Let's Code Already!
I'm going to show you how to develop three Web controls in the rest of this article. I will go into as much detail as I can considering that this is an article and not a book. Some of the things I will cover will be more general than others and I'm hoping that you get intrigued enough to read further on the subject from the books I've listed in the sidebar called "*Good Books for Learning How to Develop Custom ASP.NET Controls*."

The first custom Web control you'll learn to develop is a button that will have the ability to pop up a confirmation dialog when it is clicked. This will give the user the ability to cancel whatever processing would have taken place when the button was clicked. I'll show you how to develop this Web control as an inherited control that I'll call the ConfirmationButton.

Next you'll learn to develop a second custom Web control I'll call the FormField control. This control was in my article on Declarative Programming. This control is a combination of a label, a textbox, and a button that you can arrange in a variety of ways. The idea behind this control is that it serves as the basis for data entry on a Web page. Because of the heavy use this control may receive on a single Web page, I'll develop this as a rendered control.

The third and last custom Web control that you'll lean to develop is an EmailContact control. You no doubt have seen many Web sites that provide the user with a "Contact Us" page. This page usually includes a small form that allows the visitor of the site to send an e-mail to the site's Webmaster. You can see similar forms in today's most popular type of Web site among the develop community—the blog. You may see this kind of form when you leave feedback for a blog posting. I'll show you how to encapsulate everything necessary to build this kind of form into a custom Web control. You'll even see how to build the actual e-mailing functionality directly into the control. I'll show you how to develop this control as a composite control and will contain within it both the FormField rendered control and the ConfirmationButton inherited control.

You will enable all these controls with certain functionality during the development steps in this article. At the end of this article, I will list additional functionality that the final versions of the controls include in the downloadable code.

The WebControl class inherits form the Control class and adds some styling properties to it.

So let's get to work...

The ConfirmationButton Control
The first custom Web control will also be the simplest. The ConfirmationButton control will be a simple class that starts by inheriting from the built-in Button Web control that ships with Visual Studio.

In VB.NET:

```
Public Class ConfirmationButton
   Inherits Button
End Class
```

In C#:

```
public class ConfirmationButton : Button
{
}
```

The parent Button class of this control resides in the System.Web.UI.WebControls namespace, which you obtain by referencing the *System.Web.dll* in your project. If this class is compiled as-is, it will duplicate the functionality of the Button class that .NET provides. In fact, you can compile

this class, add it to the toolbox for a Web Form, then drop it on a Web Form and it will be no different than dropping a Button control from the Web Form toolbox.

Properties
You will make only two simple changes to your new class. The first is the addition of a property. I want to describe this process in detail, however, because this is the way you'll add most properties in all three controls you'll build in this article. I say *most properties* because when you build the composite control, you'll code "mapped" properties a bit differently, but I'll get to that later. You want to add a *ConfirmationMessage* property to your ConfirmationButton class, and a value other than an empty string will indicate that you want a confirmation dialog box to pop up when a user clicks a button. Here is the code for your new property.

In VB.NET:

```
Public Property ConfirmationMessage() As String
   Get
      If CType( _
         VieState("ConfirmationMessage"), _
         Object) Is Nothing Then
         Return String.Empty
      End If
      Return CType( _
         ViewState("ConfirmationMessage"), String)
   End Get
   Set(ByVal Value As String)
      ViewState("ConfirmationMessage") = Value
   End Set
End Property
```

In C#:

```
public string ConfirmationMessage
{
   get
   {
      if((object)
         ViewState["ConfirmationMessage"] ==
         null) return String.Empty;
      return (string)
         ViewState["ConfirmationMessage"];
   }
   set
   {
      ViewState["ConfirmationMessage"] = value;
   }
}
```

Look closely and you'll notice that this property is not much different than properties you write every day in your classes. The typical procedure for a property is to expose a member variable of some kind. You've no doubt done this time and time again. The *ConfirmationMessage* property works similarly with the difference being that you aren't using a privately declared member variable. Instead, you're using a ViewState variable as the internal storage for the *ConfirmationMessage* property; so let me talk a little more about this.

The *get* accessor in the property returns the value of the internal variable. In the property, this is

stored in *ViewState["ConfirmationMessage"]*, and the set accessor sets the *ViewState* variable to whatever you set the property to. *ViewState* is a variable of type StateBag. Like the famous Session variables and Cookie variables that you have used in Web applications, it is a dictionary that contains one or more values accessed by a key. In this case, I've standardized the naming conventions so the key is equivalent to the name of the property. Keep this in mind because you're going to see it a lot more through the rest of this article. The contents of the *ViewState* variable are stored in the __*ViewState* hidden field on your Web page in an encrypted format and are used to save and retrieve values that you need to persist in between page postbacks. The rest of the code around the 'return' statement in the get accessor checks to see if the specific *ViewState* value exists in the *ViewState* array. Because *ViewState* is a dictionary type (actually, StateBag is), checking for an entry in it that does not exist does not raise an exception; instead, it adds an empty object. The *if* statement checks the *ViewState* value for *Nothing* (or *null*). If this is the case, then it will return whatever you want the default value to be. The check for *Nothing* (or *null*) is done with an object-cast of the *ViewState* variable. The reason for this is due to the nature of the StateBag object, where an attempted access of a non-existing value automatically adds a *null* value.

Standard Attributes
You'll also have a few additions for this property that will be "standard operating procedure" for all properties you write in custom Web controls. These consist of certain attributes with which you'll decorate your property. Some properties will receive more attribute decorations than others, but all will receive at least four attributes: *Category*, *Description*, *Bindable*, and *DefaultValue*. All are found in the System.ComponentModel namespace.

In VB.NET:

```
< _
   Category("Behavior"), _
   Description("Gets or sets the message to be " & _
   "displayed in a confirmation dialog. The " & _
   "dialog will present itself if this property " & _
   "is not empty."), _
   DefaultValue(""), _
   Bindable(True) _
> _
```

In C#:

```
[
Category("Behavior"),
Description(""),
DefaultValue(""),
Bindable(true)
]
```

Category Attribute
This attribute specifies the category that your property will be listed under when you view it in the property browser. Indulge me in a comical tangent: Make sure you do not accidentally misspell the category name. I can't tell you how many times I have misspelled the word "Appearance" when using it as a Category. Instead, I've typed "Apperance" and guess what the property browser shows? That's right—two categories, "Appearance" and "Apperance."

Description Attribute
This attribute determines the description of the property that is displayed at the bottom of the property browser. It provides simple instructions for programmers using your Web control.

DefaultValue Attribute

Contrary to what the name of this attribute makes you assume, it does not determine what gets stored in the property by default. As you saw when I described the Property statement, you're setting the default value for the ViewState variable in your *get* accessor. This attribute lets Visual Studio know what the default value of the property will be for two main reasons. The first is that it lets the property browser know whether to display the value of the property in bold letters or not. If you haven't noticed before, when you drop a control on a Web Form and refer to the property browser, all the property values are displayed in normal font until you change their values, after which they are displayed in bold letters. This lets you know that the entered value is not the default value for the property. The second reason is that only properties with non-default values appear in the ASPX tags that define your Web control on a Web Form.

Bindable Attribute
This attribute determines if the property will participate in data binding functionality. The data source for the property can be defined in an expression. This topic is beyond the scope of this article, but if you want further information, look at the Data category of a typical Web control in the property browser and click on the *DataBindings* value, then click the Help button for a full explanation.

This is the technique for creating properties that you'll see in many of the properties going forward in all your Web controls, so this will be the only place where I will get into a lot of detail for Web control properties.

Rendering the Control
Now that you've added your custom property to the Web control, you need to tell the control what to do with it. Remember that your goal is to check the content of this property and if it is anything but an empty string, you want a confirmation dialog to pop up and let the user confirm or cancel the postback that the button initiates. When you create custom Web controls, it's important to understand what you want the eventual HTML result to be. Remember, I stated earlier that Web controls are essentially code generators that render standard HTML. The ConfirmationButton control, like its Button base class, renders a standard HTML input tag that looks something like this.

```
<input type="submit"... />
```

The rest of the tag is not important at this time. What is important is what you would need to add to such an HTML tag if you wanted a confirmation pop up—and that is some Jscript code in its *onclick* event, such as:

```
<input type="submit" onclick="if(!confirm(
   'Are you sure?') return false; "... />
```

This Jscript code will simply pop up a confirmation message asking "Are you sure?" Then, if the user clicks Cancel, the code returns a value of *false*. This has the effect of canceling the submission of the form originally intended by pressing the button, and effectively canceling the postback. Now you need to get this code into your Web control. You do this by overriding the *AddAttributestoRender* method. This method adds tag attributes to the rendered HTML tag your control will eventually produce.

In VB.NET:

```
Protected Overrides Sub AddAttributesToRender( _
   ByVal writer As HtmlTextWriter)
```

```
        If Me.ConfirmationMessage <> _
           String.Empty Then
           writer.AddAttribute( _
              HtmlTextWriterAttribute. _
              Onclick, "if(!confirm('" & _
              Me.ConfirmationMessage.Replace( _
              "'", "\'") + "')) return false;")
        End If
        MyBase.AddAttributesToRender(writer)
    End Sub
```

In C#:

```
    protected override void AddAttributesToRender(
        HtmlTextWriter writer)
        {
        if (this.ConfirmationMessage != String.Empty)
        {
            writer.AddAttribute(
                HtmlTextWriterAttribute.Onclick,
                "if(!confirm('" +
                this.ConfirmationMessage.Replace("'", "\'") +
                "')) return false;");
        }
        base.AddAttributesToRender(writer);
    }
```

The argument of type HtmlTextWriter is used to write to the HTML rendering engine. You will see much code like this later. One of the methods of this type allows you to add an attribute to the HTML tag that will later get rendered. As you can see, you're adding an *onclick* attribute and providing the appropriate Jscript to perform the task. The code uses the *Replace* statement to replace any single ticks with ones preceded by a backslash. This is standard Jscript syntax.

I've twice mentioned *the tag that is rendered*. The Button class code has decided this for you so it is not discussed in detail here. You're taking it on faith that Microsoft's Button control renders an *<input>* tag.

That's it for the ConfirmationButton class. If you compile it you can add it to your toolbox and use it on a Web Form. When you drag it onto a Web Form and examine the property browser, you'll notice the *ConfirmationMessage* property under the Behavior category. Set this property to a value of "Are you sure?", run the Web Form and press the button, and you should see something like Figure 1.

If you click "Yes," you should see the page postback as usual, but clicking "No" will simply make the popup window disappear and you'll notice that the postback will be cancelled. *(You can tell if the postback occurs by looking at the bottom status bar of the browser.)*

Figure 1: Confirmation Message: The figure shows the message that appears when a user presses the ConfirmationButton.

With very little work, you could have your button decide whether or not to use the *ConfirmationMessage* property based on another Boolean property called *Confirm*; instead of depending on the existence of the message text.

The assembly, or DLL file, that this class compiled into can be distributed to any Web applications and used freely. This is where the extreme reusability factor

of custom Web controls comes in. In fact, once this control exists in your toolbox, dragging it onto a Web Form will automatically add the reference to its DLL to your ASP.NET application.

Before I wrap this up, I want to touch on one other thing that will apply to all Web controls. I'm referring to the control's tag prefix. All of you have used the Visual Studio Web controls already, and I'm sure you've noticed the ASPX code that gets generated when you drag something like a textbox onto a Web Form.

```
<asp:TextBox id="TextBox1" runat="server" ... />
```

The "asp:" in this case is called the tag prefix. By default, .NET uses "cc1" for your new control but you're going to change it to something else. Since my company name is InfoTek Consulting Group, I tend to use "icg" as the tag prefix for all of my Web controls. I'll place the definition for a tag prefix in the *AssemblyInfo* file. The declaration looks like this.

```
[assembly: TagPrefix("InfoTek.Web.UI.WebControls",
   "icg")]
```

For VB.NET projects, replace the square brackets (*[]*)with angle brackets (< and >). The first argument specifies the custom control's namespace and the second argument specifies the tag prefix you want to use. By adding this attribute to the *AssemblyInfo* file, your ConfirmationButton control takes this form.

```
<icg:ConfirmationButton id="ConfirmationButton1"
runat="server" ... />
```

Congratulations. You've just developed a very simple, yet useful inherited Web control. You can use the technique I showed you here on any type control and it is only limited by your own creativity. For practice, try creating another inherited control but make it extend a Textbox control to convert its contents to upper or lower case when it loses focus. Here's a hint: the Jscript *onblur* event gets fired when the control loses focus. I'll leave the rest to you.