.Web.UI.Page
{
   /// <summary>
   /// Set the page's theme based on the user's selection
   /// </summary>
   protected void Page_PreInit(object sender, EventArgs e)
   {
      // Retrieve theme from session
      string theme = (string)Session["theme"];

      // Must make sure that session exists
      if (theme != null)
      {
         this.Page.Theme = theme;
      }
      else
      {
         // Set default theme
         this.Page.Theme = "Cool";
      }
   }
   /// <summary>
   /// Process the user's theme choice by saving it in
   /// session state
   /// </summary>
   protected void drpThemes_SelectedIndexChanged(object sender,
         EventArgs e)
   {
      // Ignore the first item ("pick a theme") in list
      if (drpThemes.SelectedIndex != 0)
      {
         // Save theme in session
         Session["theme"] = drpThemes.SelectedItem.Text;

         // Re-request page
         Server.Transfer(Request.Path);
      }
   }
}
```

By keeping your markup devoid of appearance formatting, it can be quite radically transformed by your themes. This example has two different themes: one called `Cool` and the other called `Professional`. Figure 6.7 illustrates how this page appears in its two themes (the `Cool` theme is the one on the right with two layout columns).

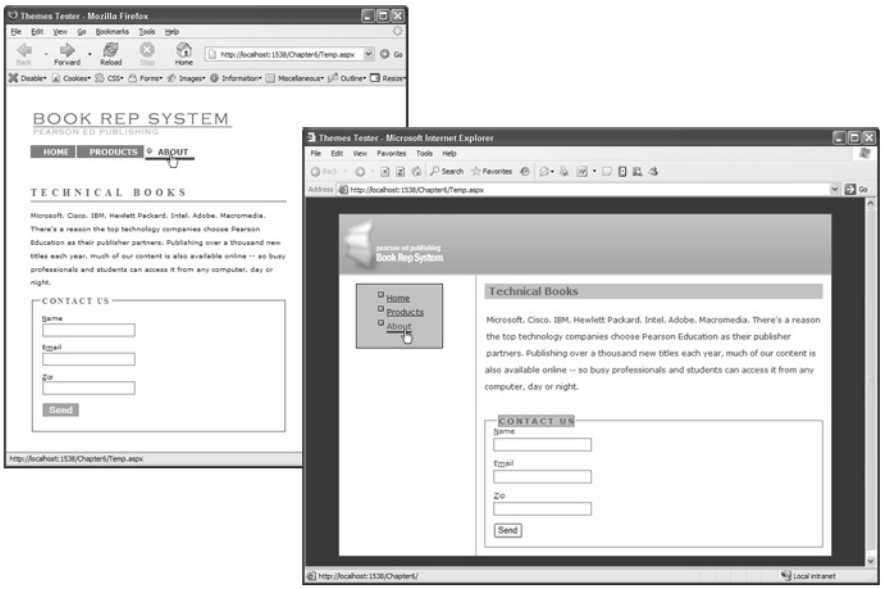

Figure 6.7 ThemeTester.aspx with its two themes

The skins for these two themes (shown in Listings 6.5 and 6.6) are quite similar as well as quite straightforward. Notice that both skins delegate the actual formatting to each theme's CSS file.

Listing 6.5 Cool.skin

```
<asp:Image runat="server" ImageUrl="images/cool.gif"
   AlternateText="Masthead Image" SkinID="logo" />

<asp:BulletedList runat="server"
   BulletStyle="CustomImage"
   BulletImageUrl="images/bullet.gif"
   DisplayMode="HyperLink" />

<asp:TextBox runat="server" CssClass="txtBox" />

<asp:Button runat="server" CssClass="myButton"/>
```

Listing 6.6 Professional.skin

```
<asp:Image runat="server" ImageUrl="images/prof.gif"
   AlternateText="Masthead Image" SkinID="logo" />

<asp:BulletedList runat="server" BulletStyle="NotSet"
   DisplayMode="HyperLink"/>

<asp:TextBox runat="server" CssClass="txtBox" />

<asp:Button runat="server" CssClass="myButton"/>
```

As mentioned, the majority of the heavy lifting for these themes is done by the CSS file for each theme. Listings 6.7 and 6.8 contain the content of each CSS file. The two CSS files are similar in that they use class, ID, and descendant selectors to specify the text formatting and the positioning for the different elements in your Web Form. The `professionalStyles.css` file is the easier of the two to understand because it contains no positioning styles. The only complexity in this style sheet is how the unordered list items are turned into a horizontal menu—that is, by turning off the bullets (`list-item: none`), keeping the list items together on a single line (`display: inline`), and by styling the link and hover pseudo elements.

Listing 6.7 professionalStyles.css

```
body {
   background-color: White;
   font-size: small;
}

h1 {
   margin-top: 3em;
   font-family: Georgia, Times New Roman, serif;
   font-size: 1.2em;
   padding-bottom: 4px;
   border-bottom: 1px solid #cc0000;
   color: #666666;
   text-transform: uppercase;
   letter-spacing: 0.4em;
}

p {
   font:  normal .8em/2.0em Verdana, Arial , sans-serif;
}

#container {
   position: relative;
```

```
   top: 30px;
   left: 30px;
   width: 400px;
}

/* style the list as a horizontal menu */
#menu {
   font-family: Verdana, Arial , sans-serif;
   font-size: 0.9em;
}

#blstSample {
   margin-left: 0;
   padding-left: 0;
   white-space: nowrap;
}

#blstSample li {
   display: inline;
   list-style: none;
   text-transform: uppercase;
}

#blstSample a {
   font-weight: bold;
   padding: 3px 10px 3px 20px;
   margin-right: 2px;
}

#blstSample a:link,  #blstSample a:visited {
   text-decoration: none;
   color: white;
   background-color: Gray;

}
#blstSample a:hover {
   color: #cc0000;
   border-bottom: 4px solid #cc0000;
   padding-bottom: 2px;
   text-decoration: none;
   background: url(images/bullet.gif) no-repeat 0 50%;
   background-color: white;
}

/* style the form elements */
fieldset {
   border: 1px solid #cc0000;
   padding: 1em;
}
```

```
.txtBox {
   border:1px solid #666666;
   margin: 0.2em 0 0.8em 0.2em;
}

.myButton {
    border:1px solid gray;
    color:#FFFFFF;
    background-color:#FF9900;
    font-size:1em;
    font-weight: bold;
    margin: 0.2em 0 0.8em 0.2em;
}

label {
   font:  normal 0.8em Verdana, Arial , sans-serif;
   margin: 0.2em 0 0 0.2em;
}

legend {
   font-family: Georgia, Times New Roman, serif;
   font-size: 1em;
   font-weight: bold;
   color: #666666;
   text-transform: uppercase;
   letter-spacing: 0.2em;
}
```

The `coolStyles.css` file is a bit more complex. It uses CSS positioning and margins to place the menu `<div>` and the content `<div>` into two separate columns. The menu `<div>` is floated to the left (`float: left`) of the subsequent content in the form (the content `<div>`) and given a specified width. The content `<div>` is given a left margin that places all of the content to the right of the menu `<div>`. One interesting feature of this CSS file is the use of the so-called Tan Hack to deal with a bug with Internet Explorer 6.0 and earlier. In this case, Internet Explorer is adding the content `<div>` left margin setting to the form's `<input>` elements (the `TextBox` and `Button` controls). To fix this problem, the style sheet uses the `* html` selector (which is only supported by IE) to override the correct margin settings and apply a negative margin. The rest of the CSS is simple text formatting.

Listing 6.8 coolStyles.css

```
body {
   background-color: #0A4581;
   font-size: small;
}
```

```
h1 {
   margin-top: 0;
   padding: 0.2em;
   font-family: Verdana, Tahoma, Helvetica, Arial, sans-serif;
   font-size: 1.2em;
   background-color: #ABC6EE;
   color: #666666;
}

P {
   font: normal .9em/2.2em Verdana, Helvetica, Arial, sans-serif;
}

/* style each of the principle div elements */
#container {
   width: 90%;
   margin: 10px auto;
   background-color: #ffffff;
   color: #333333;
   border: 1px solid gray;
}

#content {
   margin-left: 16em;
   border-left: 1px solid gray;
   padding: 1em;
}


#header {
   padding: 0;
   height: 92px;
   background: url(images/header_background.gif);
}

/* style the list as a vertical list of links */
#menu {
   float: left;
   color: black;
   width: 9em;
   margin: 1em;
   padding: 0.5em;
   border: 1px solid #000;
   font-family: Verdana, Tahoma, Helvetica, Arial, sans-serif;
   background-color: #ABC6EE;
}

#blstSample li {
   padding-top: 0.4em;
}
```

```
#blstSample a:link,  #blstSample a:visited {
   text-decoration: underline; color: blue;
}
#blstSample a:hover  {
   color: #cc0000;
   border-bottom: 3px solid #cc0000;
   text-decoration: none;
}

/* style the form elements */
#register {
   font-family: Verdana, Tahoma, Helvetica, Arial, sans-serif;
}

fieldset {
   border: 1px solid #666666;
   padding: 1em;
}

legend {
   font-size: 1em;
   font-weight: bold;
   color: #666666;
   text-transform: uppercase;
   letter-spacing: 0.2em;
   background-color: #ABC6EE;
   border: 1px solid #666666;
}

.txtBox {
   border:1px solid #666666;
   margin: 0.3em 0 0.8em 0.2em;
}

/* IE 6 adds the #content margin to the text box and button so we
   must give the text boxes and buttons a negative margin using
   a CSS format only understood by IE (which will override the
   margin set in the .txtBox rule just above).
*/
* html .txtBox { margin-left: -16em; }
* html .myButton { margin-left: -16em; }

label {
   font:  normal 0.8em Verdana, Arial , sans-serif;
}
```

One other feature worth mentioning about these two themes is their use of `em` units for sizing. By avoiding the use of pixels, these two layouts still work even if users change the display size of the text in their browser (see Figure 6.8). We could have achieved a similar result as well by using percent units (e.g., `font-size: 90%;`) for sizing.

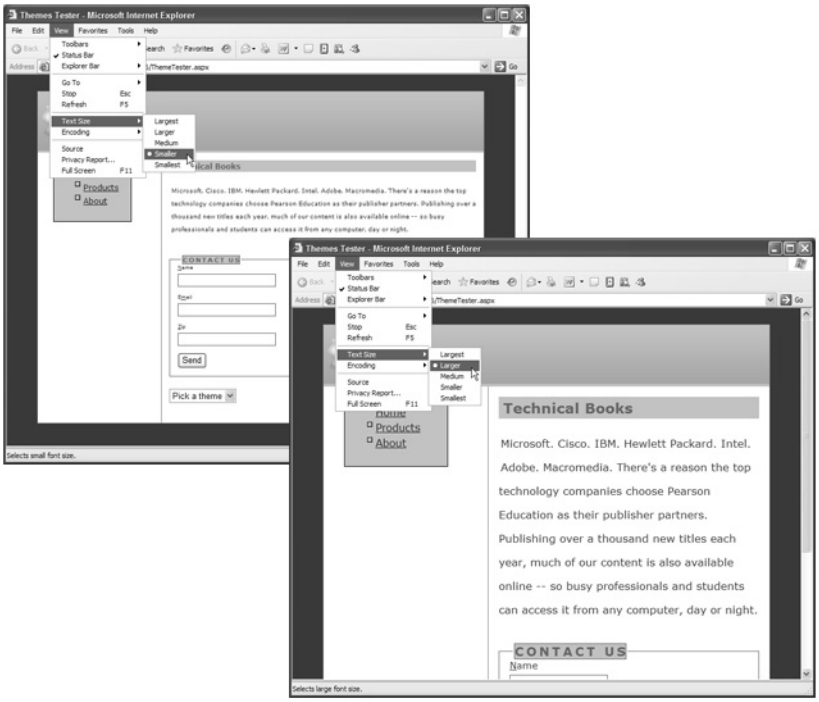

Figure 6.8 Cool theme with different user-selected font sizes

The `em` unit is the size of the character box for the element's parent. If no other font sizes are specified in other containers, a font size of `1em` is equivalent to the default font size (determined by the browser). The principle used in these styles is to set the default font size in the body to `small`, and then make all other dimensions relative to whatever the browser sets for the `small` font size.

```
body { font-size: small; }
h1 { font-size: 1.2em; }
p { font-size: 0.8em; }
```

If the browser's font size for `small` is 14px, the `h1` is displayed at around 17px (14 × 1.2) and the `p` is displayed at around 11px (14 × 0.8). If the user increases the size of her text through browser preferences, so that `small` is now 18px, the `h1` and the `p` are 21 and 14 pixels, respectively (18 × 1.2 and 18 × 0.8).

Master Pages

ASP.NET 2.0 provided a treasure trove of new features. Perhaps none of these generated as much anticipation as ***master pages***. This new feature allows the developer to define the structural layout for multiple Web Forms in a separate file and then apply this layout across multiple Web Forms. You can thus move common layout elements, such as logos, navigation systems, search boxes, login areas, and footers, out of all the individual pages and into a single master page.

Any developer who has had to create and maintain an ASP.NET Web application that consists of more than two or three pages should be able to see the value of this ability. In most Web applications, the individual pages typically share a common structure or a common look and feel across the entire site. For instance, most pages within a site may have a logo in the upper-left corner, a global navigation system across the top, a secondary navigation system down the left side of the page, and a footer at the bottom of the page. It is clearly less than ideal to replicate the markup and code for this common structure across multiple pages.

Master pages provide a solution to this problem. They allow the developer to create a consistent page structure or layout without duplicating code or markup. Master pages are created in much the same way as any other Web Form. They contain markup, server controls, and can have a code-behind class that responds to all the usual page lifecycle events. However, they do have their own unique extension (`.master`) as well as a different directive at the top of the page. As well (and most importantly), they also contain one or more `ContentPlaceHolder` controls.

The `ContentPlaceHolder` control defines a region of the master page, which is replaced by content from the page that is using this master page. That is, each Web Form that uses the master page only needs to define the content unique to it within the `Content` controls that correspond to the `ContentPlaceHolder` controls in the master page, as illustrated in Figure 6.9.

Like any Web server control, each `ContentPlaceHolder` control within a master page must have a unique `Id`.

```
<asp:ContentPlaceHolder id="contentBody" runat="server">
</asp:ContentPlaceHolder>
```

This `Id` is used to link the `Content` controls in the various Web Forms that use the master page. This link is made via the `ContentPlaceHolderId` property of the `Content` control.

```
<asp:Content id="myContent" ContentPlaceHolderId="contentBody"
   runat="server">
</asp:Content>
```

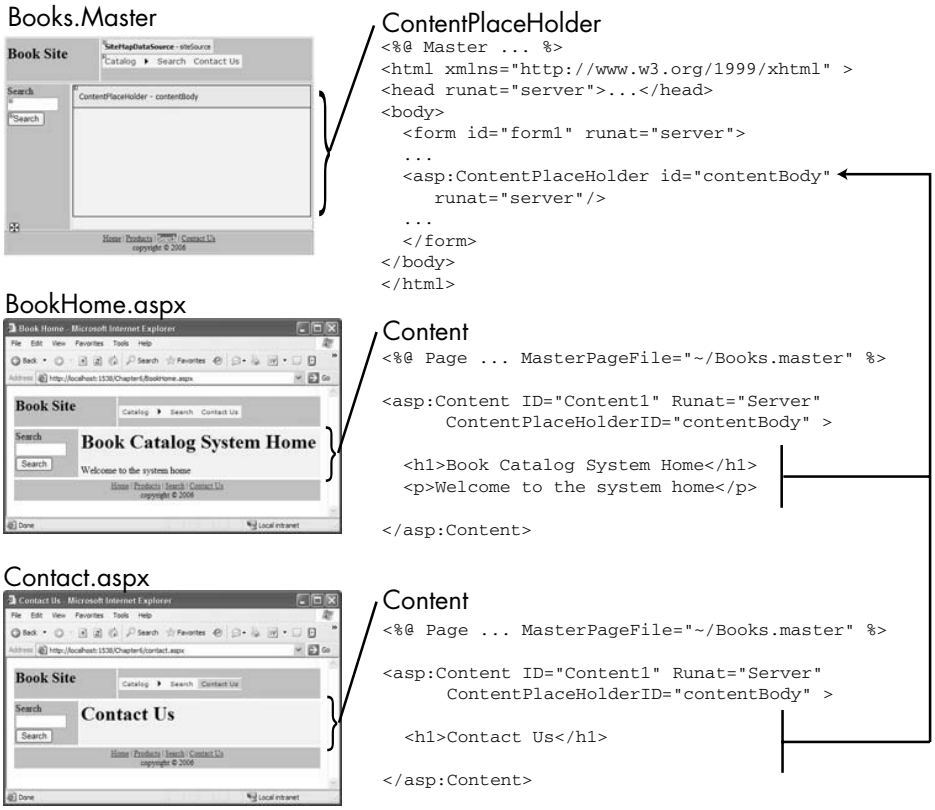

Figure 6.9 Master pages

A master page significantly simplifies the markup for pages that use it. The master page contains the complete XHTML document structure. That is, the `html`, `head`, and `body` elements are contained only within the master page. The `.aspx` files that use the master page only need define the content that will be inserted into the placeholders in the master page. In fact, these `.aspx` files can *only* contain content within `Content` controls. Unlike HTML frames (which perform a somewhat similar function in HTML), master pages are transparent to the user (that is, the user is unaware of their existence) because ASP.NET merges the content of the master page with the content of the requested page.

Visual Studio and its designer completely support master pages. You can visually edit the master page, as well as visually edit the individual `aspx` pages as they would appear within the master page. Figure 6.10 illustrates the `BookHome.aspx` page within the Visual Studio designer. Only the contents of the `Content` control are editable; the master page markup is displayed (ghosted out) but cannot be edited.

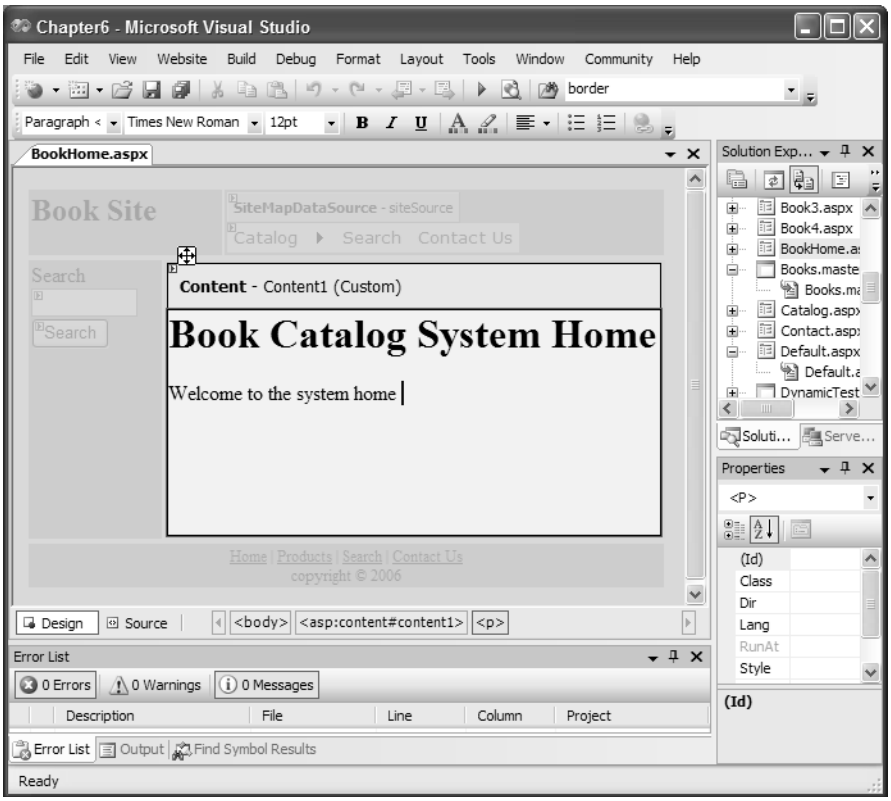

Figure 6.10 Visual Studio support for master pages

You can specify the master page to use when you add a new Web Form in Visual Studio simply by turning on the Select Master Page option in the Add New Item dialog box (see Figure 6.11).

When you choose this Select Master Page option, Visual Studio then displays the Select A Master Page dialog (see Figure 6.12). This dialog displays all the master pages in the Web site (i.e., all the files with the `.master` extension).

Of course, you can use a master page in any Web Form without the intervention of Visual Studio simply by adding the appropriate `MasterPageFile` attribute to the form's `Page` directive, as shown in the following.

```
<%@ Page MasterPageFile="~/Books.master" ... Title="Book Home" %>
```

This example also contains the `Title` attribute. Because the master page must define the `<head>` section, you need some way to specify the page title (which is displayed in the browser's title bar) in the individual pages themselves. The `Title`

directive provides one way to do this; the other is to programmatically set the page's `Title` property. If you do not set it, the page is displayed in the browser with "Untitled" in the title bar.

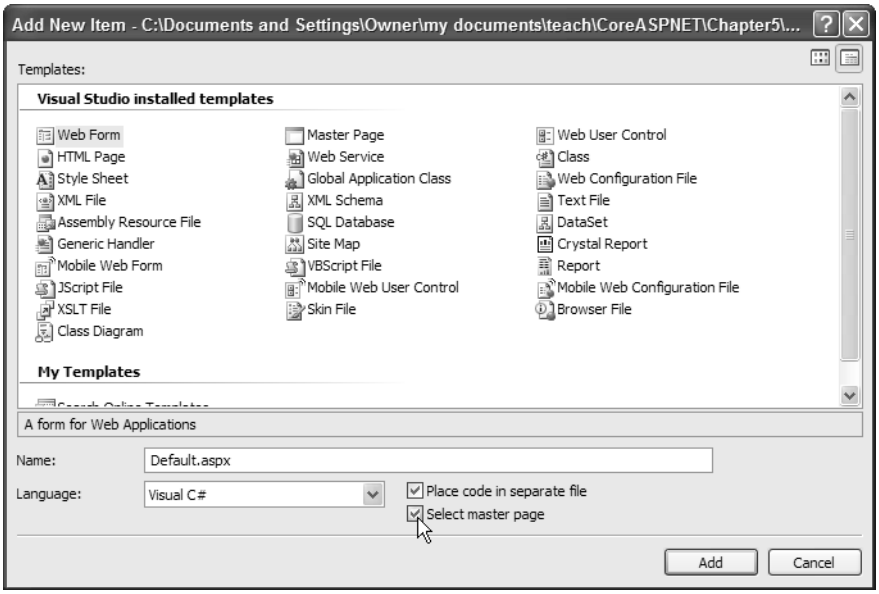

Figure 6.11 Creating a Web Form that uses a master page

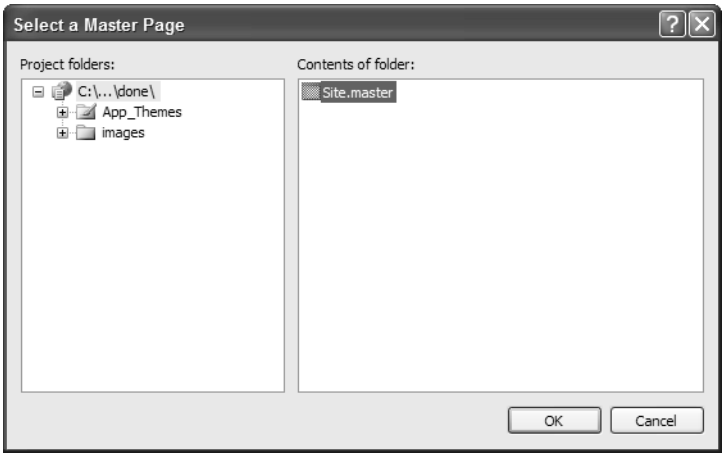

Figure 6.12 Selecting the master page

As an alternative to specifying the `MasterPageFile` attribute on each Web Form in your site, you can specify it at once globally for all pages in the site via the `Pages` element in the `Web.config` file.

```
<system.web>
   …
   <pages masterPageFile="~/Books.master" />
</system.web>
```

Note that setting the `MasterPageFile` attribute in the Web Form overrides the setting in the `Web.config` file.

Defining the Master Page

As already mentioned, a master page looks like a regular Web Form except that it has a different extension and directive. As well, a master page contains one or more `ContentPlaceHolder` controls. The following example illustrates a sample master page (the positioning and formatting is contained in its style sheet).

```
<%@ Master Language="C#" AutoEventWireup="true"
   CodeFile="Site.master.cs" Inherits="Site" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
   <title></title>
   <link rel="stylesheet" type="text/css" href="~/styles.css" />
</head>
<body>
   <form id="form1" runat="server">
      <div id="container">
         <div id="header">
         <asp:Image runat="server" ID="imgLogo"
            ImageUrl="~/images/cool.gif" />
         </div>
         <div id="sideArea">
            <div id="menu">
               <asp:BulletedList ID="blstSample" runat="server"
                     DisplayMode="HyperLink" >
                  <asp:ListItem Value="BookHome.aspx">
                    Home
                  </asp:ListItem>
                  <asp:ListItem Value="Products.aspx">
                    Products
                  </asp:ListItem>
                  <asp:ListItem Value="Contact.aspx">
```

```
                    Contact Us
                  </asp:ListItem>
               </asp:BulletedList>
            </div>
            <div id="sideAreaBox">
               <asp:ContentPlaceHolder ID="sideContent"
                  runat="server">
               <p>default side content</p>
               </asp:ContentPlaceHolder>
            </div>
         </div>
         <div id="mainArea">
            <asp:ContentPlaceHolder ID="mainContent"
               runat="server">
            <p>default main content</p>
            </asp:ContentPlaceHolder>
         </div>
         <div id="footer">
            <p>
            This site is not real. It is an example site for Core
            ASP.NET book.
            </p>
         </div>
      </div>
   </form>
</body>
</html>
```

Notice that both `ContentPlaceHolder` controls contain default content. This default content is displayed if the `aspx` files that use this master page do not provide `Content` controls for these `ContentPlaceHolder` controls.

Let us now define a Web Form that uses this master page.

```
<%@ Page Language="C#" MasterPageFile="~/Site.master"
   CodeFile="BookHome.aspx.cs" Inherits="BookHome" Title="Book Home"
%>

<asp:Content ID="Content1"
   ContentPlaceHolderID="mainContent"
   Runat="Server">

   <h1>Book Rep System Home</h1>
   <p>Welcome to the book rep system</p>
</asp:Content>

<asp:Content ID="Content2"
   ContentPlaceHolderID="sideContent"
   Runat="Server">
```

```
    <h2>New Releases</h2>
    <p>Core C#</p>
</asp:Content>
```

By removing the markup for the common elements and placing them into the master page, you are left with a Web Form that is quite striking in its clarity and conciseness. It contains only the content that is unique to this page. The result, using a style sheet quite similar to that shown in Listing 6.8, is shown in Figure 6.13.

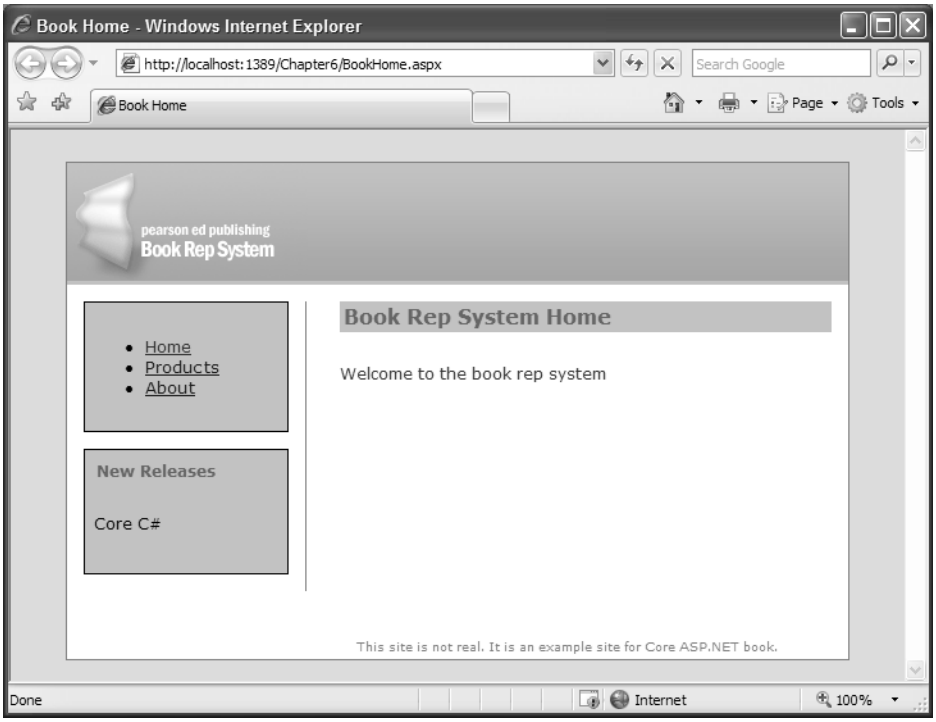

Figure 6.13 Sample Web Form and master page

Notice that all content in this page is contained within either of the two `Content` controls. Because the Web Form uses a master page, ASP.NET does not allow us to place any markup outside of these two controls. If you do place content outside of a `Content` control, you will see a Parser Error (see Figure 6.14).

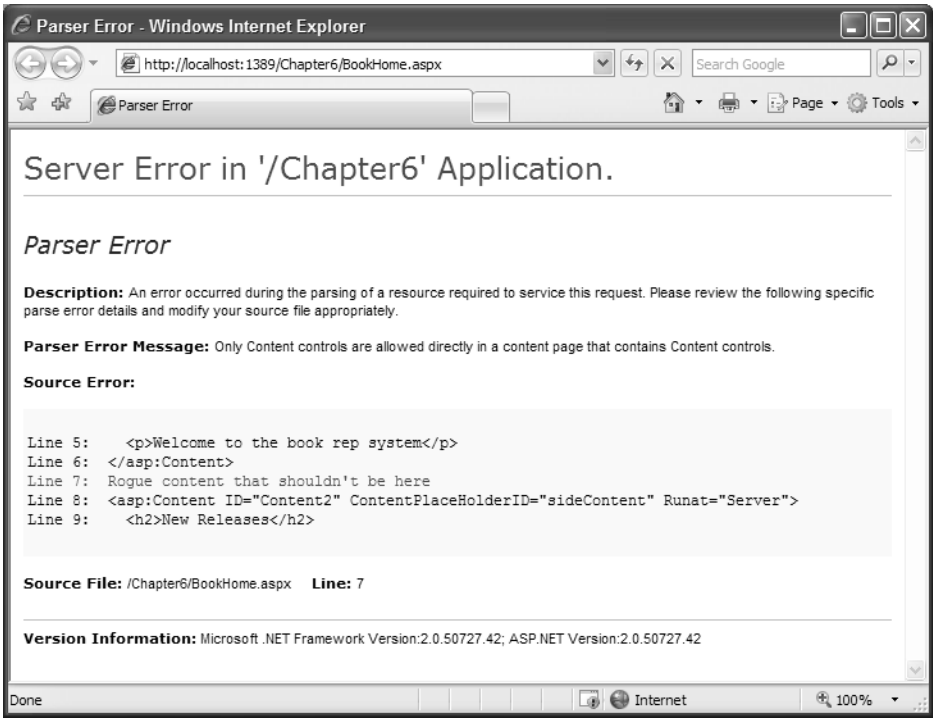

Figure 6.14 Parser error

Nested Master Pages

Master pages can be nested so that one master page contains another master page as its content. This can be particularly useful for Web sites that are part of a larger system of sites. For instance, the master page shown in Figure 6.13 could be just one intranet in a much larger system. You might thus want a way to move between these intranets. Nested master pages provide this mechanism. Figure 6.15 illustrates how the master page in Figure 6.13 can be a child nested inside another parent master page.

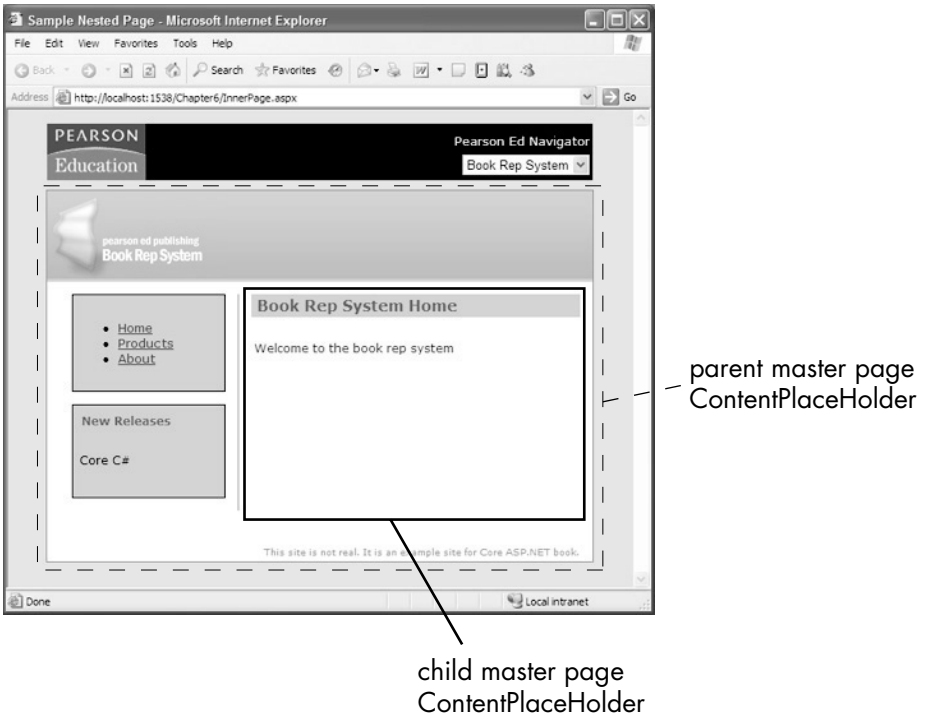

Figure 6.15 Nested master pages

To nest a master page within another master page, the child master page simply needs a `Content` control that maps to a `ContentPlaceHolder` control in the parent master page, as well as the appropriate `MasterPageFile` attribute in the `Master` directive. For instance, you could modify the previous example of a master page so that it is contained within a `Content` control, and thus becomes a child master page.

```
<%@ Master MasterPageFile="~/Parent.master"
   CodeFile="Child.master.cs" Inherits="Child" %>

<asp:Content ID="Content1" ContentPlaceHolderID="parentContent"
      Runat="Server">

   <div id="container">
      <div id="header">
      <asp:Image runat="server" ID="imgLogo"
         ImageUrl="~/images/cool.gif" />
```

```
      </div>
      <div id="sideArea">
         <div id="menu">
            <asp:BulletedList ID="blstSample" runat="server"
                  DisplayMode="HyperLink">
               <asp:ListItem Value="">Home</asp:ListItem>
               <asp:ListItem Value="">Products</asp:ListItem>
               <asp:ListItem Value="">About</asp:ListItem>
            </asp:BulletedList>
         </div>
         <div id="sideAreaBox">
            <asp:ContentPlaceHolder ID="sideContent"
               runat="server">
            <p>default side content</p>
            </asp:ContentPlaceHolder>
         </div>
      </div>
      <div id="mainArea">
         <asp:ContentPlaceHolder ID="mainContent" runat="server">
         <p>default main content</p>
         </asp:ContentPlaceHolder>
      </div>
      <div id="footer">
         <p>
          This site is not real. It is an example site for Core
          ASP.NET book.
         </p>
      </div>
   </div>
</asp:Content>
```

Your parent master page is quite straightforward with just the one `ContentPlaceHolder` control. Once again, the formatting and positioning is handled by CSS.

```
<%@ Master Language="C#" AutoEventWireup="true"
   CodeFile="Parent.master.cs" Inherits="Parent" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title></title>
    <link rel="stylesheet" type="text/css" href="~/styles.css" />
    <link rel="stylesheet" type="text/css" href="~/parent.css" />
</head>
<body>
    <form id="form1" runat="server">
```

```
    <div id="topNav">
       <p>
       Pearson Ed Navigator<br />
       <asp:DropDownList ID="drpPlaces" runat="server">
          <asp:ListItem>Book Rep System</asp:ListItem>
         <asp:ListItem>Addison-Wesley</asp:ListItem>
         <asp:ListItem>Prentice Hall</asp:ListItem>
       </asp:DropDownList>
       </p>
    </div>
    <asp:contentplaceholder id="parentContent" runat="server" />
    </form>
</body>
</html>
```

CORE NOTE

There is no designer support for nested master pages in Visual Studio.

How Master Pages Work

When a page that uses a master page (i.e., a content page) is requested, ASP.NET merges the content page and master page together (assuming of course that both have already been compiled). It does so by inserting the master page's content at the beginning of the content page's control tree. *This means that the master page content is actually a control that is added to the page.* In fact, the master page control is a subclass of the `UserControl` class (which is covered later in the chapter). Thus, the master page control is the root control in the page's control hierarchy. The content of each individual `Content` control is then added as a collection of child controls to the corresponding `ContentPlaceHolder` control in the master page control. This occurs after the `PreInit` event but before the `Init` event of the page.

Like any control, master pages have their own sequence of events that are fired as part of the page lifecycle. Because it is the root control in the page's control hierarchy, its events are always fired before the control events in any content pages. As a result, it is important that you do not conceptualize the master page as a "master" in the sense of a "controller" page. That is, the master page should *not* be orchestrating and controlling what happens in the individual content pages it contains. The master page is simply a template page and should control only those server controls it directly contains. Thus, in general, you should endeavor to keep the master page and the content pages as uncoupled as possible.

Referencing Issues with Master Pages

Because master page content is ultimately inserted into the content page's control tree, there are some potential issues to be aware of with external references inside of master pages. Because the user's request is for the content page and not the master page, all URL references are relative to the content page. This can cause some problems when the master page and the content pages are in different folders within the site.

For instance, let us imagine a situation in which your content page is in a folder under the root named `content` and your master page is contained in a folder under the root named `master`; inside this `master` folder, you also have a subfolder named `images`. Let's say you want to reference an image named `logo.gif`, which is inside the `images` folder of `master`. In such a case, the following references inside your master page do not work.

```
<img src="images/logo.gif" />
body { background-image: url(images/logo.gif); }
<img src="~/master/images/logo.gif" />
```

The first two references in fact refer to `/content/images/logo.gif`, because all relative references are relative to the content page. The second reference doesn't work either because the application relative symbol (~) only works with server controls.

One alternative to this problem is to use an absolute reference to the site root.

```
<img src="/master/images/logo.gif" />
```

The problem with this approach is its fragility; that is, the reference breaks if the site structure changes. Another approach is to use a server control, as in the following.

```
<asp:Image runat="server" id="a"
   ImageUrl="~/master/images/logo.gif" />
```

This approach works because server controls within master pages that contain URLs have their URLs modified by the ASP.NET environment.

Programming the Master Page

As you have seen, the combined master and content pages appear to the user as a single page whose URL is that of the requested content page. From a programming perspective, the two pages act as separate containers for their respective controls, with the content page also acting as the container for the master page. Yet, because the master page content becomes merged into the content page, you can also programmatically reference public master page members in the content page.

Why would you want to do this? Perhaps your master page contains a control whose content varies depending upon which content page is being viewed. One common example is a master page that contains an advertisement image; the precise advertisement to display in that control might vary depending upon which part of the site is being viewed. The master page might also contain a secondary navigation area that again differs depending upon which part of the site is being visited.

There are two principal ways of accessing content in the master page within the content page. You can do so via the `FindControl` method of the `Master` object or via public members that are exposed by the master page.

Let us begin by examining the `FindControl` approach. The `FindControl` method searches the current naming container for a specified server control and returns a typed reference to it. Thus, you can retrieve a `TextBox` named `txtOne` from the current Web Form via the following.

```
TextBox one = (TextBox)this.FindControl("txtOne");
String contents = one.Text;
```

Notice that the reference returned from the method needs to be cast to the appropriate control type. Of course, it doesn't make too much sense to search the `Page` naming container for the control when you could replace these two lines with the simpler form with which you are accustomed, namely `string contents = txtOne.Text`. Where `FindControl` is truly useful is in those situations where you need to reference a control that is "hidden" within some other naming container, such as a `Wizard` or `GridView` control or within a master page. For instance, imagine that you have the following `HyperLink` control contained within your master page.

```
<asp:HyperLink ID="imgbtnAd" runat="server" />
```

Now, let's say that you want to change the image and the destination URL for this control in each of your content pages. You could so with the following code somewhere in your content page's code-behind class.

```
HyperLink ad = (HyperLink)Master.FindControl("imgbtnAd");
if (ad != null)
{
   ad.ImageUrl = "~/Images/something.gif";
   ad.NavigateUrl = "http://www.somewhere.com";
}
```

Although this approach does work, it is not ideal. A better approach is to safely encapsulate the data you need from the master page into public properties, which you could then access within your content pages. This way, your content pages are not coupled to the implementation details of the master page. The following example adds two properties to the code-behind for the master page. It allows content pages to manipulate the image and navigation URLs of the `HyperLink` control in the master page.

```
public partial class ProgrammedContentMaster :
   System.Web.UI.MasterPage
{
   public string AdImageUrl
   {
      get { return imgbtnAd.ImageUrl; }
      set { imgbtnAd.ImageUrl = value; }
   }

   public string AdNavigateUrl
   {
      get { return imgbtnAd.NavigateUrl; }
      set { imgbtnAd.NavigateUrl = value; }
   }
}
```

Your content pages can now use these properties; to do so requires the use of the `Master` property of the `Page` class. Unfortunately, you cannot simply reference one of these properties directly from the `Master` property in your content page code-behind, as shown here.

```
Master.AdImageUrl = "~/Images/something.gif";
```

You cannot do this because the `Master` property returns an object of type `MasterPage`, which is the base class for all master pages. Of course, this general `MasterPage` class knows nothing of the properties you have just defined in your master page. Instead, you must cast the `Master` property to the class name of your master page, and then manipulate its properties.

```
ProgrammedContentMaster pcm = (ProgrammedContentMaster)Master;

pcm.AdImageUrl = "~/Images/something.gif";
pcm.AdNavigateUrl = "http://www.somewhereelse.com";
```

An alternative to casting is to add the following `MasterType` directive to your content pages.

```
<%@ MasterType VirtualPath="~/ProgrammedContentMaster.master" %>
```

This changes the `Master` property of the `Page` class so that it is strongly typed (that is, it is not of type `MasterPage`, but of type `ProgrammedContentMaster`). This eliminates the need for casting the `Master` property, and thus you can manipulate the custom properties directly.

```
Master.AdImageUrl = "~/Images/something.gif";
Master.AdNavigateUrl = "http://www.somewhereelse.com";
```

Master Pages and Themes

ASP.NET themes provide a centralized way to define the appearance of a Web site. Although your content pages can make use of themes, you can simply use master pages to set the theme for your site as a whole. That is, you cannot use the `Theme` attribute within the `Master` directive at the top of the master page (although you can still do so in the `Page` attribute of each of your content pages).

However, you can use your master page to provide a user interface element that allows the user to browse and dynamically set the theme for your content pages. You may recall back in Listing 6.4, you programmatically set the theme of a page within the `PreInit` event of the page and you used ASP.NET session state to store the currently selected theme. Unfortunately, you cannot set the theme for your content pages within the `PreInit` event of the master page. Instead, you must have the `PreInit` event handler within the content page. If you have hundreds or even thousands of content pages, does that mean you must have the same `PreInit` event handler in all of these content pages?

Fortunately, no; you can make use of some simple object-oriented inheritance so that you only need code the `PreInit` event handler once. Recall that the code-behind class for all Web Forms has the `Page` class as its base class. You can define your own class that inherits from this base class and which contains the `PreInit` event handler that sets the themes. This new class then becomes the base class for the code-behind classes in your site.

The example site contained in Listings 6.9 through 6.13 demonstrates this technique (along with the other material covered in this chapter on master pages). It uses master pages to specify the structure of the site, themes, and CSS to control the appearance of the site's pages. The master page contains both an advertisement banner image that is customized by each content page as well as a theme selector. Figures 6.16 and 6.17 illustrate how the example appears with its two themes.

Listing 6.9 contains the code for the new class that will become the base class for all your Web Forms. Like Listing 6.4, it simply retrieves the theme selected by the user from the session state and applies it to the (content) page. This class file should be saved in your project's `App_Code` folder.

Listing 6.9 ParentPage.cs

```
public class ParentPage: Page
{
    /// <summary>
    /// Sets the theme of the page based on the current
    /// session state
    /// </summary>
    protected void Page_PreInit(object sender, EventArgs e)
    {
        string themeName = (string)Session["themeName"];
```

```
        if (themeName != null)
            this.Page.Theme = themeName;
        else
            this.Page.Theme = "Cool";
    }
}
```

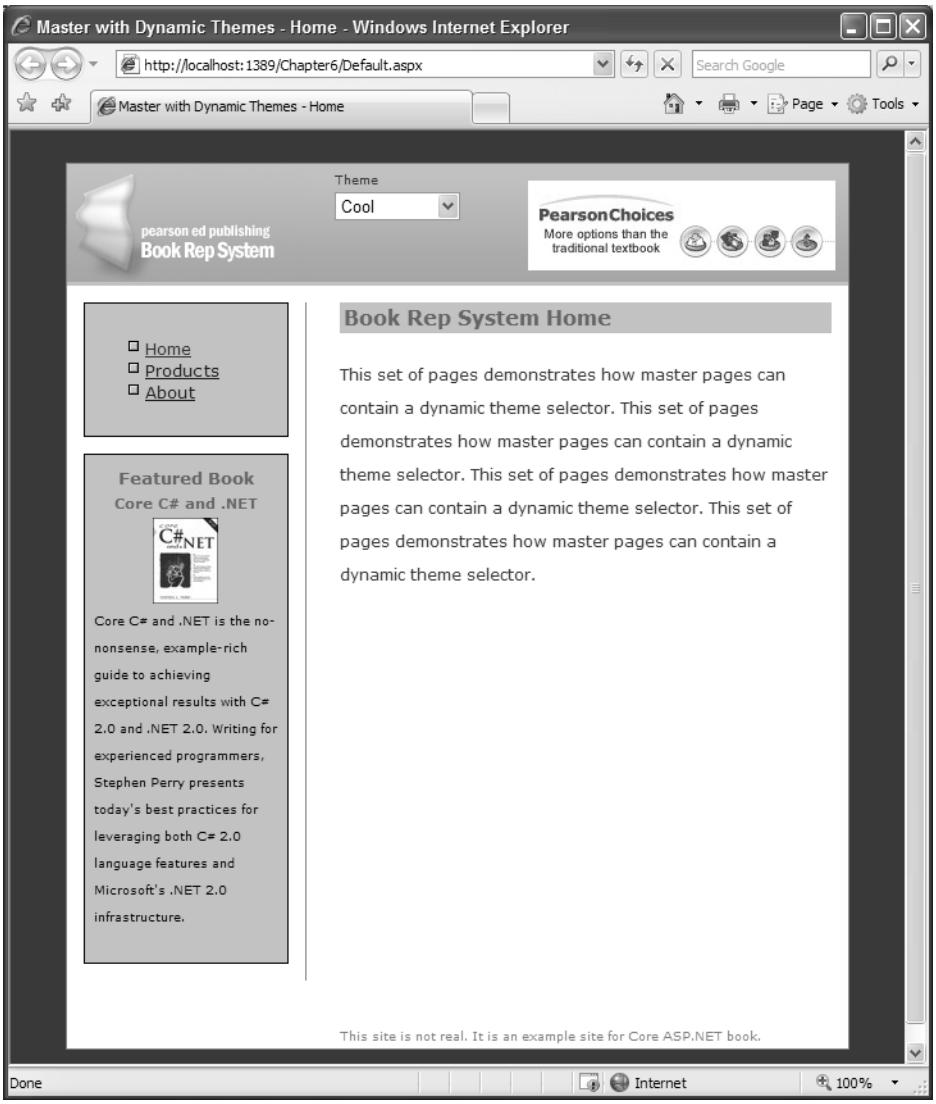

Figure 6.16 Combining cool theme with master pages

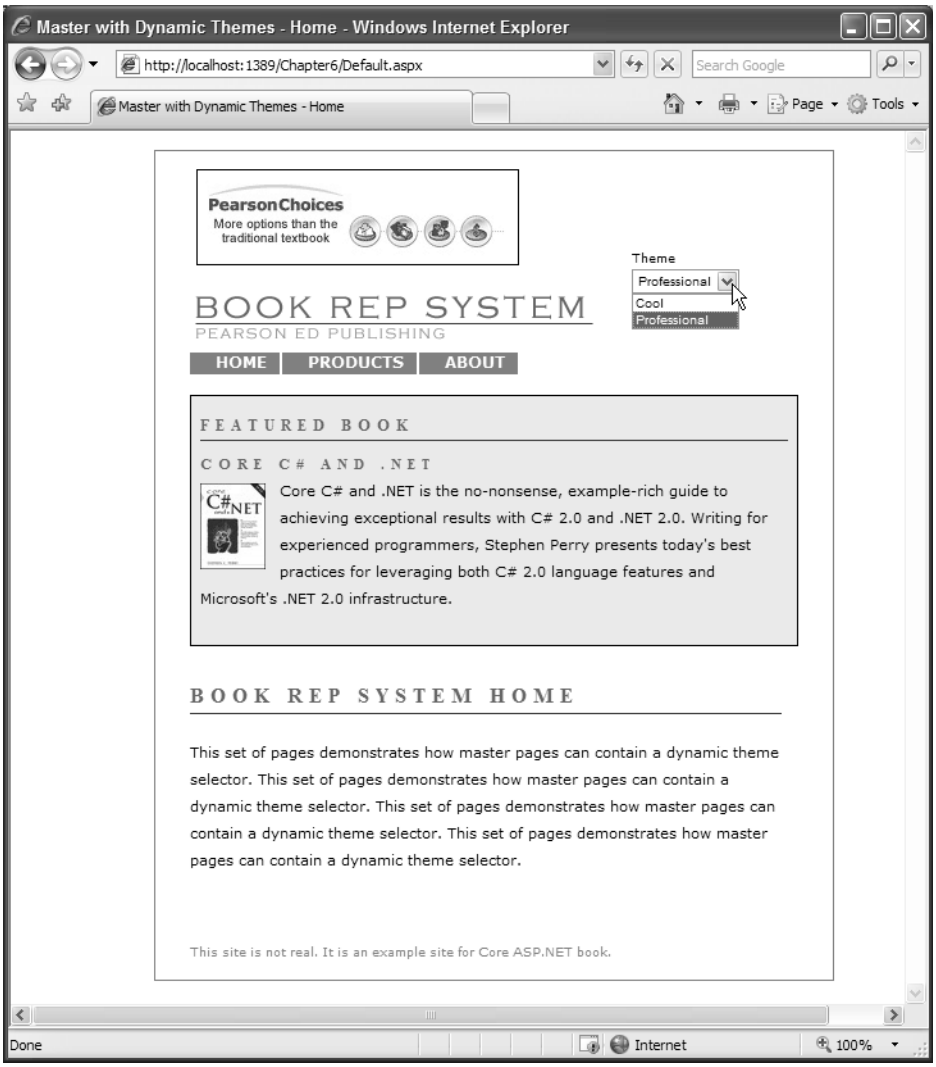

Figure 6.17 Combining professional theme with master pages

I won't bother showing you all the content pages in the site. Listing 6.10 illustrates one sample content page (the home page).

Listing 6.10 Default.aspx

```
<%@ Page Language="C#" MasterPageFile="~/BookWithThemes.master"
   AutoEventWireup="true" CodeFile="Default.aspx.cs"
```

```
   Inherits="Default"
   Title="Master with Dynamic Themes - Home" %>
<%@ MasterType VirtualPath="~/BookWithThemes.master" %>

<asp:Content ID="Content1" ContentPlaceHolderID="sideContent"
     Runat="Server">
  <h2>Featured Book</h2>
  <h3>Core C# and .NET<br />
  <asp:Image ID="imgBook" runat="server" CssClass="sideImage"
     ImageUrl="~/images/0131472275.gif"
     AlternateText="Cover image of Core C# book"/></h3>
  <p>Core C# and .NET is the no-nonsense, example-rich guide to
  achieving exceptional results with C# 2.0 and .NET 2.0. Writing
  for experienced programmers, Stephen Perry presents today's
  best practices for leveraging both C# 2.0 language features and
  Microsoft's .NET 2.0 infrastructure.</p>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="mainContent"
     Runat="Server">
  <h1>Book Rep System Home</h1>
  <p>
  This set of pages demonstrates how master pages can contain a
  dynamic theme selector. This set of pages demonstrates how
  master pages can contain a dynamic theme selector. This set of
  pages demonstrates how master pages can contain a dynamic theme
  selector. This set of pages demonstrates how master pages can
  contain a dynamic theme selector.
  </p>
</asp:Content>
```

The code-behind for this sample content page (see Listing 6.11) must be altered to use your `ParentPage` class as its base class. Notice as well that it modifies properties exposed by the master page to customize the advertisement that appears within the master page.

Listing 6.11 Default.aspx.cs

```
public partial class Default : ParentPage
{
   protected void Page_Load(object sender, EventArgs e)
   {
      Master.AdImageUrl = "~/Images/ads/ad1.gif";
      Master.AdNavigateUrl =
         "http://www.aw-bc.com/newpearsonchoices";
   }
}
```

Listing 6.12 defines the markup for the master page.

Listing 6.12 BookWithThemes.master

```
<%@ Master Language="C#" AutoEventWireup="true"
   CodeFile="BookWithThemes.master.cs"
   Inherits="BookWithThemes" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
   <title></title>
</head>
<body>
   <form id="form1" runat="server">
      <div id="container">

         <div id="header">
            <div id="themes">
               <p>Theme <br />
               <asp:DropDownList ID="drpThemes" runat="server"
                  AutoPostBack="true" OnSelectedIndexChanged=
                    "drpTheme_selectedChanged">
               </asp:DropDownList>
               </p>
            </div>
            <div id="bannerAd">
               <asp:HyperLink ID="imgbtnAd" runat="server" />
            </div>
            <div class="clearBreak"></div>
            <div id="logo">
               <asp:Image runat="server" ID="imgLogo"
                  SkinID="logo"/>
            </div>
         </div>

         <div id="sideArea">
            <div id="masterMenu">
               <asp:BulletedList ID="blstSample" runat="server" >
                  <asp:ListItem Value="Default.aspx">
                     Home</asp:ListItem>
                  <asp:ListItem Value="Products.aspx">
                    Products</asp:ListItem>
                  <asp:ListItem Value="About.aspx">
                    About</asp:ListItem>
               </asp:BulletedList>
            </div>
```

```
            <div id="sideAreaBox">
               <asp:ContentPlaceHolder ID="sideContent"
                  runat="server">
               <p>default side content</p>
               </asp:ContentPlaceHolder>
            </div>
         </div>

         <div id="mainArea">
            <asp:ContentPlaceHolder ID="mainContent"
               runat="server">
            <p>default main content</p>
            </asp:ContentPlaceHolder>
         </div>

         <div id="footer">
            <p>
            This site is not real. It is an example site for Core
            ASP.NET book.
            </p>
         </div>
      </div>
   </form>
</body>
</html>
```

Like the other master pages examined in this chapter, this master page contains no formatting. The CSS and skins of your themes control the appearance of each page. The master page contains four main sections: `<div id="header">`, `<div id="sideArea">`, `<div id="mainArea">`, and `<div id="footer">`. The `sideArea` and `mainArea` sections contain the two `ContentPlaceHolder` controls. The `header` section contains the logo image, an advertisement image, and a `DropDownList` that allows the user to change the theme. The theme list is not filled via markup, but instead is filled programmatically in the code-behind class for this master page, as shown in Listing 6.13.

Listing 6.13 BookWithThemes.master.cs

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

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

// Note that you need to include this namespace reference
using System.IO;

public partial class BookWithThemes : System.Web.UI.MasterPage
{
   /// <summary>
   /// Populate theme list based on themes in App_Themes folder
   /// </summary>
   protected void Page_Load(object sender, EventArgs e)
   {
      // Dynamically load the drop-down list only the first time
      if (!IsPostBack)
      {
         // Themes must be in this location
         string path = Server.MapPath("~/App_Themes");

         // Verify this location exists
         if (Directory.Exists(path))
         {
            // Retrieve array of theme folder names
            String[] themeFolders =
               Directory.GetDirectories(path);

            // Process each element in this array
            foreach (String folder in themeFolders)
            {
               // Retrieve information about this folder name
               DirectoryInfo info = new DirectoryInfo(folder);

               // Add this folder name to the drop-down list
               drpThemes.Items.Add(info.Name);
            }

            // Once all the themes are added to the list, you
            // must make the content page's current theme the
            // selected item in the list

            // First search the list items for the page's
            // theme name
            ListItem item =
               drpThemes.Items.FindByText(Page.Theme);

            // Now set the selected index of the list (i.e.,
            // select that list item) to the index of the list
            // item you just found.
```

```
            drpThemes.SelectedIndex =
               drpThemes.Items.IndexOf(item);
         }
      }
   }

   /// <summary>
   /// Event handler for the theme selector
   /// </summary>
   protected void drpTheme_selectedChanged(object s, EventArgs e)
   {
      // Save theme in session
      string theme = drpThemes.SelectedItem.Text;
      Session["themeName"] = theme;

      // Re-request page
      string page = Request.Path;
      Server.Transfer(page);
   }

   /// <summary>
   /// Property to get/set the url of the advertisement image
   /// </summary>
   public string AdImageUrl
   {
      get { return imgbtnAd.ImageUrl; }
      set { imgbtnAd.ImageUrl = value; }
   }

   /// <summary>
   /// Property to get/set the url for the link surrounding the
   /// advertisement image.
   /// </summary>
   public string AdNavigateUrl
   {
      get { return imgbtnAd.NavigateUrl; }
      set { imgbtnAd.NavigateUrl = value; }
   }
}
```

The only remaining files are the skins and CSS for the two themes. For space reasons, I do not include them here (although they can be downloaded from my Web site at `http://www.randyconnolly.com/core`). They are quite similar to those shown in Listings 6.5 through 6.8.

User Controls

The previous section examined the powerful master page mechanism in ASP.NET. A master page allows the developer to define the structural layout for multiple Web Forms in a separate file and then apply this layout across multiple Web Forms. You can thus move common layout elements out of all the individual pages and into a single master page. As a consequence, the master page feature is a powerful mechanism for reducing code-behind and markup duplication across a Web application.

Yet, it still is possible, even when using a master page, for presentation-level (i.e., user interface) duplication to exist in a site. For instance, consider the page shown back in Figure 6.16. In this example, the feature book box is displayed only on the home page. But imagine that you want this box to appear on several other pages, but not on every page (if it was to appear on every page, you would place it in the master page). Your inner programmer should shirk at the idea of simply copying and pasting the markup and code for implementing this featured book box to multiple pages. Although such a solution is easy in the short term, maintaining this type of approach in the long run is a real headache; every time you have to make a change to this box, you have to change it in multiple places.

User controls are the preferred ASP.NET solution to this type of presentation-level duplication. They provide a cleaner approach to user interface reuse then copying and pasting or using the server-side includes of classic ASP. In addition, user controls in ASP.NET are very simple to create and then use. As well, they follow the same familiar development model as regular Web Forms.

Creating and Using User Controls

User controls are created in a manner similar to Web Forms. As with Web Forms, a user control can contain markup as well as programming logic. Also like Web Forms, the programming for a user control can be contained within the same file as the markup, or contained within a separate code-behind file. After a user control is defined, it can then be used in Web Forms, master pages, or even other user controls.

The markup for a user control is contained in a text file with the `.ascx` extension. This file can contain any necessary programming logic within embedded code declaration block (i.e., within `<script runat="server">…</script>` tags) embedded within this `.ascx` user control file. Alternately, the programming code can be contained in a code-behind class for the user control. However, the code-behind class for a user control inherits from the `UserControl` class, rather than the `Page` class.

Walkthroughs 6.3 and 6.4 demonstrate how to create and then use a simple user control.

Walkthrough 6.3 *Creating a User Control*

1. Use the Website → Add New Item menu option in Visual Studio, or right-click the project in Solution Explorer and choose Add New Item.
2. From the Add New Item dialog box, choose the Web User Control template and name the control `FeatureBookControl.ascx`. Click Add.

   Notice that a blank user control contains no content other than a `Control` directive. Unlike with a Web Form, there is no HTML skeleton with a user control. It is completely up to the developer to decide what markup should appear in the user control.
3. Change the user control so that it has the following content.

```
<%@ Control Language="C#" AutoEventWireup="true"
  CodeFile="FeatureBookControl.ascx.cs"
  Inherits="FeatureBookControl" %>

<h2>Featured Book</h2>
<h3>Core C# and .NET</h3>
<asp:Image ID="imgBook" runat="server" CssClass="sideImage"
    ImageUrl="~/images/0131472275.gif"
    AlternateText="Cover image of Core C# book"/>

<p>
Core C# and .NET is the no-nonsense, example-rich guide to
achieving exceptional results with C# 2.0 and .NET 2.0.
Writing for experienced programmers, Stephen Perry presents
today's best practices for leveraging both C# 2.0 language
features and Microsoft's .NET 2.0 infrastructure.
</p>
```

4. Save the control.

Now that you have created a user control, you can use it in a Web Form, master page, or other user control. To make use of a user control, you must follow two steps:

1. Indicate that you want to use a user control via the `Register` directive, as shown in the following.

```
<%@ Register TagPrefix="SomePrefix" TagName="SomeName"
   Src="someControl.ascx" %>
```

   `TagPrefix` determines a unique namespace for the user control, which is necessary to differentiate multiple user controls with the same name. `TagName` is the unique name for the user control, whereas `Src` specifies the virtual path to the user control, such as `MyControl.ascx` or `~/Controls/MyControl.ascx`.

2. After registering the user control, place the user control tag in the markup just as you would with any Web server control (including the `runat="server"` and `Id` properties). For instance, if you had the user control specified in step 1, the following markup adds this control.

```
<SomePrefix:SomeName id="myUC1" runat="server" />
```

CORE NOTE

As an alternative to manually entering these two steps, you can also simply drag-and-drop the user control onto your page while it is in Design view in Visual Studio.

Walkthrough 6.4 ***Using a User Control***

1. Create a new Web Form or edit an existing Web Form. For instance, you could use the `Default.aspx` page from Listing 6.10.
2. Add the following `Register` directive to the page.

```
<%@ Register Src="FeatureBookControl.ascx"
   TagName="FeatureBookControl" TagPrefix="uc" %>
```

3. In a sensible spot, add the following markup to the page. If you are using the `Default.aspx` page from Listing 6.10, replace the markup in the first `Content` control with the following.

```
<uc:FeatureBookControl ID="myFeaturedBook" runat="server" />
```

4. Save and test the page. The user control should be displayed in the page.

Adding Data and Behaviors to the User Control

In the user control created in Walkthrough 6.3, the information on the featured book was hardcoded. A more realistic version of this user control would interact with some type of business object or database. Thus, like most Web Forms, most user controls also include some type of programming logic.

You also may want to customize some aspect of the user control, so that it appears or behaves differently when used in different pages or controls. For instance, in the example Featured Book user control, it might be useful to specify the category of book you want to see in the control. You can easily add this type of customization to a

control, by adding public properties to the user control. These public properties can then be manipulated declaratively or programmatically by the containing page.

For instance, Listing 6.14 demonstrates the definition of a `FeatureCategory` property in the code-behind class for the `FeatureBookControl` user control. The `Page_Load` in this simplified example modifies the output of the control based on the value of this property.

Listing 6.14 FeatureBookControl.ascx.cs

```
public partial class FeatureBookControl :
   System.Web.UI.UserControl
{
   // Data member for feature category property
   public string _category;

   protected void Page_Load(object sender, EventArgs e)
   {
      // Real-world example would use category values and content
      // from a database or business object. Here you
      // simply hardcode the values

      imgBook.ImageUrl = "~/images/";
      if (FeatureCategory == "Programming")
      {
         labTitle.Text = "Framework Design Guidelines";
         imgBook.ImageUrl += "0321246756.gif";
         labDescription.Text = "This book can improve …";
      }
      else
      {
         labTitle.Text = "Core C# and .NET";
         imgBook.ImageUrl += "0131472275.gif";
         labDescription.Text = "Core C# and .NET is the …";
      }
      imgBook.AlternateText = "Cover image of " + labTitle.Text;
   }

   public string FeatureCategory
   {
      get { return _category; }
      set { _category = value; }
   }
}
```

This code-behind class assumes that the markup for the `FeatureBookControl` user control has been modified, as shown in Listing 6.15.

Listing 6.15 FeatureBookControl.ascx

```
<%@ Control Language="C#" AutoEventWireup="true"
   CodeFile="FeatureBookControl.ascx.cs"
   Inherits="FeatureBookControl" %>

<h2>Featured Book</h2>
<h3><asp:Label ID="labTitle" runat="server"  /></h3>
<asp:Image ID="imgBook" runat="server" CssClass="sideImage" />
<p>
<asp:Label ID="labDescription" runat="server" />
</p>
```

This public property in the user control can now be used wherever you use this user control. For instance, the following example illustrates one way that this property could be used.

```
<uc:FeatureBookControl ID="myFeaturedBook" runat="server"
   FeatureCategory="Programming" />
```

Summary

This chapter examined the different ways to customize the visual appearance of your Web application in ASP.NET 2.0. It began with the simplest, namely the common appearance properties that are available for most server controls. These properties allow you to control colors, borders, and text formatting. Although these properties are very useful for customizing the appearance of your Web output, they do not contain the full formatting power of Cascading Style Sheets. Luckily, ASP.NET server controls fully support styles via the `CssClass` and the `Style` properties. I recommended that you minimize the amount of appearance formatting in your server controls as much as possible, and instead externalize appearance settings in your CSS files. The benefit to this approach is that your Web Form's markup becomes simpler, easier to modify, and scales better to different devices.

The chapter also covered two important features of ASP.NET 2.0: themes and master pages. Themes provide a way to customize the appearance of Web server controls on a site-wide basis. Themes can still be integrated with CSS; doing so allows the developer to completely separate style from content. The master pages mechanism provides a much sought-after templating technique to ASP.NET. With master pages, elements that are common throughout a site, such as headers, footers, navigation elements, and the basic site layout itself, can be removed from Web Forms and placed instead within a master page. This significantly simplifies the Web Forms within a site, making the site as a whole easier to create and maintain.

The brief final section in the chapter covered user controls. These are an essential part of most real-world Web sites. User controls provide a consistent, object-oriented approach to user interface reuse in ASP.NET.

The next chapter covers another vital part of your Web site's appearance, its navigation system. It examines how you can programmatically move from page to page as well as the new controls in ASP.NET 2.0 devoted to navigation: the `Menu`, `Tree`, and `SiteMap` controls.

Exercises

The solutions to the following exercises can be found at my Web site, `http://www.randyconnolly.com/core`. Additional exercises only available for teachers and instructors are also available from this site.

1. Create a page named `PropertyTester.aspx`. This page should allow the user to dynamically change various appearance properties of some sample controls on the page, as shown in Figure 6.18. Ideally, the code-behind class uses an instance of the `Style` class as a data member, whereas event handlers simply modify this instance and apply it to the controls.

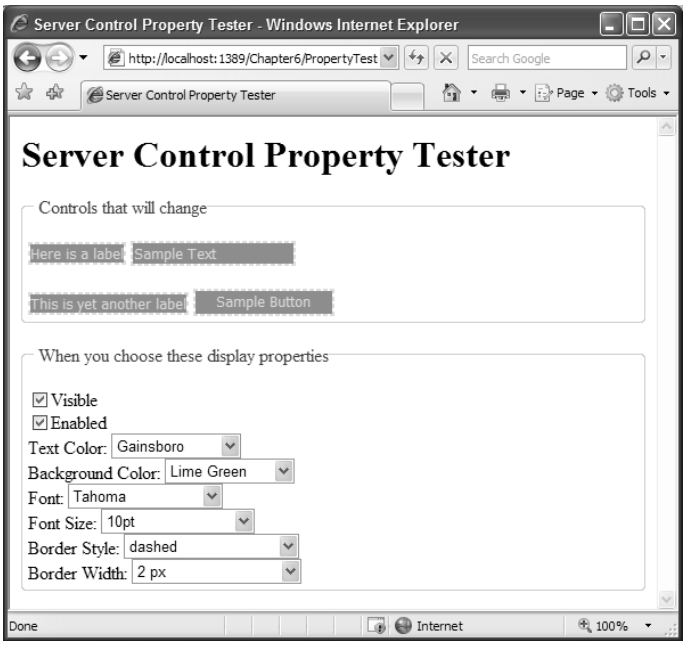

Figure 6.18 PropertyTester.aspx

2. Create a new theme to be used by the `ThemeTester.aspx` example from Listing 6.3.
3. Create a master page named `ThreeColumns.master` that contains a header, three content columns, and a footer. The three columns should each contain a `Content` control. Create a demonstration page that uses this master.
4. Create a user control named `ThemeSelector.ascx` that allows the user to change the theme used on the page. This control should only display themes that exist. Create a demonstration page that uses this user control.

Key Concepts

- Master pages
- Named collection
- Named skins
- Session state
- Skins
- Themes
- User controls

References

Allen, Scott. "Master Pages in ASP.NET." `http://odetocode.com`.

Allen, Scott. "Themes in ASP.NET." `http://odetocode.com`.

Murkoth, Jeevan C. "Master Pages in ASP.NET 2.0." *dotnetdevelopersjournal.com* (December 2004).

Onion, Fritz. "Master Your Site Design with Visual Inheritance and Page Templates." *MSDN Magazine* (June 2004).

Winstanley, Phil. "Skin Treatment: Exploiting ASP.NET 2.0 Themes." *asp.netPro Magazine* (October 2005).