MARUDHAR KESARI JAIN COLLGE FOR WOMEN , VANIYAMBADI

PG DEPARTMENT OF COMPUTER APPLICATIONS

CLASS	**: II MCA**
SUBJECT CODE	**: GCA43**
SUBJECT NAME	**: WEB APPLICATION USING C#**

SYLLABUS

UNIT - IV: CUSTOM CONTROLS Teaching Hours: 11

User Controls – Creating a Simple User Control – Visual Studio.NET Custom Control Support – Independent User Controls – Integrated User Controls – User Control Events – Limitations – Deriving Custom Controls.

Chapter 22: Custom Controls

Overview

Component-based development encourages you to divide the logic in your application into discrete, independent blocks. Once you've made the jump to custom classes and objects, you can start creating modular web applications (and even desktop applications) that are built out of reusable units of code. But while these objects help work out thorny data access procedures or custom business logic, they don't offer much when it comes to simplifying your user interface code. If you want to create web applications that reuse a customized portion of user interface, you're still stuck rewriting control tags and reconfiguring page initialization code in several different places.

It doesn't have to be this way. ASP.NET includes tools for modularizing and reusing portions of user interface code that are just as powerful as those that allow you to design custom business objects. There are two main tools at your fingertips, both of which we'll explore in this chapter:

- **User controls** allow you to reuse a portion of a page, by placing it in a special .ascx file. ASP.NET also allows you to make smart user controls that provide methods and properties, and configure their contained controls automatically.
- Custom or **derived controls** allow you to build a new control by inheriting from an ASP.NET control class. With custom controls there is no limit to what you can do, whether it's adding a new property or tweaking the rendered HTML output.

User Controls

User controls look pretty much the same as ASP.NET web forms. Like web forms, they are composed of an HTML-like portion with control tags (the .ascx file) and can optionally use a .vb code-behind file with event-handling logic. They can also include the same range of HTML content and ASP.NET controls, and they experience the same events as the Page object (such as Load and PreRender). The only differences between user controls and web pages are:

- User controls begin with a <%@ Control %> directive instead of a <%@ Page %> directive.
- User controls use the file extension .ascx instead of .aspx, and their code-behind files inherit from the System.Web.UI.UserControl class. In fact, the UserControl class and the Page class both inherit from the same base classes, which is why they share so many of the same methods and events, as shown in the inheritance diagram in Figure 22-1.

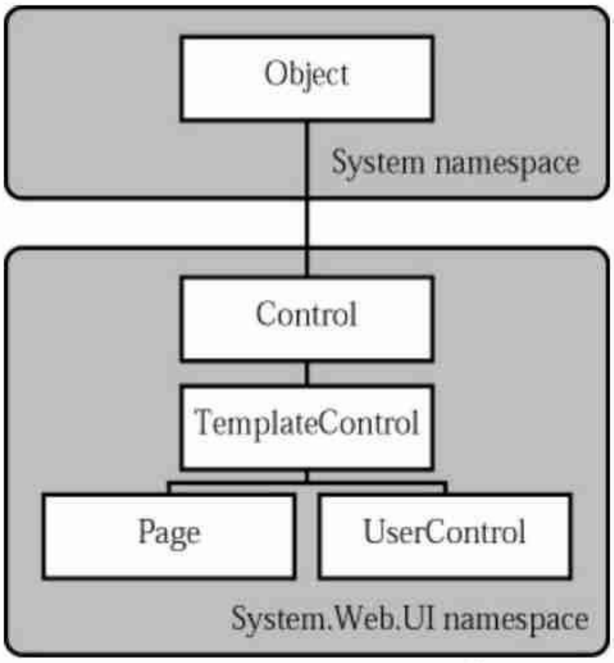


Figure 22-1: The Page and UserControl inheritance chain

- User controls can't be requested directly by a client. Instead, user controls are embedded inside other web pages.

Creating a Simple User Control

You can create a user control in Visual Studio .NET in much the same way as you add a web page. Just right-click on the project in the Solution Explorer, and select Add | Add User Control. If you aren't using Visual Studio .NET, you simply begin by creating an .ascx text file.

The following user control contains a single label control.

```
<%@ Control Language="VB" AutoEventWireup="false"
   Inherits="Footer" %>

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

Note that the Control directive uses the same attributes that are used in the Page directive for a web page, including Language, AutoEventWireup, and Inherits. Optionally, you could add a Src attribute if you aren't precompiling your code-behind class (and Visual Studio .NET will add a Codebehind attribute to track the file while editing). To refresh your memory about page attributes, refer to Chapter 5.

The code-behind class for this sample user control is similarly straightforward. It uses the UserControl.Load event to add some text to the label.

```
Public MustInherit Class Footer
  Inherits UserControl

   Protected lblFooter As Label

   Private Sub Load(sender As Object, e As EventArgs) _
     Handles MyBase.Load
       lblFooter.Text = "This page was served at "
       lblFooter.Text &= DateTime.Now.ToString()
   End Sub

End Class
```

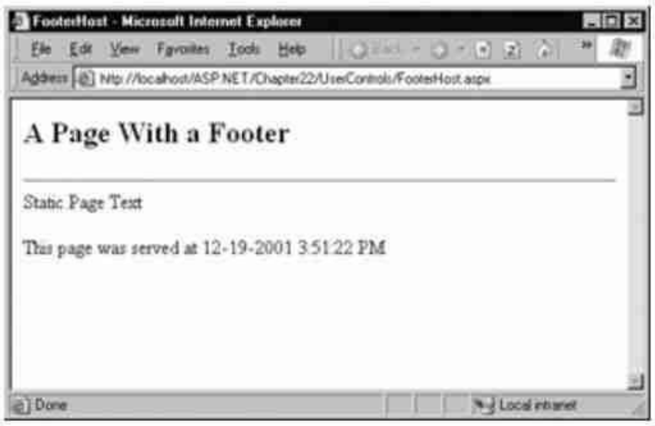
Note that the class has the addition of the MustInherit keyword in its class definition. This indicates that the user control cannot be accessed directly; instead, a control derived from this class must be placed on a web page.

To test this user control, you need to insert it into a web page. Two steps are required. First, you need to add a <%@ Register %> directive that identifies the control you want to use and associates it with a unique control prefix.

```
<%@ Register TagPrefix="cr" TagName="Footer" Src="Footer.ascx" %>
```

The Register directive specifies a tag prefix and name. Tag prefixes group sets of related controls (for example, all ASP.NET web controls use the tag prefix "asp"). Tag prefixes are usually lowercase—technically, they are case-insensitive—and should be unique for your company or organization. The Src directive identifies the location of the user control template file, not the code-behind file.

You can then add the user control whenever you want (and as many times as you want), by inserting its control tag. Consider this page example (shown in Figure 22-2).


FooterHost - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/ASP.NET/Chapter22/UserControls/FooterHost.aspx
Done
Local intranet

A Page With a Footer

Static Page Text

This page was served at 12-19-2001 3:51:22 PM

Figure 22-2: A page with a user control footer

```
<%@ Page Language="VB" AutoEventWireup="false"
   Inherits="FooterHost"%>
<%@ Register TagPrefix="cr" TagName="Footer" Src="Footer.ascx" %>

<HTML>
<body>

<form id=Form1 method=post runat="server">
   <h2>A Page With a Footer</h2><hr>
   Static Page Text<br><br>
   <cr:Footer id=Footer1 runat="server" />
</form>

</body>
</HTML>
```

This example shows a simple way that you could create a header or footer and reuse it in all the pages in your web site, simply by adding the user control declaration. In the case of our simple footer, the actual code savings is limited, but it could become much more useful for a complex control with extensive formatting or several contained controls.

Of course, this only scratches the surface of what you can do with a user control. In the following sections, you'll learn how to enhance a control with properties, methods, and events—transforming it from a simple "include file" into a full-fledged object.

Dynamically Loaded User Controls

The Page class provides a special LoadControl method that allows you to create a user control dynamically at runtime from an .ascx file. The user control is returned to you as a control object, which you can then add to the Controls collection of a container control on the web page (like PlaceHolder or Panel) to display it on the page. This technique is not recommended as a substitute for declaratively using a user control, but it does have some interesting applications if you need to generate user interface dynamically. Chapter 25 shows how the LoadControl method is used with the IBuySpy portal application.

Visual Studio .NET User Control Support

In Visual Studio .NET, you have a few shortcuts available when working with user controls. Once you've created your user control, simply build your project, and then drag the .ascx file from the Solution Explorer and drop it onto the drawing area of a web form. Visual Studio .NET will automatically add the Register directive for you, as well as an instance of the user control tag.

In the design environment, the user control will be displayed as a familiar gray box (see Figure 22-3)—the same nondescript package used for databound controls that don't have any templates configured.

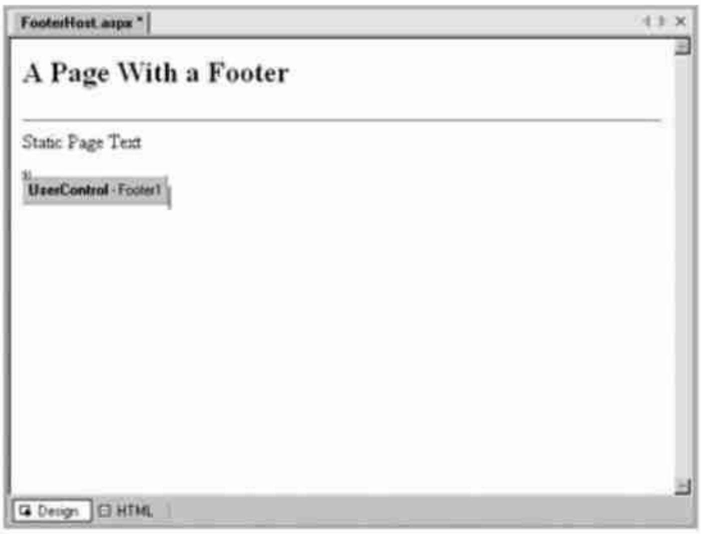


Figure 22-3: A user control at design time

Independent User Controls

Conceptually, there are really two types of user controls: independent and integrated. Independent user controls don't interact with the rest of the code on your form. The Footer user control is one such example. Another example might be a list of buttons that offer links to other pages. A Menu user control like this could handle the events for all the buttons, and then run the appropriate Response.Redirect code to move to another web page. Or it could just include ordinary HyperLink controls that don't have any associated server-side code. Every page in the web site could then include the same Menu user control—enabling painless web site navigation with no need to worry about frames. In fact, this is exactly the approach you'll see in practice in Chapter 25 with the IBuySpy e-commerce application.

The following sample defines a simple Menu footer control that just requests a given page with a different query string argument. Note that the style attribute of the <div> tag (which defines fonts and formatting) is omitted for clarity. I've also left out the Inherits attribute in the Control tag, because the user control does not require any code-behind logic.

```
<%@ Control Language="VB" AutoEventWireup="false" %>

<div>
 Products:
 <asp:HyperLink id="lnkBooks" runat="server"
  NavigateUrl="MenuHost.aspx?product=Books">Books
 </asp:HyperLink><br>
 <asp:HyperLink id=lnkToys runat="server"
  NavigateUrl="MenuHost.aspx?product=Toys">Toys
 </asp:HyperLink><br>
 <asp:HyperLink id=lnkSports runat="server"
  NavigateUrl="MenuHost.aspx?product=Sports">Sports
 </asp:HyperLink><br>
 <asp:HyperLink id=lnkFurniture runat="server"
  NavigateUrl="MenuHost.aspx?product=Furniture">Furniture
 </asp:HyperLink>
</div>
```

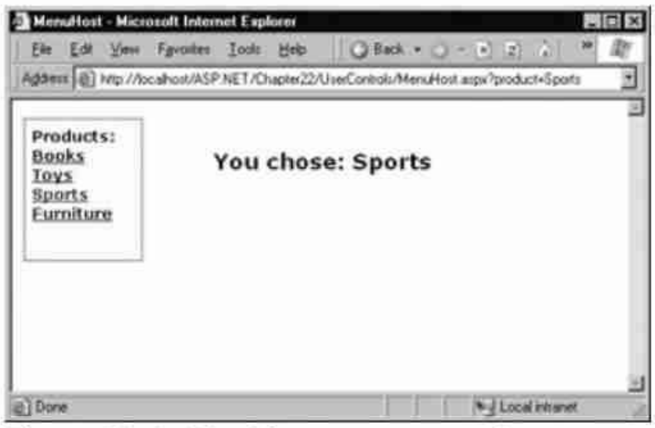
The MenuHost.aspx file includes two controls, the Menu control and a label that displays the product query string parameter.

```
<%@ Page Language="VB" AutoEventWireup="false" Inherits="MenuHost"%>
<%@ Register TagPrefix="cr" TagName="Menu" Src="Menu.ascx" %>

<HTML>
<body>

<form id=Form1 method=post runat="server">
  <cr:Menu id="Menu1" runat="server" />
  <asp:Label id="lblSelection" / >
</form>

</body>
</HTML>
```

When the MenuHost.aspx page loads, it adds the appropriate information to the lblSelection control.

```
Private Sub Page_Load(sender As Object, e As EventArgs) _
 Handles MyBase.Load
  If Request.Params("product") <> "" Then
    lblSelection.Text = "You chose: "
    lblSelection.Text &= Request.Params("product")
  End If
End Sub
```

The end result is shown in Figure 22-4. Whenever you click a link, the page is posted back, and the text updated.


MenuHost - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Address
Done
Local intranet

http://localhost/ASP.NET/Chapter22/UserControls/MenuHost.aspx?product=Sports

Products:
Books
Toys
Sports
Furniture

You chose: Sports

Figure 22-4: The Menu user control

Integrated User Controls

Integrated user controls interact with the web page that hosts them, in one way or another. When designing these controls, the class-based design tips you learned in Chapter 3 really become useful.

A typical example is a user control that allows some level of configuration through properties. For example, you could create a footer that supports two different display formats: long date and short time. To add a further level of refinement, the Footer user control allows the web page to specify the appropriate display format using an enumeration.

The first step is to create an enumeration in the custom Footer class. Remember, an enumeration is simply a type of constant that is internally stored as an integer, but is set in code by using one of the allowed names that you specify. Variables that use the FooterFormat enumeration can take the value FooterFormat.LongDate or FooterFormat.ShortTime.

```
Public Enum FooterFormat
  LongDate
  ShortTime
End Enum
```

The next step is to add a property to the Footer class that allows the web page to retrieve or set the current format applied to the footer. The actual format is stored in a private variable called _Format, which is set to the long data format by default when the class is first created. (The same effect could be accomplished, in a slightly sloppier way, by using a public member variable named Format instead of a full property procedure.) If you're hazy on how property procedures work, feel free to review the explanation in Chapter 3.

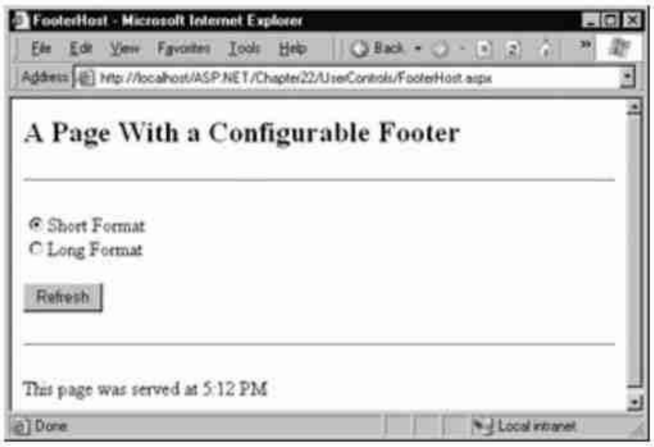
```
Private _Format As FooterFormat

Public Property Format() As FooterFormat = FooterFormat.LongDate
  Get
    Return _Format
  End Get
  Set(ByVal Value As FooterFormat)
    _Format = Value
  End Set
End Property
```

Finally, the UserControl.Load event needs to take account of the current footer state and format the output accordingly. The full Footer class code is shown here:

```
Public MustInherit Class Footer
 Inherits UserControl

  Protected lblFooter As Label

  Public Enum FooterFormat
    LongDate
    ShortTime
  End Enum

  Private _Format As FooterFormat = FooterFormat.LongDate
  Public Property Format() As FooterFormat
    Get
      Return _Format
    End Get
    Set(ByVal Value As FooterFormat)
      _Format = Value
   End Set
  End Property
```

```
    Private Sub Page_Load(sender As Object, e As EventArgs) _
      Handles MyBase.Load
        lblFooter.Text = "This page was served at "

        If _Format = FooterFormat.LongDate Then
            lblFooter.Text &= DateTime.Now.ToLongDateString()
        ElseIf _Format = FooterFormat.ShortTime Then
            lblFooter.Text &= DateTime.Now.ToShortTimeString()
        End If
    End Sub

End Class
```

To test out this footer, you need to create a page that includes the user control and defines a variable in the custom Page class that represents the user control. (Note that Visual Studio .NET does not add this variable automatically when you create the control, as it does for ordinary web controls.) You can then modify the Format property through the user control variable. An example is shown in Figure 22-5, which automatically sets the Format property for the user control to match a radio button selection whenever the page is posted back.


FooterHost - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Address http://localhost/ASP.NET/Chapter22/UserControls/FooterHost.aspx
A Page With a Configurable Footer
Short Format
Long Format
Refresh
This page was served at 5:12 PM
Done
Local intranet
Figure 22-5: The modified footer

Note that the user control property is modified in the Page.Load event handler, not the cmdRefresh.Click event handler. The reason is that the Load event occurs before the user control has been rendered each time the page is created. The Click event occurs after the user control has been rendered, and though the property change will be visible in your code, it won't affect the user control's HTML output, which has already been added to the page.

```
Public Class FooterHost
  Inherits Page

    Protected optShort As RadioButton
    Protected optLong As RadioButton
    Protected cmdRefresh As Button
    Protected Footer1 As Footer

    Private Sub Page_Load(sender As Object, e As EventArgs) _
      Handles MyBase.Load

        If optLong.Checked = True Then
            Footer1.Format = Footer.FooterFormat.LongDate
        ElseIf optShort.Checked = True Then
            Footer1.Format = Footer.FooterFormat.ShortTime
        Else
            ' The default value in the Footer class will apply.
        End If

    End Sub

End Class
```

You could also set the initial appearance of the footer in the control tag.

```
<cr:Footer Format="ShortTime" id="Footer1" runat="server" />
```

User Control Events

Another way that communication can occur between a user control and a web page is through events. Events are really the inverse of properties or methods, where the user control reacts to a change made by the web page code. With events, the user control notifies the web page about an action, and the web page code responds.

Creating a web control that uses events is fairly easy. Consider the next example, which uses a special login box that verifies a user's credentials. This type of control could be used in a variety of sources to restrict access to a variety of different pages. For that reason, you can't hardcode any logic in the user control that redirects the user to a specific page. Instead, the LoginBox user control needs to raise an event to alert the web page code if the login process was successful. At that point, the web page can display the appropriate resource or message.

The first step in creating the LoginBox user control is to define the events using the Event keyword. Remember, events are always declared at the public level, so they don't need an access qualifier keyword. (You can refer to Chapter 3 for a quick overview of how to use events in .NET.)

The LoginBox control defines two events, one that indicates a failed login attempt and one that indicates success.

```
Event LoginFailed()
Event LoginAuthenticated()
```

The LoginBox code can now fire either of these events by using the RaiseEvent command.

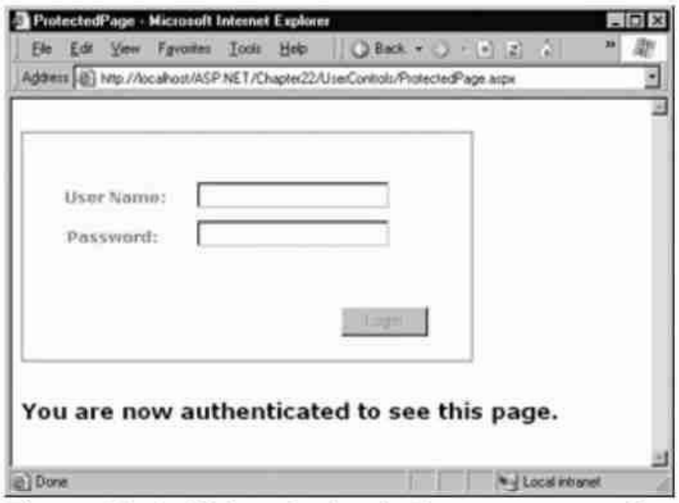
```
RaiseEvent LoginFailed()
```

These events are raised after the Login button is clicked, and the user's information is examined. The full page code is shown here:

```
Public MustInherit Class LoginBox
 Inherits UserControl

  Protected txtUser As TextBox
  Protected txtPassword As TextBox
  Protected WithEvents cmdLogin As Button

  Event LoginFailed()
  Event LoginAuthenticated()

  Private Sub cmdLogin_Click(sender As Object, e As EventArgs) _
   Handles cmdLogin.Click

    ' Typically, this code would use the FormsAuthentication
    ' class described in Chapter 24, or some custom
    ' database-lookup code to authenticate the user.
    ' Our example simply checks for a "secret" code.
    If txtPassword.Text = "opensesame" Then
      RaiseEvent LoginAuthenticated()
    Else
      RaiseEvent LoginFailed()
    End If

  End Sub

End Class
```

The page hosting this code can then add an event handler for either one of these events, in the same way that event handlers are created for any control—by declaring the control variable with the WithEvents keyword, and adding a Handles statement to a procedure with the appropriate signature.

Figure 22-6 shows a simple example of a page that uses the LoginBox control. If the login fails three times, the user is forwarded to another page. If the login succeeds, the LoginBox control is disabled and a message is displayed.


ProtectedPage - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Address http://localhost/ASP.NET/Chapter22/UserControls/ProtectedPage.aspx
User Name:
Password:
Login
Done
Local intranet

You are now authenticated to see this page.

Figure 22-6: Using the LoginBox user control

The code for this example is refreshingly straightforward.

```
Public Class ProtectedPage
 Inherits Page

  Protected lblSecretMessage As Label
  Protected pnlControls As Panel
  Protected WithEvents Login As UserControls.LoginBox

  Private Sub Failed() Handles Login.LoginFailed
     ' Retrieve the number of failed attempts from viewstate.
     Dim Attempts As Integer
     Attempts = CType(Viewstate("Attempts"), Integer)

     Attempts += 1
     If Attempts >= 3 Then Response.Redirect("default.aspx")

     ' Store the new number of failed attempts in viewstate.
     Viewstate("Attempts") = Attempts
  End Sub

  Private Sub Succeeded() Handles Login.LoginAuthenticated
     pnlControls.Enabled = False
     lblSecretMessage.Text = "You are now authenticated" & _
                   "to see this page."
  End Sub

End Class
```

Note that the user control is declared using the full namespace name (UserControls.LoginBox). This is always a good idea, and it's required if the namespace used for the user control is different than that used for the web page.

Using Events with Parameters

In the LoginBox example, there is no information passed along with the event. In many cases, however, you want to convey additional information that relates to the event. For example, you could create a LoginRequest event that passes information about the user ID and password that were entered. This would allow your code to decide whether the user has sufficient security permissions for the requested page, while retaining a common look for login boxes throughout your web application.

The .NET standard for events specifies that every event should use two parameters. The first one provides a reference to the control that sent the event, while the second one incorporates any additional information. This additional information is wrapped into a custom EventArgs object, which inherits from the System.EventArgs class. (If your event doesn't require any additional information, you can just use the generic System.EventArgs object, which doesn't contain any additional data. Many framework events, such as Page.Load or Button.Click, follow this pattern.)

The EventArgs class that follows allows the LoginBox user control to pass the name of the authenticated user to the event handler.

```
Public Class LoginAuthenticatedEventArgs
 Inherits EventArgs

  Public UserName As String

End Class
```

To use this class, you need to modify the LoginAuthenticated event definition to the .NET standard shown below. At the same, you should also modify the LoginFailed event definition, which can use the default (empty) System.EventArgs class.

```
Event LoginAuthenticated(sender As Object, _
 e As LoginAuthenticatedEventArgs)
Event LoginFailed(sender As Object, e As EventArgs)
```

Next, your code for raising the event needs to submit the required two pieces of information as event parameters.

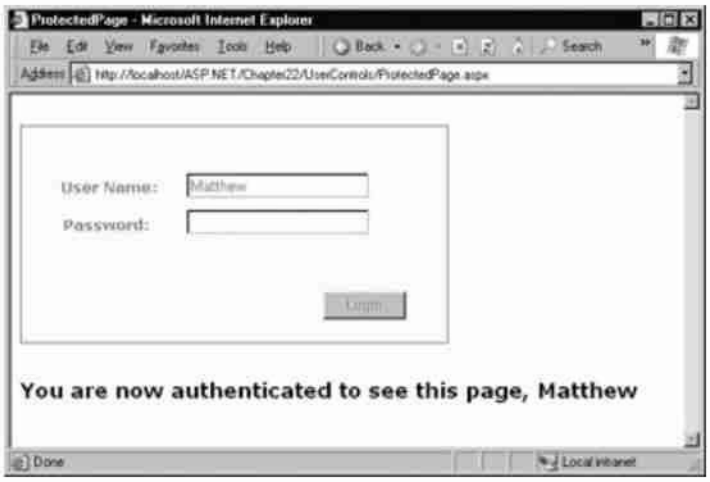
```
Private Sub cmdLogin_Click(sender As Object, e As EventArgs) _
 Handles cmdLogin.Click

  If txtPassword.Text = "opensesame" Then
    Dim EventInfo As New LoginAuthenticatedEventArgs()
    EventInfo.UserName = txtUser.Text
    RaiseEvent LoginAuthenticated(Me, EventInfo)
  Else
    ' You can define the EventArgs object on the same line
    ' that you use it to make more economical code.
    RaiseEvent LoginFailed(Me, New EventArgs())
  End If

End Sub
```

Lastly, your receiving code needs to update its event handler to use the new signature.

```
Private Sub Succeeded(sender As Object, _
 e As LoginAuthenticatedEventArgs) _
 Handles Login.LoginAuthenticated

  pnlControls.Enabled = False
  lblSecretMessage.Text = "You are now authenticated"
  lblSecretMessage.Text &= "to see this page, " & e.UserName

End Sub

Private Sub Failed(sender As Object, e As EventArgs) _
```

```
 Handles Login.LoginFailed

    ' Retrieve the number of failed attempts from viewstate.
    Dim Attempts As Integer
    Attempts = CType(Viewstate("Attempts"), Integer)

    Attempts += 1
    If Attempts >= 3 Then Response.Redirect("default.aspx")

    ' Store the new number of failed attempts in viewstate.
    Viewstate("Attempts") = Attempts

End Sub
```

Now the user will see a page like the one shown in Figure 22-7 after a successful login.


ProtectedPage - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search
Address http://localhost/ASP.NET/Chapter22/UserControls/ProtectedPage.aspx
User Name:
Matthew
Password:
Login
Done
Local intranet

You are now authenticated to see this page, Matthew

Figure 22-7: Retrieving the user name through an event

User Control Limitations

User controls provide a great deal of flexibility when you want to combine several web controls (and additional HTML content) in a single unit, and possibly add some higher-level business logic. However, they are less useful when you want to modify or extend individual web controls.

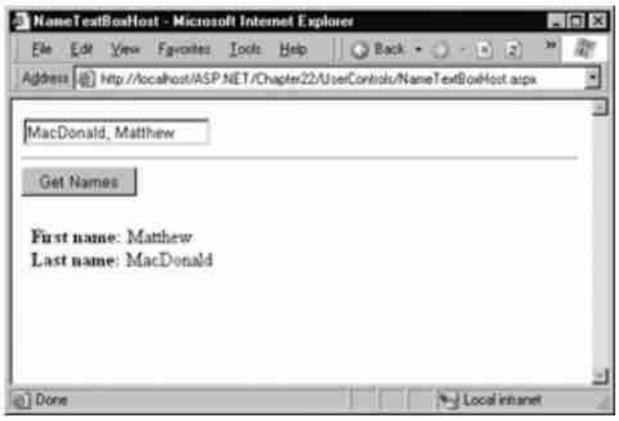
For example, imagine you want to create a text box – like control for name entry. This text box might provide GetFirstName and GetLastName methods, which examine the entered text and parse it into a first and last name using one of two recognized formats: space-separated ("FirstName LastName") or comma-separated ("LastName, FirstName"). This way the user can enter a name in either format, and your code doesn't have to go through the work of parsing the text. Instead, the user control handles it automatically.

The full code for this user control would look something like this:

```
Public MustInherit Class NameTextBox
 Inherits System.Web.UI.UserControl

  Protected txtName As TextBox

  Private _FirstName As String
  Private _LastName As String

  Public Function GetFirstName() As String
    UpdateNames()
    Return _FirstName
  End Function

  Public Function GetLastName() As String
```

```
        UpdateNames()
        Return _LastName
    End Function

    Private Sub UpdateNames()
        Dim CommaPos As Integer = txtName.Text.IndexOf(",")
        Dim SpacePos As Integer = txtName.Text.IndexOf(" ")

        Dim NameArray() As String
        If CommaPos <> -1 Then
            NameArray = txtName.Text.Split(",")
            _FirstName = NameArray(1)
            _LastName = NameArray(0)
        ElseIf SpacePos <> -1 Then
            NameArray = txtName.Text.Split(" ")
            _FirstName = NameArray(0)
            _LastName = NameArray(1)
        Else
            ' The text has no comma or space.
            ' It cannot be converted to a name.
            Throw New InvalidOperationException()
        End If
    End Sub

End Class
```

The online samples include a simple page that allows you to test this control (shown in Figure 22-8). It retrieves the first and last names that are entered in the text box when the user clicks a button.


NameTextBoxHost - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Address http://localhost/ASP.NET/Chapter22/UserControls/NameTextBoxHost.aspx
Get Names
Done
Local intranet

MacDonald, Matthew

First name: Matthew
Last name: MacDonald

Figure 22-8: A custom user control based on the text box

```
Private Sub cmdGetNames_Click(sender As Object, e As EventArgs) _
  Handles cmdGetNames.Click
    lblNames.Text = "<b>First name:</b> "
    lblNames.Text &= NameTextBox1.GetFirstName()
    lblNames.Text &= "<br><b>Last name:</b> " &
    lblNames.Text &= NameTextBox1.GetLastName()
End Sub
```

Using a full user control is overkill in this case, and it also makes it difficult for the web page programmer to further configure the text box. For example, if a programmer wants to set the text box font or change its size, you either need to add the corresponding properties to the user control class or provide access to the text box through a public member variable. (Technically, the web page programmer could access the text box by using the Controls collection of the user control, but this is a less structured approach that can run into problems if other controls are added to the user control or the ID of the text box changes.) There's also no way to set these values in the user control tag.

The next section describes a better approach for extending or fine-tuning individual controls.

Deriving Custom Controls

Thanks to the class-based framework of .NET, an easier solution for extending a specialized control is possible using inheritance. All you need to do is find the control you want to extend in the .NET class library, and derive a new class from it that adds the additional functionality you need.

Here's an example of the name text box implemented through a custom control.

```
Public Class NameTextBox
 Inherits System.Web.UI.WebControls.TextBox

  Private _FirstName As String
  Private _LastName As String

  Public Function GetFirstName() As String
    UpdateNames()
    Return _FirstName
  End Function

  Public Function GetLastName() As String
    UpdateNames()
    Return _LastName
  End Function

  Private Sub UpdateNames()
    Dim CommaPos As Integer = Text.IndexOf(",")
    Dim SpacePos As Integer = Text.IndexOf(" ")

    Dim NameArray() As String
    If CommaPos <> -1 Then
      NameArray = Text.Split(",")
      _FirstName = NameArray(1)
      _LastName = NameArray(0)
    ElseIf SpacePos <> -1 Then
      NameArray = Text.Split(" ")
      _FirstName = NameArray(0)
      _LastName = NameArray(1)
    Else
      ' The text has no comma or space.
      ' It cannot be converted to a name.
      Throw New InvalidOperationException()
    End If
  End Sub

End Class
```

In this example, the custom NameTextBox class inherits from the .NET TextBox class (which is found in the System.Web.UI.WebControls namespace). Because the NameTextBox class extends the TextBox class, all the original TextBox members (such as Font and ForeColor) are still available to the web page programmer, and can be set in code or through the control tag. The only differences in the code between the user control version and the custom control version is that the custom version adds an Inherits statement and works natively with the Text property of the NameTextBox class (not a TextBox member variable).

Consuming a Custom Control

To test the control, you should compile it into an assembly using the vbc.exe compiler, and place it in the bin directory for the application. This is the exact same process you used in Chapter 21 for custom components. You can then register the custom control using a Register directive with a slightly different set of attributes than the one you used for user controls.

```
<%@ Register TagPrefix="CR" Namespace="CustomControls"
  Assembly="CustomCtrls" %>
```

This Register directive identifies the compiled assembly file and the namespace that holds the custom control. When you register a custom control assembly in this fashion, you gain access to all the control classes in it. You can insert a control by using the tag prefix, followed by a colon (:) and the class name.

```
<CR:NameTextBox id="NameTextBox1" runat="server" />
```

You can set properties of the NameTextBox in code or through the tag. These can include any additional properties you have defined or the properties from the base class:

```
<CR:NameTextBox id="NameTextBox1" BackColor="LightYellow"
 Font="Verdana "Text="Enter Name Here" runat="server" />
```

The technique of deriving custom control classes is known as subclassing, and it allows you to easily add the functionality you need without losing the basic set of features inherent in a control. Subclassed controls can add new properties, events, and methods, or override the existing ones. For example, you could add a ReverseText or EncryptText method to the NameTextBox class that loops through the text contents and modifies them. The process of defining and using custom control events, methods, and properties is the same as with user controls.

One common reason for subclassing a control is to add default values. For example, you might create a custom Calendar control that automatically modifies some properties in its constructor. When the control is first created, these default properties will automatically be applied.

```
Public Class FormattedCalendar
 Inherits Calendar

  Public Sub New()
    ' Configure the appearance of the calendar table.
    Me.CellPadding = 8
    Me.CellSpacing = 8
    Me.BackColor = Color.LightYellow
    Me.BorderStyle = BorderStyle.Groove
    Me.BorderWidth = Unit.Pixel(2)
    Me.ShowGridLines = True

    ' Configure the font.
    Me.Font.Name = "Verdana"
    Me.Font.Size = FontUnit.XXSmall

    ' Set some special calendar settings.
    Me.FirstDayOfWeek = FirstDayOfWeek.Monday
    Me.PrevMonthText = "<--"
    Me.NextMonthText = "-->"

    ' Select the current date by default.
    Me.SelectedDate = Date.Today
  End Sub

End Class
```

You could even add additional event-handling logic to this class that will use the Calendar's DayRender event to configure custom date display. This way, the Calendar class itself handles all the required formatting and configuration; your page code does not need to work at all! Note that the Handles clause specifies the MyBase keyword, indicating an event of the current (inherited) class.

```
Private Sub FormattedCalendar_DayRender(sender As Object, _
 e As DayRenderEventArgs) Handles MyBase.DayRender
```

```
    If e.Day.IsOtherMonth Then
        e.Day.IsSelectable = False
        e.Cell.Text = ""
    Else
        e.Cell.Font.Bold = True
    End If

End Sub
```

The preferred way to handle this is actually to override the OnDayRender method, and make sure to call the base method to ensure that the basic tasks (such as notifying event recipients) are met. The effect of the following code is equivalent.

```
Protected Overrides Sub OnDayRender(cell As TableCell, _
  day As CalendarDay)

    ' Call the base Calendar.OnDayRender method.
    MyBase.OnDayRender(cell, day)

    If day.IsOtherMonth Then
        day.IsSelectable = False
        cell.Text = ""
    Else
        cell.Font.Bold = True
    End If

End Sub
```

Figure 22-9 contrasts two Calendar controls: the normal one, and the custom FormattedCalendar control class.

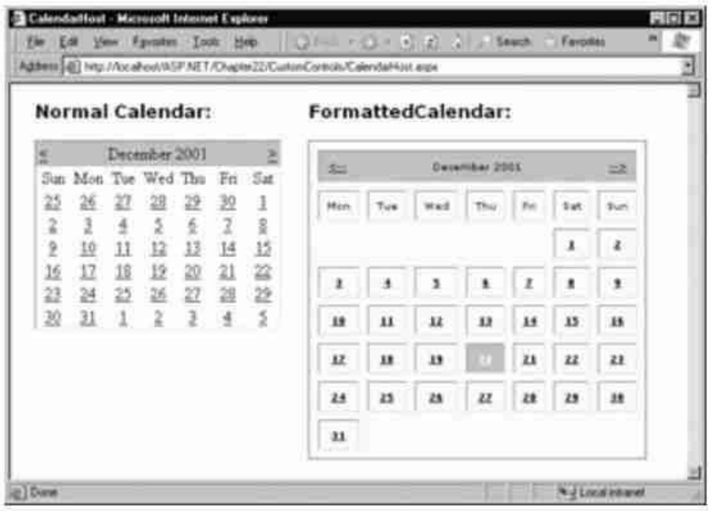


Figure 22-9: A subclassed Calendar control

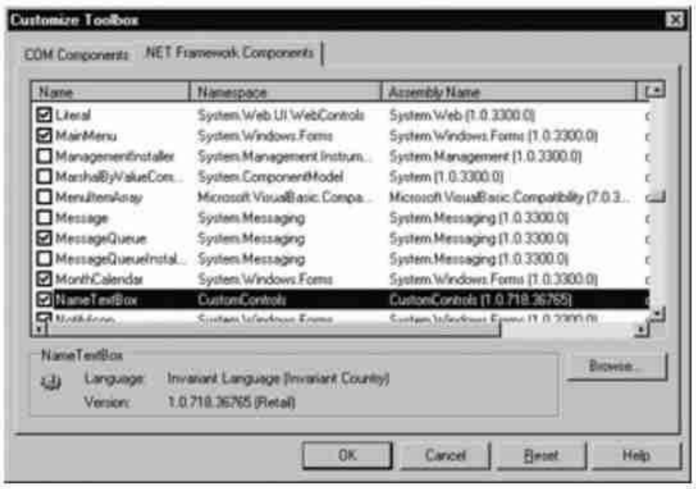
Control Tag Settings Override Constructor Defaults

Even though you set these defaults in the custom class code, that doesn't prevent you from modifying them in your web page code. The constructor code runs when the Calendar control is created, after which the control tag settings are applied, and then any event-handling code in your web page also has the chance to modify the FormattedCalendar properties.

Visual Studio .NET Custom Control Support

In Visual Studio .NET, it's often easiest to develop custom controls in a separate project, compile the assembly, and add a reference to the assembly in the new project. This allows you to easily place the custom controls and the web page classes in separate namespaces, and update them separately.

The process of adding a reference to a custom control assembly is very similar to the process you used to add a reference to a normal compiled component. Start by right-clicking on the Control Box, and choosing Customize Toolbox. Click on the .NET Framework Components tab (see Figure 22-10), and then click the Browse button. Then choose the custom control assembly from the file browser. The controls will be added to the list of available .NET controls.



Name	Namespace	Assembly Name
✓ Literal	System.Web.UI.WebControls	System.Web (1.0.3300.0)
✓ MainMenu	System.Windows.Forms	System.Windows.Forms (1.0.3300.0)
☐ ManagementInstaller	System.Management.Instrum...	System.Management (1.0.3300.0)
☐ MarshalByValueCom...	System.ComponentModel	System (1.0.3300.0)
☐ MenuItemArray	Microsoft.VisualBasic.Compa...	Microsoft.VisualBasic.Compatibility (7.0.3...
☐ Message	System.Messaging	System.Messaging (1.0.3300.0)
✓ MessageQueue	System.Messaging	System.Messaging (1.0.3300.0)
☐ MessageQueueInstal...	System.Messaging	System.Messaging (1.0.3300.0)
✓ MonthCalendar	System.Windows.Forms	System.Windows.Forms (1.0.3300.0)
✓ NameTextBox	CustomControls	CustomControls (1.0.718.36765)

NameTextBox
Language: Invariant Language (Invariant Country)
Version: 1.0.718.36765 (Retail)

Figure 22-10: Adding a reference to the NameTextBox

All checkmarked controls will appear in the Toolbox (see Figure 22-11). You can then draw them on your web forms just like you would with an ordinary control. Note that controls are not added on a per-project basis. Instead, they will remain in the Toolbox until you delete them. To remove a control, right click it and select Delete. This removes the icon only, not the referenced assembly.

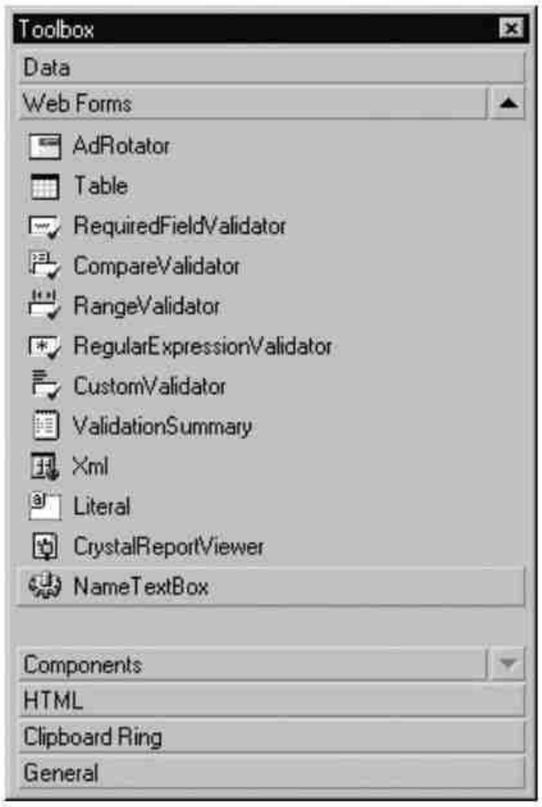


Figure 22-11: The Toolbox with a custom control

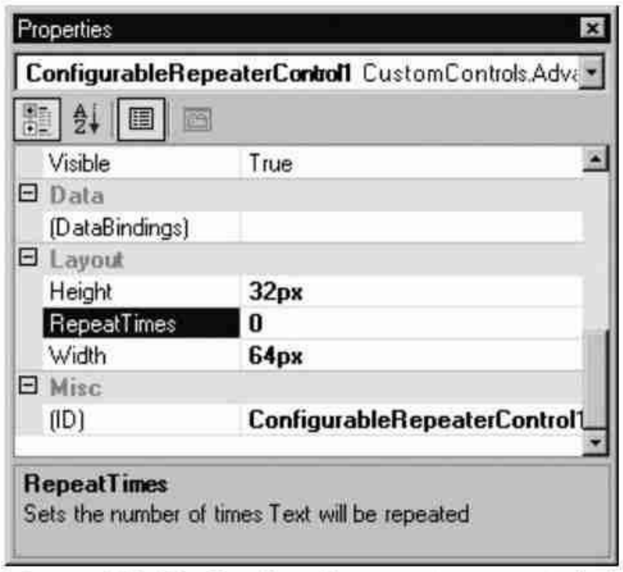
Any properties you add for the custom control automatically appear in the Properties window under the Misc heading. You can further customize the design-time behavior of your control by using various attributes from the System.ComponentModel namespace.

```
Imports System.ComponentModel
```

For example, the property defined below (part of the ConfigurableRepeater control we will consider later in this chapter) will always appear in a "Layout" category in the Properties window. It indicates this to Visual Studio .NET through a System.ComponentModel.Category attribute. Note that attributes are always enclosed in angled brackets and must appear on the same line as the code element they refer to (in this case, the property procedure declaration) or be separated from it using the underscore line continuation character. (You may remember that .NET attributes were also used when programming Web Services. In that case, the attributes were named WebMethod and WebService.)

```
<Category("Layout")> _
Public Property RepeatTimes() As Integer
   Get
      Return _RepeatTimes
   End Get
   Set(ByVal Value As Integer)
      _RepeatTimes = Value
   End Set
End Property
```

You can specify more than one attribute at a time, as long as you separate them using commas. The example here includes a Category and a Description attribute. The result is shown in the Properties window in Figure 22-12.



Visible	True
⊟ Data	
(DataBindings)	
⊟ Layout	
Height	32px
RepeatTimes	0
Width	64px
⊟ Misc	
(ID)	ConfigurableRepeaterControl1

RepeatTimes
Sets the number of times Text will be repeated

Figure 22-12: Configuring custom control design-time support

```
<Description("Sets the number of times Text will be repeated"), _
 Category("Layout"), > _
```

A list of useful attributes for configuring a control's design-time support is shown in Table 22-1.

Table 22-1: Attributes for Design-Time Support

Attribute	Description
<Browsable(True|False)>	If False, this property will not appear in the Properties window (although the programmer can still modify it in code or by manually adding the control tag attribute, as long as you include a Set property procedure).
<Bindable(True|False)>	If True, Visual Studio .NET will display this property in the

	DataBindings dialog and allow it to be bound to a field in a data source.
<Category("")>	A string that indicates the category the property will appear under in the Properties window.
<Description("")>	A string that indicates the description the property will have when selected in the Properties window.
<DefaultValue()>	Sets the default value that will be displayed for the property in the Properties window.
<ParenthesizeProperty-Name (True|False)>	If True, Visual Studio .NET will display parentheses around this property in the Properties window (as it does with the ID property).

Other Custom Control Tricks

You can also subclass a control and add low-level refinements by manually configuring the HTML that will be generated from the control. To do this, you only need to follow a few basic guidelines:

- Override one of the Render methods (Render, RenderContents, RenderBeginTag, and so on) from the base control class. To override the method successfully, you need to specify the same access level as the original method (such as Public or Protected). Visual Studio .NET will help you out on this account by warning you if you make a mistake. If you are coding in another editor, just check the MSDN reference first.

- Use the MyBase keyword to call the base method. In other words, you want to make sure you are adding functionality to the method, not replacing it with your code. Also, the original method may perform some required cleanup task that you aren't aware of, or it may raise an event that the web page could be listening for.

- Add the code to write out any additional HTML. Usually, this code will use the HtmlWriter class, which supplies a helpful Write method for direct HTML output.

The following is an example of a text box control that overrides the Render method to add a static title. (Optionally, the content for this title could be taken from a property procedure that the user can configure.)

```
Public Class TitledTextBox
 Inherits TextBox

  Protected Overrides Sub Render(writer As HtmlTextWriter)
    ' Add new HTML.
    writer.Write("<h1>Here is a title</hi><br>")

    ' Call the base method (so the text box portion is rendered).
    MyBase.Render(writer)
  End Sub

End Class
```

Figure 22-13 shows what the TitledTextBox control looks like in the design environment.

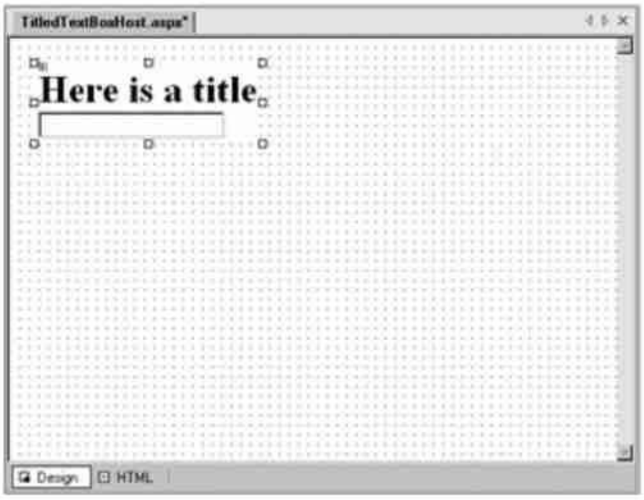


Figure 22-13: A subclassed text box with a title

Remember, ASP.NET creates a page by moving through the list of controls and instructing each one to render itself (by calling the Render method). It collects the total HTML output and sends it as a complete page (nested in root <HTML><body> tags).

You can use a similar technique to add attributes to the HTML text box tag. For example, you might want to add a piece of JavaScript code to the OnBlur attribute to make a message box appear when the control loses focus. There is no TextBox property that exposes this attribute, but you can add it manually in the AddAttributesToRender method.

```
Public Class LostFocusTextBox
 Inherits TextBox

  Protected Overrides Sub AddAttributesToRender( _
   writer As HtmlTextWriter)

    MyBase.AddAttributesToRender(writer)
    writer.AddAttribute("OnBlur", _
     "javascript:alert('You Lost Focus')")

  End Sub

End Class
```

The resulting HTML for the LostFocusTextBox looks something like this:

```
<input type="text" id="LostFocusTextBox2"
     OnBlur="javascript:alert('You Lost Focus')" />
```

Figure 22-14 shows what happens when the text box loses focus. Once again, you could extend the usefulness of this control by making the alert message configurable through a property.

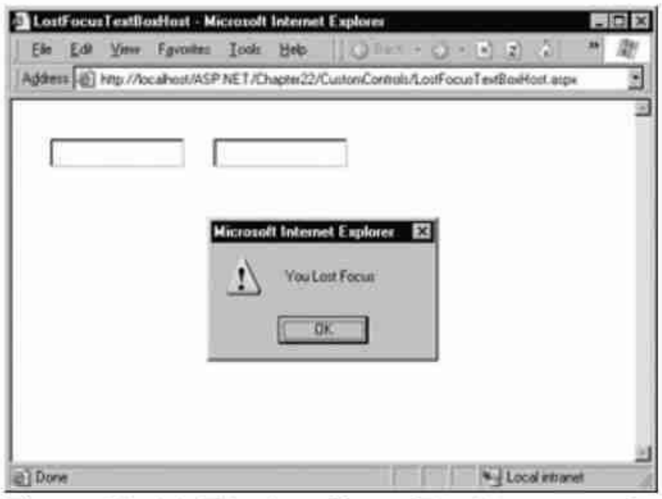

LostFocusTextBoxHost - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Address http://localhost/ASP.NET/Chapter22/CustomControls/LostFocusTextBoxHost.aspx
Microsoft Internet Explorer
OK
Done
Local intranet
You Lost Focus

Figure 22-14: The LostFocusTextBox control

```
Public Class LostFocusTextBox
 Inherits TextBox

  Private _Alert As String
  Public Property AlertCode() As String
    Get
      Return _Alert
    End Get
    Set(Value As String)
      _Alert = Value
    End Set
  End Property

  Protected Overrides Sub AddAttributesToRender( _
   writer As HtmlTextWriter)

    MyBase.AddAttributesToRender(writer)
    writer.AddAttribute("OnBlur", _
     "javascript:alert('" & _Alert & "')")

  End Sub

End Class
```

Creating a Web Control from Scratch

Once you start experimenting with altering the TextBox control's HTML, it might occur to you to design a control entirely from scratch. This is an easy task in ASP.NET—all you need to do is inherit from the System.Web.UI.WebControls.WebControl class, which is the base of all ASP.NET web controls. Figure 22-15 shows the inheritance hierarchy for web controls.

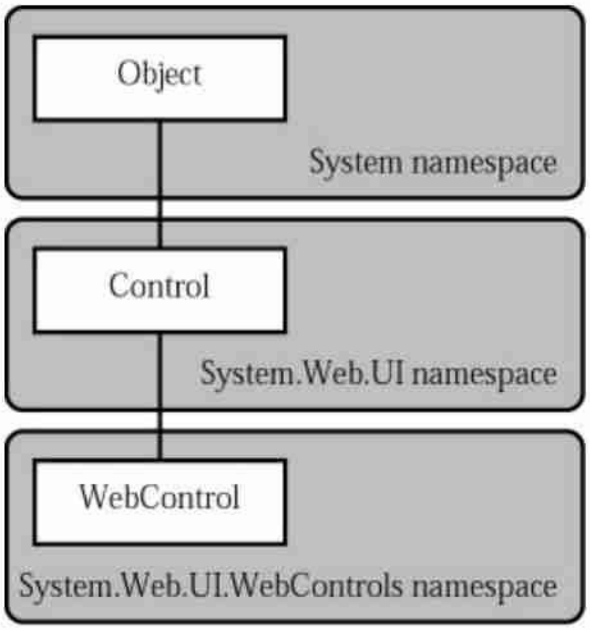


Figure 22-15: Web control inheritance hierarchy

You Could Inherit from System.Web.UI.Control Instead

Technically, you could inherit from the base System.Web.UI.Control class instead, but it provides fewer features. The main difference with the WebControl class is that it provides a basic set of formatting properties such as Font, BackColor, and ForeColor. These properties are automatically implemented by adding the appropriate HTML attributes to your HTML tag, and are also persisted in viewstate for you automatically. If you inherit from the more basic Control class, you need to provide your own style-based properties, add them as attributes using the HtmlTextWriter, and store the settings in viewstate so they will be remembered across postbacks.

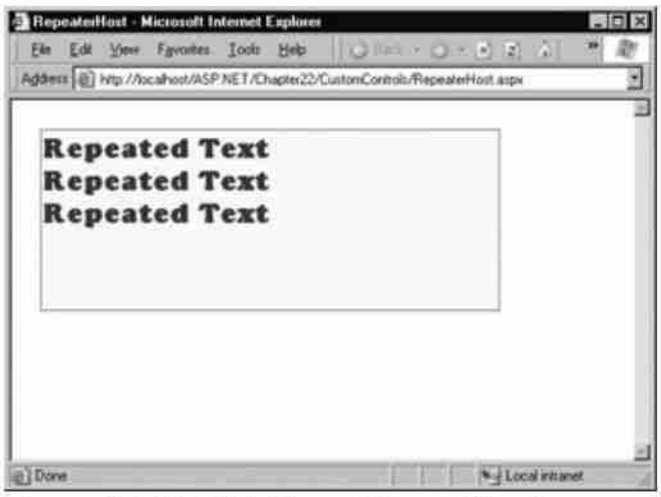
To create your own control from scratch, all you need to do is add the appropriate properties, and implement your own custom RenderContents or Render method, which writes the HTML output using the HtmlTextWriter class. The RenderContents method takes place after the Render method, which means that the formatting attributes have already been applied.

The code is quite similar to the earlier TextBox examples. It creates a repeater type of control that lists a given line of text multiple times. The number of repeats is set by the RepeatTimes property.

```
Public Class ConfigurableRepeaterControl
 Inherits WebControl

   Private _RepeatTimes As Integer = 3
   Private _Text As String = "Text"
   Public Property RepeatTimes() As Integer
      Get
         Return _RepeatTimes
      End Get
      Set(ByVal Value As Integer)
         _RepeatTimes = Value
      End Set
   End Property

   Public Property Text() As String
      Get
         Return _Text
      End Get
      Set(ByVal Value As String)
         _Text = Value
```

```
        End Set
    End Property

    Protected Overrides Sub RenderContents(writer As HtmlTextWriter)
        MyBase.RenderContents(writer)
        Dim i As Integer
        For i = 1 To _RepeatTimes
            writer.Write(_Text & "<br>")
        Next
    End Sub

End Class
```

Because we've used a WebControl-derived class instead of an ordinary Control-derived class, and because the code writes the output inside the RenderContents method, the web page programmer can set various style attributes. A sample formatted ConfigurableRepeaterControl is shown in Figure 22-16. If you want to include a title or another portion that you don't want rendered with formatting, add it to the Render method.


RepeaterHost - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/ASP.NET/Chapter22/CustomControls/RepeaterHost.aspx
Done
Local intranet

Repeated Text
Repeated Text
Repeated Text

Figure 22-16: WebControl formatting for repeating control

Creating Adaptive Controls

One interesting approach you can take with custom controls is to vary their output based on the browser. This is actually a process that ASP.NET carries out automatically for some of its more complex controls (such as the validators), ensuring that they are always rendered in the form best suited to the client's browser.

Depending on the sophistication of your control, a lot of thought may need to go into making it support multiple browsers. (Cross-browser support is one of the main criteria that will distinguish the best-of-breed custom ASP.NET controls from third-party vendors.) However, the actual implementation details are trivial—all you need to do is retrieve the browser type and respond accordingly in the Render or RenderContents method.

A sample control is shown next that simply evaluates the capabilities of its container. It generates the output shown in Figure 22-17.

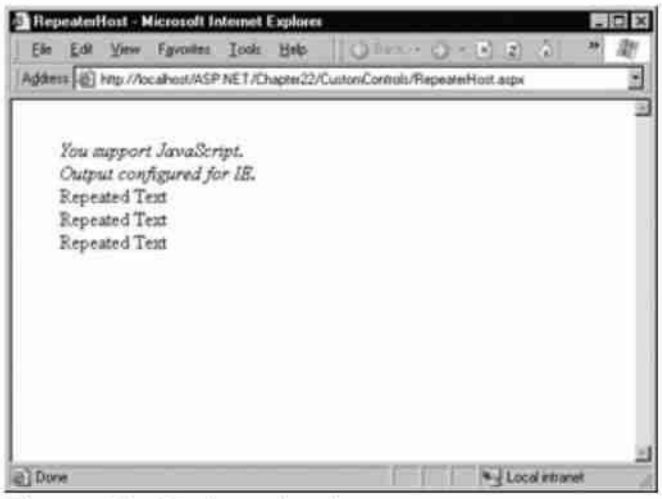

You support JavaScript.
Output configured for IE.
Repeated Text
Repeated Text
Repeated Text

Figure 22-17: An adaptive repeater

```
Protected Overrides Sub RenderContents(writer As HtmlTextWriter)
   MyBase.RenderContents(writer)

   If Me.Page.Request.Browser.JavaScript = True Then
      writer.Write("<i>You support JavaScript.</i><br>")
   End If

   If Me.Page.Request.Browser.Browser = "IE" Then
      writer.Write("<i>Output configured for IE.</i><br>")
   ElseIf Me.Page.Request.Browser.Browser = "Netscape" Then
      writer.Write("<i>Output configured for Netscape.</i><br>")
   End If

   Dim i As Integer
   For i = 1 To _RepeatTimes
      writer.Write(_Text & "<br>")
   Next

End Sub
```

Maintaining Control State

Currently, the repeater control provides an EnableViewState property, but it doesn't actually abide by it. You can test this out by creating a simple page with two buttons (see Figure 22-18). One button changes the RepeatTimes to 5, while the other button simply triggers a postback. You'll find that every time you click the Postback button, RepeatTimes is reset to the default of 3. If you change the Text property in your code, you'll also find that it reverts to the value specified in the control tag.

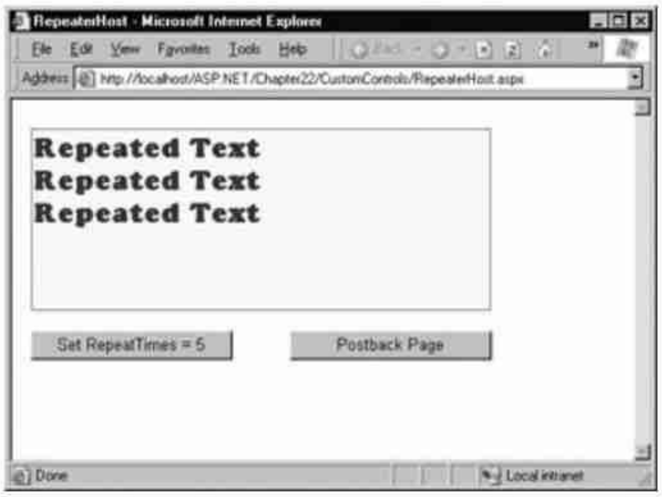


Figure 22-18: Testing viewstate

There's an easy solution to this problem: store the value in the control's viewstate. As with web pages, the value in any member variables in a custom web control class is automatically abandoned after the page is returned to the client.

Here's a rewritten control that uses a variable stored in viewstate instead of a member variable.

```
Public Class ConfigurableRepeaterControl
 Inherits WebControl

  Public Sub New()
    Viewstate("RepeatTimes") = 3
    Viewstate("Text") = "Text"
  End Sub

  Public Property RepeatTimes() As Integer
    Get
      Return CType(Viewstate("RepeatTimes"), Integer)
    End Get
    Set(ByVal Value As Integer)
      Viewstate("RepeatTimes") = Value
    End Set
  End Property

  Public Property Text() As String
    Get
      Return CType(Viewstate("Text"), String)
    End Get
    Set(ByVal Value As String)
      Viewstate("Text") = Value
    End Set
  End Property

  Protected Overrides Sub RenderContents(writer As HtmlTextWriter)
    MyBase.RenderContents(writer)
    Dim i As Integer
    For i = 1 To CType(Viewstate("RepeatTimes"), Integer)
      writer.Write(CType(Viewstate("Text"), String) & "<br>")
    Next
  End Sub

End Class
```

The code is essentially the same, although it now uses a constructor to initialize the RepeatTimes and Text values. Extra care must also be taken to make sure that the Viewstate object is converted to the correct data type. Performing this conversion manually ensures that you won't end up with bugs or quirks that are difficult to find. Note that though the code looks the same as the code used to store a variable in a Page object's viewstate, the collections are different. That means that the web page programmer won't be able to access the control's viewstate directly.

You'll find that if you set the EnableViewState property to False, changes will not be remembered, but no error will occur. When viewstate is disabled, ASP.NET still allows you to write items to viewstate, but they won't persist across postbacks.

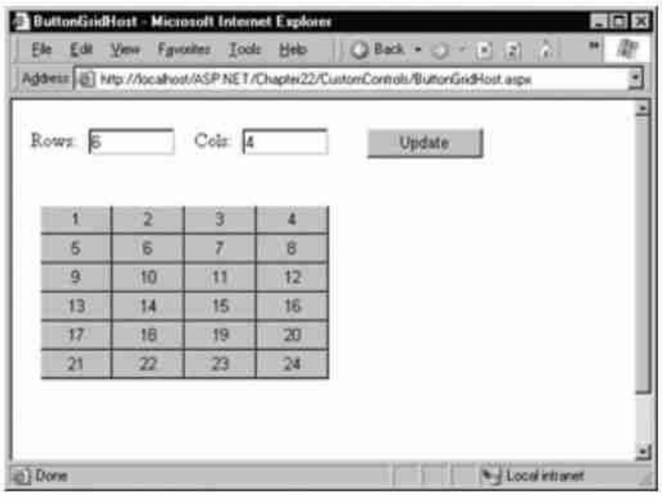
Creating a Composite Control

So far, we've seen how user controls are generally used for aggregate groups of controls with some added higher-level business logic, while custom controls provide you with more control that allows you to modify or create the final HTML output from scratch. This distinction is usually true. You'll also find that user controls are generally quicker to create, easier to work with in a single project, and simpler to program. Custom controls, on the other hand, provide extensive low-level control features we haven't even considered in this chapter, such as templates and data binding.

One technique we haven't considered is composite controls—custom controls that are built out of other controls. Composite controls are a little bit closer to user controls, because they render their user interface at least partly out of other controls. For example, you might find that you need to generate a complex user interface using an HTML table. Rather than write the entire block of HTML manually in the Render method (and try to configure it based on various higher-level properties), you could dynamically create and insert a Table web control. This pattern is quite common with ASP.NET server controls—for example, it's logical to expect that advanced controls such as the Calendar and DataGrid rely on simpler table-based controls like Table to generate their user interface.

Generating composite controls is quite easy. All you need to do is create the control objects you want to use and add them to the Controls collection of your custom control. (Optionally, you could use one or more container controls, such as the PlaceHolder or Panel, and add the controls to the Controls collection of a container control.) By convention, this task is carried out by overriding the CreateChildControls method.

The following example creates a grid of buttons, based on the Rows and Cols properties. A simple test page for this control is pictured in Figure 22-19.


ButtonGridHost - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Address http://localhost/ASP.NET/Chapter22/CustomControls/ButtonGridHost.aspx
Rows: 6
Cols: 4
Update
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
17 18 19 20
21 22 23 24
Done
Local intranet
Figure 22-19: A composite control using buttons

```
Public Class ButtonGrid
 Inherits WebControl

   Public Sub New()
      Viewstate("Rows") = 2
      Viewstate("Cols") = 2
   End Sub

   Public Property Cols() As Integer
      Get
         Return CType(Viewstate("Cols"), Integer)
      End Get
      Set(ByVal Value As Integer)
         Viewstate("Cols") = Value
      End Set
   End Property

   Public Property Rows() As Integer
      Get
         Return CType(Viewstate("Rows"), Integer)
      End Get
      Set(ByVal Value As Integer)
         Viewstate("Rows") = Value
      End Set
   End Property
```

```
  Protected Overrides Sub CreateChildControls()
    Dim i, j, k As Integer

    For i = 1 To CType(Viewstate("Rows"), Integer)

      For j = 1 To CType(Viewstate("Cols"), Integer)
        k += 1

        ' Create and configure a button.
        Dim ctrlB As New Button()
        ctrlB.Width = Unit.Pixel(60)
        ctrlB.Text = k.ToString()

        ' Add the button.
        Me.Controls.Add(ctrlB)
      Next

      ' Add a line break.
      Dim ctrlL As New LiteralControl("<br>")
      Me.Controls.Add(ctrlL)
    Next
  End Sub

End Class
```

Custom Control Events

Raising an event from a control is just as easy as it was with a user control. All you need to do is define the event (and any special EventArgs class) and then fire it with the RaiseEvent statement.

However, to raise an event, your code needs to be executing—and for your code to be triggered, the web page needs to be posted back from the server. If you need to create a control that reacts to user actions instantaneously, and then refreshes itself or fires an event to the web page, you need a way to trigger and receive a postback.

In an ASP.NET web page, web controls fire postbacks by calling a special JavaScript function called __doPostBack. The __doPostBack function accepts two parameters: the name of the control that triggered the postback, and a string representing additional postback data. You can retrieve a reference to this function using the special Page.GetPostBackEventReference property from your control. (Every control provides the Page property, which gives a reference to the web page where the control is situated.)

The GetPostBackEventReference reference allows you perform an interesting trick—namely, creating a control or HTML link that invokes the __doPostBack function. The easiest way to perform this magic is usually to add an OnClick attribute to an HTML element (anchors, images, and buttons all support this attribute).

Consider the ButtonGrid control. Currently, the buttons are created, but there is no way to receive their events. This can be changed by setting each button's OnClick event to the __doPostBack function. The single added line is highlighted in bold:

```
Protected Overrides Sub CreateChildControls()
  Dim i, j, k As Integer

  For i = 1 To CType(Viewstate("Rows"), Integer)

    For j = 1 To CType(Viewstate("Cols"), Integer)
      k += 1

      ' Create and configure a button.
      Dim ctrlB As New Button()
      ctrlB.Width = Unit.Pixel(60)
```

```
        ctrlB.Text = k.ToString()

        ' Set the OnClick attribute with a reference to
        ' __doPostBack. When clicked, ctrlB will cause a
        ' postback to the current control and return
        ' the assigned text of ctrlB.
        ctrlB.Attributes("OnClick") = _
          Page.GetPostBackEventReference(Me, ctrlB.Text)

        ' Add the button.
        Me.Controls.Add(ctrlB)
      Next

      ' Add a line break.
      Dim ctrlL As New LiteralControl("<br>")
      Me.Controls.Add(ctrlL)
   Next
End Sub
```

To handle the postback, your custom control needs to implement the IPostBackEventHandler interface.

```
Public Class ButtonGrid
  Inherits WebControl
  Implements IPostBackEventHandler
```

You then need to create a method that implements IPostBackEventHandler.RaisePostBackEvent. This method will be triggered when your control fires the postback, and it will receive the additional information submitted with the GetPostBackEventReference method (in this case, the text of the button).

```
Public Overridable Overloads Sub RaisePostBackEvent(eventArgument _
  As String) Implements IPostBackEventHandler.RaisePostBackEvent

    ' Respond to postback here.

End Sub
```

Once you receive the postback, you can modify the control, or even raise another event to the web page. To enhance the ButtonGrid example to use this method, we'll define an additional event in the control:

```
Event GridClick(ByVal ButtonName As String)
```

This event handler raises an event to the web page, with information about the button that was clicked. You'll notice that the event definition doesn't follow the .NET standard for simplicity's sake.

The RaisePostBackEvent method can then trigger the event.

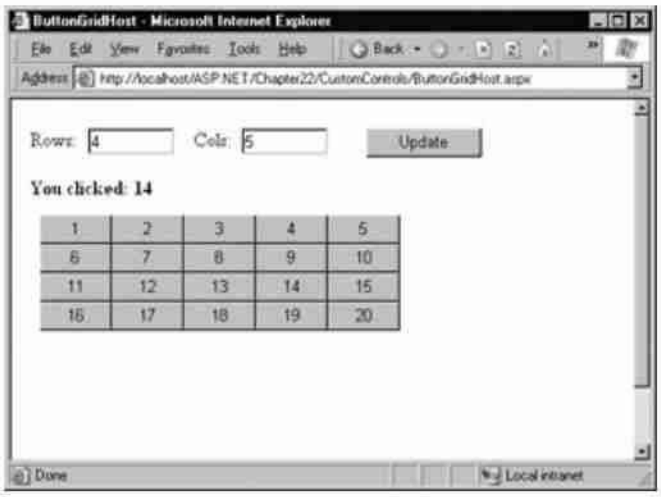
```
Public Overridable Overloads Sub RaisePostBackEvent(eventArgument _
  As String) Implements IPostBackEventHandler.RaisePostBackEvent

    RaiseEvent GridClick(eventArgument)

End Sub
```

A web page client that wants to handle the event might use event-handling code that looks something like this:

```
Private Sub ButtonGrid1_GridClick(ButtonName As String) _
  Handles ButtonGrid1.GridClick

    lblInfo.Text = "You clicked: " & ButtonName
```

```
End Sub
```

Figure 22-20 shows the ButtonGrid events in action.


ButtonGridHost - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Address http://localhost/ASP.NET/Chapter22/CustomControls/ButtonGridHost.aspx
Rows: 4
Cols: 5
Update
You clicked: 14
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
Done
Local intranet
Figure 22-20: Handling custom control events

ASP.NET custom control creation could be a book in itself. To master it, you'll want to experiment with the online samples for this chapter (provided in a CustomControls project for Visual Studio .NET users). Once you've perfected your controls, pay special attention to the attributes described in Table 22-1. These are key to making your control behave properly and work conveniently in design environments like Visual Studio .NET.

Chapter 23: Caching and Performance Tuning

Overview

ASP.NET applications are a bit of a contradiction. On one hand, because they are hosted over the Internet they have unique requirements—namely, they need to be able to serve hundreds of clients as easily and quickly as they deal with a single user. On the other hand, ASP.NET includes some remarkable "tricks" that let you design and code a web application in a similar way as you might program a desktop application. This uniqueness can lead developers into trouble. ASP.NET makes it easy to forget you are creating a web application—so easy that you might introduce programming practices that will slow or cripple your application when it's heavily used in the real world.

Fortunately, there is a middle ground. You can use the incredible time-saving features like viewstate, server postbacks, and session state that you have spent the last twenty-some chapters learning about, and still create a robust web application. But you need to finish the job, and spend the extra 10 percent of time necessary at every stage of your programming to ensure that your application will perform. This chapter discusses these strategies, which fall into three main categories:

Design for scalability There are a few key guidelines that, if kept in mind, can steer you toward efficient, scalable designs.

Profile your application One problem with web applications is that it's sometimes hard to test them under the appropriate conditions and really get an idea of what their problems may be. However, Microsoft provides several useful tools that allow you to benchmark your application and put it through a rigorous checkup.

Implement caching A little bit of caching may seem like a novelty in a single-user test, but it can make a dramatic improvement in a real-world scenario. You can easily incorporate output and fragment caching into most pages, and use data caching to replace memory-unfriendly state management.

Designing for Scalability

Throughout this book, the chapters have combined practical how-to information with some tips and insight about the best designs (and possible problems). However, now that you are a more accomplished ASP.NET programmer, it's a good idea to review a number of considerations—and a few minor ways that you can tune up all aspects of your application.

ASP versus ASP.NET

ASP.NET provides dramatically better performance than ASP, although it's hard to quote a hard statistic because the performance increases differ widely depending on the ways you structure your pages and the type of operations you perform. By far, the greatest performance increase results from the new automatic code compilation. With ASP, your web pages and their script code were processed for every client request. With ASP.NET, each page class is compiled to native code the first time it is requested, and then cached for future requests.

This system does have one noticeable side effect, however. The very first time that a user accesses a particular web page (or the first time the user accesses it after it has been modified), there will be a longer delay while the page is compiled. In some cases, this time for the first request may actually be slightly slower than the equivalent ASP page request. However, this phenomenon will be much more common during development than in an actual deployed web site.