ASP.NET Technical Articles

**Working with Client-Side Script**

August 2004

**Summary**: While ASP.NET performs most of its processing on the server, some actions are better served by client-side processing. Scott Mitchell shows how your ASP.NET pages and controls can add client-side code. (27 printed pages)

[Download the source code for this article](http://download.microsoft.com/download/A/8/C/A8C44419-05F8-450C-9FB4-F1AED34527F1/MSDNClientSideScript.msi).

**Contents**

[Introduction](http://msdn.microsoft.com/en-us/library/aa479302.aspx#clientsidescript_topic1)  
[Creating a Base Class as the Foundation for Adding Client-Side Script](http://msdn.microsoft.com/en-us/library/aa479302.aspx#clientsidescript_topic2)  
[Adding Client-Side Script from the Code-Behind Class](http://msdn.microsoft.com/en-us/library/aa479302.aspx#clientsidescript_topic3)  
[Executing Client-Side Code in Response to User Action](http://msdn.microsoft.com/en-us/library/aa479302.aspx#clientsidescript_topic4)  
[Implementing Common Client-Side Functionality](http://msdn.microsoft.com/en-us/library/aa479302.aspx#clientsidescript_topic5)  
[Conclusion](http://msdn.microsoft.com/en-us/library/aa479302.aspx#clientsidescript_topic6)  
[Related Books](http://msdn.microsoft.com/en-us/library/aa479302.aspx#clientsidescript_topic7)

**Introduction**

When working with dynamic Web-based scripting technologies, like classic ASP or PHP, developers must have a keen understanding of the logical, temporal, and physical separation between the client and the server. For a user's action to trigger the execution of server-side code, for example, a developer working with classic ASP must explicitly cause the user's browser to make a request back to the Web server. Creating such interactions can easily consume much development time and lead to unreadable code.

Microsoft ASP.NET helped ease the burden of tying user events to execution of specific server-side code through the use of Web Forms, which blur the lines between client and server. Using ASP.NET and a minimum of effort, a developer can quickly create a Web page that has a variety of interactive user interface elements—buttons, drop-down lists, and so on—which, based on the end user's actions, can cause selective server-side code to run. For example, with ASP.NET to add a drop-down list that performs some action whenever the selected drop-down list item is changed, all you need to do is add a **DropDownList** Web control, set its **AutoPostBack** property to True, and create a **SelectedIndexChanged** event handler for the drop-down list. Accomplishing this same functionality with classic ASP would require you to write oodles of intricate HTML, client-side JavaScript, and server-side script code; with ASP.NET, the necessary script code and server-side event model are provided for you.

While Web Forms in ASP.NET greatly simplify running server-side script when client-side actions are performed, such power, if misused, can lead to unacceptable performance. While Web Forms hide the complexities involved, each time server-side code needs to be executed, the end user's browser must make a request back to the Web server by resubmitting the form. When submitting the form, all form fields—textboxes, drop-down lists, check boxes, and so on—must have their values sent back as well. Additionally, the page's view state is sent back to the Web server. In total, each time the Web page is posted back, potentially several kilobytes of data will need to be sent back to the Web server. Frequent postbacks, then, can quickly lead to an unusable Web application, especially for those users still stuck on dial-up. The need for frequent postbacks can be reduced by pushing functionality to the client.

**Note A**SP.NET Web Forms emit a hidden form field titled **VIEWSTATE**, which contains a base-64 encoded representation of the changed state of the Web controls in the Web Form. Depending on the Web controls present, the view state can range anywhere from a few dozen bytes, to tens of kilobytes. To learn more about view state check out my article [Understanding ASP.NET View State](http://msdn.microsoft.com/en-us/library/ms972976.aspx).

With classic ASP, adding data-driven, custom client-side script was simple, albeit not very readable. To display a popup window in classic ASP that loads a URL based on some ID field, for instance, you would type in the appropriate client-side script, using the <%=id%> syntax to insert the value of the ID field. ASP.NET allows you to create such data-driven client-side script with some assorted methods in the **Page** class.

This article examines techniques for adding client-side script to your ASP.NET Web pages. Client-side script is, as its name implies, script code that runs in the visitor's browser. We'll see how to accomplish common client-side tasks, such as displaying alerts, confirm boxes, and popup windows. (One of the main uses of client-side script—form field validation—is a bit of a moot topic with ASP.NET, since the validator Web controls provide client-side form validation out of the box.) The focus of this article will be on the server-side classes, methods, and techniques for injecting client-side script; we will not be examining the actual client-side script in detail, as this information is covered in numerous other articles and sites around the Web.

**Creating a Base Class as the Foundation for Adding Client-Side Script**

One of the major differences between classic ASP and ASP.NET is the programming model of each technology. ASP pages are atomic, procedural scripts interpreted on each page visit. ASP.NET, however, is a fully object-oriented programming technology. All ASP.NET Web pages are classes with properties, methods, and events. All Web pages are derived, either directly or indirectly, from the **Page** class in the **System.Web.UI** namespace; the **Page** class contains the base functionality of an ASP.NET Web page.

One of the concepts of object-oriented programming is that of *inheritance*. Inheritance allows you to create a new class that extends the functionality of another class. (If class *B* inherits class *A*, it is said to extend class *A*; class *A* is said to be the base class.) When using the code-behind model for creating ASP.NET Web pages, you can quite clearly see that the code-behind class inherits the **Page** class:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl04_code');" \o "Copy Code)

Public Class WebForm1

Inherits System.Web.UI.Page

...

End Class

By having your code-behind class inherit the **Page** class, it automatically receives the functionality inherent in the **Page** class, such as the **Request**, **Response**, **Session**, and **ViewState** objects, as well as common events, like **Init**, **Load**, **Render**, as so on. As we'll see in this article, if you have a need for some common functionality to be available for all ASP.NET Web pages, one approach is to create a class that derives from the **Page** class and has additional methods and properties to accomplish these desired enhancements. Then, to have an ASP.NET Web page utilize these enhancements, all we need to do is update the **Inherits** statement in the page's code-behind class to use the class that extends the **Page** class.

In this article we'll create a class, called **ClientSidePage**, that derives from the **Page** class and provides extra methods to help with performing common client-side tasks. By having a code-behind class inherit **ClientSidePage**, rather than **Page**, adding script code will be as simple as calling a method and passing in a few parameters. Specifically, this class will contain methods for:

* Displaying a modal, client-side dialog box.
* Setting the focus to a specific form field on page load.
* Using a modal confirm dialog box to determine if a user wants to postback the form or not.
* Displaying popup windows.

Before we delve into the **ClientSidePage** class, let's first examine the pertinent methods in the **Page** class for injecting client-side script into a Web page. Once we've covered these **Page** methods, we'll jump into extending their functionality with the **ClientSidePage** class, and see how to tie everything together and use the extended class in an ASP.NET Web page.

**Adding Client-Side Script from the Code-Behind Class**

All ASP.NET Web pages must be derived directly or indirectly from the **Page** class in the **System.Web.UI** namespace. The **Page** class contains the base set of methods, properties, and events required for a functioning Web page. Among the class's many methods are a few methods designed for injecting client-side script into the rendered HTML. These methods are called from the code-behind class and can therefore be used to emit data-driven client-side script. The pertinent **Page** class methods for emitting client-side script follow.

The base class is derived from the **System.Web.UI.Page** class, so you can access the **Page** class's public methods by calling them directly from your code-behind class.

**Note** To access the **Page** class's methods, you can either type in the method name directly, or utilize IntelliSense in Microsoft Visual Studio .NET by entering **MyBase.** (for Microsoft Visual Basic .NET), **this.** (for C#), or **Page.** (for either C# or Visual Basic .NET). If you are using Visual Basic .NET as your programming language of choice, be sure to configure Visual Studio .NET to not hide advanced members, or you won't see these client-side script methods. (To show advanced members, go to **Tools | Options | Text Editor | Basic** and uncheck **Hide advanced members**.)

**RegisterClientScriptBlock(*key*, *script*)**

The **RegisterClientScriptBlock** method adds a block of client-side script after the Web Form's rendered **<form>** element, before any Web controls contained within the Web Form. The *key* input parameter allows you to specify a unique key associated with this script block, whereas the *script* parameter includes the complete script code to emit. (This *script* parameter should include the actual **<script>** element, along with the client-side JavaScript or Microsoft VBScript.)

When emitting client-side script through the code-behind class of an ASP.NET Web page, typically the value of the *key* parameter isn't of paramount importance. Simply choose a descriptive key value. The *key* parameter is more pertinent when injecting client-side script code through a custom, compiled server control. There may be instances where a compiled control requires that there be a set of client-side functions. Multiple instances of the server control on one page might be able to share these common client-side script functions, so these functions need only be emitted once for the entire page, and not once per control instance. For example, the validation controls utilize client-side code to enhance the user experience. This client-side code must be present if there are any validation controls on the page, but if there are multiple validation controls, all can use this single set of shared functions.

By giving a script block a key, a control developer building a control that utilizes a set of common client-side functions can check to see if the required set of common functions has already been added by another instance of the control on the page. If so, it need not re-add the common script. To check if a script block has been added with the same key, use the **IsClientScriptBlockRegistered(key)** method, which will return a Boolean value indicating whether or not a script block with the same key has been registered. Realize that you can add a script block without first checking if it's registered. If you attempt to add a script block with a key that's already registered, the added script block will be ignored and the original script block will remain assigned to that key.

**Note** The **IsClientScriptBlockRegistered** method is particularly useful in two situations. First, it comes in handy when you're adding similar, but unique script blocks, and you need to insure that each script block is given a unique key. The code we'll examine later on in this article illustrates the utility of the "is registered" method. A second use is when building a control that needs some common script, especially if the script is not trivially generated. By using the **IsClientScriptBlockRegistered** method, you can ensure that the script common to all instances of the server control on the page is generated only once per page load, rather than once per control instance on the page.

The **RegisterClientScriptBlock** method is useful for adding client-side script that does not rely on any of the form fields present within the Web Form. A common use of this method is to display a client-side alert box. For example, imagine that you had a Web page with some **TextBox** Web controls and a Save Button. The **TextBox** controls might have particular values from a database. Imagine that this page allowed the user to modify these values and commit their changes by clicking the Save button. When clicking Save, the Web page would be posted back, and the Button's **Click** event would fire. You could create a server-side event handler for this event that updates the database. To let the user know that their changes had been saved, you might want to display an alert box that says, "Your changes have been saved." This could be accomplished by adding the following line of code to the Button's **Click** event handler:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl05_code');" \o "Copy Code)

RegisterClientScriptBlock("showSaveMessage", \_

"<script language=""JavaScript""> \_

alert('Your changes have been saved.'); \_

</script>")

The above will add the specified script content within the page's **<form>**, but before the content within the form. When the page is rendered in the user's browser they will see a client-side alert box displayed upon page load, as shown in Figure 1.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl06_code');" \o "Copy Code)

<form method="post" ...>

<script language="JavaScript">

alert('Your changes have been saved.');

</script>

...

</form>



**Figure 1. Result of the client-side JavaScript**

**Note** One potentially undesirable side effect of the above example is that the alert box will be displayed right after the browser received the **<form>** tag. The browser will suspend rendering of the Web page until the user clicks the alert box's OK button. This means that the user will see a white browser page until they click OK. If you want to have the page displayed completely before displaying the alert box, you can have the JavaScript inserted at the end of the **<form>** element using the **RegisterStartupScript** method, which we'll discuss next.

**RegisterStartupScript(*key*, *script*)**

The **RegisterStartupScript** method is quite similar to the **RegisterClientScriptBlock** method. The main difference is the location where the client-side script is emitted. Recall that with the **RegisterClientScriptBlock** the script is emitted after the start of the **<form>** element, but before the form's contents. **RegisterStartupScript**, on the other hand, adds the specified script at the *end* of the form, after all form fields. Use **RegisterStartupScript** to place client-side script that interacts with the rendered HTML elements. (Later we'll look at an example of setting the focus to a form field upon page load; to accomplish this you'll use the **RegisterStartupScript** method.)

Like **RegisterClientScriptBlock**, the script blocks added by **RegisterStartupScript** need a unique key value. Again, this key value is primarily used by custom control developers. Not surprisingly, there is an **IsStartupScriptRegistered(key)** method as well, which returns a Boolean value indicating if a script block with the specified *key* has already been registered or not.

**Note** For more information on using **RegisterStartupScript** and **RegisterClientScriptBlock** in creating custom, compiled server controls, read an earlier article of mine: [Injecting Client-Side Script from an ASP.NET Server Control](http://msdn.microsoft.com/en-us/library/aa478975.aspx).

**RegisterArrayDeclaration(*arrayName*, *arrayValue*)**

If you need to create a client-side JavaScript **Array** object with some set values, use this method to add a value to a specific array. For example, when using validation controls in an ASP.NET Web page, an **Array** object (**Page\_Validators**) is built that contains references to the set of validation controls on the page. When the form is submitted, this array is enumerated to check if the various validation controls are valid or not.

To add the values 1, 2, and 3 to a client-side **Array** object named **FavoriteNumbers**, you'd use the following server-side code:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl09_code');" \o "Copy Code)

RegisterArrayDeclaration("FavoriteNumbers", "1")

RegisterArrayDeclaration("FavoriteNumbers", "2")

RegisterArrayDeclaration("FavoriteNumbers", "3")

This code would emit the following client-side script:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl10_code');" \o "Copy Code)

<script language="javascript">

<!--

var FavoriteNumbers = new Array(1, 2, 3);

// -->

</script>

Notice that each array value passed in must be a string; however, the client-side script rendered sets the values of the **Array** object as the contents of the string. That is, if you wanted to create an **Array** with the string values "Scott" and "Jisun", you'd use:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl11_code');" \o "Copy Code)

RegisterArrayDeclaration("FavoriteFolks", "'Scott'")

RegisterArrayDeclaration("FavoriteFolks ", "'Jisun'")

Notice that the second input parameters are strings that contain 'Scott' and 'Jisun'—text delimited by a single apostrophe. This would render the following client-side script:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl12_code');" \o "Copy Code)

<script language="javascript">

<!--

var FavoriteFolks = new Array('Scott', 'Jisun');

// -->

</script>

**RegisterHiddenField(*hiddenFieldName*, *hiddenFieldValue*)**

In classic ASP there was often the need to pass around various bits of information from one page to another. A common way of accomplishing this was using hidden form fields. (A hidden form field is a form field that is not displayed, but whose value is sent on the form's submission. The syntax for creating a hidden form field is <input type="hidden" name="name" value="value" />.) The need for passing information around by custom hidden form fields in ASP.NET is greatly reduced since the state of the controls in the page is automatically persisted. If, however, you find that you need to create a custom hidden form field, you can do so through the **RegisterHiddenField** method.

The **RegisterHiddenField** method accepts two input parameters: the name of the hidden field and the value. For example, to create a hidden form field with the name foo and the value bar, use the following code:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl13_code');" \o "Copy Code)

RegisterHiddenField("foo", "bar")

This would add a hidden form field within the page's **<form>** element, like so:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl14_code');" \o "Copy Code)

<form name="\_ctl0" method="post" action="test.aspx" id="\_ctl0">

<input type="hidden" name="foo" value="bar" />

...

</form>

**Understanding How Client-Side Elements Are Rendered**

The **Page** class contains two **internal** methods responsible for rendering client-side script registered in the methods discussed above: **OnFormRender** and **OnFormPostRender**. (A method marked **internal** can only be called by other classes in the same assembly. Therefore, you cannot call the **Page**'s **internal** methods from the code-behind classes in your ASP.NET Web application.) Both of these methods are called in the **HtmlForm** class's **RenderChildren** method. The **HtmlForm** class, in the **System.Web.UI.HtmlControls** namespace, represents a Web Form; that is, the server-side form in an ASP.NET Web page—<form runat="server">...</form>—is loaded as an instance of the **HtmlForm** class during the page's instantiation stage.

Since the client-side script registered by the **Page** class's assorted methods is rendered in the **OnFormRender** and **OnFormPostRender** methods, and since these methods are only called by the **HtmlForm** class, the client-side script you programmatically add by these methods is only rendered if the Web page contains a Web Form. That is, any script elements you programmatically add through any of the discussed methods above will only be emitted in the page's final markup if the ASP.NET Web page contains a server-side form (a <form runat="server">...</form>).

A Web Form on an ASP.NET Web page is rendered by first adding the starting **<form>** element. Following that, the Web Form's **RenderChildren** method is called, which contains three lines of code:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl15_code');" \o "Copy Code)

Page.OnFormRender(...)

MyBase.RenderChildren(...)

Page.OnFormPostRender(...)

The call to the **Page** class's **OnFormRender** method adds the following markup:

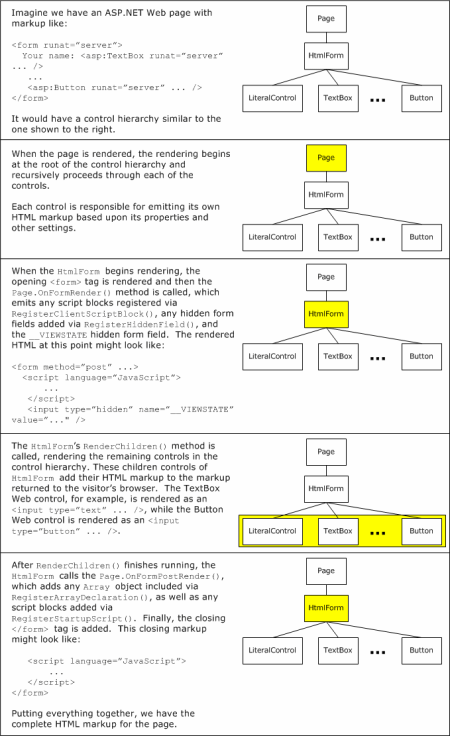
* Any hidden form fields added by calls to **RegisterHiddenField**.
* The base-64 encoded view state in a hidden form field named **\_\_VIEWSTATE**.
* Any script blocks added by calls to **RegisterClientScriptBlock**.

The second line of code in the Web Form's **RenderChildren** method calls the base class's **RenderChildren** method, which renders the contents within the Web Form. After rendering all of the form's contents, a call is made to the **Page** class's **OnFormPostRender** method, which adds the following client-side content:

* Any **Array** objects added by the **RegisterArrayDeclaration** method.
* Any script blocks added by calls to **RegisterStartupScript**.

Finally, after the Web Form's **RenderChildren** method completes, the closing form tag (**</form>**) is rendered. Figure 2 illustrates this rendering process graphically.

**Note** Figure 2 assumes you are somewhat familiar with the ASP.NET page life cycle. If you are interested in learning more about the page life cycle, consider reading [Understanding ASP.NET View State](http://msdn.microsoft.com/en-us/library/ms972976.aspx), focusing on the section titled, "The ASP.NET Page Life Cycle."

[](http://msdn.microsoft.com/en-us/library/Aa479302.clientside_fig02(en-us,MSDN.10).gif)

**Figure 2. Page rendering in ASP.NET**

**Examining the Rendering Order of Script Blocks**

Upon first glance at the register methods of the **Page** class, it might seem that the order with which the registered elements are rendered in the Web page correspond to the order with which they were added in the code. That is, imagine that you had the following two lines of code in your ASP.NET Web page's code-behind class:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl19_code');" \o "Copy Code)

RegisterClientScriptBlock("showSaveMessage", \_

"<script language=""JavaScript"">var name='" & \_

someDataDrivenValue & "'; </script>")

RegisterClientScriptBlock("showSaveMessage", \_

"<script language=""JavaScript"">alert('Hello, ' + name + '!');

</script>")

You wouldn't be too surprised when you found that the page rendered the following client-side script blocks (assuming the value of *someDataDriveValue* was Sam):

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl20_code');" \o "Copy Code)

<script language="JavaScript">var name='Sam';</script>

<script language="JavaScript">alert('Hello, ' + name + '!');</script>

The user visiting this page would see an alert box saying "Hello, Sam!"

Based on this test, you might then assume that it was always the case that the order the script blocks were emitted in the HTML page was the order they were specified in the server-side code. However, this would be an incorrect assumption, and one that could cause your pages to break. For example, imagine that the script blocks added above were emitted in the HTML page in the reverse order. Then you'd have:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl21_code');" \o "Copy Code)

<script language="JavaScript">alert('Hello, ' + name + '!');</script>

<script language="JavaScript">var name='Sam';</script>

This would display an alert box reading "Hello, !", since the variable **name** has not yet been assigned a value. Clearly, there are times when the order with which script blocks are emitted is very important.

The register methods of the **Page** class—**RegisterClientScriptBlock**, **RegisterStartupScript**, **RegisterArrayDeclaration**, and **RegisterHiddenFields**—all write the supplied script content to an internal **HybridDictionary**. A **HybridDictionary** is a data structure found in the **System.Collections.Specialized** namespace, and is designed for storing items in a dictionary where the number of items in the dictionary is not known. For a small collection of items, a **ListDictionary** is the most efficient data structure, but for larger dictionaries, a **Hashtable** is more efficient. A **HybridDictionary** splits the difference—it starts by storing items using a **ListDictionary**. Once the **ListDictionary** has its ninth item added, the **HybridDictionary** switches from using a **ListDictionary** to using a **Hashtable**.

While this approach is ideal for performance, it can wreck havoc if you are using several script blocks where the order of the script blocks is important. That's because while a **ListDictionary** maintains the order with which the elements were added, a **Hashtable** does not. So, if you add eight or fewer items to any one of the particular register methods, the items will be emitted in the order with which they were added. However, if you add a ninth item, the order that the script is emitted will be seemingly random.

**Note** The **ListDictionary** stores its elements using a linked list, while the **Hashtable** stores its elements in an array, where the contents are ordered by the hashed value of a sting key. A thorough discussion on linked lists and hashtables is far beyond the scope of this article. For more information, including an analysis of their performance, consider reading [An Extensive Examination of Data Structures](http://msdn.microsoft.com/en-us/library/aa289148.aspx), specifically [Part 2](http://msdn.microsoft.com/en-us/library/aa289149.aspx) and [Part 4](http://msdn.microsoft.com/en-us/library/aa289151.aspx).

If you plan on having cases where there may be more than eight client-side elements added using a particular register method, and the order with which the elements appear matters, you might want to take a look at [Peter Blum](http://www.peterblum.com/Home.aspx)'s free [RegisterScripts](http://www.peterblum.com/RegScripts/RegScriptsHome.aspx) library. RegisterScripts provides greater control over the order with which the client-side elements are emitted, and also provides the option to not have to manually add the **<script>** tags, which you have to add when including client-side scripts with the **RegisterClientScriptBlock** or **RegisterStartupScript** methods.

**Executing Client-Side Code in Response to User Action**

The **Page** class's register methods are ideal for injecting client-side code that runs when the page loads, but in many situations we want to run code in response to an end user's action. For example, we might want to display a confirm dialog box when a user clicks a button, or call a particular client-side JavaScript function when a drop-down list's selected item is changed.

HTML elements have a variety of client-side events that you can tap into and have client-side code execute when the event fires. The required markup simply goes in the HTML element's tag as an attribute. For example, to display an alert box when a button is clicked you can do:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl27_code');" \o "Copy Code)

<input type="button"

value="Click me to see an alert box!"

onclick="alert('Here it is!');" />

To run client-side code when a client-side event transpires, you can add the appropriate attribute to an HTML element. For a Web control, you can programmatically add a client-side attribute using the **Attributes** collection. For example, imagine that you had a **TextBox** Web control that you wanted to be highlighted yellow whenever the rendered textbox gains focus. To accomplish this you'd want the **TextBox** Web control's rendered HTML to look something like the following:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl28_code');" \o "Copy Code)

<input type="text"

onfocus="this.style.backgroundColor='yellow';"

onblur="this.style.backgroundColor='white';" />

To achieve this markup, we can programmatically set the **TextBox** Web control's **onfocus** and **onblur** client-side attributes by the **Attributes** collection, like so:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl29_code');" \o "Copy Code)

TextBoxControl.Attributes("onfocus") = "this.style.backgroundColor='yellow';"

TextBoxControl.Attributes("onblur") = "this.style.backgroundColor='white';"

This technique of tying client-side code with client-side events is commonly used to provide a rich, interactive user experience. Later on in this article we'll see how to employ this technique to display confirm dialog boxes based on a user's actions.

**Implementing Common Client-Side Functionality**

Now that we've looked at the ASP.NET methods involved in dynamically adding client-side script to a Web page, let's turn our attention to applying this knowledge. The remainder of this article focuses on common client-side tasks, such as displaying alert boxes, confirm boxes, popup windows, and so on. Specifically we'll create a class that contains a set of methods that can be used in an ASP.NET project to quickly and easily provide such functionality.

The Visual Basic .NET code we will be examining throughout the remainder of this article is available in this article's code download.

**Displaying an Alert Box**

A common client-side requirement is to display an alert box. An alert box is a client-side, modal dialog box typically used to provide some important bit of information to the end user. An example of an alert box can be seen in Figure 1. An alert box is displayed through the client-side JavaScript **alert** function, which accepts a single parameter—the message to display. Displaying an alert box is fairly simple and straightforward; in fact, an example was shown earlier in the article.

In order to make it as easy as possible for a page developer to display an alert box, let's create a class called **ClientSidePage** that contains a method called **DisplayAlert(message)**. This class will inherit the **Page** class. A page developer that wants to utilize these client-side helper methods, then, will need to have their code-behind class inherit this **ClientSidePage** class rather than the default **Page** class. The following code shows this **ClientSidePage** class with its first method, **DisplayAlert**.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl30_code');" \o "Copy Code)

Public Class ClientSidePage

Inherits System.Web.UI.Page

Public Sub DisplayAlert(ByVal message As String)

RegisterClientScriptBlock(Guid.NewGuid().ToString(), \_

"<script language=""JavaScript"">" & GetAlertScript(message) & "</script>")

End Sub

Public Function GetAlertScript(ByVal message As String) As String

Return "alert('" & message.Replace("'", "\'") & "');"

End Function

End Class

Notice that this class is derived from the **System.Web.UI.Page** class. The **DisplayAlert** method simply uses the **RegisterClientScriptBlock** method to display the supplied *message* in an alert box. Since this method may be called multiple times by a single page, each call uses a GUID (*G*lobally *U*nique *Id*entifier) for its key. The string being passed to the **alert** function is delimited by apostrophes, any apostrophes in *message* must be escaped (JavaScript escapes apostrophes as \').

To use this code in your ASP.NET Web application, you will need to add a new class to your ASP.NET application. In Visual Studio .NET, right click on the ASP.NET Web application project name in the Solution Explorer and choose to add a new class. Then, cut and paste the above code into the class. Next, in your ASP.NET Web pages where you want to utilize this code, you'll need to modify the code-behind class so that it inherits from the **ClientSidePage** class rather than from **Page**. The following code shows a sample code-behind class derived from **ClientSidePage** and that uses the **DisplayAlert** method.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl31_code');" \o "Copy Code)

Public Class WebForm1

Inherits ClientSidePage

Private Sub Page\_Load(ByVal sender As System.Object, \_

ByVal e As System.EventArgs) \_

Handles MyBase.Load

DisplayAlert("Hello, World!")

End Sub

...

End Class

Note that the **ClientSidePage** class not only has the **DisplayAlert** method, which generates a full client-side **<script>** element, but also has a **GetAlertScript** method, which returns just the client-side script, sans the **<script>** tag. This second method can be used in cases where you want to display an alert based on some client-side event. For example, if you want to have an alert displayed anytime a particular text box received focus, you could add the following code to your server-side code-behind class:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl32_code');" \o "Copy Code)

TextBoxControlID.Attributes("onfocus") = GetAlertScript(message)

**Setting Focus to a Form Field on Page Load**

Have you ever noticed that when visiting Google the focus is automatically set to the search text box? This one little "feature" makes searching Google all the more faster—upon visiting Google you don't have to take the second or two to move your mouse and click on the text box. Rather, you can simply start typing away upon page load. Setting the focus to a form field, be it a text box, radio button, check box, or drop-down list, requires just a few lines of client-side JavaScript code. Let's add a method to the **ClientSidePage** class that will automatically add focus to a specified Web control on page load. This method will need to emit client-side script that looks like:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl33_code');" \o "Copy Code)

<script language="JavaScript">

function CSP\_focus(id) {

var o = document.getElementById(id);

if (o != null)

o.focus();

}

</script>

... Form fields ...

<input type="..." id="id of element to focus" ... />

... Form fields ...

<script language="JavaScript">

CSP\_focus(id of element to focus);

</script>

The client-side function **CSP\_focus** accepts a string parameter, the **ID** of the form field to set to focus, and retrieves the HTML element from the DOM. The retrieved element's **focus()** function is then called. At the bottom of the Web page, after all of the form fields have been specified, we need to call the **CSP\_focus** method passing in the **ID** of the form field where we want the focus set.

The following method, **GiveFocus(Control)**, uses the **RegisterClientScriptBlock** and **RegisterStartupScript** methods to generate the needed client-side script.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl34_code');" \o "Copy Code)

Public Sub GiveFocus(ByVal c As Control)

RegisterClientScriptBlock("CSP-focus-function", \_

"<script language=""JavaScript"">" & vbCrLf & \_

"function CSP\_focus(id) {" & \_

" var o = document.getElementById(id); " & \_

"if (o != null) o.focus(); " & \_

"}" & vbCrLf & \_

"</script>")

RegisterStartupScript("CSP-focus", \_

"<script language=""JavaScript"">CSP\_focus('" & \_

c.ClientID & "');</script>")

End Sub

To use the **GiveFocus** method from an ASP.NET Web page whose code-behind class inherits **ClientSidePage**, simply call **GiveFocus** in the **Page\_Load** event handler and pass in the Web control that should have its focus set on page load. For example, to set the focus to the **TextBox** Web control **TextBoxControl**, use the following **Page\_Load** event handler:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl35_code');" \o "Copy Code)

Private Sub Page\_Load(ByVal sender As System.Object, \_

ByVal e As System.EventArgs) \_

Handles MyBase.Load

GiveFocus(TextBoxControl)

End Sub

**Opening a Popup Window**

While popup windows have gotten a bad rap on the Internet as a nefarious tool for advertisers, popup windows are used in many Web applications to a good end. For example, you might want a page that displays a list of database items in a **DataGrid**, with a link to edit each particular item. Rather than using the **DataGrid**'s in-line editing capabilities, you might want to have a popup window opened when the user opts to edit a **DataGrid**, where the popup window contains a list of text boxes with the editable fields of the **DataGrid**. (One reason you might want to do this is because there may be a very large number of editable fields, but you only want to show the most pertinent fields in the **DataGrid**, thereby eliminating the possibility of using the **DataGrid**'s built-in editing features.)

To display a popup window, use the JavaScript function **window.open()**, which takes a number of optional input parameters, the three germane ones being:

* The URL to load in the popup window.
* A string name for the popup window.
* The features for the popup window, such as its height and width, whether or not the window is resizable, and so on.

A thorough discussion of the **window.open()** function is beyond the scope of this article; to learn more refer to the [technical documentation](http://msdn.microsoft.com/library/default.asp?url=/workshop/author/dhtml/reference/methods/open_0.asp).

Like the method to display the alert box, the **ClientSidePage** class contains two methods for displaying a popup window—one that renders a self-contained **<script>** block that displays the window, and one that returns just the JavaScript script itself. In addition to the methods to open a popup window, there is also a set of methods to close the current window. (It may be the case that you want to programmatically close the popup window based on some client-side or server-side event.)

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl37_code');" \o "Copy Code)

Public Sub DisplayPopup(ByVal url As String, ByVal options As String)

RegisterStartupScript(Guid.NewGuid().ToString(), \_

"<script language=""JavaScript"">" & \_

GetPopupScript(url, options) & \_

"</script>")

End Sub

Public Function GetPopupScript(ByVal url As String, \_

ByVal options As String) As String

Return "var w = window.open(""" & \_

url & """, null, """ & options & """);"

End Function

Public Sub CloseWindow(Optional ByVal refreshParent As Boolean = False)

RegisterClientScriptBlock("CSP-close-popup", \_

"<script language=""JavaScript"">" & \_

GetCloseWindowScript(refreshParent) & "</script>")

End Sub

Public Function GetCloseWindowScript(Optional \_

ByVal refreshParent As Boolean = False) As String

Dim script As String

If refreshParent Then

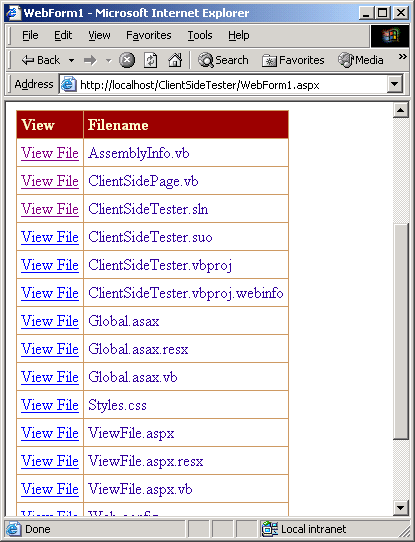
script &= "window.opener.location.reload();"

End If

Return "self.close();"

End Function

An example of this code in action can be seen in the code download for this article. There you'll find a sample Web page that has a **DataGrid** listing the files in the same directory as the ASP.NET Web page. This **DataGrid** has two columns: a TemplateColumn that displays a hyperlink that, when clicked, opens a popup window showing the contents of the selected file; and the name of the file (see Figure 3).



**Figure 3. DataGrid with popup window**

The **DataGrid**'s markup utilizes the **GetPopupScript** method, as shown below:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl39_code');" \o "Copy Code)

<asp:DataGrid id="dgFiles" runat="server" ...>

<Columns>

<asp:TemplateColumn HeaderText="View">

<ItemTemplate>

<a href='javascript:<%#

GetPopupScript("ViewFile.aspx?FileName=" &

DataBinder.Eval(Container.DataItem, "Name"),

"scrollbars=yes,resizable=yes,width=500,height=400")

%>'>

View File</a>

</ItemTemplate>

</asp:TemplateColumn>

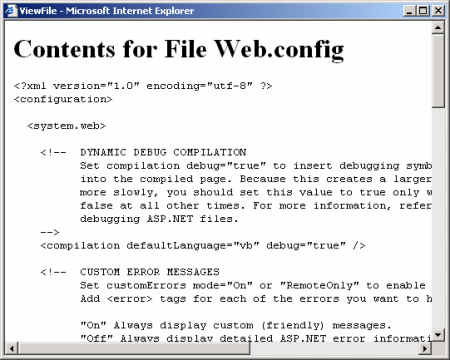
<asp:BoundColumn DataField="Name"

HeaderText="Filename"></asp:BoundColumn>

</Columns>

</asp:DataGrid>

The ASP.NET Web page ViewFile.aspx opens the file whose name is specified in the querystring, and displays its contents (see Figure 4).



**Figure 4. Displaying the contents of Web.config in a popup window**

**Note** Popup windows are best suited for intranet applications only, because a number of Internet users utilize some sort of popup blocking software, such as [Google Toolbar](http://toolbar.google.com/). In fact, with the Microsoft Windows XP Service Pack 2, Microsoft Internet Explorer will be configured to block popups by default. However, popups will still work when a user visits a site in the Trusted Sites or Local Intranet zones. For more information on the popup blocking features for Internet Explorer in the Windows XP Service Pack 2, be sure to read [Changes to Functionality in Microsoft Windows XP Service Pack 2](http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2brows.mspx).

**Confirming Before Postback**

Earlier in this article, we looked at how to display a client-side alert box, which is a modal dialog box with an OK button. JavaScript offers a more interactive flavor of the alert box called a confirm dialog box. The confirm dialog box is displayed using the **confirm(message)** function and has the effect of displaying a dialog box with the text specified by the *message* input parameter along with OK and Cancel buttons. The **confirm(message)** function returns true if the user clicks OK and false if they click Cancel.

Typically confirm dialog boxes are used to ensure that the user wants to continue before submitting a form. When an HTML element is clicked to submit a form (such as a submit button), if that HTML element fires a client-side event handler that returns false, the form submission is canceled. Commonly the confirm dialog box is used in a Web page as follows:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl43_code');" \o "Copy Code)

<form ...>

<input type="submit" value="Click Me to Submit the Form"

onclick="return confirm('Are you sure you want

to submit this form?');" />

</form>

When the user clicks the "Click Me to Submit the Form" button, they'll see a confirm dialog box that asks them if they are sure they want to submit the form (see Figure 5). If the user clicks OK, **confirm()** will return true and the form will be submitted. If, however, they click Cancel, **confirm()** will return false and the form's submission will be canceled.



**Figure 5. Results of the Confirm JavaScript**

Imagine you had a **DataGrid** with a column of buttons labeled "Delete." Upon clicking this button, the form will postback and the selected record will be deleted. In this instance, you might want to double-check that the user really wanted to delete this record. Here would be a great place to use a client-side confirm dialog box. You could prompt the user with a dialog box stating something like: "This will permanently delete the record. Are you sure you want to continue?" If the user clicks OK, the form will postback and the record will be deleted; if they click Cancel, the form will not be posted back, and hence the record will not be deleted.

To add the client-side JavaScript necessary to display a confirm dialog box on button's click, simply use the **Attributes** collection to add a client-side **onclick** event handler. Specifically, set the **onclick** event handler code to: return confirm(message);. In order to provide such functionality for a **DataGrid**'s **ButtonColumn**, you'll need to programmatically reference the **Button** or **LinkButton** control in either the **DataGrid**'s **ItemCreated** or **ItemDataBound** event handlers and set the **onclick** attribute there. For more information, see <http://aspnet.4guysfromrolla.com/articles/090402-1.aspx>.

**Confirmation with AutoPostBack DropDownLists**

While confirm dialog boxes are typically used when a button is clicked, they can also be used when a drop-down list is changed. For example, you might have a Web page that automatically posts back when a particular **DropDownList** Web control is changed. (The **DropDownList** Web control has an **AutoPostBack** property that, if set to True, causes the form to postback whenever the **DropDownList**'s selected item is changed.)

Intuitively you might think adding a confirm dialog box for a **DropDownList** is identical to adding such a dialog box for a **Button** Web control. That is, simply set the D**ropDownList**'s client-side **onchange** attribute to something like: return confirm(...);. using:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl46_code');" \o "Copy Code)

DropDownListID.Attributes("onchange") = "return confirm(...);"

Unfortunately, this won't work as desired because an **AutoPostBack** **DropDownList**'s **onchange** attribute will be set to a bit of JavaScript that causes a postback, namely a call to the client-side **\_\_doPostBack** function. When setting the **onchange** attribute programmatically yourself, the end result is that the rendered client-side **onchange** event handler has both your code and the call to **\_\_doPostBack**:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl47_code');" \o "Copy Code)

<select onchange="return confirm(...);\_\_doPostBack(...);">

...

</select>

Noting this, what we really want to happen is have the **\_\_doPostBack** function called if confirm returns true, because then the page will be posted back. We can accomplish this by setting the **onchange** event by the **Attributes** collection to: if (confirm(...)), which will render the following markup, which is what we are after:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl48_code');" \o "Copy Code)

<select onchange="if (confirm(...)) \_\_doPostBack(...);">

...

</select>

At first glance this will seem to have the desired effect. If a user selects a different item from the drop-down list, a confirm box appears. If the user clicks OK, the form will postback; if the user clicks Cancel, the form's postback is halted. The problem, though, is that the drop-down list retains the item the user selected to initiate the drop-down list's **onchange** event. For example, imagine the drop-down list loads with item *x* being selected, and then the user chooses item *y*. This will trigger the drop-down list's client-side **onchange** event, which will display the confirm dialog box. Now, imagine that the user hits Cancel—the drop-down list will still be selected on item *y*. What we want is to have the selection reverted back to item *x*.

To accomplish this we need to do two things:

1. Write a JavaScript function that "remembers" the selected drop-down list item.
2. In the drop-down list's client-side **onchange** event, if the user clicks Cancel, you need to revert the drop-down list back to the "remembered" value.

Step 1 entails creating a global script variable for the drop-down list and a function that runs when the page loads that will record the drop-down list's value. Step 2 requires changing the client-side **onchange** attribute for the drop-down list to look like:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl49_code');" \o "Copy Code)

if (!confirm(...)) resetDDLIndex(); else \_\_doPostBack();

Where **resetDDLIndex** is a JavaScript function that reverts the drop-down list's selected value back to the "remembered" value. The client-side script for this needs to look like the following:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl50_code');" \o "Copy Code)

<select id="ddlID"

onchange="if (!confirm(...)) resetDDLIndex(); else \_\_doPostBack(...);">

...

</select>

<script language="JavaScript">

var savedDDLID = document.getElementById("ddlID").value;

function resetDDLIndex() {

document.getElementById("ddlID").value = savedDDLID;

}

</script>

This necessary script can be easily generated by creating a helper method in the **ClientSidePage** class.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl51_code');" \o "Copy Code)

Public Sub ConfirmOnChange(ByVal ddl As DropDownList, ByVal message As String)

'Register the script block

If Not IsStartupScriptRegistered("CSP-ddl-onchange") Then

RegisterStartupScript("CSP-ddl-onchange", \_

"<script language=""JavaScript"">" & \_

"var CSP\_savedDDLID = " & \_

document.getElementById('" & \_

ddl.ClientID & "').value;" & vbCrLf & \_

"function resetDDLIndex() {" & vbCrLf & \_

" document.getElementById('" & " & \_

" ddl.ClientID & "').value = CSP\_savedDDLID;" &

vbCrLf & \_

"}" & vbCrLf & \_

"</script>")

End If

ddl.Attributes("onchange") = \_

"if (!confirm('" & message.Replace("'", "\'") & \_

"')) resetDDLIndex(); else "

End Sub

To use this, simply call this method for each **AutoPostBack** **DropDownList** on the Web page that you want to display a confirm dialog box for when its selected item changes.

**Confirming When Exiting Without Saving**

In most every data-driven Web application I've created there's always been some page where users can edit particular bits of information from the database. A very simple example might be a page with a series of **TextBox** and **DropDownList** Web controls, with the database data populated within these controls. The user can make any suitable changes and click the Save button to persist their changes to the database.

When I create these pages, I usually end the page with two **Button** Web controls: a Save button and a Cancel button. The Save button saves any changes back to the database, while the Cancel button exits the page without persisting any changes. While two buttons may seem like a perfect design, sometimes users accidentally click the Cancel button when they meant to click the Save button, thereby losing any changes they made to the data. To prevent this from happening, you can use a confirm box on the Cancel button that only appears if any of the textboxes or drop-down lists on the Web page have been changed. That is, if the user makes any changes to the data and then clicks Cancel, a confirm box will prompt them to see if they are sure they want to exist without saving. (If the user just clicks Cancel without changing any data, no such confirm box is shown.)

This user experience can be accomplished by a bit of client-side JavaScript. Basically, it entails a JavaScript global variable, **isDirty**, that is initially false but is set to true whenever any of the form fields' **onchange** events fire. There's also a JavaScript function that displays a confirm dialog box if **isDirty** is true. The Cancel button's **onclick** client-side event handler is wired up to return the result from this JavaScript function. The following HTML illustrates this concept:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl52_code');" \o "Copy Code)

<script language="JavaScript">

var isDirty= false;

function checkForChange(msg) {

if (isDirty) return confirm(msg); else return true;

}

</script>

Name: <input type="text" onchange="isDirty = true;" />

<input type="submit" name="btnSave" value="Save" id="btnSave" />

<input type="submit" name="btnCancel" value="Cancel"

id="btnCancel"

onclick="return checkForChange('You have made changes to the data

since last saving. If you continue, you will lose these

changes.');" />

This script can be easily generated by moving its generation to the **ClientSidePage** class. Specifically, we can create the following three methods:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl53_code');" \o "Copy Code)

Protected Sub RegisterOnchangeScript()

If Not IsClientScriptBlockRegistered("CSP-onchange-function") Then

RegisterClientScriptBlock("CSP-onchange-function", \_

"<script language=""JavaScript"">" & \_

"var isDirty= false;" & vbCrLf & \_

"function CSP\_checkForChange(msg) {" & vbCrLf & \_

" if (isDirty) return confirm(msg); " & \_

"else return true;" & vbCrLf & \_

"}" & vbCrLf & \_

"</script>")

End If

End Sub

Public Sub MonitorChanges(ByVal c As WebControl)

RegisterOnchangeScript()

If TypeOf c Is CheckBox Or TypeOf c Is CheckBoxList \_

Or TypeOf c Is RadioButtonList Then

c.Attributes("onclick") = "isDirty = true;"

Else

c.Attributes("onchange") = "isDirty = true;"

End If

End Sub

Public Sub ConfirmOnExit(ByVal c As WebControl, ByVal message As String)

RegisterOnchangeScript()

c.Attributes("onclick") = \_

"return CSP\_checkForChange('" & message.Replace("'", "\'") &

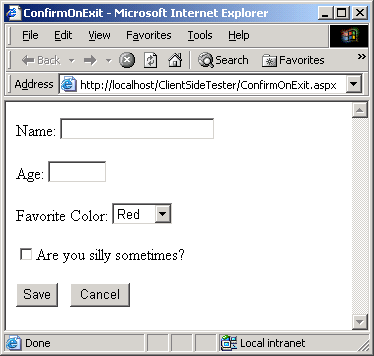
"');"

End Sub

To create a Web page that exhibits this behavior, we simply need to have its server-side code-behind class derive from **ClientSidePage** and in the **Page\_Load** event handler have a call to **MonitorChanges** for each Web control that needs a client-side **onchange** event and a call to **ConfirmOnExit** for each **Button**, **LinkButton**, and **ImageButton** that, when clicked, should display a warning if the user has made changes and is exiting the page.

**Note** Notice that the **MonitorChanges** method uses the **onclick** client-side event instead of **onchange** for the **CheckBox**, **CheckBoxList**, and **RadioButtonList** Web controls. This is because these controls wrap a **<span>** tag or **<table>** around the check box or series of check boxes or radio buttons. In my tests with Internet Explorer, I found that the **onchange** event, when applied to the **<span>** or **<table>**, was not picked up when a check box or radio button was clicked, but the **onclick** event was raised.

Figure 6 shows an example ASP.NET Web page with two **TextBox** Web controls, a **DropDownList** Web control, and a **CheckBox** Web control. As the **Page\_Load** event handler below shows, all of these Web controls are being monitored for changes. The Cancel button, **btnCancel**, is configured so that if it's clicked after changes have been made, a confirm dialog box will be displayed.



**Figure 6. Dialog with confirmation**

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl55_code');" \o "Copy Code)

Public Class ConfirmOnExit

Inherits ClientSidePage

Private Sub Page\_Load(ByVal sender As System.Object, \_

ByVal e As System.EventArgs) \_

Handles MyBase.Load

'Specify what controls to check for changes

MonitorChanges(name)

MonitorChanges(age)

MonitorChanges(favColor)

MonitorChanges(chkSilly)

ConfirmOnExit(btnCancel, \_

"You have made changes to the data since last saving." & \_

" If you continue, you will lose these changes.")

End Sub

...

End Class

**Note** The client-side **onchange** event does not work in older versions of Netscape. Also, Internet Explorer 5.0 has had [some reported problems](http://support.microsoft.com/default.aspx?scid=kb;EN-US;238391) with the **onchange** event (which were fixed in Internet Explorer 5.01, SP 1).

Furthermore, this approach won't work as desired with **DropDownList** Web controls with **AutoPostBack** set to True, as the postback will reset the value of **isDirty**. There are a couple of workarounds for this problem, such as using a hidden form field that indicates whether or not the postedback form data is dirty to begin with or not. I leave implementing this as an exercise for the reader.

**Creating a Client-Side MessageBox Control**

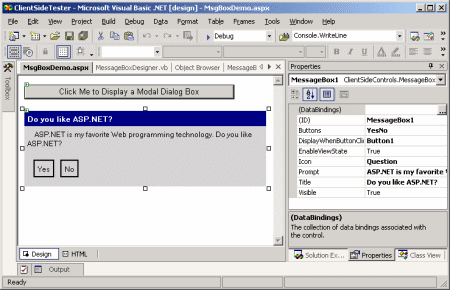
While confirm dialog boxes are a great way to prevent accidental clicks and to potentially reduce the number of postbacks to the Web server, there are certain scenarios where you might want to display a confirm dialog box and be able to determine on the server-side whether or not they clicked OK or Cancel. (Recall that with the confirm dialog box, if the user clicks Cancel, the form is not posted back.) Furthermore, the alert and confirm boxes in JavaScript are rather limited in their appearance. Fortunately client-side VBScript offers a richer message box experience through its **MsgBox** function.

In a past project, I had a need for a client-side, modal message box that would cause a postback no matter what button was clicked. In response, I built a custom compiled ASP.NET server control that met these requirements. In addition, the client-side message box uses VBScript's **MsgBox** function to provide a richer message box experience.

**Note** VBScript only works as a client-side scripting language in Microsoft's Internet Explorer browser. To account for this, my server control only uses VBScript if the visiting browser is Internet Explorer. Non-Internet Explorer browsers are sent JavaScript.

A thorough discussion of this custom server control could warrant an entire article in itself, so rather than focus on the inner workings of the control, let's examine how to use the **MessageBox** control in an ASP.NET Web page. (The complete source for the control, as well as a sample ASP.NET Web page using the control, is available in this article's download.)

To use the **MessageBox** control in an ASP.NET Web page, first add the **MessageBox** control to the Visual Studio .NET Toolbox. This can be accomplished by right-clicking on the Toolbox and choosing to Add/Remove Items from the Toolbox, and then browsing to the **MessageBox** assembly. To add the client-side message box to a Web page, simply drag it from the Toolbox onto the Designer. Figure 7 shows the **MessageBox** control in the Visual Studio .NET Designer.

[](http://msdn.microsoft.com/en-us/library/Aa479302.clientside_fig07(en-us,MSDN.10).gif)

**Figure 7. Displaying a modal messagebox**

The **MessageBox** class has a number of properties that you can configure to tweak the appearance of the message box:

* **Buttons**. Specifies what buttons are displayed. The options are defined in the **ButtonOptions** enumeration and can be: OkOnly, OkCancel, AbortRetryIgnore, YesNoCancel, YesNo, or RetryCancel.
* **DisplayWhenButtonClicked**. The **ID** of the Button Web control that, when clicked, will display the client-side message box. Use this property if you want the message box displayed due to a specific button being clicked.
* **Icon**. The icon displayed in the message box; the options are defined in the **IconOptions** enumeration. The legal values are: Critical, Question, Exclamation, and Information.
* **Prompt**. The text displayed within the message box.
* **Title**. The title of the message box.

Once you have added the **MessageBox** control to an ASP.NET Web page, the next challenge is having it displayed due to some client-side action. The **DisplayWhenButtonClicked** property allows you to specify the **ID** of a Button Web control on the page that, when clicked, will cause the message box to be displayed. Alternatively, you can have the message box displayed by calling the client-side function **mb\_show(id)**, where *ID* is the **ID** of the **MessageBox** control.

Regardless of what button configuration you choose to display in the message box, when any button is clicked, a postback ensues and the **MessageBox** control's **Click** event fires. You can create an event handler for this event by simply double-clicking on the **MessageBox** in the Designer. The event handler's second input parameter is of type **MessageBoxClickedEventArgs**, which contains a **ButtonClicked** property that returns information on what message-box button the user clicked.

The **MessageBox** control is useful in situations where you want to quickly present the user with a modal dialog box that, regardless of the user's choice, results in a postback. To see the **MessageBox** control in action, check out the MsgBoxDemo.aspx page in the source code download.

**Conclusion**

This article began with a look at common uses of client-side script in a Web page, and then turned to examine the methods and techniques for injecting client-side script into an ASP.NET Web page. As we saw, the **Page** class contains a number of methods designed for programmatically inserting client-side script blocks from the server-side code-behind class. These methods are also commonly used from custom, compiled server controls, as discussed in an earlier article of mine: [Injecting Client-Side Script from an ASP.NET Server Control](http://msdn.microsoft.com/en-us/library/aa478975.aspx).

In addition to adding script blocks, client-side functionality often must be tied to a client-side event raised by some HTML element. To programmatically specify a Web control's client-side event handler through the server-side code-behind class, use the **Attributes** collection, which is available as a property to all Web controls.

The second half of this article applied the topics covered in the first half, showing how to implement common client-side functionality in an ASP.NET Web page. We saw how to extend the **Page** class so that from a code-behind class we could easily display an alert box, set the focus upon page load to a specific Web control, how to display a popup, and how to display a confirm dialog box. We also looked at creating a custom server control that used VBScript to provide a richer client-side message box user experience that caused a postback regardless of the button clicked.