Lecture 10

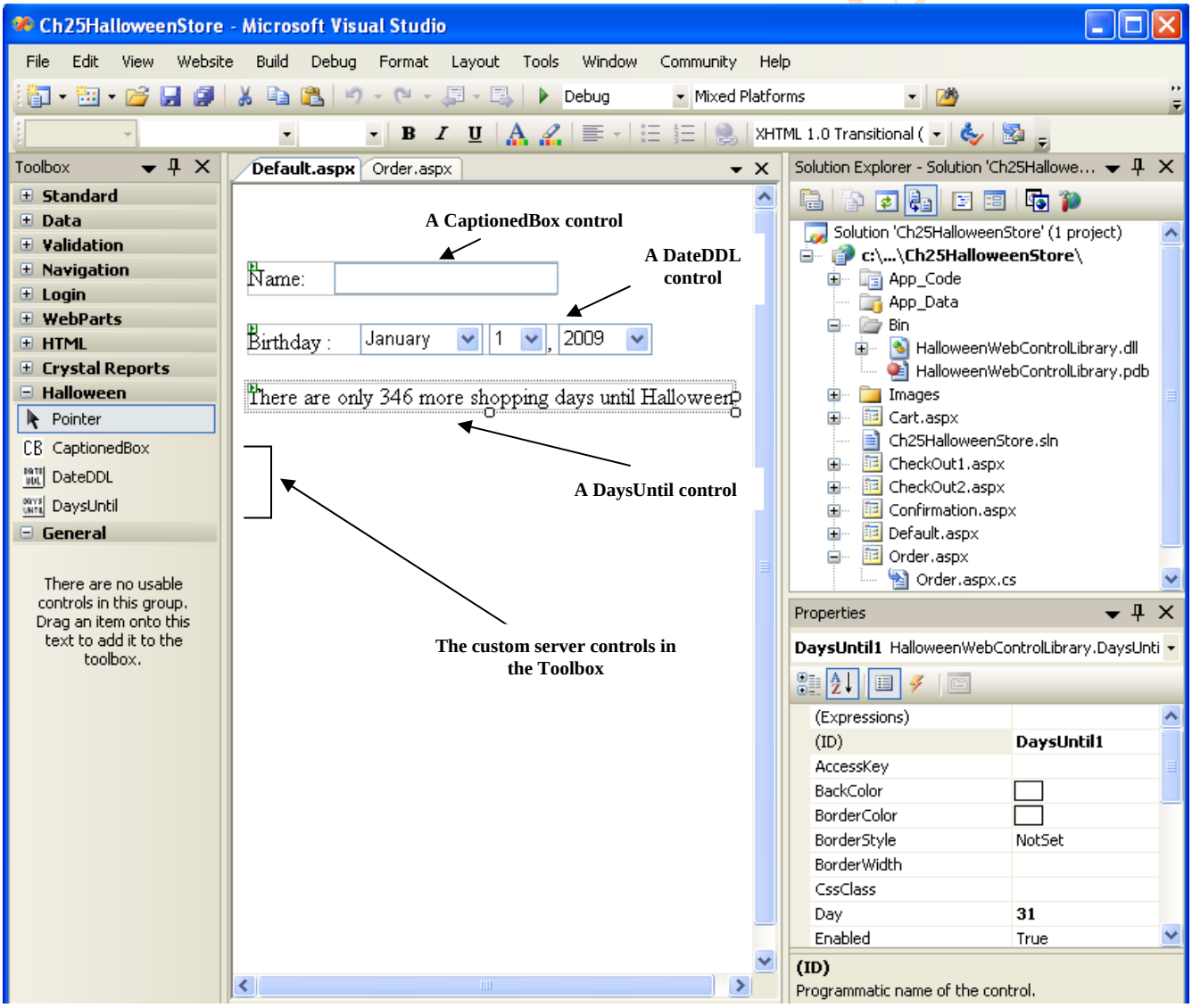
Custom server controls development

Custom server controls are similar to user controls, but are more powerful. Since custom server controls are an advanced feature, ***you can't use the Standard Edition of Visual Studio or the Express Edition Web Developer to develop them***.

An introduction to custom server controls

A custom server controls are similar to the server controls that come with ASP.NET. Because you create custom server controls yourself, however, the can include features that aren't available with the standard web server controls.

An example of three custom server controls displayed in the Web Forms Designer



The *CaptionedBox* control is similar to *TextBox* control but include a caption to the of the text box. The second control, called a *DateDDL* control, consists of the three drop-down lists that let the user select a month, day, and year. Like the *CaptionedBox* control, the *DateDDL* control also includes a caption. The third control, called a *DaysUntil* control, calculates the number of days until a specified date. In addition, text can be displayed before and after the calculated days.

The *aspx* code for the page shown above

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

<%@ Register Assembly="HalloweenWebControlLibrary"
Namespace="HalloweenWebControlLibrary"  TagPrefix="cc1" %>

<!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>Untitled Page</title>
</head>
<body>
  <form id="form1" runat="server">
  <div>
     </div>
     
    <cc1:CaptionedBox ID="CaptionedBox1" CaptionPadding="20px"
      Caption="Name: " runat="server">
    </cc1:CaptionedBox><br />
    <cc1:DateDDL ID="DateDDL1" CaptionPadding="20px"
      Caption="Birthday :" runat="server" />
    <br />
    <br />
    <cc1:DaysUntil ID="DaysUntil1"
       TextBefore="There are only "
       TextAfter="more shopping days until Halloween"
       runat="server" />
  </form>
</body>
</html>
```

When you create a custom server control, you base it on the *WebControl* class or a class that's derived from this class. That way, the control inherits the properties, methods, and events of that class. For example:

1. The *DaysUntil* control is based on the *WebControl* class.
2. The *CaptionedBox* control is based on the *TextBox* class so it has access to the additional properties, methods, and events defined by that class.

3. The *DateDDL* control is based on the *CompositControl* class.

In addition to the properties, methods, and events that are inherited from the base class, a custom server control can define its own custom properties, methods, and events. For example, the *DaysUntil* control has properties that let you set the month and day that's used in the calculation, as well as properties that let you specify the text that's displayed before and after the calculated number of days. You can also override the properties, methods and events defined by the base class.

Custom server controls are typically stored in a ***web control library***. Then, if you add the web control library to a web site as shown above, the custom server controls become available from the Toolbox. To use a control, you simply drag it to a web form just as you would any other control.

Although custom server controls are easy to use, they can be difficult to develop. That's because Visual Studio doesn't provide a designer interface to help you create them. Instead, you have to develop custom server controls entirely in code. As a result, you can't see how the control will appear until you build it and add it to a web form.

Custom server controls vs. user controls

Custom server controls...

- Are best used to create general-purpose controls you want to reuse in several applications
- Are more difficult to develop that user controls
- Are created entirely in code and consist of a single *cs/vb* class file
- Are usually created in Web Control Library projects
- Are available from the Toolbox

User controls...

- Are best used when you want to create elements you can reuse on several pages within the same applications
- Are easier to develop than custom server controls
- Are created with the User Control Designer and consist of an *ascx* file that defines the control's user interface along with a code-behind file
- Aren't available from the Toolbox

User controls tend to be application-specific. For example, a typical use of user controls is to create the banner, navigation menu, and footer for an application. Then, you can include those user controls in each page of the application. Since you're unlikely to use the same user controls in another application, you usually add them directly to the application that uses them.

Custom sever controls usually more general-purpose functions than user controls. For example, the *CaptionedBox* and *DateDDL* user controls from figure shown above could be used in variety of applications. To make them available to other applications, custom server controls are usually created in a separated *Web Control Library* project. Then, Visual Studio lets you add the controls in the library to the *Toolbox* so they're available to any application. To make them available to other applications, custom server controls are usually created in a separate Web Control Library project. Then, if you add the Web Control Library project to a web site, Visual Studio adds the controls in the library to the Toolbox so they're available to all of the forms of the web site.

Three types of custom server controls

Below we will describe the three basic types of custom server controls. You implement each of these controls using different programming techniques. But you have to keep in mind, however, that they're not mutually exclusive. In other words, you can use one or more of these techniques within the same custom server control.

A rendered control is a control that contains code that generates all or most of the HTML that's used to render the control.
The render control, for example, is shown in figure above. It is the *DaysUntil* control. Rendered controls are usually based on the *WebControl* class, which provides the basic functionality of an ASP.Net Server control. Then, the custom control overrides the *Render* method of this class to generate its own HTML.

A superclassed control is a control that adds functionality to an existing ASP.Net Server control. The *CaptionedBox* control you in figure above, for example, adds a caption to a text box control. Because this control is based on the *TextBox* class rather than the *WebControl* class, it has all the properties, methods, and events of a standard text box control.

A composite control is a control that's made up of two or more ASP.Net Server controls. For example, the *DateDDL* control consists of three drop-down list controls. Composite controls are based on the *WebControl* class. Then, the ***child controls*** that up the control are added to the collection of controls defined by the *CompositeControl* class.

Note that a control can sometimes be implemented using more than one of the control types. For example, you could implement the *DaysUntil* control using any three of the control types. To implement it as a superclassed control, you would base it on a *Label* control. Then, you would calculate the days remaining until the target date and assign the result to the *Text* property of the control, along with the text you want to display before and after the calculated days. To implement this control as a composite control, you should use three *Label* controls: one for the text you want to display before the calculated value, one for the calculated value, and one for the text you want to display after the calculated value. Before you create a control, then, you should consider which type will provide the greatest flexibility and ease of use.

Creating a rendered control

When you add a custom server control to a project, Visual Studio creates a class like the one shown below.

The starting code for a custom server control

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Text;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
```

```
namespace HalloweenWebControlLibrary
{
   [DefaultProperty("Text")]
   [ToolboxData("<{0}:WebCustomControl1 runat=server></{0}:WebCustomControl1>")]
   public class WebCustomControl1 : WebControl
   {
      [Bindable(true)]
      [Category("Appearance")]
      [DefaultValue("")]
      [Localizable(true)]
      public string Text
      {
         get
         {
            String s = (String)ViewState["Text"];
            return ((s == null) ? String.Empty : s);
         }

         set
         {
            ViewState["Text"] = value;
         }
      }

      protected override void RenderContents(HtmlTextWriter output)
      {
         output.Write(Text);
      }
   }
}
```

This class inherits the *WebControl* class and contains a single property named *Text* that gets and sets the value of a variable that's stored in the view state of the page. This class also contains a method named *RenderContents* that overrides the *RenderContents* method of the *WebControl* class.

As it is, the default class doesn't provide a useful function. However, it does provide a good start point for building useful controls. Before you use this class, though, you need to understand how it works. If you've created your own class files using C#, you shouldn't have much trouble understanding most of this code. However, a few of the details might be confusing.

To start, notice the code within the square brackets at the beginning of the class and the property declarations. This code defines ***designer attributes*** that supply the Web Forms Designer with information about the control when it's used on a web form.

Next, notice that the class overrides the *RenderContents* method of the *WebControl* class that it inherits. That way, it can generate the HTML needed to display the control. In this case, the *RenderContents* method uses the *Write* method of the *HtmlTextWriter* object that's passed to it to write the value of the *Text* property to the control's HTNL output stream.

You should realized that because the *RenderContents* method in the default class overrides the *RenderContents* method of the *WebControl* class, the properties of the *WebControl* class that the default class inherits, such as *ID, Height, Width*, and *BorderStyle*, never get rendered. To render these properties, you have to call one of the *Render* methods of the *WebControl* class.

Overriding the *Render* or/and *RenderContents* method for a custom control

Every web server control has a *Render* or *RenderContents* method that's responsible for generating the HTML that's sent to the browser to display the control. In a custom server control, you override one of these methods so you can generate the HTML that renders the control.

Methods of the *WebControl* class for rendering controls

Method	Description
RenderContents(output)	Renders the contents of the control to the specified *HtmlTextWriter* object, but does not include its start and end tags
Render(output)	Renders the HTML for the control to specified *HtmlTextWriter* object, including the start and end tags
RenderBeginTag(output)	Renders the begin tag for the control to specified *HtmlTextWriter* object
RenderEndTag(output)	Renders the end tag for the control to the specified *HtmlTextWriter* object

When ASP.NET receives a request for a web page, it creates an *HtmlTextWriter* object that's used to write HTML to the output stream. This object is passed to the *Render* and *RenderContents* methods, and you can use it within these methods to write the HTML that renders the control.

The example below shows how to override the *RenderContent* method to add content to the HTML element for the base class.

```
protected override void RenderContents(HtmlTextWriter output)
{
   output.Write(Text);
}
```

Here, the *Write* method of the *HtmlTextWriter* object adds the value of the *Text* property between the start and end tags that are generated automatically for the base class. The start tag that's generated includes attributes that correspond to any control properties you set as you work with the control in the designer. As a result, you can use the *RenderContent* method to render the properties the custom control inherits from its base class.

The second example shows how to override the *Render* method to add content to the HTML element for the base class.

```
protected override void Render(HtmlTextWriter output)
{
   //begin writing table
   output.WriteBeginTag("table");
```

```
        output.Write(HtmlTextWriter.TagRightChar);

        //finish writing table
        output.WriteEndTag("table");

        }
```

The *Render* method is similar the *RenderContents* method, but it renders the start and end tags as well as the content. Because of that, when you override this method to add content to the HTML element, you must use the *RenderBeginTag* and *RenderEndTag* methods of the base class to render the start and end tags for the control. To access methods from the base class, you can use keyword as illustrated in this example.

The third example shows how to override the *Render* method to add additional HTML elements before or after the element for the base class.

```
protected override void Render(HtmlTextWriter output)
{
        output.Write("Caption");

        base.Render(output);
}
```

Here, the first statement calls the *Write* method of the *HtmlTextWriter* object to add an HTML element before the element for the base class. This statement sends a string that includes an HTML element for the caption to the browser. Then, the second statement calls the *Render* method of the base class. This method generates the HTML necessary to the render the base class, including the start and end tags, and sends it to the browser. In this case, the class is based on the *TextBox* class. As a result, calling the *Render* method of the base class renders an HTML element for a text box and sends it to the browser.

The design of the DaysUntil control

This control displays the number of days remaining until the target date specified by the *Month* and *Day* properties. The default values for the *Month* and *Day* properties are 10 and 31.

The *aspx* code for the *DaysUntil* control

```
<cc1:DaysUntil ID="DaysUntil1" runat="server"
     TextAfter="more shopping days until Halloween"
    TextBefore="There are only" />
```

Properties of the *DaysUntil* control

Property	Description
Month	The target date's month. The default value is 10.
Day	The target date's day. The default is 31.
TextBefore	The text that is displayed before the days until the target date.
TextAfter	The text that is displayed after the days until the target date.

The C# code for the *DaysUntil* control

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Text;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace HalloweenWebControlLibrary
{
  [ToolboxData("<{0}:DaysUntil runat=server></{0}:DaysUntil>")]
  public class DaysUntil : WebControl
  {
    [Bindable(true)]
    [Category("Appearance")]
    [DefaultValue("")]
    [Localizable(true)]
    public string TextBefore
    {
      get
      {
        String s = (String) ViewState["TextBefore"];
```

```
        return ((s == null) ? String.Empty : s);
    }

    set
    {
        ViewState["TextBefore"] = value;
    }
}

[Bindable(true)]
[Category("Appearance")]
[DefaultValue("")]
[Localizable(true)]
public string TextAfter
{
    get
    {
        String s = (String) ViewState["TextAfter"];

        return ((s == null) ? String.Empty : s);
    }

    set
    {
        ViewState["TextAfter"] = value;
    }
}

[Bindable(true)] [Category("Appearance")]
[DefaultValue("")] [Localizable(true)]
public int Month
{
    get
    {
        String s = (String) ViewState["Month"];

        if (s == null || s.Equals(""))
            return 10;
        else
            return Convert.ToInt32(ViewState["Month"]);
    }
    set
    {
        ViewState["Month"] = value.ToString();
    }
}
```

```
    [Bindable(true)] [Category("Appearance")]
    [DefaultValue("")] [Localizable(true)]
    public int Day
    {
      get
      {
        String s = (String)ViewState["Day"];

        if (s == null || s.Equals(""))
          return 31;
        else
          return Convert.ToInt32(ViewState["Day"]);
      }

      set
      {
        ViewState["Day"] = value.ToString();
      }
    }

    protected override void RenderContents(HtmlTextWriter output)
    {
      output.Write(TextBefore + " ");
      output.Write(DaysUntilDate());
      output.Write(" " + TextAfter);
    }

    private int DaysUntilDate()
    {
      DateTime targetDate = new DateTime(DateTime.Today.Year, Month, Day);

      if (DateTime.Today > targetDate)
        targetDate = targetDate.AddYears(1);

      TimeSpan timeUntil = targetDate - DateTime.Today;

      return timeUntil.Days;
    }

  }
}
```

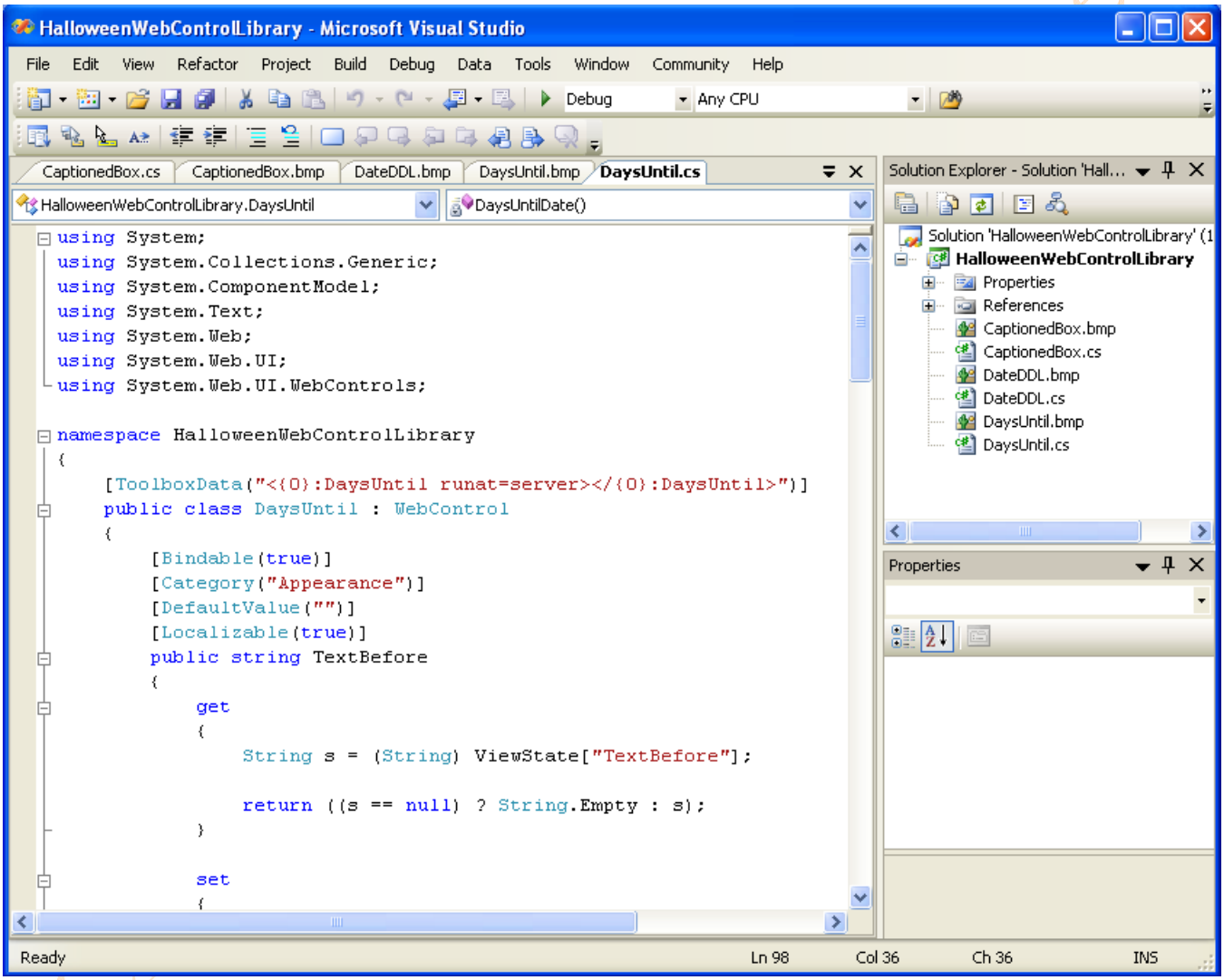
This class finishes with the overridden *RenderContents* method. This method uses the *HtmlTextWriter* object to write the value of the *TextBefore* property, followed by number of days remaining until the

target date, followed by the value of the *TextAfter* property. To calculate the number of days remaining until the target date, the *RenderContents* method calls the *DaysUntilDate* method.

How to create and test a web control library

To create and use custom server controls in Visual Studio you can create a Web Control Library project that contains one or more custom server controls. Then, you can add this project to any web site to test these controls.

Figure below presents a procedure for creating and testing a web control library.



Step 1: Create a web control library

The first step is to create a new Web Control Library project. To do that, you have to do as follows:

1. Display the *New Project* dialog box *(File → New Project)*
2. Select the *Web Control Library* template that's available from the *Window* group of the C# group, and enter a name (for example: *HalloweenWebControlLibrary*) and location for the

project. By default, a Web Control Library project is stored in the Visual Studio projects location directory.

When you create a Web Control Library project, the project is created with a control named *WebCustomControl1* by default. This control contains the starting code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Text;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace HalloweenWebControlLibrary
{
  [DefaultProperty("Text")]
  [ToolboxData("<{0}:WebCustomControl1 runat=server></{0}:WebCustomControl1>")]
  public class WebCustomControl1 : WebControl
  {
    [Bindable(true)]
    [Category("Appearance")]
    [DefaultValue("")]
    [Localizable(true)]
    public string Text
    {
      get
      {
        String s = (String)ViewState["Text"];
        return ((s == null) ? String.Empty : s);
      }

      set
      {
        ViewState["Text"] = value;
      }
    }

    protected override void RenderContents(HtmlTextWriter output)
    {
      output.Write(Text);
    }
  }
}
```

Step 2: Create a custom server controls

To create the first control, you can modify the *WebCustomControl1.cs* file that's generated by default. Alternatively, you can delete this file and add a new file for each custom control.

To add a custom server control you have to:

1. Select the *Add → New* Item command to display the *Add New Item* dialog box.
2. Select the *Web Custom Control* template and enter a name for the control.

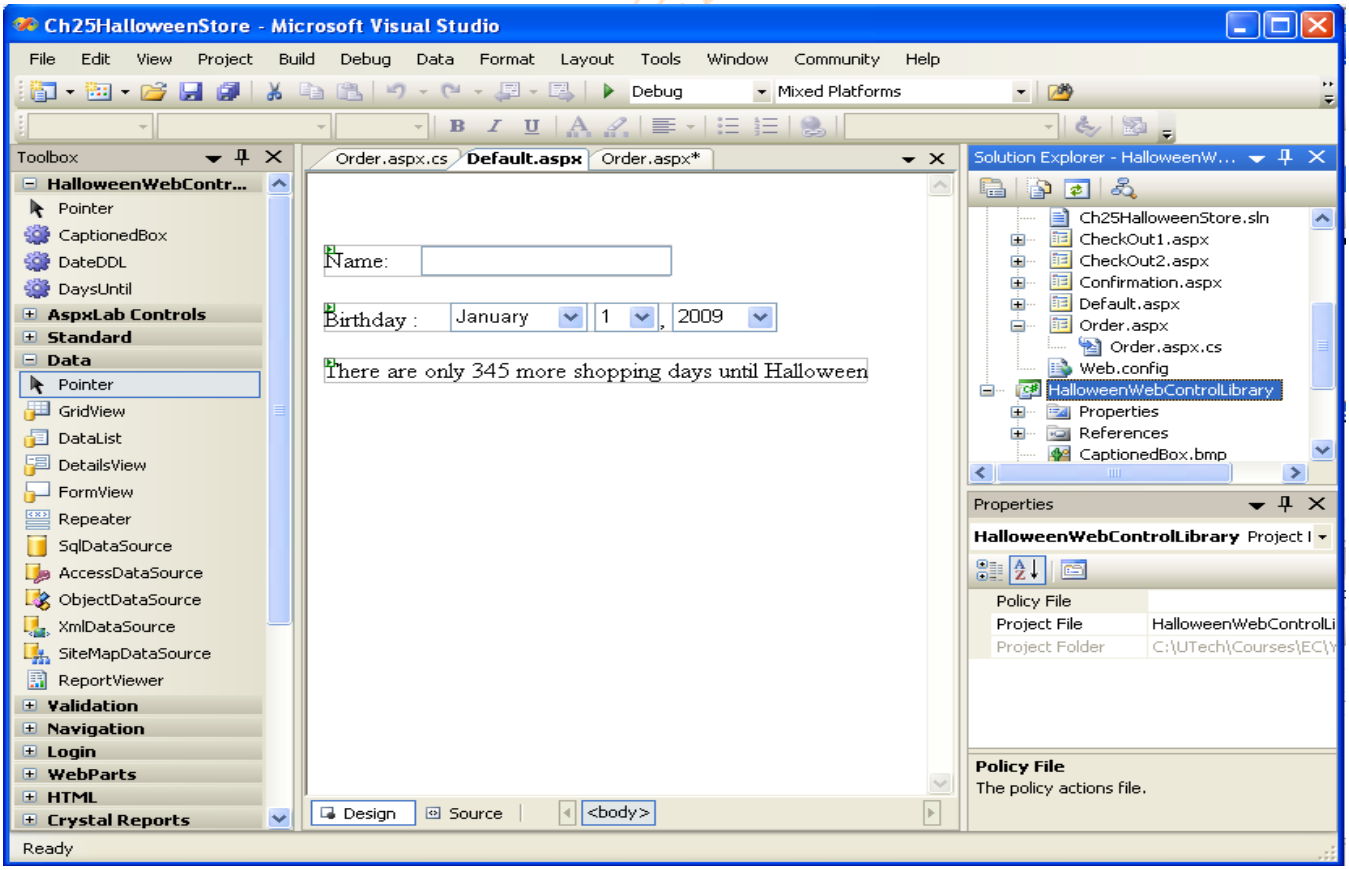
After you add a new control to a web control library, you need to add the code that implements the control.

Step 3: Build the web control library

The next step is to build the web control library. To do that, just right-click the project and select the *Build* command. Then, Visual Studio creates the files you'll need to use the web control library in a web site.

Step 4: Add the library to a web site

One way to test your custom server controls is to add the web control library to a new or existing web site. To do that, create or open the web site and then use the *File → Add → Existing Project* command to display the *Add Existing Project* dialog box. Then, locate and select the project file for the web control library. In figure below, for example, the *HalloweenWebControlLibrary* project shown in the previous figure has been added to a web site.



Step 5: Test the controls

When you add a Web Control Library project to a web site, Visual studio automatically adds the custom controls in the library to the Toolbox. The controls are included in a group with the name defined by developer (in our case it is *HalloweenWebControls* tab).

At this point, you can test a control by dragging it from the Toolbox to the form. The first time you add a custom control to a form, Visual studio creates a *Bin* folder. This folder contains the ***dll*** and ***pdb*** files for the web control library.

After you add a custom control to a form, you can set the control properties using the *Properties* window.

If you find that a control doesn't look or work the way you want it to, you can change its code and then rebuild the web control library that contains it. You may also need to rebuild a form that uses a custom control or refresh the custom controls on a form if you change the code or properties for the control. To rebuild a form, just, right-click on it in the Solution Explorer and select the *Build* Page command. To refresh a control, right-click on it in the form and select the *Refresh* command.

When you test your custom controls, you should realize that coding errors sometimes manifest themselves in strange ways. Because of that, they can be difficult to debug. For example, if a control generates invalid HTML, it won't display correctly in the Web Forms Designer. Typically, the designer will simply display the name of the control in brackets. In other cases, the control will appear correctly in the designer, but it won't display properly when you run the application. And in still other cases, the control will throw an exception when you run the application.

Fortunately, you can use the debugger to help you determine the case of an error. If a control isn't rendered properly, for example, you can set a breakpoint in the *Render* or *RenderContents* method. Then, you can review variable values and monitor program execution to determine the case of the problem.

If the control displays in the browser but doesn't appear or work the way it should, you can also look at the HTML that's generated to be sure it's correct. To do that, use the *View* → *Source* command in your browser's menu.

The *aspx* code for these three controls

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

<%@ Register Assembly="HalloweenWebControlLibrary"
Namespace="HalloweenWebControlLibrary"
   TagPrefix="cc1" %>

<!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>Untitled Page</title>
</head>
<body>
```

```
  <form id="form1" runat="server">
  <div>
     </div>
     
    <cc1:CaptionedBox ID="CaptionedBox1" CaptionPadding="20px"
      Caption="Name: " runat="server">
    </cc1:CaptionedBox><br />
    <cc1:DateDDL ID="DateDDL1" CaptionPadding="20px"
      Caption="Birthday :" runat="server" />
    <br />
    <br />
    <cc1:DaysUntil ID="DaysUntil1"
       TextBefore="There are only "
       TextAfter="more shopping days until Halloween"
       runat="server" />
  </form>
</body>
</html>
```

Using the *HtmlTextWriter* class to render HTML

As you know before, the *Render* and *RenderContents* methods of a custom server control receive an *HtmlTextWriter* object named output as an argument. You use this argument to generate the HTML needed to display the control. You've already seen how to use the *Write* method of this class to write simple text to the output stream. You can use this method to render all of the HTML for a custom server control by carefully constructing the HTML using C# string handling features. Below we present some additional methods of the *HtmlTextWriter* class that make it easier to generate HTML output.

Common methods of the *HtmlTextWriter* class

Method	Description
Write(string)	Writes the specified string to the HTML output
WriteBeginTag(tagname)	Writes the start tag for the specified HTML element. This method does not write the closing brackets (>) for the start tag. That way, you can include attribute within the tag.
WriteAttribute(attribute, value)	Writes the specified attribute and assigns the specified value. This method should be coded after the *WriteBeginTag* method but before the Write method, that adds the closing brackets for the start tag.
WriteEndTag(tagname)	Writes the end tag for the specified HTML element

Common fields of the *HtmlTextWriter* class

Field	Description
TagRightChar	The closing bracket of an HTML tag *(>)*
SelfClosingTagEnd	The closing slash and bracket of a self-closing HTML tag *(/>)*

The methods shown below simplify the process of creating valid HTML. The *WriteBeginTag* method writes an HTML start tag using the element name you supply. Note, however, that this method doesn't write the closing bracket for the start tag. You have to use the *WriteAttribute* method to add attribute to the tag. Then, you can use the *Write* method to add the closing bracket for the tag specifying *HtmlTextWriter.TagRightChar* for the string value. You can also use the *Write* method to add content after the start tag. Then, you can use the *WriteEndTag* method to write the end tag. This is illustrated in the example below. This code creates a td element with a *Style* attribute that specifies the width and alignment of the content between the start and end tags.

If an element doesn't include any content, you can code the start tag as a self-closing tag. To do that, you use the *HtmlTextWriter.SelfClosingTagEnd* field to close the start tag instead of *HtmlTextWriter.TagRightChar*. This field adds the character *"/>"* to the output stream, which indicates the end of the element. Then, you can omit the end tag.

A method that renders a *Span* element with text

```
protected override void RenderCaption(HtmlTextWriter output)
{
        output.WriteBeginTag("td");

        string style = "width: " + CaptionWidth.ToString() + ";";

        style += "text-align: right;";

        output.WriteAttribute("Style", style);
        output.Write(HtmlTextWriter.TagRightChar);
        output.Write(Caption);
        output.WriteEndTag("td");
}
```

Designer attributes

Common attributes for custom control classes

Attribute	Description
DefaultProperty(name)	The name of the default property for the control
ToolboxData(tagName)	The name of the tag that's generated when the control is added to a web form the Toolbox

The figures above present some of the designer attributes you can use with custom control classes and properties. The only two class attributes you'll typically use are the ones that are added by default to a custom control. As its name implies, the *DefaultProperty* attribute identifies the default property for the control. This attribute is set to *Text* by default since this is the only property that's defined for a control when you first create it. If you delete the *Text* property or change its name, you should delete the *DefaultProperty* attribute or change it to another property.

The second class attribute, *ToolboxData*, provides information that Visual Studio uses when you add the control to a form. Specifically, it provides the name of the tag that's used for the control. By default, this is the class name that you specify when you create the web control, which is usually what you want.

The property attributes provide the Web Forms Designer with additional information about the custom properties you define. For example, the attributes that are included on the *Text* property of a custom server controls by default specify that the property can be used for binding, that the property should be listed in the *Appearance* category of the *Properties* window, that the property has no default value, and that the property can be localized to use a specific user's language and culture. You can modify or delete any of these attributes or add any of the other attributes.

One property attribute you're likely to add is *Description*. The value you specify for this attribute is displayed in the *Description* pane at the bottom of the *Properties* window when you select the control in the designer window and then select the property. In this figure, you can see the description that's specified for the *Month* property of the *DaysUntil* control.

Common attributes for custom control properties

Attribute	Description
Bindable(boolean)	Specifies whether the property is typically used for building
Browserable(boolean)	Specifies whether the property should be visible in the *Properties* window. The default is *true*.
Category(string)	The category the property should appear under in the *Properties* window.
DefaultValue(string)	The default value for the property.
Description(string)	A description for the property.
DesignOnly(boolean)	Specifies whether the property is available only at design time. The default is *false*.
Localizable(boolean)	Specifies whether the property can be localized so that it uses the user's language and culture

A property definition with property attributes

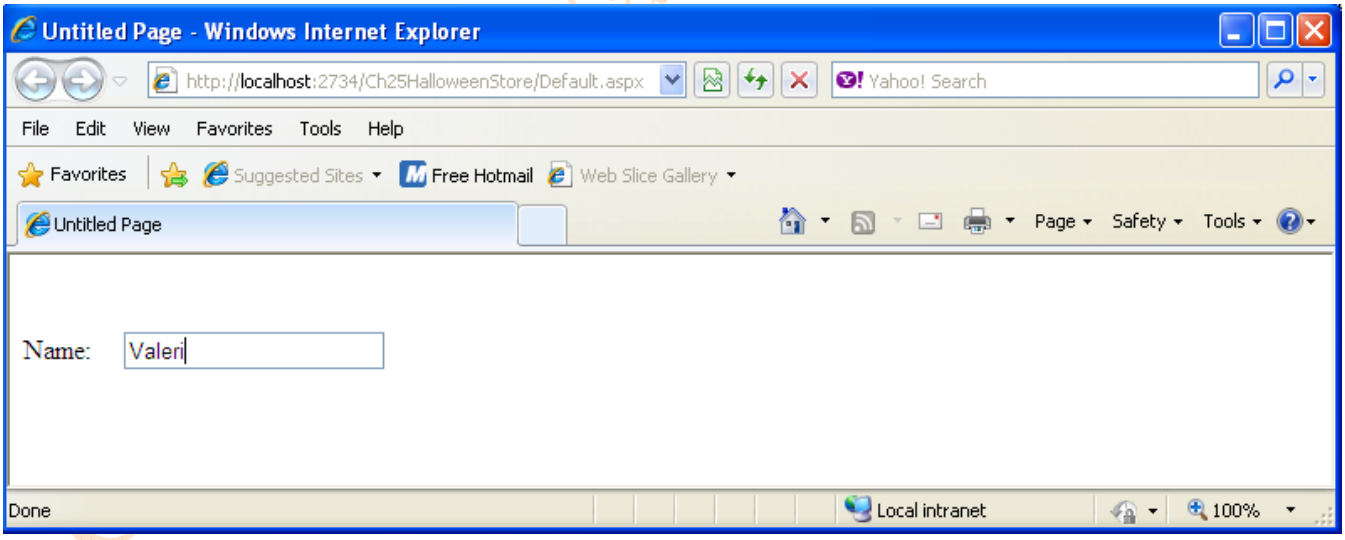
```
[Bindable(true)] [Category("Appearance")] [DefaultValue("")]
[Description("The month used to determine the target date.")]
public int Month
{
    get
    {
```

```
            String s = (String) ViewState["Month"];

            if (s == null || s.Equals(""))
            {
                  return 10;
            }
            else
                  return Convert.ToInt32(ViewState["Month"]);
      }


      set
      {
            ViewState["Month"] = value.ToString();
      }
}
```

Creating a superclassed control

A superclassed control is a control that inherits a specific web server rather than generic *WebControlClass*. The programming techniques for creating a superclassed control are similar to the techniques for created a rendered control.

A web page with the *CaptionedBox* control



Before you create a superclassed control, you should carefully consider which control it inherits. In general, you'll want it to inherit the control that most resembles the appearance and function of the custom control you're creating. For example, if the primary purpose of the control is to display text, it should inherit the *Label* control. On the other hand, if the primary purpose of the control is to accept text input from the user, it should inherit the *TextBox* control.

When you create a superclassed control, you typically use the *Render* method to render its HTML. How you code this method depends on how you're extending the base control. If you're simply adding some HTML before or after the control or if you're just changing some of the base control's property values, your *Render* method can call the *Render* method of the base class to render the entire base control. If your enhancements are more substantial, however, you may need to use the *RenderBeginTag* and *RenderEndTag* methods so you can render additional HTML between the control's start and end tags.

Properties of the *CaptionedBox* control

Property	Description
Caption	The text that's displayed to the left of the text box
CaptionWidth	The width of the control's caption area
CaptionPadding	The width of the area between the caption and the text box
RightAlignCaption	Determines whether the caption text is right-aligned. The default value is *False*.

The *aspx* code for the *CaptionedBox* control

```
<cc1:CaptionedBox ID="CaptionedBox1" CaptionPadding="20px" runat="server"
    Caption="Name: " CaptionWidth="75px" RightAlignCaption="true">
</cc1:CaptionedBox>
```

The C# programming code for the *CaptionedBox* control

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Text;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace HalloweenWebControlLibrary
{
  [DefaultProperty("Text")]
  [ToolboxData("<{0}:CaptionedBox runat=server></{0}:CaptionedBox>")]
  public class CaptionedBox : TextBox
  {
    private string caption;
    private Unit captionWidth;
    private Unit captionPadding;
    private bool rightAlignCaption;

    [Category("Appearance")]
```

```
[Description("The caption displayed to the left of the text box.")]
public string Caption
{
   get { return caption; }
   set { caption = value; }
}

[Category("Appearance")]
[Description("The width of the caption.")]
public Unit CaptionWidth
{
   get { return captionWidth; }
   set { captionWidth = value; }
}

[Category("Appearance")]
[Description("The space between the caption and the text box.")]
public Unit CaptionPadding
{
   get { return captionPadding; }
   set { captionPadding = value; }
}

[Category("Appearance")]
[Description("Determines if the caption is right-aligned.")]
public bool RightAlignCaption
{
   get { return rightAlignCaption; }
   set { rightAlignCaption = value; }
}

protected override void Render(HtmlTextWriter output)
{
   //begin writing table
   output.WriteBeginTag("table");
   output.WriteAttribute("cellpadding", "0");
   output.WriteAttribute("cellspacing", "0");

   output.Write(HtmlTextWriter.TagRightChar);

   //begin writing row
   output.WriteBeginTag("tr");
   output.Write(HtmlTextWriter.TagRightChar);

   //write first column
   output.WriteBeginTag("td");
```

```
            string style = "width: " + CaptionWidth.ToString() + ";";

            if (RightAlignCaption)
               style += "text-align: right;";

            output.WriteAttribute("Style", style);
            output.Write(HtmlTextWriter.TagRightChar);
            output.Write(Caption);
            output.WriteEndTag("td");

            //write second column
            output.WriteBeginTag("td");

            style = "width: " + CaptionPadding.ToString() + ";";

            output.WriteAttribute("Style", style);

            output.Write(HtmlTextWriter.TagRightChar);
            output.WriteEndTag("td");

            //write third column
            output.WriteBeginTag("td");
            output.Write(HtmlTextWriter.TagRightChar);

            base.Render(output);   // this renders the TextBox

            output.WriteEndTag("td");

            //finish writing row
            output.WriteEndTag("tr");

            //finish writing table
            output.WriteEndTag("table");
        }
    }
}
```

You can see that the control inherits the *TextBox* class. In addition, you can see the custom properties that are defined for this class. Notice that, unlike the properties for the *DaysUntil* control, each of these properties includes a *Description* attribute that describes the property. In addition, these properties simply get and set the values of class variables. Because that simplifies the code for the properties, we prefer this technique over saving the properties in view state.

Also, notice that the *CaptionWidth* and *CaptionPadding* properties and the class variables that hold their values are described are declared with the *Unit* data type. It represents a length measurement that

consists of both the length and the unit of measure. By default, the unit of measure is a pixel, which is usually what you want.

The overridden *Render* method uses an HTML table that contains a single row with three columns to align the caption and text box. To accomplish this, this method uses the *Write* method of the *HtmlTextWriter* object to write the tags and attributes necessary to render the table as HTML. Notice that, within the *Table* element, the *CellPadding* and *CellSpacing* attributes specify *0* pixels of padding and spacing.

With the *Td* element for the first column, the *Style* attribute specifies the width of the captioned using the *CaptionWidth* property. Notice that the *ToString* method is used to convert the length and unit of measure of this property to a string value. If the caption width is set to *10* pixels, for example, the result is *10px*. In addition, if the *RightAlignCaption* property is set to *True*, the *Style* attribute also specifies that the next text should be right-aligned. The start tag for the first *Td* element is followed by its content, which is the value of the *Caption* property. The content is then followed by an end tag for the *Td* element.

Next, the *Render* method renders the *Td* element for the second column. This *Td* element creates the space between the caption and the text box. To do that, it includes a *Style* attribute that specifies the value of the *CaptionPadding* property as the width of the element.

Finally, the *Render* method renders the *Td* element for the third column. This column contains the text box. To render the text box, the shaded statement executes the *Render* method of the base class. Since this class is based on the *TextBox* class, this method renders the *TextBox* control.

Creating a composite control

A composite control is a control that includes two or more child controls. A composite control typically inherits the *CompositControl* class, as shown here:

public class DateDDL : CompositeControl

A property and methods of the *CompositeControl* class used with child controls

Property	**Description**
Controls	Accesses the collection of controls contained by the web control
Method	**Description**
CreateChildControls()	Called automatically by ASP.NET when a page is posted back or before it's rendered

A composite control typically inherits the CompositeControl class, as shown above. This class inherits the WebControl class and adds functionality to that class. In particular, it ensures that all the child controls defined by the composite control are created before they're accessed.

The *CompositeControl* class also implements the *INamingContainer* interface. This interface defines properties, methods, and events just like classes. Unlike classes, however, an interface doesn't include the details of how the properties, methods, and events are implemented. That's left to the class that use them.

The *INamingContainer* interface is unusual in that it doesn't define any properties, methods, or events. Instead, it simply identifies a control as a container, which ensures that its child controls will be unique. That way, you can include two or more instances of the control on the same page.

To work with the collection of child controls in a composite control, you use the *Controls* property of the *CompositeControl* class. To add a control to the collection, for example, you can use the *Add* method of the control collection object, and to remove all of the controls from the collection, you use the *Clear* method. This is illustrated by the example below.

Common methods of control collection objects

Property	Description
Add(control)	Adds the specified child control
Remove(control)	Removes the specified child control
Clear()	Removes all child controls

```
protected override void CreateChildControls()
    {
      Controls.Clear();

      Label lbl = new Label();

      Lbl.Text = labelText;

      Controls.Add(lbl);

      TextBox txt = new TextBox();

      Controls.Add(txt);
    }
```

This method starts by cleaning the controls collection. Then, it inherits an instance of the *Label* class and an instance of the *TextBox* class and adds them both to the controls colleection.

Notice that these statments are codede within a method named *CreatChildControls*. As you can see by the method declaration, this method overrides the *CreateChildControls* method of the base class. ASP.NET calls this method automatically to create the child controls of the base control, if it has any. To create a composite control, then, you need to override this method to create the child controls you want to include in the composite control.

The last example, shown below, shows how to add literal text to the controls collection. In this example, the literal control contains a single space *( )*.

```
Controls.Add(new LiteralControl(" "));
```

Note that when you use a literal control, you can't apply a style to the text it contains. If you need to do that, you'll want to use a *Label* control instead.

The design of the *DateDDL* control

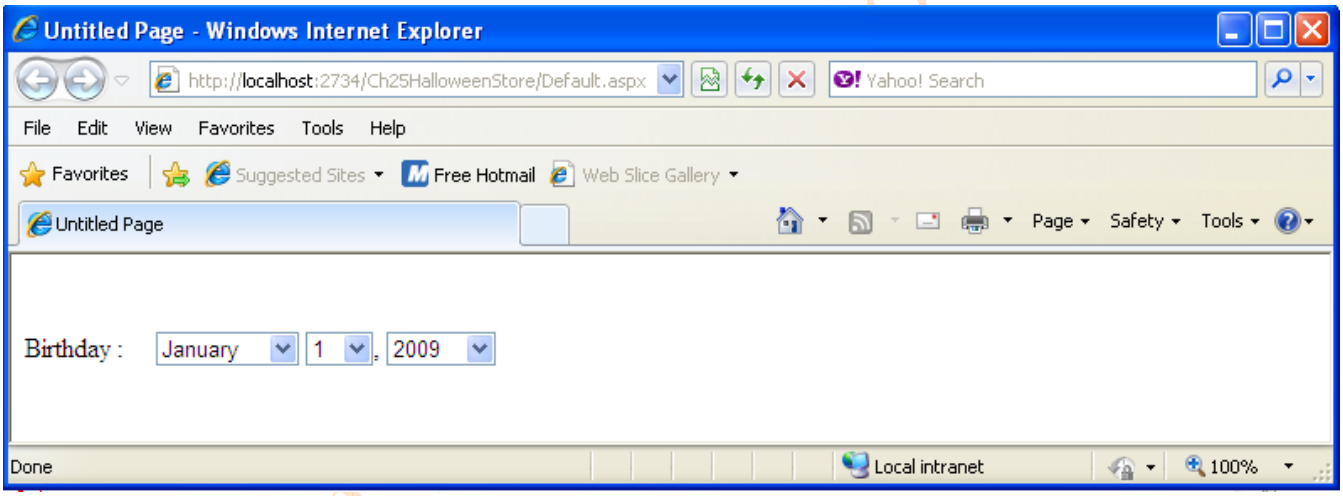
This control includes three drop-down lists that let the user select a month, day, and year. When the control is displayed, the Month drop-down list is populated with the names of the months, the Day drop-down list is populated with the number 1 through 31, and the year drop-down list is populated with the 10 years that begin with the current year. This control also includes a caption just like the CaptionedBox control.

In addition to the *Caption, CaptionWidth CaptionPadding*, and *RightAlignCaption* properties that are used to implement the caption, the *DateDDL* control has four properties that store date information.

The last property, *AutoPostBack*, lets you specify whether the page that contains the control should be posted back to the server when the user selects a different month, day, or year.

The *DateDDL* control also raised an event named *DateChanged* whenever the user selects a different month, day, or year. The *Month, Day*, and *Year* properties for this control are coded so that this event is aised only once each time the page is posted.

A web page that uses a *DateDDL* control



Properties of the *DateDDL* control

Property	Description
Caption	The text that's displayed to the left of the drop-down lists
CaptionWidth	The width of the control's caption area
CaptionPadding	The width of the area between the caption and the drop-down lists
RightAlignCaption	Determines whether the caption is right-alighned. The default value is *False*.
Date	The date selected by the user

Month	An integer representing the month (1 to 12)
Day	An ingereg representing the day of the month (1 to 31)
Year	An integer represnting the year
AutoPostBack	Determines whether the page should be posted when the user selects a different month, day, or year. The default is *False*.

Events raised by the *DateDDL* control

Event	Description
DateChanged	Raised when the user selects a different month, day, or year

The *aspx* code for the *DateDDL* control

```
<cc1:DateDDL ID="DateDDL1" CaptionPadding="20px" CaptionWidth="120px"
        OnDateChanged="DateDDL1_DateChanged"
       Caption="Birthday :" runat="server">
</cc1:DateDDL>
```

The C# programming code for the *DateDDL* control

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Text;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace HalloweenWebControlLibrary
{
  [DefaultProperty("Text")]
  [ToolboxData("<{0}:DateDDL runat=server></{0}:DateDDL>")]
  public class DateDDL : CompositeControl
  {
    private Label lblCaption = new Label();
    private Label lblMessage = new Label();
    private Label lblCaptionPadding = new Label();
    private DropDownList ddlMonth = new DropDownList();
    private DropDownList ddlDay = new DropDownList();
    private DropDownList ddlYear = new DropDownList();

    private string caption;
```

```
private Unit captionWidth;
private Unit captionPadding;
private bool rightAlignCaption;

private int month;
private int day;
private int year;
private bool autoPostBack;
private bool dateChangedEventRaised = false;

public event EventHandler DateChanged;

public DateDDL()
{
   ddlDay.SelectedIndexChanged += new EventHandler(ddlDay_SelectedIndexChanged);
   ddlMonth.SelectedIndexChanged += new EventHandler(ddlMonth_SelectedIndexChanged);
   ddlYear.SelectedIndexChanged += new EventHandler(ddlYear_SelectedIndexChanged);
}

[Category("Appearance")]
[Description("The caption displayed to the left of the text box.")]
public string Caption
{
   get { return caption; }
   set { caption = value; }
}

[Category("Appearance")]
[Description("The width of the caption.")]
public Unit CaptionWidth
{
   get { return captionWidth; }
   set { captionWidth = value; }
}

[Category("Appearance")]
[Description("The space between the caption and the text box.")]
public Unit CaptionPadding
{
   get { return captionPadding; }
   set { captionPadding = value; }
}

[Category("Appearance")]
[Description("Determines if the caption is right-aligned.")]
public bool RightAlignCaption
```

```
{
   get { return rightAlignCaption; }
   set { rightAlignCaption = value; }
}

public DateTime Date
{
   get { return Convert.ToDateTime(Month + "/" + Day + "/" + Year); }
}

[Category("Misc"), Description("Gets or set the month.")]
public int Month
{
   get
   {
      month = ddlMonth.SelectedIndex + 1;

      return month;
   }
   set
   {
      month = value;

      ddlMonth.SelectedIndex = month - 1;

      if (!dateChangedEventRaised)
      {
         DateChanged(this, new EventArgs());

         dateChangedEventRaised = true;
      }
   }
}

[Category("Misc"), Description("Gets or sets the day.")]
public int Day
{
   get
   {
      day = ddlDay.SelectedIndex + 1;

      return day;
   }
   set
   {
      day = value;
```

```
            ddlDay.SelectedIndex = day - 1;

            if (!dateChangedEventRaised)
            {
                DateChanged(this, new EventArgs());

                dateChangedEventRaised = true;
            }
        }
    }

    [Category("Misc"), Description("Gets or sets the year.")]
    public int Year
    {
        get
        {
            year = Convert.ToInt32(ddlYear.SelectedValue);

            return year;
        }
        set
        {
            year = value;

            ddlYear.SelectedValue = year.ToString();

            if (!dateChangedEventRaised)
            {
                DateChanged(this, new EventArgs());

                dateChangedEventRaised = true;
            }
        }
    }

    [Category("Misc")]
    [Description("Determines whether a postback occurs "
        + "when the selection is changed.")]
    [DefaultValue(false)]
    public bool AutoPostBack
    {
        get { return autoPostBack; }
        set { autoPostBack = value; }
    }
```

```
protected override void Render(HtmlTextWriter output)
{
   base.Render(output);
}

protected override void CreateChildControls()
{
   this.MakeCaption();
   this.MakeCaptionPadding();
   this.MakeMonthList();
   this.MakeDayList();
   this.MakeYearList();

   Controls.Clear();
   Controls.Add(lblCaption);
   Controls.Add(lblMessage);
   Controls.Add(lblCaptionPadding);
   Controls.Add(ddlMonth);
   Controls.Add(new LiteralControl(" "));
   Controls.Add(ddlDay);
   Controls.Add(new LiteralControl(", "));
   Controls.Add(ddlYear);
}

private void MakeCaption()
{
   lblCaption.Text = Caption;
   lblCaption.Width = CaptionWidth;

   if (RightAlignCaption)
      lblCaption.Style["text-align"] = "right";
}

private void MakeCaptionPadding()
{
   lblCaptionPadding.Width = CaptionPadding;
}

private void MakeMonthList()
{
   ddlMonth.Width = Unit.Pixel(85);
   ddlMonth.Items.Clear();
   ddlMonth.Items.Add("January");
   ddlMonth.Items.Add("February");
   ddlMonth.Items.Add("March");
   ddlMonth.Items.Add("April");
```

```
        ddlMonth.Items.Add("May");
        ddlMonth.Items.Add("June");
        ddlMonth.Items.Add("July");
        ddlMonth.Items.Add("August");
        ddlMonth.Items.Add("September");
        ddlMonth.Items.Add("October");
        ddlMonth.Items.Add("November");
        ddlMonth.Items.Add("December");
        ddlMonth.SelectedIndex = Month - 1;
        ddlMonth.AutoPostBack = AutoPostBack;
    }

    private void MakeDayList()
    {
        ddlDay.Width = Unit.Pixel(40);
        ddlDay.Items.Clear();

        for (int i = 1; i <= 31; i++)
            ddlDay.Items.Add(i.ToString());

        ddlDay.SelectedValue = Day.ToString();

        ddlDay.AutoPostBack = AutoPostBack;
    }

    private void MakeYearList()
    {
        ddlYear.Width = Unit.Pixel(65);
        ddlYear.Items.Clear();

        for (int i = DateTime.Today.Year; i <= DateTime.Today.Year + 10; i++)
            ddlYear.Items.Add(i.ToString());

        ddlYear.SelectedValue = Year.ToString();
        ddlYear.AutoPostBack = AutoPostBack;
    }

    private void ddlMonth_SelectedIndexChanged(object sender, EventArgs e)
    {
        this.Month = ddlMonth.SelectedIndex + 1;
        this.AdjustDay();
    }

    private void ddlDay_SelectedIndexChanged(object sender, EventArgs e)
    {
        this.Day = Convert.ToInt32(ddlDay.SelectedValue);
```

```
        this.AdjustDay();
    }

    private void ddlYear_SelectedIndexChanged(object sender, EventArgs e)
    {
        this.Year = Convert.ToInt32(ddlYear.SelectedValue);
        this.AdjustDay();
    }

    private void AdjustDay()
    {
        switch (Month)
        {
            case 4:
            case 6:
            case 9:
            case 11:
                if (Day == 31) Day = 30;

                break;
            case 2:
                if (Year % 4 == 0)
                    if (Day > 29) Day = 29;
                else
                    if (Day > 28) Day = 28;

                break;
        }
    }

    protected void ddlDay_DateChanged(object sender, EventArgs e)
    {
        WebControl control = (WebControl)sender;

        lblMessage.Text = "The DateChanged event was raised by the "
            + control.ID + " control.";
    }
  }
}
```


The first five child controls that are included in this control are declared at the beginning of the class. Then, this control includes a constructor that wires the *SelectedIndexChanged* event to each of the three drop-down lists. That way, this class can respond to any of these events.

Second, the *Date* property listing is a read-only property that consists of just a get accessor. As you can see, this accessor returns a formatted date that's constructed using the *Month, Day*, and *Year* properties. Because you can't set the value of this property, it doesn't include any designer attributes.

You can see the *Render* method for the this control:

```
protected override void Render(HtmlTextWriter output)
{
   base.Render(output);
}
```

It simply calls the *Render* method of the base class to render the control and the child controls it contains.

Next on this listing, you can see the overridden *CreateChildControls* method:

```
protected override void CreateChildControls()
{
   this.MakeCaption();
   this.MakeCaptionPadding();
   this.MakeMonthList();
   this.MakeDayList();
   this.MakeYearList();

   Controls.Clear();
   Controls.Add(lblCaption);
   Controls.Add(lblMessage);
   Controls.Add(lblCaptionPadding);
   Controls.Add(ddlMonth);
   Controls.Add(new LiteralControl(" "));
   Controls.Add(ddlDay);
   Controls.Add(new LiteralControl(", "));
   Controls.Add(ddlYear);
}
```

This method is responsible for creating the two *Label* controls and the month, day, anf year drop-down lists and adding them to the controls collection. To do that, it starts by executing the private *MakeCaption* and *MakeCaptionPadding* methods to initialize the caption and the spaceing between the caption and drop-down lists. Then, it executes the private *MakeMonthList*, *MakeDayList*, and *MakeYearList* method to initialize the drop-down lists. Next, it clears the controls collection and adds the controls. Notice that literal controls that contains a space are added between the drop-down lists so that these lists don't appear right next to each other when they're displayed.

When the user selects a different month, day, or year from a drop-down list, the event handler for the *SelectedIndexChanged* event of that control executed. The code for these event handledrs here:

```
private void ddlMonth_SelectedIndexChanged(object sender, EventArgs e)
{
   this.Month = ddlMonth.SelectedIndex + 1;
```

```
    this.AdjustDay();
}

private void ddlDay_SelectedIndexChanged(object sender, EventArgs e)
{
    this.Day = Convert.ToInt32(ddlDay.SelectedValue);
    this.AdjustDay();
}

private void ddlYear_SelectedIndexChanged(object sender, EventArgs e)
{
    this.Year = Convert.ToInt32(ddlYear.SelectedValue);
    this.AdjustDay();
}
```

They simply set the *Month, Day*, and *Year* properties to the selection made by the user, and then call the *AdiustDay* method:

```
private void AdjustDay()
{
    switch (Month)
    {
        case 4:
        case 6:
        case 9:
        case 11:
            if (Day == 31) Day = 30;

            break;
        case 2:
            if (Year % 4 == 0)
                if (Day > 29) Day = 29;
            else
                if (Day > 28) Day = 28;

            break;
    }
}
```

At the *Set* routines for these properties, you can see that they in turn raise the *DateChanged* event. Notice that a *Boolean* variable is used to keep track of whether the *DateChanged* event has already been raised to make sure it isn't raised more than once.

The *AdjustDay* method that's called by the *SelectedIndexChanged* event handles prevents the date from being set to an invalid date. If the user selects *April 31*, for example, this method afjust the date to *April 30* since *April* only has *30* days. Or, if the user selects *February 29* and it isn't a leap year, this method adjust the date to *February 28* since only has *29* days in a leap year.