

PETER'S INTERACTIVE PAGES USER'S GUIDE



Click on any of these topics to jump to them:

◆ FieldStateController	Using	Adding	Properties
◆ MultiFieldStateController	Using	Adding	Properties
◆ FSCOnCommand	Using	Adding	Properties
◆ MultiFSCOnCommand	Using	Adding	Properties
◆ CalculationController	Using	Adding	Properties
• NumericTextBoxCalcItem	ConstantCalcItem	ListConstantCalcItem	CheckStateCalcItem
• ConditionCalcItem	ParenthesisCalcItem	CalcControllerCalcItem	TotalingCalcItem
◆ TextCounter Control	Using	Adding	Properties
◆ ContextMenu	Using	Adding	Properties
◆ DropDownMenu	Using	Adding	Properties
◆ Interactive Hints	Using	Adding	PopupViews Adding On Page
		Adding	SharedHintFormatters
◆ Enhanced ToolTips	Using		
◆ Enhanced Buttons	Using	Adding	Properties
◆ ChangeMonitor	Using	Page-Level	Properties Button Properties
◆ Direct Keystrokes to Click Buttons			
◆ Custom Submit Function			
📄 Learn online			
◆ Page Level Properties and Methods			
◆ JavaScript Support Functions			
◆ Table Of Contents		Troubleshooting	

License Information

This document includes information for the Peter's Interactive Pages module in Peter's Data Entry Suite. If you licensed the complete Suite or the "Peter's Interactive Pages" module, you have all features found in this User's Guide, unless otherwise noted.

Note: The FieldStateController and MultiFieldStateController work best when they have access to the Condition objects that are part of the Peter's Professional Validation module.

The CalculationController works best when it has access to the TextBoxes that are part of the Peter's TextBoxes module.

Peter's Interactive Pages Overview

Peter's Data Entry Suite ("DES") focuses on enhancing data entry on your ASP.NET web forms. When you build desktop (Windows, Mac, Linux, etc) applications, the operating systems provide an interactive graphical user interface, making it easy to assist the user. HTML has a very limited set of behaviors described in its tags. For example, while it's easy to set focus to the first textbox in a Windows application, there is no HTML equivalent.

JavaScript and the document object models specified by the W3C (called "DOM") and Microsoft (called "DHTML") overcomes some of HTML's deficiencies. DES packages many of the techniques, so that you don't have to figure them out (a task made more difficult by the differences between DHTML, DOM, and each browser's implementation of them).

The **Peter's Interactive Pages** uses JavaScript to enhance how the web form interacts with the user and assist them with their data entry.

- When loading the page:
 - Setting focus to a field
 - Using the FieldStateControllers to prepare the look of a page
 - Use the ChangeMonitor to disable buttons until an edit occurs
- When the user edits the page:
 - The FieldStateControllers monitor edits and update the appearance of other controls, such as enabling them, making them visible, or changing their style.
 - The CalculationController calculates and updates values as the user changes numbers in textboxes.
 - The Interactive Hints system showing hints as the user puts focus in a data entry field
 - The Enhanced ToolTips replaces the browser's tooltip with an HTML-driven popup.
 - The TextCounter tells the user how close they are to the maximum size of a textbox.
 - The ChangeMonitor enables buttons after an edit occurs.
 - The ContextMenu lets you offer a right click menu with your own commands
 - The DropDownMenu offers a menu of commands available by clicking a button or label.
- When submitting a page:
 - Prompting the user to confirm before submitting
 - Direct the ENTER key to click a button
 - Disable the submit buttons as the page is submitted

Click on any of these topics to jump to them:

- ◆ [FieldStateControllers Overview](#)
- ◆ [CalculationController Overview](#)
- ◆ [TextCounter Overview](#)
- ◆ [ContextMenu and DropDownMenu Overview](#)
- ◆ [ChangeMonitor Overview](#)
- ◆ [Interactive Hints Overview](#)
- ◆ [Enhanced ToolTips Overview](#)
- ◆ [Enhanced Buttons Overview](#)
- ◆ [Direct Keystrokes to Click Buttons Overview](#)

FieldStateControllers Overview

The four FieldStateController controls make changes to the HTML elements on your page. They can change almost any element on your page: visibility, enabled state, readonly state, the value, and more.

There are two types of FieldStateControllers: those that apply a single field state and those that apply one of two field states, based on a [Condition](#).

The FSCOnCommand and MultiFSCOnCommand are invoked by a click on a “command” such as a button and apply a single field state. Any HTML tag that supports the “onclick” event can be used to fire it. Examples:

- You have a CheckBoxList and use a button titled “Select All” to mark all checkboxes.
- In a tabbed interface, an image that represents a “tab” can show or hide a panel containing the tab’s “page”

The FieldStateController and MultiFieldStateController support two field states based on a [Condition](#). The Condition serves two purposes:

- It monitors the controls assigned to the **ControlIDToEvaluate** and **SecondControlIDToEvaluate** properties of the Condition. When they are changed or clicked (for non-data entry fields), it invokes the FieldStateController.
- It selects which of the two field states to apply, based on whether the Condition evaluates as “success” or “failed”.

They can use the extensive list of [Conditions](#) from DES’s Validators. For example, you set it up to use the CheckStateCondition to monitor a checkbox and a RequiredTextCondition to monitor a textbox.

Once the FieldStateControllers have done their task, they can optionally run the Validators on the field whose state was changed or run an entire validation group.

See “[FieldStateController and MultiFieldStateController](#)” and “[FSCOnCommand and MultiFSCOnCommand](#)”.

 [Online examples](#)

CalculationController Overview

The CalculationController lets you describe calculations that involve numbers in textboxes, constants and other logic. Using javascript, it interactively calculates and updates its result as the user edits the page. The values from these calculations can be used in the following ways:

- Displayed on the page, whether in a Label or a textbox.
- Validators that compare numbers can evaluate the value simply by setting their **ControlIDToEvaluate** property to this control’s ID. Supported validators include: CompareToValueValidator, CompareTwoFieldsValidator, RangeValidator, and DifferenceValidator. In addition, the RequiredTextValidator can determine if the calculation had an error.
- Like Validators, their [Conditions](#) can evaluate the value. For example, the Enabler property on various controls use Conditions. Now those Conditions can enable their control based on the result of a calculation.

See “[CalculationController](#)”.

 [Online examples](#)

TextCounter Overview

The TextCounter control displays the number of characters or words within a textbox. It assists users when there are limits to the size of text they can enter. It compliments, but does not replace the TextLengthValidator/WordCountValidator, because it does not impose a limit. It merely communicates the count and if a limit is exceeded.

The user interface of the TextCounter can be like an interactive label control. It also can present itself in the Hint feature of DES TextBoxes.

See “[TextCounter Control](#)”

 [Online examples](#)

ContextMenu and DropDownMenu Overview

The ContextMenu is a client side menu that is the basis for the context and help menus in the Peter's Date and Time controls. It is designed to look like a browser's context menu with gaps on both sides, a column with the command name, a column with keystroke for that command, and a frame around it. When the mouse passes over rows, the row is highlighted. When the user clicks on a row, your client side script is run. It can appear open full time or be hooked up to one or more objects on the page to pop up on your choice of left or right mouse clicks.

The DropDownMenu attaches the ContextMenu to a button or label. The user clicks the button to see the Menu.

See "[ContextMenu and DropDownMenu Control](#)".

 [Online examples](#)

ChangeMonitor Overview

The ChangeMonitor watches for edits in the form and changes the appearance of buttons and other fields upon the first detected edit.

The classic case is to have a disabled OK button that gets enabled as you start typing. Another case is to show a message like "This form has changed" in a label. Both of these cases are handled.

DES's enhanced buttons are already capable of showing a confirmation message. With the ChangeMonitor in use, that message can be shown based on whether or not the user has edited the form.

See "[ChangeMonitor](#)".

 [Online examples](#)

Interactive Hints Overview

As users work with textboxes, there are fields that require specific entries. Perhaps they require a pattern (like a date is day/month/year) or they have limits ("Keep values between 1 and 5"). Web pages often communicate this information by adding a label on the page, near the field. The label is always present, taking up valuable screen real estate.

The Interactive Hint allows a "PopupView" or label to show a message for the field currently with focus. When using labels, as the user tabs around, the text changes or completely disappears. This allows for much better screen usage.

This feature also puts the hint in the browser's status bar and the control's tooltip. It optionally can include a validation error in the hint. This gives the user the error as they tab into the field.

See "[Interactive Hints](#)".

Enhanced ToolTips Overview

The browser provides the ToolTip to describe almost any field as the mouse passes over it. That tooltip is very limited. For most browsers, it cannot be multiline. It has one style (yellow). It cannot support HTML.

Using the same PopupView feature found in Interactive Hints and the DES Validator's PopupErrorFormatter, DES gives you a better tooltip. You control its appearance and supply it with HTML to convey the information better.

See "[Enhanced ToolTips](#)".

Enhanced Buttons Overview

DES provides replacements for the native Button, LinkButton and ImageButton controls. While it needs to do this to invoke its validation as the page is submitted, there are many ways to enhance buttons using javascript.

The DES buttons provide these enhancements:

- Use the **ConfirmMessage** property to display a confirmation message. If the user answers No to the prompt, it will prevent the postback. Combined with the ChangeMonitor and the **ChangeMonitorUsesConfirm** property, you can have the message shown only after an edit occurred.
- Use the **ChangeMonitorEnables** property to determine when the button is enabled as the ChangeMonitor determines the page has been edited. When setup, the button is disabled as the page is loaded.

- Use the **DisableOnSubmit** property to disable the button after the user clicks, to limit the chance of a double submission.
- Use the **MayMoveOnClick** property when validation is causing the user to click the button twice before it will submit. The button is actually moving after the first click because validation is removing its error messages causing the page to reposition its contents. *This property does not require any license.*
- Built in support for “Interactive Hints” and “Enhanced ToolTips”.
- When using the DES Validation Framework, validation groups support special tokens to match to all groups (“*”) and assign group names based on their naming container (“+”). With the **SkipPostBackEventsWhenInvalid** property, they can skip calling your **Click** and **Command** event handler methods if validation errors are detected.
- ImageButtons will actually dim (using style sheet opacity) when disabled by the ChangeMonitor, DisableOnSubmit property, or the FieldStateController.
- LinkButtons normally show the contents of their href= attribute, which is javascript code, in the browser’s status bar. Unless prevented by the browser, DES’s LinkButtons will hide the script from the status bar. If you have a tooltip assigned, its text is used as a replacement.

The DES buttons are direct subclasses of the native buttons, making it very easy to switch to them.

See “[Enhanced Buttons](#)”.

Direct Keystrokes to Click Buttons Overview

DES’s TextBoxes and the MultiSegmentDataEntry control offer the **EnterSubmitsControlID** property, which lets you direct the ENTER key to click a specific button or control. It’s useful when you have several Submit buttons on the page, each with their own task.

Additional, the `PeterBlum.DES.Globals.WebFormDirector.RegisterKeyClicksControl()` lets you attach this capability to any control. This method has several advantages:

- It allows you to define the keystroke that clicks the button. For example, ESC can hit a “Cancel” button.
- Instead of setting it up for individual controls, you can set it up for a group of controls by attaching this to a container control, like a Panel or Table. The browsers are designed to let the onkeypress event, used here, to “bubble up” until consumed (which is what the container will do).

See “[Direct Keystrokes to Click Buttons](#)”.

FieldStateController and MultiFieldStateController Controls

The FieldStateController and MultiFieldStateController controls provide an interactive client-side interface by monitoring user actions on fields and changing the attributes and styles of other controls on the page. For example, when the user clicks a checkbox, show a previously hidden <div>. The FieldStateControllers often eliminate the effort to build the browser-independent JavaScript required to monitor events, look at field data, and change attributes and styles.

These controls do most of their work on the client-side. If the browser does not support the client-side scripting needed to run a FieldStateController, it is disabled. That will leave your controls with the state that you define in their properties on the server side.

The FieldStateController adds no HTML to your page as it does it work through JavaScript. You can add them anywhere to your web form.

Click on any of these topics to jump to them:

- ◆ [Features](#)
- ◆ [Using the FieldStateControllers](#)
 - [The Condition](#)
 - [Controls that run the FieldStateController](#)
 - [Controls To Change](#)
 - [Attribute Values To Change](#)
 - [Extending the Attributes with Your Own Code](#)
 - [Updating Validators](#)
 - [Changing Visibility on a Complex Control](#)
 - [Toggling States](#)
 - [JavaScript: Running FieldStateControllers on demand](#)
 - [Controls That Have Child Controls](#)
- ◆ [Example: FieldStateController](#)
- ◆ [Example: MultiFieldStateController](#)
- ◆ [Adding the FieldStateController Control](#)
- ◆ [Adding the MultiFieldStateController Control](#)
- ◆ [Properties of FieldStateController And MultiFieldStateController](#)
- ◆ [Online examples](#)

Features

The FieldStateController and MultiStateController controls make changes to the HTML elements on your page. They can change almost any element on your page:

- Show or hide
- Enable or disable form controls (textboxes, lists, buttons, etc)
- Change the ReadOnly state of a textbox
- Change the style sheet class name, which can deliver an entirely different appearance through style sheets
- Change the textual value of a textbox
- Change the value of the selected element in a listbox or dropdownlist
- Change the “innerHTML” of a Label, , or any other HTML tag that supports “innerHTML”
- Change the URL associated with hyperlinks, images and other HTML tags that have an href= or src= attribute.
- Change the mark in a checkbox or radiobutton
- Mark or unmark all checkboxes in a CheckBoxList
- Change the value of any document object model attribute that has a datatype of string, boolean or integer
- Change the value of any document object model style
- Run your own JavaScript to handle special situations

You can see how powerful these controls are. You only need to set properties on the controls and you have enhanced your user interface.

The FieldStateController and MultiFieldStateController support two field states based on a [Condition](#). The Condition serves two purposes:

- It monitors the controls assigned to the **ControlIDToEvaluate** and **SecondControlIDToEvaluate** properties of the Condition. When they are changed or clicked (for non-data entry fields), it invokes the FieldStateController.
- It selects which of the two field states to apply, based on whether the Condition evaluates as “success” or “failed”.

They can use the extensive list of Conditions from DES’s Validators. For example, you set it up to use the CheckStateCondition to monitor a checkbox and a RequiredTextCondition to monitor a textbox.

The FieldStateController and MultiFieldStateController initialize the look of the page by running the Condition and applying the appropriate field state. This way, the page has the correct look and you don’t have to write any code on the server side to establish the initial appearance.

Once the FieldStateControllers have done their task, they can optionally run the Validators on the field whose state was changed or run an entire validation group.

Using the FieldStateControllers

There are four elements that always must be set up on a FieldStateController:

- A [Condition](#) object determines what control to monitor and selects between two sets of state settings. See “[The Condition classes](#)”.
- The controls that run the FieldStateController when clicked or changed
- The control or controls whose attributes that you want to change
- The attribute values that will change

Click on any of these topics to jump to them:

- ◆ [The Condition](#)
- ◆ [Controls that run the FieldStateController](#)
- ◆ [Controls To Change](#)
- ◆ [Attribute Values To Change](#)
- ◆ [Extending the Attributes with Your Own Code](#)
 - [Client-Side Function: The Change State Function](#)
 - [Server Side Event Handler](#)
- ◆ [Updating Validators](#)
- ◆ [Changing Visibility on a Complex Control](#)
- ◆ [Toggling States](#)
- ◆ [JavaScript: Running FieldStateControllers on demand](#)
- ◆ [Controls That Have Child Controls](#)
 - [GetChild Method](#)
 - [Installing the GetChild Method](#)
- ◆ [Online examples](#)

The Condition

A Condition object evaluates something on the page and determines if it indicates “success”, “failure”, or “cannot be evaluated”. See “[The Condition classes](#)”. Conditions include references to the controls whose data that the user will change, such as textboxes, lists, and checkboxes. The FieldStateController changes one or more attributes of other controls based on whether the Condition indicates “success” or “failure”.

The Condition object is assigned to the [Condition](#) property. It can be assigned in ASP.NET markup or programmatically.

FieldStateControllers use the same Condition objects as Validators. This can be any Condition class including the MultiCondition used to build Boolean expressions. You can use the CustomCondition to create your own rules as well. If you do, you must create client-side code for your Condition as the FieldStateController does most of its work on the client-side.

Note: FieldStateControllers are easier to set up when you have a license for the Peter’s Professional Validation module. Otherwise, you are limited to creating your own Condition code using the CustomController class and using any of the Non-Data Entry Conditions. Both are described in the Validation User’s Guide but do not require a license for any Validators module.

```
<des:FieldStateController id="FSC1" runat="server" properties>
  <ConditionContainer>
    <des:ConditionClass properties>
  </ConditionContainer>
</des:FieldStateController>
```

Controls that run the FieldStateController

Usually, the [Condition](#) dictates which field runs the FieldStateController when the user clicks or edits the control specified by the **ControlIDToEvaluate** or **SecondControlIDToEvaluate** properties. If you want to let the user click on a non-data entry field, like a label, button, or image, to run the FieldStateController, add the desired control to the [ExtraControlsToRunThisAction](#) property. *When **ExtraControlsToRunThisAction** is used, consider setting the **EvaluateOnClickOrChange** property to false on each Condition.*

By default, the FieldStateController evaluates after focus leaves the textbox, list, or dropdownlist. Sometimes you want it to evaluate as the user types. For example, a FieldStateController that displays another field so long as a textbox has text might want to display it as soon as the first letter is entered into the textbox. Set the [UpdateWhileEditing](#) property to `true` for this behavior.

The FieldStateControllers run as the page is first loaded into the browser. This establishes an initial appearance based on the Condition at the time. You do not have to establish the state yourself. For example, if a `<div>` should be invisible initially, the FieldStateController will handle it for you. You can override this behavior with the [RunOnPageLoad](#) property.

Controls To Change

You must assign the ID or an object reference to the control(s) that you want to change. The FieldStateController control requires one control to change. Use **ControlIDToChange** when you have an ID or **ControlToChange** when you have an object reference.

The MultiFieldStateController changes as many controls as you want. Add PeterBlum.DES.Web.WebControls.FSAControlConnection objects to the **ControlConnections** property. The PeterBlum.DES.Web.WebControls.FSAControlConnection class can be assigned an ID to its **ControlID** property and an object reference to its **ControlInstance** property.

Note: All controls must have an ID and runat=server property.

```
<des:FieldStateController id="FSC1" runat="server" ControlIDToChange="ID"
  Other properties >
  <ConditionContainer>
    <des:ConditionClass properties >
  </ConditionContainer>
</des:FieldStateController>

<des:MultiFieldStateController id="MFSC1" runat="server" properties>
  <ControlConnections>
    <des:FSAControlConnection ControlID="ID1" />
    <des:FSAControlConnection ControlID="ID2" />
  </ControlConnections>

  <ConditionContainer>
    <des:ConditionClass properties>
  </ConditionContainer>
</des:MultiFieldStateController>
```

Attribute Values To Change

You can change any of these attributes:

- Visibility – changes the **style:visibility** and **style:display** attributes. For a special situation, see “[Changing Visibility on a Complex Control](#)”.
- Enabled – changes the **disabled** attribute on the controls that support it (which varies by browser)
- ReadOnly – changes the **readOnly** attribute on textboxes
- CssClass – changes the style sheet class name
- FieldValue – changes the **value** attribute of <input>, <textarea>, and <select> tags
- InnerHTML – changes the **innerHTML** attribute on any control. **innerHTML** is found in tags that permit contents between their begin and end tags, like this: <tag>*innerHTML*</tag>. A Label, Panel, and TableCell are web controls that generate tags that support InnerHTML (, <div>, and <td> respectively.)
- URL – changes the **href** or **src** attribute to a new URL on , <input type=image>, <frame>, <iframe>, and <a> tags.
- Checked – changes the **checked** attribute on a checkbox or radiobutton
- If you know the name and legal values of an attribute or style, there is an all-purpose property, **Other**, which will modify the attribute or style with the value as the [Condition](#) changes.

You define them in the [ConditionTrue](#) and [ConditionFalse](#) properties. When the [Condition](#) evaluates as “success”, it applies the **ConditionTrue** attributes; when it evaluates as “failed”, it applies the **ConditionFalse** attributes. A change is applied only when the attribute differs between **ConditionTrue** and **ConditionFalse**. For example, **ConditionTrue.Visible** must differ from **ConditionFalse.Visible** for a visibility change to occur. See “[Properties of ConditionTrue and ConditionFalse](#)”.

```
<des:FieldStateController id="FSC1" runat="server" ControlIDToChange="ID"
  ConditionFalse-AttributeName="value" Other properties >
  <ConditionContainer>
    <des:ConditionClass properties >
  </ConditionContainer>
</des:FieldStateController>
```

Example 1: Hiding a control when a checkbox is unchecked

```
<des:FieldStateController id="FSC1" runat="server" ControlIDToChange="TextBox1"
  ConditionFalse-Visible="false" >
  <ConditionContainer>
    <des:CheckStateCondition ControlIDToEvaluate="CheckBox1" />
  </ConditionContainer>
</des:FieldStateController>
```

Example 2: Disabling 2 controls when a checkbox is unchecked

```
<des:MultiFieldStateController id="MFSC1" runat="server"
  ConditionFalse-Enabled="false" >
  <ControlConnections>
    <des:FSAControlConnection ControlID="TextBox1" />
    <des:FSAControlConnection ControlID="TextBox2" />
  </ControlConnections>
  <ConditionContainer>
    <des:CheckStateCondition ControlIDToEvaluate="CheckBox1" />
  </ConditionContainer>
</des:MultiFieldStateController>
```

Extending the Attributes with Your Own Code

Sometimes you need to write your own code to change attributes. For example, you have a complex control that needs to hide several related controls when it is hidden. You can write client- and server-side code to handle this.

Here are some of the cases to consider:

- A third party custom control uses its own JavaScript to adjust its properties.
- The control is created by JavaScript on the client side and has no server-side ID.
- A calculation must be performed before the setting can be determined.

You assign your function to the **ChangeStateFunctionName** property.

Client-Side Function: The Change State Function

Create a client-side function in JavaScript and assign its name to the **ChangeStateFunctionName** property.

***ALERT:** Many users make the mistake of assigning JavaScript code to this property. This will cause JavaScript errors. **GOOD:** "MyFunction". **BAD:** "MyFunction();" and "alert('stop it')".*

Note: JavaScript is case sensitive. Be sure the value of this property exactly matches the function definition.

Your function must take these three parameters in the order shown:

- An object reflecting this FieldStateController's properties on the client-side. It allows you to get properties like **CssClass**, **Enabled** and **Visible** for use in your function. They are client-side properties on the object with these names:
 - CT_Vis (Boolean) – ConditionTrue.Visible
 - CF_Vis (Boolean) – ConditionFalse.Visible
 - CT_Enab (Boolean) – ConditionTrue.Enabled
 - CF_Enab (Boolean) – ConditionFalse.Enabled
 - CT_RO (Boolean) – ConditionTrue.ReadOnly
 - CF_RO (Boolean) – ConditionFalse.ReadOnly
 - CT_Css (string) – ConditionTrue.CssClass
 - CF_Css (string) – ConditionFalse.CssClass
 - CT_Html (string) – ConditionTrue.InnerHTML
 - CF_Html (string) – ConditionFalse.InnerHTML
 - CT_URL (string) – ConditionTrue.URL
 - CF_URL (string) – ConditionFalse.URL
 - CT_Chk (Boolean) – ConditionTrue.Checked
 - CF_Chk (Boolean) – ConditionFalse.Checked
 - CT_Val (string) – ConditionTrue.FieldValue
 - CF_Val (string) – ConditionFalse.FieldValue
 - InvPS (Boolean) – InvisiblePreservesSpace
- ControlToChange element reference – The element that is being operated upon. It is an object for the element. If you need the object's ID, this object has a property called **id**.
- ConditionValue – a Boolean. When **true**, the Condition indicates success.

It does not return a result.

If you need to know how to add JavaScript to your page, see "[Adding Your JavaScript to the Page](#)".

Example

This function will call a fictitious JavaScript function, `TPF_SetVisibility()`, which changes the visibility of the control ID. It makes the control visible when the Condition indicates success.

```
function MyFSCFunction(DESObj, ControlToChange, ConditionValue)
{
    TPF_SetVisibility(ControlToChange.id, ConditionValue);
}
```

Server Side Event Handler

The FieldStateController performs some of its state changes as the page is first created on the server side. If you write a client-side function for the **ChangeStateFunctionName** property to use, use the **StateChange** event handler property to attach an equivalent server side method. The **StateChange** property expects your method to match the `PeterBlum.DES.Web.WebControls.ChangeStateEventHandler` delegate.

*Note: The **StateChange** property only handles one event handler and must be assigned programmatically (it does not appear in the Properties Editor.)*

The `ChangeStateEventHandler` is defined here:

[C#]

```
public delegate void ChangeStateEventHandler(
    PeterBlum.DES.IBaseFieldStateAction sender,
    PeterBlum.DES.Web.WebControls.ChangeStateEventArgs args);
```

[VB]

```
Public Delegate Sub ChangeStateEventHandler( _
    ByVal sender As PeterBlum.DES.IBaseFieldStateAction, _
    ByVal args As PeterBlum.DES.Web.WebControls.ChangeStateEventArgs)
```

Parameters

sender

An internal representation of the FieldStateControllers. It contains the same properties but is of the interface `PeterBlum.DES.IBaseFieldStateAction`. If you used the `MultiFieldStateController`, this represents a single control to change and the event handler will be called once for each control to change.

args

The `PeterBlum.DES.Web.WebControls.ChangeStateEventArgs` class provides additional inputs that are useful to your event handler. The **ControlToChange** property is reference to the control that is being changed. **Success** is a Boolean where it indicates success of the Condition when true and failure when false.

[C#]

```
public class ChangeStateEventArgs : System.EventArgs
{
    public Control ControlToChange { get; }
    public bool Success { get; }
}
```

[VB]

```
Public Class ChangeStateEventArgs Inherits System.EventArgs
    Public ReadOnly Property ControlToChange As Control
    Public ReadOnly Property Success As Boolean
End Class
```

Updating Validators

Sometimes a field hidden or disabled by the FieldStateController has an associated Validator whose error message is showing. That message is no longer appropriate. To remove it, first set up the **Enabler** property on the Validator to detect that the control is visible or enabled. Use the `VisibleCondition` or `EnabledCondition`.

Then set the **ValidateChangedControls** property to `true`.

If the FieldStateController affects controls that are used in the **Enabler** properties of other Validators, let it run all Validators in the validation group associated with those Validators. Set the **UseValidationGroup** property to `true` and the group name of the Validator in the **ValidatorGroup** property.

Use the **RevalidateOnly** property to evaluate only validators that have previously been evaluated on the page. This prevents validator errors from appearing further down the page, where the user has not edited.

Changing Visibility on a Complex Control

Some web controls include a number of HTML tags. The control's ID property may refer to just one of the HTML tags it generates. If you use the `FieldStateController` to show and hide that control by its ID, you will only show or hide the one tag associated with the control ID.

Solution

Look at the HTML output of any web control to see which HTML tag is assigned the ID (specifically the **ClientID** property value.) If that tag encloses all HTML for that control, you can use the web control's ID with the **FieldStateController.ControlIDToEvaluate** property.

If the tag does not enclose the control, add a `` or `<div>` tag around the web control. Set the `runat="server"` property and assign an ID value. Set the **FieldStateController.ControlIDToEvaluate** property to the ID of that `` or `<div>` tag.

Example: The DateTextBox Control

ALERT: DES's own controls – including the `DateTextBox` – do not need the this technique as they automatically account for the issue here. This is merely an example.

The textboxes in Peter's Date And Time use multiple HTML tags. For example, the `DateTextBox` has an `` tag to the right of the textbox which is used to toggle a popup calendar. The textbox in these controls is associated with the control's ID.

```
<input type='text' id='control_clientid' ><img src='calendar.jpg' />
```

If you assign the **ControlIDToChange** property to the `DateTextBox`'s ID, it will only show and hide the textbox, leaving the image visible.

Here's the solution.

```
<span runat="server" id="DateTextBox1Container">
  <des:DateTextBox runat="server" id="DateTextBox1" />
</span>

<des:FieldStateController runat="server" id="FSC1"
  ControlIDtoChange="DateTextBox1Container" ConditionTrue-Visible="true">
  <Condition>
    [you determine this]
  </Condition>
</des:FieldStateController>
```

Toggle States

You can easily set up a checkbox to show and hide other fields using the `FieldStateController` because the checkbox has a clear two-state value. As you work with `FieldStateControllers` you will learn to use `Conditions` to build almost any way to toggle between two states such as when a particular field is invisible, toggle it to visible.

Suppose that you want to use a `Button` to show or hide another field each time it's clicked. Here's how you would set up a `FieldStateController` to handle this case.

- Add the `Button` control to the `ExtraControlsToRunThisAction` collection.
- Assign the `Condition` property to the `VisibilityCondition`. Set its `IsVisible` property to `true`. Set its `ControlIDToEvaluate` to the field whose visibility will change.
- Assign `ControlIDToChange` to the field whose visibility will change.
- Assign `ConditionTrue.Visible` to `false` and `ConditionFalse.Visible` to `true`.

JavaScript: Running FieldStateControllers on demand

When you add your own JavaScript on a page, you may change the state of the page. FieldStateControllers will not notice your changes and the page may no longer be consistent with how you want it to work. DES provides the JavaScript function `DES_RunAllFSC()` that runs all FieldStateControllers on the page whose **RunOnPageLoad** property is `true`. Call the function from within your JavaScript code.

You can set up your JavaScript in two ways:

1. Embed the function call `DES_RunAllFSC()` into your code.

```
<script type='text/javascript' language='javascript'>
function ChangeMyFields()
{
    // change some fields here
    DES_RunAllFSC();    // let DES catch up
}
</script>
```

For details on adding scripts to your page, see [“Adding Your JavaScript to the Page”](#).

2. Let `PeterBlum.DES.Globals.WebFormDirector.GetRunAllFSCScript()` return a short script that calls this function.

```
Button1.Attributes["onclick"] =
    PeterBlum.DES.Globals.WebFormDirector.GetRunAllFSCScript()
```

Controls That Have Child Controls

The FieldStateControllers usually interact with a single HTML tag, such as a `<input>`, `<table>` or ``. Some web controls, such as the RadioButtonList and CheckBoxLayout, really do most of their work with child tags. For example, both the RadioButtonList and CheckBoxLayout have unique `<input type=radio|checkbox>` tags for the buttons. When you use the FieldStateController to change their state, you often want to modify the state of these controls. The FieldStateController has an expandable mechanism to handle these kinds of web controls.

It automatically supports `System.Web.UI.WebControls.RadioButtonList` and `System.Web.UI.WebControls.CheckBoxLayout`. You can provide a JavaScript function to handle the child controls of other web controls.

This feature only updates these state settings on child controls:

- Visibility
- Enabled
- ReadOnly
- CssClass
- Checked (not recommended for RadioButtonLists)

All other state settings are only applied to the HTML tag associated with the ID specified in [ControlIDToChange](#) or [ControlConnections](#).

GetChild Method

The GetChild method is a client-side function that is associated with a specific web control class and returns each child control for the FieldStateController to use. You will define this function when you are using a custom control whose child elements should change their visibility, enabled, read only, classname, or checked state settings.

The GetChild method is written in JavaScript and added into your web page. See [“Adding Your JavaScript to the Page”](#). It has the following format:

```
function FunctionName(pID, pIndex, pMode)
{
    return [a field based on Index and Mode or null];
}
```

Your function will be called with incrementing values of *Index* until you return `null`.

Parameters

pID (string)

The ClientID assigned by the user to the control in the **ControlIDToChange** or **ControlConnections** property. Your function will use this ID to create the ID of a child control. For example, the CheckBoxLayout uses `pFieldID + "_" + Index`. See [“Embedding the ClientID into your Script”](#).

pIndex (integer)

A value starting at 0 that selects a control, either the main control or one of its children. The method should locate a specific child control matching that *Index* and return it. If the *Index* does not identify a child control, return `null`. The FieldStateController starts with *Index* = 0 and increments it until `null` is returned. So do not return `null` unless the *Index* is beyond the range of available child controls.

pMode (integer)

Your function will also be used to attach onclick and onchange event handlers to validators that may use your custom web control. This parameter determines if the FieldStateController or a validator calls your function. The values are:

0 – Return all controls whose states should be modified by the FieldStateController.

1 – Return data entry oriented child controls. These will be set up with onclick and onchange event handlers.

Return value

A reference to a control or null. Your method should create a FieldID based on the three parameters and pass it to `DES_GetById()`. That function returns the matching control reference or null.

Example

This is the function used by both the `CheckBoxList` and `RadioButtonList` web controls. Child controls have the format `FieldID_#` where # is 0 and up.

```
function DES_GCCheckRadioList(pID, pIndex, pMode)
{
    var vID = "";
    if (pMode == 0) // FSC needs the container (index 0)
        // and the child controls (index 1 and up)
        vID = pIndex == 0 ? pID: pID + "_" + (pIndex - 1);
    else
        vID = FieldID + "_" + pIndex;
    return DES_GetById(vID);
}
```

Installing the GetChild Method

You must map your `GetChild` method to the web control class in the `custom.DES.config` file. When DES sees a web control that matches an entry in the `custom.DES.config` file, it installs the `GetChild` method.

1. Locate the `<GetChildMethods>` section of the `custom.DES.config` file.
2. Add a new `<GetChildMethod>` element. It has this format:

```
<GetChildMethod type="[full classname, qualified assembly name]"
    method="[methodname]"/>
```

As an example, here are the `<GetChildMethod>` definitions for `CheckBoxList` and `RadioButtonList`:

```
<GetChildMethod type="System.Web.UI.WebControls.CheckBoxList, System.Web,
    Version=2.0.50727.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
    method="DES_GCCheckRadioList" />
<GetChildMethod type="System.Web.UI.WebControls.RadioButtonList, System.Web,
    Version=2.0.50727.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
    method="DES_GCCheckRadioList" />
```

Example: FieldStateController

Note: For more examples, see the **Tutorials** document and <http://www.peterblum.com/DES/DemoFSC.aspx>.

Suppose you have a checkbox that shows a Panel when the user marks it and hides the Panel when unmarked. The Panel contains numerous textboxes representing an address.

Condition: The CheckStateCondition, which is designed to monitor the state of a checkbox and indicate success when the mark matches the value you supply in the **Checked** property. Assign it to the **Condition** property.

Control That Runs This FieldStateController: The checkbox, assigned to the **CheckStateCondition.ControlIDToEvaluate** property.

Control To Change: Assign the Panel control's ID to **ControlIDToChange**.

Attributes to Change: When the Condition indicates "success", **ConditionTrue.Visible** is true. When the Condition indicates "failed", **ConditionFalse.Visible** is false.

```
<p><asp:CheckBox id="CheckBox1" runat="server"
    Text="Alternate shipping address"></asp:CheckBox></p>
<p><asp:Panel id="Panell" runat="server">TextBoxes for an Address</asp:Panel></p>
<des:FieldStateController id="FieldStateController1" runat="server"
    ConditionFalse-Visible="False" ControlIDToChange="Panell">
    <ConditionContainer>
        <des:CheckStateCondition ControlIDToEvaluate="CheckBox1" />
    </ConditionContainer>
</des:FieldStateController>
```

Example: MultiFieldStateController

This is a modification of the previous example. Instead of having a Panel, you have TextBoxes whose Enabled state will be changed as the checkbox is marked.

The differences from the previous example are:

- Use of the MultiFieldStateController
- Adding TextBoxes to the **ControlConnections** property
- Using the **Enabled** attribute instead of the **Visible** attribute on **ConditionTrue** and **ConditionFalse**.
- Textboxes replace the Panel

```
<p><asp:CheckBox id="CheckBox1" runat="server"
    Text="Alternate shipping address"></asp:CheckBox></p>
<p><asp:TextBox id="AddressLine1" runat="server"></asp:TextBox></p>
<p><asp:TextBox id="AddressLine2" runat="server"></asp:TextBox></p>
<p><asp:TextBox id="AddressLine3" runat="server"></asp:TextBox></p>
<des:MultiFieldStateController id="MultiFieldStateController1" runat="server"
    ConditionFalse-Enabled="False">
    <ControlConnections>
        <des:FSAControlConnection ControlID="AddressLine1"></des:FSAControlConnection>
        <des:FSAControlConnection ControlID="AddressLine2"></des:FSAControlConnection>
        <des:FSAControlConnection ControlID="AddressLine3"></des:FSAControlConnection>
    </ControlConnections>
    <ConditionContainer>
        <des:CheckStateCondition ControlIDToEvaluate="CheckBox1" />
    </ConditionContainer>
</des:MultiFieldStateController>
```

Adding the FieldStateController Control



These steps ask you to jump around the document using clicks on links. Adobe Reader offers a **Previous View** command to return to the link. Look for this in the Adobe Reader (shown v6.0)

1. Prepare the page for DES controls. See “Preparing a page for DES controls” in the **General Features Guide**. It covers issues like style sheets, AJAX, and localization.
2. Set up all the fields involved: the data entry controls that will toggle the state and the controls whose state will change. Be sure that the controls whose state will change include an ID and `runat="server"` property.

Test your page without adding any FieldStateControllers. This is how your page will operate when the browser does not have JavaScript available. Your users should be able to work with it in this state. Your server-side code should have its logic set up correctly to know when controls should be avoided because they are supposed to be invisible or disabled.

3. Add the FieldStateController control to the page. Its location does not matter as it contributes no HTML to the page.

Visual Studio and Visual Web Developer Design Mode Users

Drag the FieldStateController control from the Toolbox onto your web form. It will look like this:



Text Entry Users

Add the control (inside the `<form>` area):

```
<des:FieldStateController id="[YourControlID]" runat="server" />
```

Programmatically Creating the Control

- Identify the control which you will add the FieldStateController control to its **Controls** collection. Like all ASP.NET controls, the FieldStateController control can be added to any control that supports child controls, like Panel, User Control, or TableCell. If you want to add it directly to the Page, first add a Placeholder and use the Placeholder.
- Create an instance of the FieldStateController control class. The constructor takes no parameters.
- Assign the **ID** property.
- Add the FieldStateController control to the **Controls** collection.

In this example, the FieldStateController control is created with an **ID** of “FieldStateController1”. It is added to Placeholder1.

[C#]

```
PeterBlum.DES.Web.WebControls.FieldStateController vFSC =
    new PeterBlum.DES.Web.WebControls.FieldStateController();
vFSC.ID = "FieldStateController1";
Placeholder1.Controls.Add(vFSC);
```

[VB]

```
Dim vFSC As PeterBlum.DES.Web.WebControls.FieldStateController = _
    New PeterBlum.DES.Web.WebControls.FieldStateController()
vFSC.ID = "FieldStateController1"
Placeholder1.Controls.Add(vFSC)
```

Guidelines for setting properties

- Design mode users can use the Properties Editor or the Expanded Properties Editor. (See “Expanded Properties Editor” in the **General Features Guide**.) The SmartTag  also offers some of the most important properties.
- Text entry users should add the properties into the <des:ControlClass> tag in this format:
propertyname="value"
- When setting a property programmatically, have a reference to the control’s object and set the property according to your language’s rules.

4. Determine which Condition class will provide the controls to monitor and selects whether to use the field states of **ConditionTrue** or **ConditionFalse**. You can use any Condition class supplied with DES or create your own using the CustomCondition class. See “The Condition classes” to choose a class.

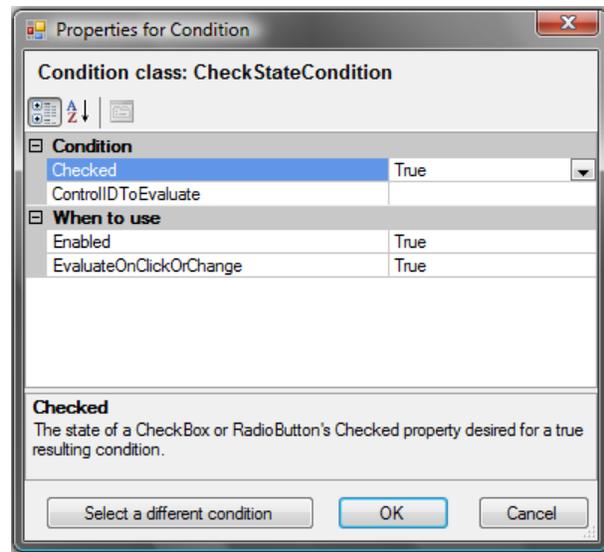
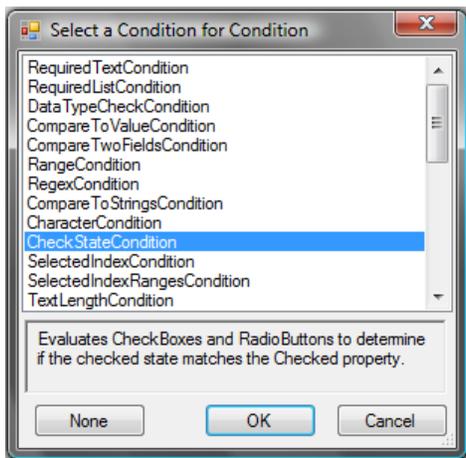
Note: You can use the Non-Data Entry Conditions, which monitor non-data oriented attributes of visibility, enabled, readonly and more. When you do, you will need to establish a control to monitor in the ExtraControlsToRunThisAction property.

Note: If you are changing the visibility on a web control, see “Changing Visibility on a Complex Control”.

5. Set the Condition object on the **Condition** property. Set its properties.

Visual Studio and Visual Web Developer Design Mode Users

The Properties Editor for the **Condition** property provides a window where you can select Condition objects and establish their properties.



- Select the Condition from the List and click **OK**.
- Establish the properties in the Properties grid.
- Click **OK**.

ASP.NET Text Formatting for the Conditions Property

You add the **Condition** as child of the <ConditionContainer> tag (*not* <Condition>).

The following example represents a CheckStateCondition.

```
<des:FieldStateController id="FieldStateController1" runat="server">
  <ConditionContainer>
    <des:CheckStateCondition ControlIDToEvaluate="CheckBox1" />
  </ConditionContainer>
</des:FieldStateController>
```

Notice that the **Condition** property name never appears in the attributes of the `<des:FieldStateController>` tag. (It will be added when using the Properties Editor but it's completely meaningless.) Instead, the `<ConditionContainer>` tag is a child of the `FieldStateController` control tag. That tag has no attributes. The child to `<ConditionContainer>` defines the class and all properties of the Condition:

```
<des:classname[all properties] />
```

- `des:classname` – Use any Condition class for the classname. If you create your own classes, you must declare the namespace using the `<% @REGISTER %>` tag at the top of the page.
- `[all properties]` – Enter the properties into the tag the same way you do for any other control.

Programmatically Adding Conditions

Here are the steps to set the **Condition**.

- Create an instance of the desired Condition class. There is a constructor that takes no parameters.
Note: There are also constructors that take parameters representing some of the control's properties. Each demands an "owner" in the first parameter. That value must be the FieldStateController object.
- Assign property values.
- Assign the Condition object to the **Condition** property.

In this example, add the `CheckStateCondition`, which is checking the mark of `CheckBox1`, to `FieldStateController1`.

[C#]

```
PeterBlum.DES.Web.WebControls.CheckStateCondition vCond =
    new PeterBlum.DES.Web.WebControls.CheckStateCondition();
vCond.ControlToEvaluate = CheckBox1;
vCond.Checked = true;
FieldStateController1.Condition = vCond;
```

[VB]

```
Dim vCond As PeterBlum.DES.Web.WebControls.CheckStateCondition = _
    New PeterBlum.DES.Web.WebControls.CheckStateCondition()
vCond.ControlToEvaluate = CheckBox1
vCond.Checked = True
FieldStateController1.Condition = vCond
```

6. Usually, the Conditions dictate which field runs the `FieldStateController` when the user clicks or edits the control specified by the **ControlIDToEvaluate** or **SecondControlIDToEvaluate** properties. If you want to let the user click on a non-data entry field, like a label, button, or image, to run the `FieldStateController`, add the desired control to the ExtraControlsToRunThisAction property. Also use **ExtraControlsToRunThisAction** when the Condition is a Non-Data Entry Condition. *When **ExtraControlsToRunThisAction** is used, consider setting the **EvaluateOnClickOrChange** property to false on each condition.*
7. Set the ID of control whose attributes will change with the ControlIDToChange property or a reference to the control in the **ControlToChange** property.
8. Assign the attributes that will change, depending on the Condition, in the **ConditionTrue** and **ConditionFalse** properties. **ConditionTrue** will be used when the Condition evaluates as "success". **ConditionFalse** will be used when the Condition evaluates as "failed".

The **ControlIDToChange** will change its settings when the attribute differs between **ConditionTrue** and **ConditionFalse**. For example, **ConditionTrue.Visible** must differ from **ConditionFalse.Visible** for a visibility change to occur.

See "Properties of ConditionTrue and ConditionFalse".

ASP.NET Text Formatting for the ConditionTrue and ConditionFalse Properties

The properties contained in **ConditionTrue** and **ConditionFalse** are added directly to the `<des:FieldStateController>` tag. Use this format: `<des:FieldStateController ConditionTrue-propertyname="value" ConditionFalse-propertyname="value">`. For example:

```
<des:FieldStateController id="FieldStateController1" runat="server"
    ConditionFalse-Visible="False">
```

9. Sometimes a field hidden or disabled by the `FieldStateController` has an associated `Validator` whose error message is showing. That message is no longer appropriate. To remove it, first set up the **Enabler** property on the `Validator` to detect that the control is visible or enabled. Use the `VisibleCondition` or `EnabledCondition` class. See the “[Evaluating non-data value states of controls](#)”.

Then set the **ValidateChangedControls** property to `true`.

10. Here are some other considerations:

- If you are using an AJAX system to update this control, set the **InAJAXUpdate** property to `true`. Also make sure the `PageManager` control or `AJAXManager` object has been setup for AJAX. See “Using these Controls With AJAX” in the **General Features Guide**. *Failure to follow these directions can result in incorrect behavior and javascript errors.*
- This control does not preserve most of its properties in the `ViewState`, to limit its impact on the page. If you need to use the `ViewState` to retain the value of a property, see “The `ViewState` and Preserving Properties for PostBack” in the **General Features Guide**.
- If you encounter errors, see the “[Troubleshooting](#)” section for extensive topics based on several years of tech support’s experience with customers.
- See also “[Additional Topics for Using These Controls](#)”.

Adding the MultiFieldStateController Control



These steps ask you to jump around the document using clicks on links. Adobe Reader offers a **Previous View** command to return to the link. Look for this in the Adobe Reader (shown v6.0)

1. Prepare the page for DES controls. See “Preparing a page for DES controls” in the **General Features Guide**. It covers issues like style sheets, AJAX, and localization.
2. Start by setting up all the fields involved: the data entry controls that will toggle the state and the controls whose state will change. Be sure that the controls whose state will change include an ID and `runat="server"` property.

Test your page without adding any MultiFieldStateControllers. This is how your page will operate when the browser does not have JavaScript available. Your users should be able to work with it in this state. Your server-side code should have its logic set up correctly to know when fields should be avoided because they are supposed to be invisible or disabled.

3. Add the MultiFieldStateController control to the page. Its location does not matter as it contributes no HTML to the page.

Visual Studio and Visual Web Developer Design Mode Users

Drag the MultiFieldStateController control from the Toolbox onto your web form. It will look like this:



Text Entry Users

Add the control (inside the `<form>` area):

```
<des:MultiFieldStateController id="[YourControlID]" runat="server" />
```

Programmatically Creating the Control

- Identify the control which you will add the MultiFieldStateController control to its **Controls** collection. Like all ASP.NET controls, the MultiFieldStateController control can be added to any control that supports child controls, like Panel, User Control, or TableCell. If you want to add it directly to the Page, first add a Placeholder and use the Placeholder.
- Create an instance of the MultiFieldStateController control class. The constructor takes no parameters.
- Assign the **ID** property.
- Add the MultiFieldStateController control to the **Controls** collection.

In this example, the MultiFieldStateController control is created with an **ID** of “MultiFieldStateController1”. It is added to Placeholder1.

[C#]

```
PeterBlum.DES.Web.WebControls.MultiFieldStateController vMFSC =
    new PeterBlum.DES.Web.WebControls.MultiFieldStateController();
vMFSC.ID = "MultiFieldStateController1";
Placeholder1.Controls.Add(vMFSC);
```

[VB]

```
Dim vMFSC As PeterBlum.DES.Web.WebControls.MultiFieldStateController = _
    New PeterBlum.DES.Web.WebControls.MultiFieldStateController()
vMFSC.ID = "MultiFieldStateController1"
Placeholder1.Controls.Add(vMFSC)
```

Guidelines for setting properties

- Design mode users can use the Properties Editor or the Expanded Properties Editor. (See “Expanded Properties Editor” in the **General Features Guide**.) The SmartTag  also offers some of the most important properties.
- Text entry users should add the properties into the <des:ControlClass> tag in this format:
propertyname="value"
- When setting a property programmatically, have a reference to the control’s object and set the property according to your language’s rules.

4. Determine which Condition class will provide the controls to monitor and selects whether to use the field states of **ConditionTrue** or **ConditionFalse**. You can use any Condition class supplied with DES or create your own using the CustomCondition class. See “About Conditions” in the **Validation User’s Guide** to choose a class.

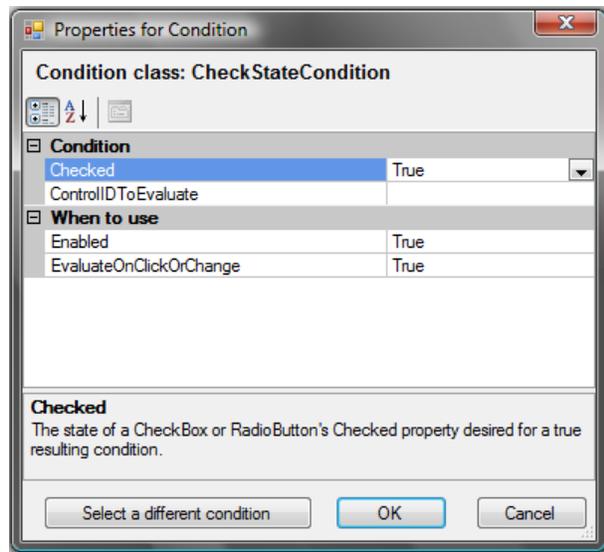
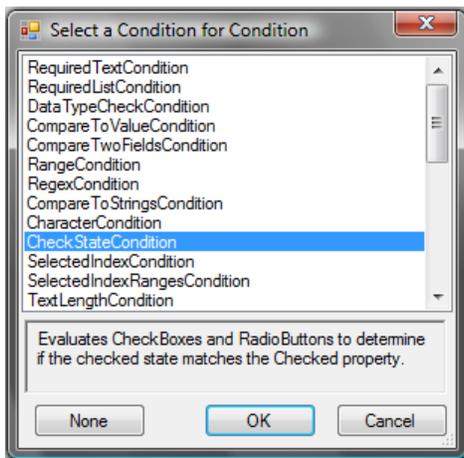
Note: You can use the Non-Data Entry Conditions, which monitor non-data oriented attributes of visibility, enabled, readonly and more. When you do, you will need to establish a control to monitor in the ExtraControlsToRunThisAction property.

Note: If you are changing the visibility on a web control, see “Changing Visibility on a Complex Control”.

5. Set the Condition object on the **Condition** property. Set its properties.

Visual Studio and Visual Web Developer Design Mode Users

The Properties Editor for the **Condition** property provides a window where you can select Condition objects and establish their properties.



- Select the Condition from the List and click **OK**.
- Establish the properties in the Properties grid.
- Click **OK**.

ASP.NET Text Formatting for the Conditions Property

You add the **Condition** as child of the <ConditionContainer> tag.

The following example represents a CheckStateCondition.

```
<des:MultiFieldStateController id="MultiFieldStateController1" runat="server">
  <ConditionContainer>
    <des:CheckStateCondition ControlIDToEvaluate="CheckBox1" />
  </ConditionContainer>
</des:MultiFieldStateController>
```

Notice that the **Condition** property never appears in the attributes of the `<des:MultiFieldStateController>` tag. (It will be added when using the Properties Editor but it's completely meaningless.) Instead, the `<ConditionContainer>` tag is a child of the `FieldStateController` control tag. That tag has no attributes. The child to `<ConditionContainer>` defines the class and all properties of the Condition:

```
<des:classname[all properties] />
```

- `des:classname` – Use any **Condition** class for the classname. If you create your own classes, you must declare the namespace using the `<% @REGISTER %>` tag at the top of the page.
- `[all properties]` – Enter the properties into the tag the same way you do for any other control.

Programmatically Adding Conditions

Here are the steps to set the **Condition**.

- Create an instance of the desired Condition. There is a constructor that takes no parameters.
Note: There are also constructors that take parameters representing some of the control's properties. Each demands an "owner" in the first parameter. That value must be the FieldStateController object.
- Assign property values.
- Assign the Condition object to the **Condition** property.

In this example, add the `CheckStateCondition`, which is checking the mark of `CheckBox1`, to `FieldStateController1`.

[C#]

```
PeterBlum.DES.Web.WebControls.CheckStateCondition vCond =
    new PeterBlum.DES.Web.WebControls.CheckStateCondition();
vCond.ControlToEvaluate = CheckBox1;
vCond.Checked = true;
FieldStateController1.Condition = vCond;
```

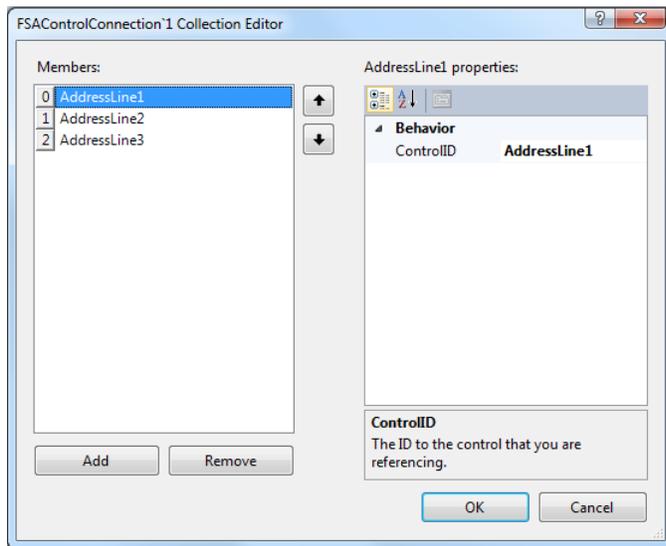
[VB]

```
Dim vCond As PeterBlum.DES.Web.WebControls.CheckStateCondition = _
    New PeterBlum.DES.Web.WebControls.CheckStateCondition()
vCond.ControlToEvaluate = CheckBox1
vCond.Checked = True
FieldStateController1.Condition = vCond
```

6. Usually, the Conditions dictate which field runs the `FieldStateController` when the user clicks or edits the control specified by the **ControlIDToEvaluate** or **SecondControlIDToEvaluate** properties. If you want to let the user click on a non-data entry field, like a label, button, or image, to run the `FieldStateController`, add the desired control to the **ExtraControlsToRunThisAction** property. Also use **ExtraControlsToRunThisAction** when the Condition is a Non-Data Entry Condition. *When **ExtraControlsToRunThisAction** is used, consider setting the **EvaluateOnClickOrChange** property to false on each condition.*
7. Create a list of control whose attributes will change within the **ControlConnections** property. Add `PeterBlum.DES.Web.WebControls.FSAControlConnection` objects. Assign the ID of the control to the **ControlID** property or a reference to the control in the **ControlInstance** property. *ControlInstance can only be assigned programmatically.*

Visual Studio and Visual Web Developer Design Mode Users

The Properties Editor for the **ControlConnections** property provides a window where add `PeterBlum.DES.Web.WebControls.FSAControlConnection` objects and assign the **ControlID** property to the ID of the control.



ASP.NET Text Formatting for the ControlConnections Property

ControlConnections is a type of collection. Therefore its ASP.NET Markup is nested as a series of `PeterBlum.DES.Web.WebControls.FSAControlConnection` objects within the `<ControlConnections>` tag. Each `PeterBlum.DES.Web.WebControls.FSAControlConnection` is a tag with `<des:FSAControlConnection>` followed by the **ControlID** property.

The following example represents the same `ControlConnections` shown in the editor window above.

```
<des:MultiFieldStateController id="MultiFieldStateController1" runat="server">
  <ControlConnections>
    <des:FSAControlConnection ControlID="AddressLine1">
    </des:FSAControlConnection>
    <des:FSAControlConnection ControlID="AddressLine2">
    </des:FSAControlConnection>
    <des:FSAControlConnection ControlID="AddressLine3">
    </des:FSAControlConnection>
  </ControlConnections>
</des:MultiFieldStateController>
```

Programmatically Adding Conditions

Use the `Add()` method on the **ControlConnections** property. Pass the ID, reference to the control, or `PeterBlum.DES.Web.WebControls.FSAControlConnection` object. In this example, "AddressLine1" is an ID to a control and `AddressLine2` is a reference to the control object.

[C#]

```
MultiFieldStateController1.ControlConnections.Add("AddressLine1");
MultiFieldStateController1.ControlConnections.Add(AddressLine2);
```

[VB]

```
MultiFieldStateController1.ControlConnections.Add("AddressLine1")
MultiFieldStateController1.ControlConnections.Add(AddressLine2)
```

8. Assign the field state attributes that will change, depending on the [Condition](#), in the [ConditionTrue](#) and [ConditionFalse](#) properties. **ConditionTrue** will be used when the Condition evaluates as “success”. **ConditionFalse** will be used when the Condition evaluates as “failed”.

The controls in **ControlConnections** will change their settings when the attribute differs between **ConditionTrue** and **ConditionFalse**. For example, **ConditionTrue.Visible** must differ from **ConditionFalse.Visible** for a visibility change to occur.

See “[Properties of ConditionTrue and ConditionFalse](#)”.

ASP.NET Text Formatting for the ConditionTrue and ConditionFalse Properties

The properties contained in **ConditionTrue** and **ConditionFalse** are added directly to the `<des:MultiFieldStateController>` tag. Use this format: `<des:MultiFieldStateController ConditionTrue-propertyname="value" ConditionFalse-propertyname="value">`. For example:

```
<des:MultiFieldStateController id="MultiFieldStateController1" runat="server"
  ConditionFalse-Visible="False">
```

9. Sometimes a field hidden or disabled by the MultiFieldStateController has an associated Validator whose error message is showing. That message is no longer appropriate. To remove it, first set up the [Enabler](#) property on the Validator to detect that the control is visible or enabled. Use the [VisibleCondition](#) or [EnabledCondition](#) class. See the “[Evaluating non-data value states of controls](#)”.

Then set the [ValidateChangedControls](#) property to true.

10. Here are some other considerations:

- If you are using an AJAX system to update this control, set the **InAJAXUpdate** property to true. Also make sure the PageManager control or AJAXManager object has been setup for AJAX. See “Using these Controls With AJAX” in the **General Features Guide**. *Failure to follow these directions can result in incorrect behavior and javascript errors.*
- This control does not preserve most of its properties in the ViewState, to limit its impact on the page. If you need to use the ViewState to retain the value of a property, see “The ViewState and Preserving Properties forPostBack” in the **General Features Guide**.
- If you encounter errors, see the “Troubleshooting” section for extensive topics based on several years of tech support’s experience with customers.
- See also “[Additional Topics for Using These Controls](#)”.

Properties of FieldStateController And MultiFieldStateController

Click on any of these topics to jump to them:

- ◆ [Invoke the Change Properties](#)
- ◆ [Controls To Change Properties](#)
- ◆ [Attributes To Change Properties](#)
 - [Properties of ConditionTrue and ConditionFalse](#)
- ◆ [Update Validators Properties](#)
- ◆ [When to Use the Control Properties](#)
- ◆ [Behavior Properties](#)

Invoke the Change Properties

The Properties Editor shows these properties in the “Invokes The Change” category.

- **Condition** (PeterBlum.DES.IBaseCondition) – Pick any Condition class and set its properties. Set “[The Condition classes](#)”. Be sure to be thorough with properties on the Condition that you select. It is easy to forget to assign the **ControlIDToEvaluate** or rules of the condition. When **ControlIDToEvaluate** is forgotten, expect to get a runtime error.

You can see how to set this property in design mode, in ASP.NET markup, and programmatically in step 5 of “[Adding the FieldStateController Control](#)”.

- **ExtraControlsToRunThisAction** (PeterBlum.DES.Connections) – Identifies additional controls and elements on the page that run this FieldStateController when clicked or changed.

The **Condition** already identifies controls through its **ControlIDToEvaluate** and **SecondControlIDToEvaluate** properties so this is rarely needed. The most common usages are:

- If you want the user to click on a non-data field, like a label, image, or button, use this property.
- [Conditions](#) that reference radiobuttons such as the CheckStateCondition. The browser only runs a radiobutton’s onclick event when the button is clicked. It doesn’t run when clicking another radiobutton unmarks the radiobutton specified in **ControlIDToEvaluate**. Assign the other radiobuttons to this property.
- If your CustomCondition uses controls that are not specified by **ControlIDToEvaluate** and **SecondControlIDToEvaluate**, add those controls to this property.

This property is a collection of PeterBlum.DES.Web.WebControls.ControlConnection objects. You can assign the control’s ID to the **ControlConnection.ControlID** property or a reference to the control in the **ControlConnection.ControlInstance** property. When using the **ControlID** property, the control must be in the same or an ancestor naming container. If it is in another naming container, use **ControlInstance**.

Here are some considerations:

- Be sure that the control assigned to this collection has the `runat="server"` property.
- You may want to disable the Conditions from setting up other fields from running the FieldStateController by setting the **Condition.EvaluateOnClickOrChange** property to `false`. For example, if your Condition evaluates a checkbox with the CheckBoxCondition, by default, CheckBoxCondition will run the FieldStateController whenever the checkbox is clicked.

ASP.NET Markup for the ExtraControlsToRunThisAction Property

ExtraControlsToRunThisAction is a type of collection. Therefore its ASP.NET Markup is nested as a series of child controls within the `<ExtraControlsToRunThisAction>` tag. Here is an example.

```
<des:FieldStateController id="FieldStateController1" runat="server">
  <ExtraControlsToRunThisAction>
    <des:ControlConnection ControlID="TextBox1" />
    <des:ControlConnection ControlID="Label1" />
  </ExtraControlsToRunThisAction>
</des:FieldStateController>
```

Programmatically adding to the ExtraControlsToRunThisAction Property

Use the `ExtraControlsToRunThisAction.Add()` method to add an entry. This overloaded method takes one parameter. Choose from the following:

- A reference to the control itself. This is the preferred form.
- A string giving the ID of the control. Do not use this when the control is not in the same naming container.
- An instance of the class `PeterBlum.DES.Web.WebControls.ControlConnection`.

This example shows how to update an existing `PeterBlum.DES.Web.WebControls.ControlConnection` and add a new entry. Suppose the ASP.NET code looks like the text above and the `Label1` control is not in the same or ancestor naming container. Also suppose the control referenced in the property `TextBox2` control must be added.

[C#]

```
uses PeterBlum.DES;
...
ControlConnection vConnection = (ControlConnection)
    FieldStateController1.ExtraControlsToRunThisAction[1];
vConnection.ControlInstance = Label1;
// add TextBox2. It can be either a control reference or its ID
FieldStateController1.ExtraControlsToRunThisAction.Add(TextBox2);
```

[VB]

```
Imports PeterBlum.DES
...
Dim vConnection As ControlConnection = _
    CType(FieldStateController1.ExtraControlsToRunThisAction(1), ControlConnection)
vConnection.ControlInstance = Label1
' add TextBox2. It can be either a control reference or its ID
FieldStateController1.ExtraControlsToRunThisAction.Add(TextBox2)
```

Controls To Change Properties

The Properties Editor shows these properties in the “Control To Change” category.

- **ControlIDToChange** (string) – *Only on FieldStateController*. The ID to the control whose state will be changed. If this ID is unassigned, the FieldStateController is disabled.

This ID must be in the same or an ancestor naming container. If it is in another naming container, use **ControlToChange**. *Be sure that the control whose ID is used here has the `runat=server` property.*

Note: If you are changing the visibility on a web control, see “[Changing Visibility on a Complex Control](#)”.

Nested controls syntax

Throughout DES, any property that accepts a control ID supports a special syntax that allows you to refer to a child property inside of the control you specify. That child property must provide the data entry control you are trying to validate, or have its own child property with that data entry control.

```
[ID to the container control].[property with the data entry control]
```

```
[ID to the container control].[property exposing another container control].[property with the data entry control]
```

If any parent object is also a `System.Web.UI.Controls` subclass, you can specify the ID of a control it contains. It will call `FindControls(id)` to find the child control.

```
[ID to the container control].[ID of the child control]
```

```
[ID to the container control].[property exposing another container control].[ID of the child control]
```

- **ControlToChange** (`System.Web.UI.Control`) – *Only on FieldStateController*. A reference to the control whose state will be changed. It is an alternative to **ControlIDToChange** that you must assign programmatically. It accepts controls in any naming container.

When programmatically assigning properties to a `FieldStateController`, if you have access to `Control To Change`'s object, it is better to assign it here than assign its ID to the **ControlIDToChange** property because DES operates faster using **ControlIDToChange**.

- **ControlConnections** (`PeterBlum.DES.Web.WebControls.ControlConnectionCollection`) - *Only on MultiFieldStateController*. This collection contains `PeterBlum.DES.Web.WebControls.FSAControlConnection` objects that defines a reference to a control, either by its ID or an object reference. When setting up the `MultiFieldStateController`, you should add a `FSAControlConnection` object for each control whose state will change. Set the `FSAControlConnection.ControlID` property to the ID of the control if it's in the same or ancestor naming container. Set the `FSAControlConnection.ControlInstance` property programmatically to the control in any other naming container.

Be sure that each control whose ID is used has the `runat=server` property.

You can see how to set this property in design mode, in the webform and programmatically in step 7 of “[Adding the MultiFSConCommand Control](#)”.

The `ControlConnectionCollection` class is subclassed from `System.Collections.ArrayList` and inherits all of its properties, methods and events.

Attributes To Change Properties

The Properties Editor shows these properties in the “Attributes to Change” category.

- **ConditionTrue** (PeterBlum.DES.StateSettings) – The values of attributes when the **Condition** evaluates as “success”. See “Properties of **ConditionTrue** and **ConditionFalse**” below. Each attribute will be compared to the same attribute in **ConditionFalse**. If the two values differ, the control’s state will be changed.

ASP.NET Text Formatting

The properties contained in **ConditionTrue** are added directly to the `<des:FieldStateController>` tag. Use this format: `<des:FieldStateController ConditionTrue-propertyname="value">`. For example:

```
<des:FieldStateController id="FieldStateController1" runat=server  
ConditionTrue-Visible="False">
```

- **ConditionFalse** (PeterBlum.DES.StateSettings) – The values of attributes when the **Condition** evaluates as “failed”. See “Properties of **ConditionTrue** and **ConditionFalse**”. Each attribute will be compared to the same attribute in **ConditionTrue**. If the two values differ, the control’s state will be changed.

ASP.NET Text Formatting

The properties contained in **ConditionFalse** are added directly to the `<des:FieldStateController>` tag. Use this format: `<des:FieldStateController ConditionFalse-propertyname="value">`. For example:

```
<des:FieldStateController id="FieldStateController1" runat=server  
ConditionFalse-Visible="False">
```

- **InvisiblePreservesSpace** (Boolean) – Determines if a control takes up space on the page when it is invisible. When the **ConditionTrue.Visible** or **ConditionFalse.Visible** property causes the field to be hidden, there are two ways the field can be hidden: Preserve the space of the element or remove the element entirely. This depends on the **display** style attribute.

When `true`, space is preserved. (The style is set to **visibility:hidden** with no change to the **display** style.)

When `false`, the element is removed. (The style is **visibility:hidden;display:none**.) When the element becomes visible once again, the **display** style is restored to its original value.

It defaults to `true`.

Properties of ConditionTrue and ConditionFalse

The `PeterBlum.DES.StateSettings` class is used by the **ConditionTrue** and **ConditionFalse** properties to define most of the attributes that you can change on a control. When these values are set differently between **ConditionTrue** and **ConditionFalse**, the field state is applied based on the `Condition`. Here are the properties of the `StateSettings` class:

- **Visible** (Boolean) – Sets the visibility. When `true`, it is visible. When `false`, it is hidden. `FieldStateController.InvisiblePreservesSpace` determines if it also retains or loses the space it occupies when it is hidden. It defaults to `true`.

This property changes **style:visibility**. If `InvisiblePreservesSpaces` is `true`, it also changes **style:display**.

Note: If you are changing the visibility on a web control, including Peter's Date Package controls, see "[Changing Visibility on a Complex Control](#)".

- **Enabled** (Boolean) – Sets the enabled state on controls that support the HTML property **disabled**. When `true`, the control is enabled (`disabled=false`). When `false`, the control is disabled.

Most browsers support the `disabled` attribute on data entry controls and buttons (`<input>`, `<select>` and `<textarea>` tags). Internet Explorer supports it on most tags. When you disable a `<div>`, for example, all data entry controls it contains appear disabled. Yet, textboxes may still be editable because they don't actually have their own `disabled` attribute set up. To provide cross browser compatibility, limit this property to data entry controls and buttons.

It defaults to `true`.

When the **ControlToChange** is not a data entry control, setting **Enabled** to `false` works differently. Normally the `Enabled` property adds the `disabled=true` attribute to the control. But that doesn't have any effect on Firefox and many other browsers. Instead, this feature switches to using a style sheet. It uses the style `"DES_DisabledLabel"` which is defined in **DES/Appearance/Interactive Pages/InteractivePages.css**.

Here is its definition:

```
.DES_DisabledLabel
{
    color: #C0C0C0; /* light gray */
}
```

Effectively it becomes the same as using `Enabled=true` and `CssClass="+DES_DisabledLabel"`. (The `+` character lets it merge its style with any already defined on the **ControlToChange**.)

You can change this class name by adding the `PageManager` control and setting its `DisabledLabelCssClass` property to the desired class name. You can also change it globally by editing the `DefaultDisabledLabelCssClass` property in the **Global Settings Editor**.

- **ReadOnly** (Boolean) – Sets the **readOnly** attribute on controls that support the HTML property **readOnly**: `TextBoxes`, `<input type='text'>`, and `<textarea>` tags.

When `true`, the control is read only. When `false`, the control is editable.

It defaults to `false`.

- **CssClass** (string) – Sets the `className` attribute to a style sheet class name. Since the style includes so many visual attributes, this is recommended over setting individual styles (which can be done in the **Other** property.)

When `"{ORIG}"`, it automatically uses the initial value found on the page.

When `""`, it sets the value to `""`.

If it starts with a `+` character, it merges with the existing style of the control. If it does not start with a `+` character, it is the only style applied to the control.

It defaults to `"{ORIG}"`.

- **FieldValue** (string) – Sets the `value` property of these form elements: `<input>`, `<textarea>`, or `<select>`. This includes `TextBoxes`, `CheckBoxes`, `RadioButtons`, `Buttons`, `Lists`, and `DropDownLists`. `Lists` and `DropDownLists` must have a matching value associated with an item in their lists to update the text shown.

When "{ORIG}", it automatically uses the initial value found on the page.

When "", it sets the value to "". If the ControlToChange is a TextBox that supports the **ValueWhenBlank** property, it will apply the **ValueWhenBlank** instead of "".

It defaults to "{ORIG}".

- **InnerHTML** (string) – Changes the **innerHTML** attribute. You can supply HTML or straight text. InnerHTML is the text contained inside of the tags: `<tag>innerHTML</tag>`.

InnerHTML can be very harmful. For example, if you assign it to a `<table>` tag, it will overwrite all `<tr>` and `<td>` tags it contains. You can easily input something that is not valid for the tag whose innerHTML you are modifying.

Good candidates are the Label and ``. If a Panel, `<div>`, TableCell, or `<td>` only contain text, they work well too.

When "{ORIG}", it automatically uses the initial value found on the page.

When "", it sets the value to "".

It defaults to "{ORIG}".

- **URL** (string) – Changes the **href** or **src** attributes on ``, `<input type=image>`, `<frame>`, `<iframe>`, and `<a>` tags. This includes the Image and HyperLink web controls.

Provide a valid URL. If a hyperlink uses a script in its **href** attribute, scripts should start with "javascript:".

When "", it automatically uses the initial value found on the page.

It defaults to "".

If your URL refers to a file within your web application, you can use the tilde "~" character as the first character to make your web application more portable. The "~" is replaced by the web application path. Normally during development, that folder is just below the domain root. In production, it is the domain root. For example, if you have an "Images" folder in your web application root, declare the **URL** property like this: "~/Images/file.gif".

- **Checked** (Boolean) – Changes the **checked** attribute of CheckBoxes, CheckBoxLists, and RadioButtons.

When `true`, it marks the control.

When `false`, it unmarks the control.

It defaults to `true`.

- **Other** (PeterBlum.DES.CSAttributeDesc) – Changes any attribute or style to a value you specify. You must know the name of the attribute or style and supply a legal value for it.

Any attribute you specify may not be compatible with all browsers. See [Microsoft's DHTML Reference](#). Each DHTML attribute topic will identify whether it is also supported in DOM by indicating its support in the W3C standards.

You do not have to define the **Other** property in both **FieldStateController.ConditionTrue** and **FieldStateController.ConditionFalse**. When you leave one without an **AttributeName**, DES will automatically capture the current value from the browser as the page is loaded and use it. You can also assign entirely different **AttributeNames** in **ConditionTrue** and **ConditionFalse**.

Note: If you want to specify more than one attribute or style, you must create one FieldStateController for each.

To set up the **Other** property, you must specify four values: **AttributeName**, **Value**, **DataType**, and **AttributeType**.

- **AttributeName** (string) – The name of the attribute or style. If "", the **Other** property is not used. It defaults to "".

Note: Attribute names are case sensitive. Enter them exactly as specified in the DHTML or DOM specification.

- **Value** (string) – The value to assign to the attribute or style. It defaults to "". When the **DataType** is **Boolean**, assign 'false' or 'true'. When the **DataType** is **Integer**, assign only digits.
- **DataType** (enum PeterBlum.DES.AttributeDataType) – Specifies the data type of the attribute or style. You must be sure to choose the correct type or you may get JavaScript errors at runtime. This enumerated type has these values:

- String – This is the default
- Integer
- Boolean
- **AttributeType** (enum PeterBlum.DES.AttributeType) – Specifies whether the attribute is on the field or on the style of the field. This enumerated type has these values:
 - Attribute – This is the default.
 - Style

ASP.NET Text Formatting for the Other Property

Here is the ASP.NET formatting for entering these properties:

```
<des:FieldStateController id="FieldStateController1" runat="server"
  ConditionTrue-Other-AttributeName="title"
  ConditionTrue-Other-Value="This is a tooltip"
  ConditionTrue-Other-AttributeType="Attribute"
  ConditionTrue-Other-DataType="String">

  ConditionFalse-Other-AttributeName="height"
  ConditionFalse-Other-Value="30"
  ConditionFalse-Other-AttributeType="Style"
  ConditionFalse-Other-DataType="Integer">
</des:FieldStateController>
```

- **ChangeStateFunctionName** (String) – Sometimes you need to write your own code to change attributes. For example, you have a complex control that needs to hide several related controls when it is hidden. You can write client-side code to handle this.

Use it to run JavaScript code that may do special things. See “[Client-Side Function: The Change State Function](#)” for details.

This property is name of a JavaScript function that will run.

ALERT: Many users make the mistake of assigning JavaScript code to this property. This will cause JavaScript errors.

GOOD: “MyFunction”. **BAD:** “MyFunction(,)” and “alert(‘stop it’)”.

Note: JavaScript is case sensitive. Be sure the value of this property exactly matches the function definition.

Update Validators Properties

The Properties Editor shows these properties in the “Update Validators” category.

Note: These properties are supported by the DES Validation Framework but not the native Validation Framework.

- **ValidateChangedControls** (Boolean) – When `true`, validate the controls after applying changes. It validates controls defined in **ControlIDToChange**, **ControlToChange**, and **ControlConnections** properties.

Generally this is done when the controls to change are data entry controls that have their visibility or enabled state changed. Usually your Validators will have their **Enabler** properties set to detect the visibility or enabled state of the control they are validating by using the `VisibleCondition` or `EnabledCondition`. See the “[Evaluating non-data value states of controls](#)”.

When `true`, validate. When `false`, do not validate, but you can still use **UseValidationGroup**.

It defaults to `false`.

- **UseValidationGroup** (Boolean) – Runs all Validators whose group matches the **ValidationGroup** property upon completion of the field state change. It behaves just like the user clicked a submit button for a particular group, including an update of `ValidationSummary` controls.

Set this to `true` when you have a Validator control associated with a control that this `FieldStateController` has hidden or disabled, usually in its **Enabler** property. As a result, the Validator will update itself.

This is an alternative to **ValidateChangedControls**.

It defaults to `false`.

The Validator control must include an **Enabler** property setting that detects the control it evaluates is hidden or disabled using the `VisibilityCondition` or `EnabledCondition`.

- **ValidationGroup** (string) – Defines a group name used by Validators that you want to run when this `FieldStateController` changes a field. When **UseValidatorGroup** is `true`, all Validators matching this group name are run after the field state has changed. This allows validators to remove themselves when the state no longer supports them. Use "*" to run through all groups. It defaults to "".
- **RevalidateOnly** (Boolean) – When using either [ValidateChangedControls](#) or [UseValidationGroup](#) properties, this determines if all validators are evaluated or just those that were already evaluated once.

Each validator knows if it has previously been evaluated on this page, even if a postback occurs (so long as the server side calls `PeterBlum.DES.Globals.WebFormDirector.Validate()` or on an AJAX callback. This can improve the user interface by avoiding having error messages appear on fields that are part of the validation group but have yet to be edited.

Set this to `true` when you don't want a validator to appear on fields the user hasn't edited based on the `FieldStateControllers` action.

Set it to `false` to include all validators determined by the **ValidateChangedControls** or **UseValidationGroup** properties.

It defaults to `true`.

When to Use the Control Properties

The Properties Editor shows these properties in the “When To Use” category.

- **Enabled** (Boolean) – If you have added the FieldStateController to the page but need to disable it completely, set this property to `false`. It defaults to `true`.

If the FieldStateController references another control whose **Visible** property is set to `false`, the FieldStateController is automatically disabled. This is because when **Visible** is `false`, the web control does not generate any HTML and the **Condition** cannot evaluate it or the FieldStateController cannot modify it on the client-side.

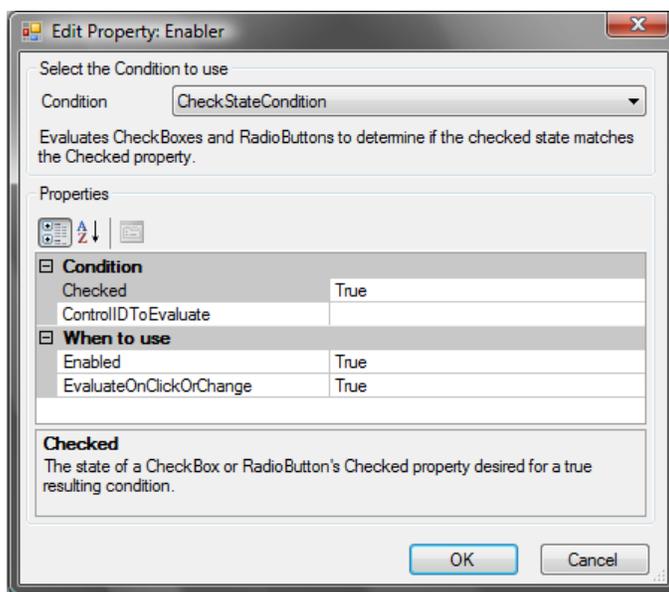
- **Enabler** (PeterBlum.DES.IBaseCondition) – There are times when a FieldStateController should be disabled. For example, don’t change the state because the textbox is invisible or a checkbox is unmarked. These rules are formed by **Condition** classes with the **Enabler** property on each FieldStateController. By default, the **Enabler** property is set to “None”, where it doesn’t disable the control. You can set it to any Condition, including those you may create programmatically.

Consider these issues when using the Enabler:

- Most Conditions have a property called **EvaluateOnClickOrChange** which defaults to `true`. Change it to `false` when using it in an Enabler.
- Do not use this to detect a control whose **Visible** property is set to `false`. Such a control does not create HTML for the client-side to use. Instead, set the **Enabled** property to `false` when the control is invisible.

Visual Studio and Visual Web Developer Design Mode Users

The Properties Editor offers this window to select a Condition and to edit its properties.



1. Select the Condition from the DropDownList. See the section “[The Condition classes](#)”.
2. Establish the properties in the Properties grid.
*Reminder: Be sure to set **EvaluateOnClickOrChange** to false on all Conditions within the Enabler as shown.*
3. Click **OK**.

Adding an Enabler with ASP.NET Markup

If you want to enter the **Enabler** property and its child properties into the web form using the HTML mode, there are special considerations. The format is very unusual, in part because the .Net framework doesn't support changing the class of a property (polymorphism) without an interesting hack.

Here is the FieldStateController with the **Enabler** set to the CheckStateCondition.

```
<des:FieldStateController id="FieldStateController1" runat="server"
  ControlIDToChange="Span1" >

  <EnablerContainer>
    <des:CheckStateCondition ControlIDToEvaluate="CheckBox1"
      EvaluateOnClickOrChange="false" >
    </des:CheckStateCondition>
  </EnablerContainer>

</des:FieldStateController>
```

*Reminder: Be sure to set **EvaluateOnClickOrChange** to false on all Conditions within the Enabler as shown.*

Notice that the **Enabler** property never appears in the attributes of the <des:FieldStateController> tag. (*It will be added when using the Properties Editor but it's completely meaningless.*) Instead, the <EnablerContainer> tag is a child of the FieldStateController tag. That tag never has any attributes. The child to <EnablerContainer> defines the class and all properties of the Condition:

```
<des:classname [all properties] />
```

- o des:classname – Use any Condition class for the classname. If you create your own classes, you must declare the namespace using the <% @REGISTER %> tag at the top of the page.
- o [all properties] – Enter the properties into the tag the same way you do for any other control.

Adding an Enabler Programmatically

Here are the steps to set the **Enabler**.

1. Create an instance of the desired Condition. There is a constructor that takes no parameters.

Note: There are also constructors that take parameters representing some of the control's properties. Each demands an "owner" in the first parameter. That value must be the FieldStateController object.

2. Assign property values.

Reminder: Be sure to set EvaluateOnClickOrChange to false on all Conditions within the Enabler.

3. Assign the Condition object to the **Enabler** property.

In this example, add the CheckStateCondition, which is checking CheckBox1, to FieldStateController1.

[C#]

```
PeterBlum.DES.Web.WebControls.CheckStateCondition vCond =
  new PeterBlum.DES.Web.WebControls.CheckStateCondition();
vCond.ControlToEvaluate = CheckBox1;
vCond.EvaluateOnClickOrChange = false;
FieldStateController1.Enabler = vCond;
```

[VB]

```
Dim vCond As PeterBlum.DES.Web.WebControls.CheckStateCondition = _
  New PeterBlum.DES.Web.WebControls.CheckStateCondition()
vCond.ControlToEvaluate = CheckBox1
vCond.EvaluateOnClickOrChange = False
FieldStateController1.Enabler = vCond
```

Behavior Properties

The Properties Editor shows these properties in the “Behavior” category.

- **InAJAXUpdate** (Boolean) – When the page uses AJAX callbacks to add, update, or remove this control, set this to `true`. It defaults to `false`.

In addition, if any of these properties identify a non-DES control that participates in the AJAX callback, set this to `true`:

- **ControlIDToChange** and **ControlToChange**
- Controls in **ControlConnections**.
- **Condition**. Look at the **ControlIDToEvaluate** and **SecondControlIDToEvaluate**.
- **Enabler**. Look at the **ControlIDToEvaluate** and **SecondControlIDToEvaluate**.
- **ExtraControlsToRunThisAction**.

Note: This is only needed for non-DES controls. DES controls will tell the FieldStateController if their own IsAJAXUpdate property is true.

See “Using These Controls with AJAX” in the **General Features Guide**.

- **RunOnPageLoad** (Boolean) – When `true`, apply the field state based on the [Condition](#) as the page is loading. This establishes an initial appearance. You normally do not set any properties on your controls that are controlled by the FieldStateController.

Recommended in most cases. Set it to `false` only when you only want the controls being monitored to apply the field state.

It defaults to `true`.

- **UpdateWhileEditing** (Boolean) – Determines if the FieldStateController is triggered as the user types into a textbox that it uses to evaluate its [Condition](#). By default, it does not and only triggers when focus leaves the textbox. Set this to `true` to evaluate the FieldStateController with each keystroke.

It defaults to `false`.

- **SupportClientSideLookupByID** (Boolean) - Allows JavaScript programmers to get to the client-side representation of the FieldStateController object by the ClientID of the owner control.

Use the client-side function `DES_FindAObById(ClientID)` to search for the “Action object” that matches the ClientID you specify. That function will return `null` if not found.

Use the Action object to invoke the FieldStateController as if the user changed a field associated with it. Pass the Action object to the client-side function `DES_DoAction(action object)`. The Action object contains the result of evaluating the [Condition](#) in the property **CondResult**. It is an integer whose values are:

1=success; 0=failed; -1=cannot evaluate.

Here is a function that invokes the FieldStateController and returns the result:

```
<script type='text/javascript' language='javascript'>
function InvokeFSC(pClientID)
{
    var vAO = DES_FindAObById(pClientID);
    DES_DoAction(vAO);
    return vAO.CondResult; //1=success; 0=failed; -1=cannot evaluate
}
</script>
```

Suppose you have a Button called Button1 that will be used to invoke FieldStateController1. Here is how you set it up to call InvokeFSC in `Page_Load()`:

```
Button1.Attributes.Add("onclick",
    "InvokeFSC('" + FieldStateController1.ClientID + "');return false;")
```

When **SupportClientSideLookupByID** is true, two things happen:

1. The ID is written as a property, CID, into the control. (It isn't written by default to avoid adding excess text to the page.)
2. If the **Enabled** property is false, normally no code is written to the client side. This is overridden and code is generated so users can toggle the **Enabled** property.

It defaults to false.

- **ViewStateMgr** (PeterBlum.DES.Web.WebControls.ViewStateMgr) – Enhances the ViewState on this control to provide more optimal storage and other benefits. Normally, the properties of this control and its segments are not preserved in the ViewState. When working in ASP.NET markup, define a pipe delimited string of properties in the **PropertiesToTrack** property. When working in code, call `ViewStateMgr.TrackProperty("propertyname")` to save the property. Individual segments have a similar method: `TrackPropertyInViewState("propertyname")`.

For more details, see “The ViewState and Preserving Properties forPostBack” in the **General Features User's Guide**.

- **PropertiesToTrack** (string) – A pipe delimited list of properties to track. Designed for use in markup and the properties editor. The ViewState is not automatically used by most of these properties. To include a property, add it to this pipe delimited list.

For example, "Group|MayMoveOnClick".

When working programmatically, use `ViewStateMgr.TrackProperty("PropertyName")`.

FSCOnCommand and MultiFSCOnCommand Controls

The `PeterBlum.DES.Web.WebControls.FSCOnCommand` and `PeterBlum.DES.Web.WebControls.MultiFSCOnCommand` controls are `FieldStateControllers` that the user fires when they click on a button or other command. While the [FieldStateController](#) and [MultiFieldStateController](#) Controls apply two states, based on a [Condition](#), these two controls apply only one. A typical case is to have a `SelectAll` button that marks all checkboxes in a `CheckBoxList`.

`FSCOnCommand` and `MultiFSCOnCommand` are actually subclasses of `FieldStateController` and `MultiFieldStateController`. They were created to simplify the setup of a special case, where a button is clicked and a single state is applied. You assign the controls that are the commands, the controls to change, and the changes to make.

These controls do most of their work on the client-side. If the browser does not support the client-side scripting needed to run a `FieldStateController`, it is disabled. That will leave your controls with the state that you define in their properties on the server side.

The `FieldStateController` adds no HTML to your page as it does it work through JavaScript. You can add them anywhere to your web form.

Click on any of these topics to jump to them:

- ◆ [Features](#)
- ◆ [Using the FSCOnCommand Controls](#)
 - [Controls that run the FSCOnCommand control](#)
 - [Attribute Values To Change](#)
 - [Updating Validators](#)
 - [Changing Visibility on a Complex Control](#)
 - [Selectively Running the Control](#)
 - [Extending the Attributes with Your Own Code](#)
- ◆ [Example: FSCOnCommand](#)
- ◆ [Example: MultiFSCOnCommand](#)
- ◆ [Adding the FSCOnCommand Control](#)
- ◆ [Adding the MultiFSCOnCommand Control](#)
- ◆ [Properties of FSCOnCommand And MultiFSCOnCommand](#)
- ◆ [Online examples](#)

Features

The FSCOnCommand and MultiFSCOnCommand controls make changes to the HTML elements on your page based on a button click. They can change almost any element on your page:

- Show or hide
- Enable or disable form controls (textboxes, lists, buttons, etc)
- Change the ReadOnly state of a textbox
- Change the style sheet class name, which can deliver an entirely different appearance through style sheets
- Change the textual value of a textbox
- Change the value of the selected element in a listbox or dropdownlist
- Change the “innerHTML” of a Label, , or any other HTML tag that supports “innerHTML”
- Change the URL associated with hyperlinks, images and other HTML tags that have an href= or src= attribute.
- Change the mark in a checkbox or radiobutton
- Mark or unmark all checkboxes in a CheckBoxList
- Change the value of any document object model attribute that has a datatype of string, boolean or integer
- Change the value of any document object model style
- Run your own JavaScript to handle special situations

You can see how powerful these controls are. You only need to set properties on the controls and you have enhanced your user interface.

The FSCOnCommand and MultiFSCOnCommand are invoked by a click on a “command” such as a button and apply a single field state. Any HTML tag that supports the “onclick” event can be used to fire it. Examples:

- You have a CheckBoxList and use a button titled “Select All” to mark all checkboxes.
- In a tabbed interface, an image that represents a “tab” can show or hide a panel containing the tab’s “page”

Once these controls have done their task, they can optionally run the Validators on the field whose state was changed or run an entire validation group.

Using the FSCOnCommand Controls

There are three elements that always must be set up on a FSCOnCommand or MultiFSCOnCommand control:

- The “commands” - controls that run the FSCOnCommand control when clicked
- The control or controls whose attributes that you want to change
- The attribute values that will change

Click on any of these topics to jump to them:

- ◆ [Controls that run the FSCOnCommand control](#)
- ◆ [Controls To Change](#)
- ◆ [Attribute Values To Change](#)
- ◆ [Updating Validators](#)
- ◆ [Changing Visibility on a Complex Control](#)
- ◆ [Selectively Running the Control](#)
- ◆ [Extending the Attributes with Your Own Code](#)
- ◆ [Online examples](#)

Controls that run the FSCOnCommand control

Add an element to the page that will run the FSCOnCommand Control. It can be almost any HTML tag that supports the client-side onclick event. Buttons, Labels, Tables, Images, Hyperlinks and more are all usable. So create the interface that you prefer. If you use an HTML tag in your web form, be sure it has an ID and `runat=server`.

Special concerns:

- The Button class can select between submit and non-submit styles with the **UseSubmitBehavior** property. Set it to `false`. In addition, set the button's **OnClick** property to `return: false;`.
- When using a HyperLink, set the **NavigationUrl** property to `javascript: return false;`.
- When using a `` tag, set the **href** attribute to `javascript: return false;`

Assign the command control to the **ControlIDToRunThisAction** property.

If you have several command controls that run the same FSCOnCommand control, add them to the **ExtraControlsToRunThisAction** property.

Controls To Change

You must assign the ID or an object reference to the control(s) that you want to change. The FSCOnCommand control requires one control to change. Use **ControlIDToChange** when you have an ID or **ControlToChange** when you have an object reference.

The MultiFSCOnCommand control changes as many controls as you want. Add `PeterBlum.DES.Web.WebControls.FSAControlConnection` objects to the **ControlConnections** property. The `PeterBlum.DES.Web.WebControls.FSAControlConnection` class can be assigned an ID to its **ControlID** property and an object reference to its **ControlInstance** property.

Note: All controls must have an ID and `runat=server` property.

Attribute Values To Change

With the following properties, you can change any of these attributes:

- **VisibleState** – changes the **style:visibility** and **style:display** attributes. Use True, False, or Ignore for its value. (Programmers will find these on the enumerated type `PeterBlum.DES.TrueFalseIgnore`.) For a special case, see “[Changing Visibility on a Complex Control](#)”.
- **EnabledState** – changes the **disabled** attribute on the controls that support it (which varies by browser). Use True, False, or Ignore for its value. (Programmers will find these on the enumerated type `PeterBlum.DES.TrueFalseIgnore`.)
- **ReadOnly** – changes the **readOnly** attribute on textboxes. Use the `ReadOnly` property. Use True, False, or Ignore for its value. (Programmers will find these on the enumerated type `PeterBlum.DES.TrueFalseIgnore`.)
- **CssClass** – changes the style sheet class name.
- **FieldValue** – changes the **value** attribute of `<input>`, `<textarea>`, and `<select>` tags
- **InnerHTMLState** – changes the **innerHTML** attribute on any control. **InnerHTML** is found in tags that permit contents between their begin and end tags, like this: `<tag>innerHTML</tag>`. A Label, Panel, and TableCell are web controls that generate tags that support InnerHTML (``, `<div>`, and `<td>` respectively.)
- **URL** – changes the **href** or **src** attribute to a new URL on ``, `<input type=image>`, `<frame>`, `<iframe>`, and `<a>` tags.
- **Checked** – changes the **checked** attribute on a checkbox or radiobutton. Use True, False, or Ignore for its value. (Programmers will find these on the enumerated type `PeterBlum.DES.TrueFalseIgnore`.)
- If you know the name and legal values of an attribute or style, there is an all-purpose property, **Other**, which will modify the attribute or style with the value.

For details on the above properties, see “[Attributes To Change Properties](#)”.

In addition, you can supply a JavaScript function to run to handle unusual cases:

- A third party custom control uses its own JavaScript to adjust its properties.
- The control is created by JavaScript on the client side and has no server-side ID.
- A calculation must be performed before the setting can be determined.

You assign your function to the **ChangeStateFunctionName** property. See “[Client-Side Function: The Change State Function](#)”.

Updating Validators

Sometimes a field hidden or disabled by the FSCOnCommand control has an associated Validator whose error message is showing. That message is no longer appropriate. To remove it, first set up the **Enabler** property on the Validator to detect that the control is visible or enabled. Use the VisibleCondition or EnabledCondition class. See “[Evaluating non-data value states of controls](#)”.

Then set the **ValidateChangedControls** property to true.

If the FSCOnCommand control affects controls that are used in the **Enabler** properties of other Validators, let it run all Validators in the validation group associated with those Validators. Set the **UseValidatorGroup** property to true and the group name of the Validator in the **ValidatorGroup** property.

Use the **RevalidateOnly** property to evaluate only validators that have previously been evaluated on the page. This prevents validator errors from appearing further down the page, where the user has not edited.

Changing Visibility on a Complex Control

Some web controls include a number of HTML tags. The control's ID property may refer to just one of the HTML tags it generates. If you use the `FSCOnCommand` to show and hide that control by its ID, you will only show or hide the one tag associated with the control ID.

Example: The `DateTextBox` control

ALERT: *DES's own controls – including the `DateTextBox` - do not need the this technique as they automatically account for the issue here. This is merely an example.*

The textboxes in Peter's Date And Time use multiple HTML tags. For example, the `DateTextBox` has an `` tag to the right of the textbox which is used to toggle a popup calendar. The textbox in these controls is associated with the control's ID.

```
<input type='text' id='control_clientid' /><img src='calendar.jpg' />
```

If you assign the `ControlIDToChange` property to the `DateTextBox`'s ID, it will only show and hide the textbox, leaving the image visible.

Solution

Look at the HTML output of any web control to see which HTML tag is assigned the ID (specifically the `ClientID` property value.) If that tag encloses all HTML for that control, you can use the web control's ID with the `FSCOnCommand.ControlIDToChange` property.

If the tag does not enclose the control, add a `` or `<div>` tag around the web control. Set the `runat=server` property and assign an ID value. Set the `FSCOnCommand.ControlIDToChange` property to the ID of that `` or `<div>` tag.

Example: `DateTextBox`

ALERT: *DES's own controls – including the `DateTextBox` - do not need the this technique as they automatically account for the issue here. This is merely an example.*

```
<span runat="server" id="DateTextBox1Container">
  <des:DateTextBox runat="server" id="DateTextBox1" />
</span>

<des:FSCOnCommand runat="server" id="FSC1"
  ControlIDtoChange="DateTextBox1Container" VisibleState="true">
  <Condition>
    [you determine this]
  </Condition>
</des:FSCOnCommand>
```

Selectively Running the Control

Sometimes, you will want the command button to run only in certain situations. For example, if you use a checkbox as a command to mark a CheckBoxList only when the user marks the checkbox, the FSCOnCommand should not run when the user unmarks the checkbox.

Use the **Enabler** property to establish a **Condition**. When the Condition evaluates as “success”, the FSCOnCommand runs. When it evaluates as “failed”, it does not run.

Example

This is a checkbox that marks a CheckBoxList when the user marks the checkbox.

Select all Bananas Oranges Peaches

```
<asp:CheckBox id=CheckBox1 runat="server" Text="Select all"></asp:CheckBox>
<asp:CheckBoxList id=CheckBoxList1 runat="server"
  RepeatDirection="Horizontal" RepeatLayout="Flow">
  <asp:ListItem Value="Bananas">Bananas</asp:ListItem>
  <asp:ListItem Value="Oranges">Oranges</asp:ListItem>
  <asp:ListItem Value="Peaches">Peaches</asp:ListItem>
</asp:CheckBoxList>
<des:FSCOnCommand id=FSCOnCommand1 runat="server"
  ControlIDToChange="CheckBoxList1" ControlIDToRunThisAction="CheckBox1"
  Checked="True" >
  <EnablerContainer>
    <des:CheckStateCondition ControlIDToEvaluate="CheckBox1">
    </des:CheckStateCondition>
  </EnablerContainer>
</des:FSCOnCommand>
```

Example: FSCOnCommand

Suppose you have a "Select All" button that marks all checkboxes in a CheckBoxList.

```
<des:button id="SelectAllBtn" runat="server" Text="Select All"
    UseSubmitBehavior="false" />
<br />
Check all <asp:CheckBoxList id=CheckBoxList1 runat="server"
    RepeatLayout="Flow" RepeatDirection="Horizontal">
    <asp:ListItem Value="1">1</asp:ListItem>
    <asp:ListItem Value="2">2</asp:ListItem>
    <asp:ListItem Value="3">3</asp:ListItem>
</asp:CheckBoxList>
<des:FSCOnCommand id="FSCOnCommand1" runat="server"
    ControlIDToRunThisAction="SelectAllBtn" ControlIDToChange="CheckBoxList1"
    Checked="True"></des:FSCOnCommand>
```

Example: MultiFSCOnCommand

This is a modification of the previous example. Instead of having a CheckBoxList, it has individual CheckBoxes and uses the MultiFSCOnCommand to update them.

```
<des:button id="SelectAllBtn" runat="server" Text="Select All"
  UseSubmitBehavior="false" />
<br />
<asp:CheckBox id="CheckBox1" runat="server" Text="1" />
<asp:CheckBox id="CheckBox2" runat="server" Text="2" />
<asp:CheckBox id="CheckBox3" runat="server" Text="3" />
<des:MultiFSCOnCommand id="MultiFSCOnCommand1" runat="server"
  ControlIDToRunThisAction="SelectAllBtn" Checked="True">
  <ControlConnections>
    <des:FSAControlConnection ControlID="CheckBox1" />
    <des:FSAControlConnection ControlID="CheckBox2" />
    <des:FSAControlConnection ControlID="CheckBox3" />
  </ControlConnections>
</des:MultiFSCOnCommand>
```

Adding the FSCOnCommand Control



These steps ask you to jump around the document using clicks on links. Adobe Reader offers a **Previous View** command to return to the link. Look for this in the Adobe Reader (shown v6.0)

1. Prepare the page for DES controls. See “Preparing a page for DES controls” in the **General Features Guide**. It covers issues like style sheets, AJAX, and localization.
2. Set up the controls whose attributes will change. You don’t have to set the properties that will be changed to their initial (“off”) value. The FSCOnCommand control will do that for you, unless you set **SetInitialAppearance** to *false*.
3. Add the “command” control, such as the button. Special concerns:
 - The Button class can select between submit and non-submit styles with the **UseSubmitBehavior** property. Set it to *false*. In addition, set the button’s **OnClickClient** property to `return: false;`.
 - When using a HyperLink, set the **NavigationUrl** property to `javascript:return false;`.
 - When using a `` tag, set the **href** attribute to `javascript:return false;`
4. Add the FSCOnCommand control to the page. Its location does not matter as it contributes no HTML to the page.

Visual Studio and Visual Web Developer Design Mode Users

Drag the FSCOnCommand control from the Toolbox onto your web form. It will look like this:



Text Entry Users

Add the control (inside the `<form>` area):

```
<des:FSCOnCommand id="[YourControlID]" runat="server" />
```

Programmatically Creating the Control

- Identify the control which you will add the FSCOnCommand control to its **Controls** collection. Like all ASP.NET controls, the FSCOnCommand control can be added to any control that supports child controls, like Panel, UserControl, or TableCell. If you want to add it directly to the Page, first add a Placeholder and use the Placeholder.
- Create an instance of the FSCOnCommand control class. The constructor takes no parameters.
- Assign the **ID** property.
- Add the FSCOnCommand control to the **Controls** collection.

In this example, the FSCOnCommand control is created with an **ID** of “FSCOnCommand1”. It is added to Placeholder1.

[C#]

```
PeterBlum.DES.Web.WebControls.FSCOnCommand vFSC =
    new PeterBlum.DES.Web.WebControls.FSCOnCommand();
vFSC.ID = "FSCOnCommand1";
Placeholder1.Controls.Add(vFSC);
```

[VB]

```
Dim vFSC As PeterBlum.DES.Web.WebControls.FSCOnCommand = _
    New PeterBlum.DES.Web.WebControls.FSCOnCommand()
vFSC.ID = "FSCOnCommand1"
Placeholder1.Controls.Add(vFSC)
```

Guidelines for setting properties

- Design mode users can use the Properties Editor or the Expanded Properties Editor. (See “Expanded Properties Editor” in the **General Features Guide**.) The SmartTag  also offers some of the most important properties.
- Text entry users should add the properties into the `<des:ControlClass>` tag in this format:
propertyname="value"
- When setting a property programmatically, have a reference to the control’s object and set the property according to your language’s rules.

5. Assign the command control to the **ControlIDThatRunsThisAction** property. If you have several command controls, add the rest to the **ExtraControlsToRunThisAction** property.
6. Set the ID of control whose attributes will change with the **ControlIDToChange** property or a reference to that control with the **ControlToChange** property.
7. Assign the attributes to change. They are in the **VisibleState**, **EnabledState**, **ReadOnly**, **CssClass**, **FieldValue**, **InnerHTMLState**, **URL**, **Checked**, and **Other** properties.
 - **VisibleState**, **EnabledState**, **ReadOnly**, **Checked** use an enumerated type that has three values: Ignore, False, and True. They default to Ignore. Change to False or True as needed. (Programmers will find these on the enumerated type `PeterBlum.DES.TrueFalseIgnore`.)
Note: If you are changing the visibility on a web control, see “[Changing Visibility on a Complex Control](#)”.
 - **CssClass**, **FieldValue**, and **InnerHTMLState** default to “{ORIG}” and will not be applied until you change their value.
8. Sometimes a field hidden or disabled by the FSCOnCommand control has an associated Validator whose error message is showing. That message is no longer appropriate. To remove it, first set up the **Enabler** property on the Validator to detect that the control is visible or enabled. Use the **VisibleCondition** or **EnabledCondition** class. See “[Evaluating non-data value states of controls](#)”.
Then set the **ValidateChangedControls** property to `true`.
9. When the page uses AJAX to update any of its controls, you **must** do some additional setup. See “Here are some other considerations”:
 - If you are using an AJAX system to update this control, set the **InAJAXUpdate** property to `true`. Also make sure the PageManager control or AJAXManager object has been setup for AJAX. See “Using these Controls With AJAX” in the **General Features Guide**. *Failure to follow these directions can result in incorrect behavior and javascript errors.*
 - This control does not preserve most of its properties in the ViewState, to limit its impact on the page. If you need to use the ViewState to retain the value of a property, see “The ViewState and Preserving Properties forPostBack” in the **General Features Guide**.
 - If you encounter errors, see the “[Troubleshooting](#)” section for extensive topics based on several years of tech support’s experience with customers.
 - See also “[Additional Topics for Using These Controls](#)”.

Adding the MultiFSCOnCommand Control

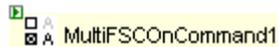


These steps ask you to jump around the document using clicks on links. Adobe Reader offers a **Previous View** command to return to the link. Look for this in the Adobe Reader (shown v6.0)

1. Prepare the page for DES controls. See “Preparing a page for DES controls” in the **General Features Guide**. It covers issues like style sheets, AJAX, and localization.
2. Set up the controls whose attributes will change. You don’t have to set the properties that will be changed to their initial (“off”) value. The FSCOnCommand control will do that for you, unless you set **SetInitialAppearance** to *false*.
3. Add the “command” control, such as the button. Special concerns:
 - The Button class can select between submit and non-submit styles with the **UseSubmitBehavior** property. Set it to *false*.
 - When using a HyperLink, set the **NavigationUrl** property to "javascript: return false;".
 - When using a `` tag, set the **href** attribute to "javascript: return false;".
4. Add the MultiFSCOnCommand control to the page. Its location does not matter as it contributes no HTML to the page.

Visual Studio and Visual Web Developer Design Mode Users

Drag the MultiFSCOnCommand control from the Toolbox onto your web form. It will look like this:



When you view the control in design mode, sometimes you will see the following:

Text Entry Users

Add the control (inside the `<form>` area):

```
<des:MultiFSCOnCommand id="[YourControlID]" runat="server" />
```

Programmatically Creating the Control

- Identify the control which you will add the MultiFSCOnCommand control to its **Controls** collection. Like all ASP.NET controls, the MultiFSCOnCommand control can be added to any control that supports child controls, like Panel, User Control, or TableCell. If you want to add it directly to the Page, first add a Placeholder and use the Placeholder.
- Create an instance of the MultiFSCOnCommand control class. The constructor takes no parameters.
- Assign the **ID** property.
- Add the MultiFSCOnCommand control to the **Controls** collection.

In this example, the MultiFSCOnCommand control is created with an **ID** of “MultiFSCOnCommand1”. It is added to Placeholder1.

[C#]

```
PeterBlum.DES.Web.WebControls.MultiFSCOnCommand vMFSC =
    new PeterBlum.DES.Web.WebControls.MultiFSCOnCommand();
vMFSC.ID = "MultiFSCOnCommand1";
Placeholder1.Controls.Add(vMFSC);
```

[VB]

```
Dim vMFSC As PeterBlum.DES.Web.WebControls.MultiFSCOnCommand = _
    New PeterBlum.DES.Web.WebControls.MultiFSCOnCommand()
vMFSC.ID = "MultiFSCOnCommand1"
Placeholder1.Controls.Add(vMFSC)
```

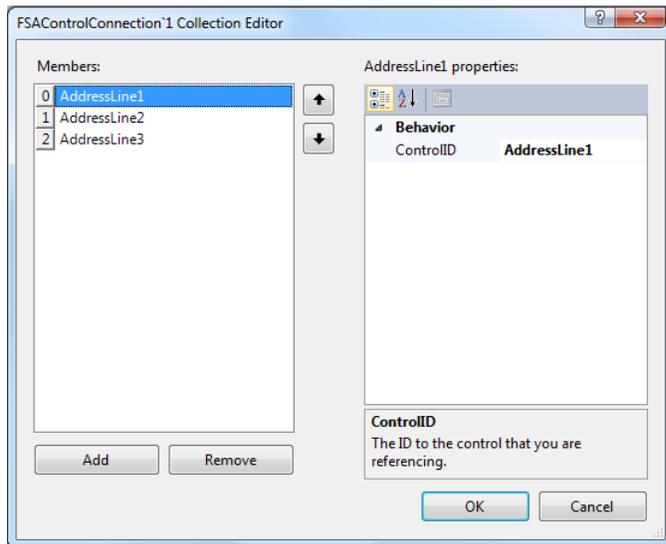
Guidelines for setting properties

- Design mode users can use the Properties Editor or the Expanded Properties Editor. (See “Expanded Properties Editor” in the **General Features Guide**.) The SmartTag  also offers some of the most important properties.
- Text entry users should add the properties into the `<des:ControlClass>` tag in this format:
propertyname="value"
- When setting a property programmatically, have a reference to the control’s object and set the property according to your language’s rules.

5. Assign the command control to the **ControlIDThatRunsThisAction** property. If you have several command controls, add the rest to the **ExtraControlsToRunThisAction** property.
6. Create a list of control whose attributes will change within the **ControlConnections** property. Add `PeterBlum.DES.Web.WebControls.FSAControlConnection` objects. Assign the ID of the control to the **ControlID** property or a reference to the control in the **ControlInstance** property. *ControlInstance can only be assigned programmatically.*

Visual Studio and Visual Web Developer Design Mode Users

The Properties Editor for the **ControlConnections** property provides a window where add `PeterBlum.DES.Web.WebControls.FSAControlConnection` objects and assign the **ControlID** property to the ID of the control.



ASP.NET Text Formatting for the ControlConnections Property

ControlConnections is a type of collection. Therefore its ASP.NET Markup is nested as a series of `PeterBlum.DES.Web.WebControls.FSAControlConnection` objects within the `<ControlConnections>` tag. Each `PeterBlum.DES.Web.WebControls.FSAControlConnection` is a tag with `<des:FSAControlConnection>` followed by the **ControlID** property.

The following example represents the same **ControlConnections** shown in the editor window above.

```
<des:MultiFSCOnCommand id="MultiFSCOnCommand1" runat="server">
  <ControlConnections>
    <des:FSAControlConnection ControlID="AddressLine1" />
    <des:FSAControlConnection ControlID="AddressLine2" />
    <des:FSAControlConnection ControlID="AddressLine3" />
  </ControlConnections>
</des:MultiFSCOnCommand>
```

Programmatically Adding Conditions

Use the `Add()` method on the `ControlConnections` property. Pass the ID, reference to the control, or `PeterBlum.DES.Web.WebControls.FSAControlConnection` object. In this example, “AddressLine1” is an ID to a control and `AddressLine2` is a reference to the control object.

[C#]

```
MultiFSCOnCommand1.ControlConnections.Add("AddressLine1");  
MultiFSCOnCommand1.ControlConnections.Add(AddressLine2);
```

[VB]

```
MultiFSCOnCommand1.ControlConnections.Add("AddressLine1")  
MultiFSCOnCommand1.ControlConnections.Add(AddressLine2)
```

7. Assign the attributes to change. They are in the [VisibleState](#), [EnabledState](#), [ReadOnly](#), [CssClass](#), [FieldValue](#), [InnerHTMLState](#), [URL](#), [Checked](#), and [Other](#) properties.
 - **VisibleState**, **EnabledState**, **ReadOnly**, **Checked** use an enumerated type that has three values: `Ignore`, `False`, and `True`. They default to `Ignore`. Change to `False` or `True` as needed. (Programmers will find these on the enumerated type `PeterBlum.DES.TrueFalseIgnore`.)

Note: If you are changing the visibility on a web control, including Peter’s Date Package controls, see “[Changing Visibility on a Complex Control](#)”.
 - **CssClass**, **FieldValue**, and **InnerHTMLState** default to “{ORIG}” and will not be applied until you change their value.
8. Sometimes a field hidden or disabled by the `MultiFSCOnCommand` control has an associated `Validator` whose error message is showing. That message is no longer appropriate. To remove it, first set up the [Enabler](#) property on the `Validator` to detect that the control is visible or enabled. Use the `VisibleCondition` or `EnabledCondition` class. See “[Evaluating non-data value states of controls](#)”.

Then set the [ValidateChangedControls](#) property to `true`.
9. Here are some other considerations:
 - If you are using an AJAX system to update this control, set the **InAJAXUpdate** property to `true`. Also make sure the `PageManager` control or `AJAXManager` object has been setup for AJAX. See “Using these Controls With AJAX” in the **General Features Guide**. *Failure to follow these directions can result in incorrect behavior and javascript errors.*
 - This control does not preserve most of its properties in the `ViewState`, to limit its impact on the page. If you need to use the `ViewState` to retain the value of a property, see “The `ViewState` and Preserving Properties for `PostBack`” in the **General Features Guide**.
 - If you encounter errors, see the “[Troubleshooting](#)” section for extensive topics based on several years of tech support’s experience with customers.
 - See also “[Additional Topics for Using These Controls](#)”.

Properties of FSCOnCommand And MultiFSCOnCommand

Click on any of these topics to jump to them:

- ◆ [Invoke the Change Properties](#)
- ◆ [Controls To Change Properties](#)
- ◆ [Attributes To Change Properties](#)
- ◆ [Update Validators Properties](#)
- ◆ [When To Use The Control Properties](#)
- ◆ [Behavior Properties](#)

Invoke the Change Properties

The Properties Editor shows these properties in the “Invokes The Change” category.

- **ControlIDToRunThisAction** (string) – The ID to the control that will run this FSCOnCommand control. If this ID is unassigned, the FSCOnCommand control will do nothing.

This ID must be in the same or an ancestor naming container. If it is in another naming container, use **ControlToRunThisAction**. *Be sure that the control whose ID is used here has the `runat=server` property.*

- **ControlToRunThisAction** (System.Web.UI.Control) – A reference to the control whose state will be changed. It is an alternative to **ControlIDToRunThisAction** that you must assign programmatically. It accepts controls in any naming container.
- **ExtraControlsToRunThisAction** (PeterBlum.DES.Web.WebControls.ControlConnectionCollection) – Identifies additional controls and elements on the page that run this FSCOnCommand control when clicked.

This property is a collection of `PeterBlum.DES.Web.WebControls.ControlConnection` objects. You can assign the control’s ID to the **ControlConnection.ControlID** property or a reference to the control in the **ControlConnection.ControlInstance** property. When using the **ControlID** property, the control must be in the same or an ancestor naming container. If it is in another naming container, use **ControlInstance**.

Be sure that each control assigned to this collection has the `runat=server` property.

ASP.NET Markup for the ExtraControlsToRunThisAction Property

ExtraControlsToRunThisAction is a type of collection. Therefore its ASP.NET Markup is nested as a series of child controls within the `<ExtraControlsToRunThisAction>` tag. Here is an example.

```
<des:FSCOnCommand id="FSCOnCommand1" runat="server">
    <ExtraControlsToRunThisAction>
        <des:ControlConnection ControlID="Button2" />
        <des:ControlConnection ControlID="Label1" />
    </ExtraControlsToRunThisAction>
</des:FSCOnCommand>
```

Programmatically adding to the ExtraControlsToRunThisAction Property

Use the `ExtraControlsToRunThisAction.Add()` method to add an entry. This overloaded method takes one parameter. Choose from the following:

- A reference to the control itself. This is the preferred form.
- A string giving the ID of the control. Do not use this when the control is not in the same naming container.
- An instance of the class `PeterBlum.DES.Web.WebControls.ControlConnection`.

This example shows how to update an existing `PeterBlum.DES.Web.WebControls.ControlConnection` and add a new entry. Suppose the ASP.NET code looks like the text above and the `Label1` control is not in the same or ancestor naming container. Also suppose the control referenced in the property `Button2` control must be added.

[C#]

```
uses PeterBlum.DES;
...
ControlConnection vConnection = (ControlConnection)
    FSCOnCommand1.ExtraControlsToRunThisAction[1];
vConnection.ControlInstance = Label1;
// add Button2. It can be either a control reference or its ID
FSCOnCommand1.ExtraControlsToRunThisAction.Add(Button2);
```

[VB]

```
Imports PeterBlum.DES
...
Dim vConnection As ControlConnection = _
    CType(FSCOnCommand1.ExtraControlsToRunThisAction(1), ControlConnection)
vConnection.ControlInstance = Label1
' add Button2. It can be either a control reference or its ID
FSCOnCommand1.ExtraControlsToRunThisAction.Add(Button2)
```

Controls To Change Properties

The Properties Editor shows these properties in the “Control To Change” category.

- **ControlIDToChange** (string) – *Only on FSCOnCommand.* The ID to the control whose state will be changed. If this ID is unassigned, the FSCOnCommand control is disabled. This ID must be in the same or an ancestor naming container. If it is in another naming container, use **ControlToChange**. *Be sure that the control whose ID is used here has the `runat=server` property.*

Note: If you are changing the visibility on a web control, see “[Changing Visibility on a Complex Control](#)”.

Nested controls syntax

Throughout DES, any property that accepts a control ID supports a special syntax that allows you to refer to a child property inside of the control you specify. That child property must provide the data entry control you are trying to validate, or have its own child property with that data entry control.

```
[ID to the container control].[property with the data entry control]
```

```
[ID to the container control].[property exposing another container control].[property with the data entry control]
```

If any parent object is also a `System.Web.UI.Controls` subclass, you can specify the ID of a control it contains. It will call `FindControls(id)` to find the child control.

```
[ID to the container control].[ID of the child control]
```

```
[ID to the container control].[property exposing another container control].[ID of the child control]
```

- **ControlToChange** (`System.Web.UI.Control`) – *Only on FSCOnCommand.* A reference to the control whose state will be changed. It is an alternative to **ControlIDToChange** that you must assign programmatically. It accepts controls in any naming container.
- **ControlConnections** (`PeterBlum.DES.Web.WebControls.ControlConnectionCollection`) - *Only on MultiFSCOnCommand.* Add a `FSAControlConnection` object for each control whose state will change.

This collection contains `PeterBlum.DES.Web.WebControls.FSAControlConnection` objects that defines a reference to a control, either by its ID or an object reference. Set the `FSAControlConnection.ControlID` property to the ID of the control if it’s in the same or ancestor naming container. Set the `FSAControlConnection.ControlInstance` property programmatically to the control in any other naming container.

Be sure that each control whose ID is used has the `runat=server` property.

You can see how to set this property in design mode, in the webform and programmatically in step 6 of “[Adding the MultiFSCOnCommand Control](#)”.

The `ControlConnectionCollection` class is subclassed from `System.Collections.ArrayList` and inherits all of its properties, methods and events.

Attributes To Change Properties

The Properties Editor shows these properties in the “Attributes to Change” category.

- **VisibleState** (enum PeterBlum.DES.TrueFalseIgnore) – Sets the visibility. The enumerated type PeterBlum.DES.TrueFalseIgnore has these values:
 - Ignore – Do not use this. This value is the default.
 - True – Visible
 - False – Hidden

The **InvisiblePreservesSpaces** property determines if it also retains or loses the space it occupies when it is hidden. It defaults to true.

This property changes **style:visibility**. If **InvisiblePreservesSpaces** is true, it also changes **style:display**.

Note: If you are changing the visibility on a web control, including Peter’s Date Package controls, see “[Changing Visibility on a Complex Control](#)”.

Note: The name VisibleState differs from the name “Visible” on the ConditionTrue and ConditionFalse properties of the FieldStateController. This was done only because a Visible property already exists for a different purpose on web controls.

- **EnabledState** (enum PeterBlum.DES.TrueFalseIgnore) – Sets the enabled state on controls that support the HTML property **disabled**. The enumerated type PeterBlum.DES.TrueFalseIgnore has these values:
 - Ignore – Do not use this. This value is the default.
 - True – Enabled (disabled=false)
 - False – Disabled

Most browsers support the disabled attribute on data entry controls and buttons (<input>, <select> and <textarea> tags). Internet Explorer supports it on most tags. When you disable a <div>, for example, all data entry controls it contains appear disabled. To provide cross browser compatibility, limit this property to data entry controls and buttons.

Note: The name EnabledState differs from the name “Enabled” on the ConditionTrue and ConditionFalse properties of the FieldStateController. This was done only because a Enabled property already exists for a different purpose on web controls.

- **ReadOnly** (enum PeterBlum.DES.TrueFalseIgnore) – Sets the **readOnly** attribute on controls that support the HTML property **readOnly**: TextBoxes, <input type='text'>, and <textarea> tags. The enumerated type PeterBlum.DES.TrueFalseIgnore has these values:
 - Ignore – Do not use this. This value is the default.
 - True – Read only
 - False – Editable

- **CssClass** (string) – Sets the **className** attribute to a style sheet class name. Since the style includes so many visual attributes, this is recommended over setting individual styles (which can be done in the **Other** property.)

When "{ IGNORE }", this property is not used.

It defaults to "{ IGNORE }".

- **FieldValue** (string) – Sets the **value** property of these form elements: <input>, <textarea>, or <select>. This includes TextBoxes, CheckBoxes, RadioButtons, Buttons, Lists, and DropDownLists. Lists and DropDownLists must have a matching value associated with an item in their lists to update the text shown.

When "{ IGNORE }", this property is not used.

It defaults to "{ IGNORE }".

- **InnerHTMLState** (string) – Changes the **innerHTML** attribute. You can supply HTML or straight text. InnerHTML is the text contained inside of the tags: `<tag>innerHTML</tag>`.

InnerHTML can be very harmful. For example, if you assign it to a `<table>` tag, it will overwrite all `<tr>` and `<td>` tags it contains. You can easily input something that is not valid for the tag whose innerHTML you are modifying.

Good candidates are the Label and ``. If a Panel, `<div>`, TableCell, or `<td>` only contain text, they work well too.

When "{ IGNORE }", this property is not used.

It defaults to "{ IGNORE }".

- **URL** (string) – Changes the **href** or **src** attributes on ``, `<input type=image>`, `<frame>`, `<iframe>`, and `<a>` tags. This includes the Image and HyperLink web controls.

Provide a valid URL. If a hyperlink uses a script in its **href** attribute, scripts should start with "javascript:".

When "{ IGNORE }", this property is not used.

It defaults to "{ IGNORE }".

If your URL refers to a file within your web application, you can use the tilde "~" character as the first character to make your web application more portable. The "~" is replaced by the web application path. Normally during development, that folder is just below the domain root. In production, it is the domain root. For example, if you have an "Images" folder in your web application root, declare the **URL** property like this: "~/Images/file.gif".

- **Checked** (enum PeterBlum.DES.TrueFalseIgnore) – Sets the **checked** attribute on radiobuttons and checkboxes (but not a RadioButtonList or CheckBoxList). The enumerated type PeterBlum.DES.TrueFalseIgnore has these values:
 - Ignore – Do not use this. This value is the default.
 - True – Mark the checkbox
 - False – Unmark the checkbox
- **Other** (PeterBlum.DES.CSAttributeDesc) – Changes any attribute or style to a value you specify. You must know the name of the attribute or style and supply a legal value for it.

Any attribute you specify may not be compatible with all browsers. See [Microsoft's DHTML Reference](#). Each DHTML attribute topic will identify whether it is also supported in DOM by indicating its support in the W3C standards.

Note: If you want to specify more than one attribute or style, you must create one FSCOnCommand for each.

To set up the **Other** property, you must specify four values: **AttributeName**, **Value**, **Data Type**, and **AttributeType**.

- **AttributeName** (string) – The name of the attribute or style. If "", the **Other** property is not used. It defaults to "".

Note: Attribute names are case sensitive. Enter them exactly as specified in the DHTML or DOM specification.

- **Value** (string) – The value to assign to the attribute or style. It defaults to "". When the **Data Type** is Boolean, assign 'false' or 'true'. When the **Data Type** is Integer, assign only digits.
- **Data Type** (enum PeterBlum.DES.AttributeDataType) – Specifies the data type of the attribute or style. You must be sure to choose the correct type or you may get JavaScript errors at runtime. This enumerated type has these values:
 - String – This is the default
 - Integer
 - Boolean
- **AttributeType** (enum PeterBlum.DES.AttributeType) – Specifies whether the attribute is on the field or on the style of the field. This enumerated type has these values:
 - Attribute – This is the default.
 - Style

ASP.NET Text Formatting For the Other Property

Here is the ASP.NET formatting for entering these properties:

```
<des:FSCOnCommand id="FieldStateController1" runat="server"
  Other-AttributeName="title"
  Other-Value="This is a tooltip"
  Other-AttributeType="Attribute"
  Other-DataType="String">
</des:FSCOnCommand>
```

- **InvisiblePreservesSpace** (Boolean) – Determines if a control takes up space on the page when it is invisible. When the **VisibleState** property causes the field to be hidden, there are two ways the field can be hidden: Preserve the space of the element or remove the element entirely. This depends on the **display** style attribute.

When `true`, space is preserved. (The style is set to **visibility:hidden** with no change to the **display** style.)

When `false`, the element is removed. (The style is **visibility:hidden;display:none**.) When the element becomes visible once again, the **display** style is restored to its original value.

It defaults to `true`.

- **ChangeStateFunctionName** (String) – Sometimes you need to write your own code to change attributes. For example, you have a complex control that needs to hide several related controls when it is hidden. You can write client-side code to handle this.

Use it to run JavaScript code that may do special things. See [“Client-Side Function: The Change State Function”](#) for details.

This property is name of a JavaScript function that will run. It is only the function name, not

ALERT: Many users make the mistake of assigning JavaScript code to this property. This will cause JavaScript errors.

GOOD: `“MyFunction”`. **BAD:** `“MyFunction();”` and `“alert(‘stop it’);”`.

Note: JavaScript is case sensitive. Be sure the value of this property exactly matches the function definition.

Update Validators Properties

The Properties Editor shows these properties in the “Update Validators” category.

Note: These properties are supported by the DES Validation Framework but not the native Validation Framework.

- **ValidateChangedControls** (Boolean) – When `true`, validate the controls after applying changes. It validates controls defined in **ControlIDToChange**, **ControlToChange**, and **ControlConnections** properties.

Generally this is done when the controls to change are data entry controls that have their visibility or enabled state changed. Usually your Validators will have their **Enabler** properties set to detect the visibility or enabled state of the control they are validating by using the `VisibleCondition` or `EnabledCondition` class. See “[Evaluating non-data value states of controls](#)”.

When `true`, validate. When `false`, do not validate, but you can still use **UseValidationGroup**.

It defaults to `false`.

- **UseValidationGroup** (Boolean) – Runs all Validators whose group matches the **ValidationGroup** property upon completion of the field state change. It behaves just like the user clicked a submit button for a particular group, including an update of `ValidationSummary` controls.

Set this to `true` when you have a Validator control associated with a control that this `FSCOnCommand` control has hidden or disabled, usually in its **Enabler** property. As a result, the Validator will update itself.

This is an alternative to **ValidateChangedControls**.

It defaults to `false`.

The Validator control must include an **Enabler** property setting that detects the control it evaluates is hidden or disabled using the `VisibilityCondition` or `EnabledCondition`.

- **ValidationGroup** (string) – Defines a group name used by Validators that you want to run when this `FSCOnCommand` control changes a field. When **UseValidatorGroup** is `true`, all Validators matching this group name are run after the field state has changed. This allows validators to remove themselves when the state no longer supports them. Use "*" to run through all groups. It defaults to "".
- **RevalidateOnly** (Boolean) – When using either [ValidateChangedControls](#) or [UseValidationGroup](#) properties, this determines if all validators are evaluated or just those that were already evaluated once.

Each validator knows if it has previously been evaluated on this page, even if a postback occurs (so long as the server side calls `PeterBlum.DES.Globals.WebFormDirector.Validate()` or on an AJAX callback. This can improve the user interface by avoiding having error messages appear on fields that are part of the validation group but have yet to be edited.

Set this to `true` when you don't want a validator to appear on fields the user hasn't edited based on this control's action.

Set it to `false` to include all validators determined by the **ValidateChangedControls** or **UseValidationGroup** properties.

It defaults to `true`.

When To Use The Control Properties

The Properties Editor shows these properties in the “When To Use” category.

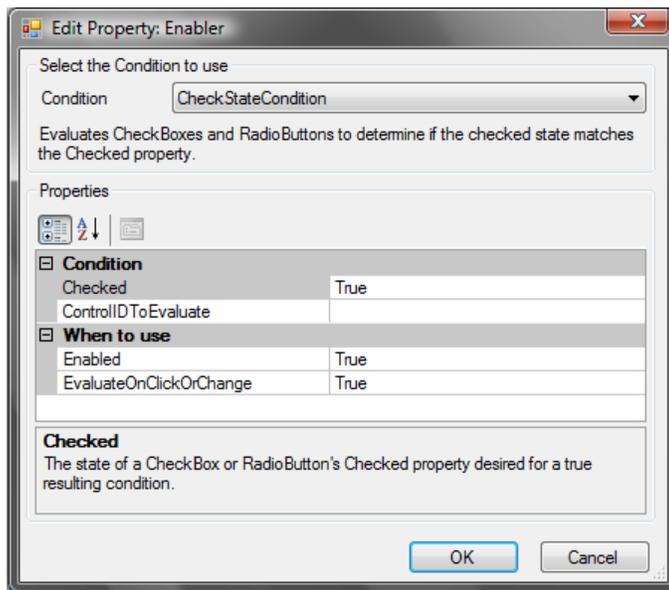
- **SetInitialAppearance** (Boolean) – Determines if the opposite settings are applied to the control to change when the page is first loaded. When `true`, it applies the opposite settings. When `false`, you are responsible to set control to change with the desired initial appearance.
It defaults to `true`.
- **Enabled** (Boolean) – If you have added the FSCOnCommand control to the page but need to disable it completely, set this property to `false`. It defaults to `true`.
- **Enabler** (PeterBlum.DES.IBaseCondition) – There are times when a FSCOnCommand control should be disabled. For example, your command button is a checkbox that should only apply attributes when the checkbox is marked. These rules are formed by [Condition](#) classes with the **Enabler** property on each FSCOnCommand control. By default, the **Enabler** property is set to “None”, where it doesn’t disable the control. You can set it to any Condition, including those you may create programmatically.

Consider these issues when using the Enabler:

- Most Conditions have a property called **EvaluateOnClickOrChange** which defaults to `true`. Change it to `false` when using it in an Enabler.
- Do not use this to detect a control whose **Visible** property is set to `false`. Such a control does not create HTML for the client-side to use. Instead, set the **Enabled** property to `false` when the control is invisible.

Visual Studio and Visual Web Developer Design Mode Users

The Properties Editor offers this window to select a Condition and to edit its properties.



1. Select the Condition from the DropDownList. See “[The Condition classes](#)”.
2. Establish the properties in the Properties grid.
3. Reminder: Be sure to set **EvaluateOnClickOrChange** to false on all Conditions within the Enabler as shown.
4. Click **OK**.

ASP.NET Markup for the Enabler Property

If you want to enter the **Enabler** property and its child properties into the web form using the HTML mode, there are special considerations. The format is very unusual, in part because the .Net framework doesn't support changing the class of a property (polymorphism) without an interesting hack.

Here is the FSCOnCommand with the **Enabler** set to the CheckStateCondition.

```
<des:FSCOnCommand id="FSCOnCommand1" runat="server"
  ControlIDToChange="Span1" >

  <EnablerContainer>
    <des:CheckStateCondition ControlIDToEvaluate="CheckBox1"
      EvaluateOnClickOrChange="false" >
    </des:CheckStateCondition>
  </EnablerContainer>

</des:FSCOnCommand >
```

*Reminder: Be sure to set **EvaluateOnClickOrChange** to false on all Conditions within the Enabler as shown.*

Notice that the **Enabler** property never appears in the attributes of the <des:FSCOnCommand> tag. (It will be added when using the Properties Editor but it's completely meaningless.) Instead, the <EnablerContainer> tag is a child of the FSCOnCommand tag. That tag never has any attributes. The child to <EnablerContainer> defines the class and all properties of the Condition:

```
<des:classname [all properties] />
```

- o des:classname – Use any Condition class for the classname. If you create your own classes, you must declare the namespace using the <% @REGISTER %> tag at the top of the page.
- o [all properties] – Enter the properties into the tag the same way you do for any other control.

Programmatically Setting The Enabler

Here are the steps to set the **Enabler**.

1. Create an instance of the desired Condition. There is a constructor that takes no parameters.

Note: There are also constructors that take parameters representing some of the control's properties. Each demands an "owner" in the first parameter. That value must be the FieldStateController object.

2. Assign property values.

Reminder: Be sure to set EvaluateOnClickOrChange to false on all Conditions within the Enabler.

3. Assign the Condition object to the **Enabler** property.

In this example, add the CheckStateCondition, which is checking CheckBox1, to FSCOnCommand1.

[C#]

```
PeterBlum.DES.Web.WebControls.CheckStateCondition vCond =
  new PeterBlum.DES.Web.WebControls.CheckStateCondition();
vCond.ControlToEvaluate = CheckBox1;
vCond.EvaluateOnClickOrChange = false;
FSCOnCommand1.Enabler = vCond;
```

[VB]

```
Dim vCond As PeterBlum.DES.Web.WebControls.CheckStateCondition = _
  New PeterBlum.DES.Web.WebControls.CheckStateCondition()
vCond.ControlToEvaluate = CheckBox1
vCond.EvaluateOnClickOrChange = False
FSCOnCommand1.Enabler = vCond
```

Behavior Properties

The Properties Editor shows these properties in the “Behavior” category.

- **InAJAXUpdate** (Boolean) – When the page uses AJAX callbacks to add, update, or remove this control, set this to `true`. It defaults to `false`.

In addition, if any of these properties identify a control that participates in the AJAX callback, set this to `true`:

- **ControlIDToRunThisAction** and **ControlToRunThisAction**
- Controls in **ControlConnections**
- **Enabler**. Look at the **ControlIDToEvaluate** and **SecondControlIDToEvaluate**.
- **ExtraControlsToRunThisAction**.

Note: This is only needed for non-DES controls. DES controls will tell the FSCOnCommand control if their own IsAJAXUpdate property is true.

See “Using These Controls with AJAX” in the **General Features Guide**.

- **UpdateWhileEditing** (Boolean) – Determines if the FSCOnCommand is triggered as the user types into a textbox that is specified in the **ControlIDToRunThisAction** property. By default, it does not and only triggers when focus leaves the textbox. Set this to `true` to evaluate the FSCOnCommand with each keystroke.

It defaults to `false`.

- **SupportClientSideLookupByID** (Boolean) - Allows JavaScript programmers to get to the client-side representation of the FSCOnCommand object by the ClientID of the owner control.

Use the client-side function `DES_FindAOById(ClientID)` to search for the “Action object” that matches the ClientID you specify. That function will return `null` if not found.

Use the Action object to invoke the FSCOnCommand as if the user changed a field associated with it. Pass the Action object to the client-side function `DES_DoAction(action object)`.

Suppose you have a Button called `Button1` that will be used to invoke `FSCOnCommand1`. Here is how you set it up to call `InvokeFSC` in `Page_Load()`:

```
Button1.Attributes.Add("onclick",
    "InvokeFSC('" + FSCOnCommand1.ClientID + "');return false;")
```

When **SupportClientSideLookupByID** is `true`, two things happen:

1. The ID is written as a property, `CID`, into the control. (It isn’t written by default to avoid adding excess text to the page.)
2. If the **Enabled** property is `false`, normally no code is written to the client side. This is overridden and code is generated so users can toggle the **Enabled** property.

It defaults to `false`.

- **ViewStateMgr** (`PeterBlum.DES.Web.WebControls.ViewStateMgr`) – Enhances the ViewState on this control to provide more optimal storage and other benefits. Normally, the properties of this control and its segments are not preserved in the ViewState. When working in ASP.NET markup, define a pipe delimited string of properties in the **PropertiesToTrack** property. When working in code, call `ViewStateMgr.TrackProperty("propertyname")` to save the property. Individual segments have a similar method: `TrackPropertyInViewState("propertyname")`.

For more details, see “The ViewState and Preserving Properties forPostBack” in the **General Features User’s Guide**.

- **PropertiesToTrack** (string) – A pipe delimited list of properties to track. Designed for use in markup and the properties editor. The ViewState is not automatically used by most of these properties. To include a property, add it to this pipe delimited list.

For example, `"Group|MayMoveOnClick"`.

When working programmatically, use `ViewStateMgr.TrackProperty("PropertyName")`.

CalculationController

The CalculationController lets you create a calculation involving IntegerTextBox, DecimalTextBox, CurrencyTextBox, and PercentTextBox controls. Its value can be displayed on the page and used by Validators and [Conditions](#).

The CalculationController has a powerful expression building tool. You can include constants, subexpressions, and IF statement logic based on Conditions. You can even provide your own custom functions for more advanced calculations.

The CalculationController itself does not generate any HTML. It creates javascript that monitors edits made to textboxes and displays results in another control, such as a Label or DecimalTextBox. So you can drop it anywhere on the page. It provides support both on the client and server side. So after post back, Validators can still evaluate themselves against this control and your own code can extract the calculation result.

Click on any of these topics to jump to them:

- ◆ [Features](#)
- ◆ [Using the CalculationController](#)
 - [Creating the Expression: The CalcItem classes](#)
 - [Displaying The Result](#)
 - [Using the Result in Validators and Conditions](#)
 - [Using the Result in Your Server-Side Code](#)
 - [JavaScript: Running CalculationControllers On Demand](#)
- ◆ [Adding the CalculationController Control](#)
- ◆ [Properties on CalculationController](#)
- ◆ [Properties on CalcItem Classes](#)
 - [Properties Common To All CalcItem Classes](#)
 - [Properties for the PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem Class](#)
 - [Properties for the PeterBlum.DES.Web.WebControls.ListConstantsCalcItem Class](#)
 - [Properties for the PeterBlum.DES.Web.WebControls.CheckStateCalcItem Class](#)
 - [Properties for the PeterBlum.DES.Web.WebControls.ConstantCalcItem Class](#)
 - [Properties for the PeterBlum.DES.Web.WebControls.ConditionCalcItem Class](#)
 - [Properties for the PeterBlum.DES.Web.WebControls.ParenthesisCalcItem Class](#)
 - [Properties for the PeterBlum.DES.Web.WebControls.CalcControllerCalcItem Class](#)
 - [Properties for the PeterBlum.DES.Web.WebControls.TotalingCalcItem Class](#)
 - [Properties for the PeterBlum.DES.BLD.DataFieldTotalingCalcItem Class \(DES Dynamic Data only\)](#)
 - [Adding Custom Code to a CalcItem](#)
- ◆ [Online examples](#)

Features

The CalculationController lets you describe calculations that involve numbers in textboxes, constants and other logic. The values from these calculations can be used in the following ways:

- Displayed on the page, whether in a Label or a textbox.
- Validators that compare numbers can evaluate the value simply by setting their **ControlIDToEvaluate** property to this control's ID. Supported validators include: CompareToValueValidator, CompareTwoFieldsValidator, RangeValidator, and DifferenceValidator. In addition, the RequiredTextValidator can determine if the calculation had an error.
- Like Validators, their **Conditions** can evaluate the value. For example, the **Enabler** property on various controls use Conditions. Now those Conditions can enable their control based on the result of a calculation.
- The calculations can include the values of other CalculationControllers on the page. This allows reuse of a common calculation, reducing the size of the client-side code and slightly improving performance by reducing the number of times the code executes a calculation.

While it is typical to add together the values of textboxes to form a total, the CalculationController can handle far more powerful expressions. Here are the elements that you can use to develop your expressions:

- Use these textboxes: IntegerTextBox, DecimalTextBox, CurrencyTextBox, and PercentTextBox.
- Lists, DropDownLists, RadioButtonLists, and CheckBoxLists can have numeric values assigned to each selectable element that are used when selected.
- Checkboxes and RadioButtons can have numeric values for their checked and uncheck states.
- Constants (numbers)
- Subexpressions which are the same idea as putting parenthesis around an expression to have it calculate together.
- Use any **Condition** object to select between two expressions. One runs if the Condition evaluates as “success”. The other, if the Condition evaluates as “failed”. This allows your expression to have different logic based on the settings on the screen. Basically, you are developing “IF” statements.

Since the CompareToValueCondition and RangeCondition now can evaluate the values of CalculationControllers, your IF statements can be decided by calculations too.

- Each element – textbox, constant, subexpression, and “IF” statement – can use these operators: add, subtract, multiply, and divide.

Here is an example of two CalculationControllers on a page, using the image shown in design mode. They refer to three controls: two DecimalTextBoxes and a CheckBox. The DecimalTextBoxes are used for the calculation. The CheckBox is used for the Condition object (a CheckStateCondition):

```
CalculationController1: IF ( CheckBox1 is checked ) THEN CalculationController2 ELSE DecimalTextBox1 - DecimalTextBox2
```

```
CalculationController2: DecimalTextBox1 + DecimalTextBox2
```

To further refine your expressions, the CalculationController has these features:

- The result, which is initially a decimal value, can be rounded in several ways. It can round to a certain number of decimal places and use different rounding rules.
- For blank textboxes, you can determine if it's an error or treated as 0.
- You can supply client- and server-side functions that let you customize the result of any element. For example, if you want the value of a textbox to be run through the Sin() function, you write a function to use that calculation. The CalculationController will pass you the value. Your function can report an error for an illegal value and correct errors.
- It gracefully handles errors, such as divide by zero.

Using the CalculationController

There are several aspects to using this control:

- Create a mathematical expression in the **Expression** property.
- Optionally display the result of the calculation. There are formatting rules to consider.
- Optionally let a Validator evaluate the result of the calculation.
- Optionally use the result of the calculation in your own code.
- Display a different number based on a selection in a list, DropDownList, or RadioButtonList.

Click on any of these topics to jump to them:

- ◆ [Creating the Expression: The CalcItem classes](#)
 - [PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem](#)
 - [PeterBlum.DES.Web.WebControls.ConstantCalcItem](#)
 - [PeterBlum.DES.Web.WebControls.ListConstantsCalcItem](#)
 - [PeterBlum.DES.Web.WebControls.CheckStateCalcItem](#)
 - [PeterBlum.DES.Web.WebControls.ParenthesisCalcItem](#)
 - [PeterBlum.DES.Web.WebControls.ConditionCalcItem](#)
 - [PeterBlum.DES.Web.WebControls.CalcControllerCalcItem](#)
 - [PeterBlum.DES.Web.WebControls.TotalingCalcItem](#)
 - [PeterBlum.DES.BLD.DataFieldTotalingCalcItem](#) (DES Dynamic Data only)
 - [General Guidelines for CalcItem objects](#)
- ◆ [Displaying The Result](#)
- ◆ [Using the Result in Validators and Conditions](#)
- ◆ [Using the Result in Your Server-Side Code](#)
- ◆ [JavaScript: Running CalculationControllers On Demand](#)
- ◆ [Online examples](#)

Creating the Expression: The CalcItem classes

The calculation is created within the **Expression** property on the CalculationController. This property is a collection which holds a list of the following objects.

Click on any of these topics to jump to them:

- ◆ [PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem](#)
- ◆ [PeterBlum.DES.Web.WebControls.ConstantCalcItem](#)
- ◆ [PeterBlum.DES.Web.WebControls.ListConstantsCalcItem](#)
- ◆ [PeterBlum.DES.Web.WebControls.CheckStateCalcItem](#)
- ◆ [PeterBlum.DES.Web.WebControls.ParenthesisCalcItem](#)
- ◆ [PeterBlum.DES.Web.WebControls.ConditionCalcItem](#)
- ◆ [PeterBlum.DES.Web.WebControls.CalcControllerCalcItem](#)
- ◆ [PeterBlum.DES.Web.WebControls.TotalingCalcItem](#)
- ◆ [General Guidelines for CalcItem objects](#)

PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem

 [Online example](#)

Identifies one IntegerTextBox, DecimalTextBox, CurrencyTextBox, or PercentTextBox that supplies a number into the calculation. Usually your calculation will have one or more of these. The TextBox will be set up so that any time it changes (after focus is lost), the calculation will run.

Note: If you are totaling a column of numeric textboxes in a ListView, GridView, DataList, or Repeater, consider using the [TotalingCalcItem object](#).

Set the numeric TextBox in the [TextBoxControlID](#) property. Since a textbox can be blank or contain an illegal (non-numeric) value, use the [BlankIsZero](#) and [InvalidIsZero](#) properties to determine if these states use the value of 0 or indicate an error.

See "[Properties for the PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem Class](#)".

Example

Expression: (IntegerTextBox1 + IntegerTextBox2)

The **Expression** property contains:

NumericTextBoxCalcItem for IntegerTextBox1

NumericTextBoxCalcItem for IntegerTextBox2 with **Operator** = Add (which is the default value for that property)

The result is shown in a Label control called Label1.

```
<des:CalculationController id="CalculationController1" runat="server"
  ShowValueControlID="Label1" >
  <Expression>
    <des:NumericTextBoxCalcItem TextBoxControlID="IntegerTextBox1" />
    <des:NumericTextBoxCalcItem TextBoxControlID="IntegerTextBox2" />
  </Expression>
</des:CalculationController>
```

PeterBlum.DES.Web.WebControls.ConstantCalcItem

 [Online example](#)

Supply a number constant into the calculation. Assign the constant to the **Constant** property.

See "[Properties for the PeterBlum.DES.Web.WebControls.ConstantCalcItem Class](#)".

Example

Expression: (IntegerTextBox1 + IntegerTextBox2) * 25

The **Expression** property contains:

[NumericTextBoxCalcItem](#) for IntegerTextBox1

[NumericTextBoxCalcItem](#) for IntegerTextBox2 with **Operator** = Add (which is the default value for that property)

[ConstantCalcItem](#) for 25 with **Operator** = Multiply

The result is shown in a Label control called Label1.

```
<des:CalculationController id="CalculationController1" runat="server"
  ShowValueControlID="Label1" >
  <Expression>
    <des:NumericTextBoxCalcItem TextBoxControlID="IntegerTextBox1" />
    <des:NumericTextBoxCalcItem TextBoxControlID="IntegerTextBox2" />
    <des:ConstantCalcItem Constant="25" Operator="Multiply" />
  </Expression>
</des:CalculationController>
```

PeterBlum.DES.Web.WebControls.ListConstantsCalcItem

 [Online example](#)

Associate the items in a ListBox or DropDownList with constants. As the user changes the list's selection, the calculation gets a different constant. For example, when SelectedIndex is 0, the value is 25 and when SelectedIndex is 1 through 4, the value is 30.

Set the ListBox or DropDownList controls in the **ListControlID** property. Add PeterBlum.DES.ConstantForSelectedIndex objects to the **ConstantsForSelectedIndexes** property to define the SelectedIndexes that are associated with a specific value.

You can make a selected index report an error to the CalculationController. You can also define a default number when an index is selected but has no matching item in the **ConstantsForSelectedIndexes** property.

See "[Properties for the PeterBlum.DES.Web.WebControls.ListConstantsCalcItem Class](#)".

Example

Expression: (value from the selected item in ListBox1) * 25

The **Expression** property contains:

ListConstantsCalcItem for ListBox1 where index 0 is 5, indices 1 and 2 are 10, and index 3 is 25.

[ConstantCalcItem](#) for 25 with **Operator** = Multiply

The result is shown in a Label control called Label1.

```
<des:CalculationController id="CalculationController1" runat="server"
  ShowValueControlID="Label1" >
  <Expression>
    <des:ListConstantsCalcItem ListControlID="ListBox1" />
    <ConstantsForSelectedIndexes>
      <des:ConstantForSelectedIndex StartIndex="0" Constant="5" />
      <des:ConstantForSelectedIndex StartIndex="1" EndIndex="2"
        Constant="10" />
      <des:ConstantForSelectedIndex StartIndex="3" Constant="25" />
    </ConstantsForSelectedIndexes>
  </des:ListConstantsCalcItem>
  <des:ConstantCalcItem Constant="25" Operator="Multiply" />
</Expression>
</des:CalculationController>
```

PeterBlum.DES.Web.WebControls.CheckStateCalcItem

 [Online example](#)

Associated two values with a CheckBox or RadioButton, where one value is used when its checked and the other when it isn't. It can be used with CheckBoxList and RadioButtonList controls so long as you identify a specific button in the list by its index.

Set the CheckBox, RadioButton, CheckBoxList, or RadioButtonList in the **CheckedStateControlID** property. When using a CheckBoxList or RadioButtonList, specify which button with its **Index** property.

Define the values within the **ValueWhenChecked** and **ValueWhenUnchecked** properties.

One common usage is to add a series of checkboxes that are checked and ignore those that are not checked. To do this, include a CheckStateCalcItem object for every checkbox, adding them together. Since this class also returns a value for the unchecked state, usually you use **ValueWhenUnchecked** = 0 when adding or subtracting. You use **ValueWhenUnchecked** = 1 when multiplying or dividing. You can use the same technique with a list of RadioButtons to determine a value for the one that is checked.

See "[Properties for the PeterBlum.DES.Web.WebControls.CheckStateCalcItem Class](#)".

Example 1

Suppose there are 3 radiobuttons grouped together and each has its own value. This determines the value of the selected one by evaluating all 3 and adding their values together. Those that are unchecked have a value of 0 (from their **ValueWhenUnchecked** property which defaults to 0.)

The **Expression** property contains:

CheckStateCalcItem for RadioButton1 with a value of 10 when checked and 0 when unchecked.

CheckStateCalcItem for RadioButton2 with a value of 20 when checked and 0 when unchecked.

CheckStateCalcItem for RadioButton3 with a value of 30 when checked and 0 when unchecked.

The result is shown in a Label control called Label1.

```
<des:CalculationController id="CalculationController1" runat="server"
  ShowValueControlID="Label1" >
  <Expression>
    <des:CheckStateCalcItem CheckStateControlID="RadioButton1"
      ValueWhenChecked="10" />
    <des:CheckStateCalcItem CheckStateControlID="RadioButton2"
      ValueWhenChecked="20" />
    <des:CheckStateCalcItem CheckStateControlID="RadioButton3"
      ValueWhenChecked="30" />
  </Expression>
</des:CalculationController>
```

EXAMPLES CONTINUE ON THE NEXT PAGE

Example 2

Suppose there is a `CheckBoxList` where each checkbox has a numeric value. This determines the sum of all checked buttons by evaluating all 3 and adding their values together. Those that are unchecked have a value of 0 (from their `ValueWhenUnchecked` property which defaults to 0.)

The **Expression** property contains:

`CheckStateCalcItem` for `CheckBoxList1` at index 0 with a value of 10 when checked and 0 when unchecked.

`CheckStateCalcItem` for `CheckBoxList1` at index 1 with a value of 20 when checked and 0 when unchecked.

`CheckStateCalcItem` for `CheckBoxList1` at index 2 with a value of 30 when checked and 0 when unchecked.

The result is shown in a `Label` control called `Label1`.

```
<des:CalculationController id="CalculationController1" runat="server"
  ShowValueControlID="Label1" >
  <Expression>
    <des:CheckStateCalcItem CheckStateControlID="CheckBoxList1"
      Index="0" ValueWhenChecked="10" />
    <des:CheckStateCalcItem CheckStateControlID="CheckBoxList1"
      Index="1" ValueWhenChecked="20" />
    <des:CheckStateCalcItem CheckStateControlID="CheckBoxList1"
      Index="2" ValueWhenChecked="30" />
  </Expression>
</des:CalculationController>
```

Example 3

You can see a real-world example on the DES Ordering page: <http://www.peterblum.com/des/order.aspx>. To see the source code, view <http://www.peterblum.com/des/orderpagemarkup.aspx>.

PeterBlum.DES.Web.WebControls.ParenthesisCalcItem

[Online example](#)

Create a subexpression whose items are calculated together in its own **Expression** property. It's the same idea as using parenthesis in computer code. For example, in `(TextBox1 * 4) + (TextBox2 * 3)` the **Expression** property contains two **ParenthesisCalcItem** objects, each which hold their own **NumericTextBoxCalcItem** and **ConstantCalcItem** objects.

See "[Properties for the PeterBlum.DES.Web.WebControls.ParenthesisCalcItem Class](#)".

Note: If you are totaling a column of numeric textboxes or CalculationControllers in a ListView, GridView, DataList, or Repeater, consider using the [TotalingCalcItem](#) object.

Example

Expression: `(DecimalTextBox1 + DecimalTextBox2) * 25`

The **Expression** property contains:

ParenthesisCalcItem with its own expression that adds the two textboxes like this:

NumericTextBoxCalcItem for `IntegerTextBox1`

NumericTextBoxCalcItem for `IntegerTextBox2` with **Operator** = Add (which is the default value for that property)

ConstantCalcItem for 25 with **Operator** = Multiply

The result is shown in a Label control called `Label1`.

```
<des:CalculationController id="CalculationController1" runat="server">
  <Expression>
    <des:ParenthesisCalcItem>
      <Expression>
        <des:NumericTextBoxCalcItem TextBoxControlID="DecimalTextBox1" />
        <des:NumericTextBoxCalcItem TextBoxControlID="DecimalTextBox2" />
      </Expression>
    </des:ParenthesisCalcItem>
    <des:ConstantCalcItem Constant="25" Operator="Multiply" />
  </Expression>
</des:CalculationController>
```

PeterBlum.DES.Web.WebControls.ConditionCalcItem

 [Online example](#)

Create an IF statement that uses any [Condition](#) object to determine whether to run subexpressions located in [ExpressionWhenTrue](#) and [ExpressionWhenFalse](#) properties. This allows great flexibility in your calculations. The Condition object is assigned to the [Condition](#) property. See [“The Condition classes”](#) to learn more.

See [“Properties for the PeterBlum.DES.Web.WebControls.ConditionCalcItem Class”](#).

Example 1

Suppose you have 4 IntegerTextBoxes. You want a CheckBox to determine whether to add only two or all four of them. Use a CheckStateCondition pointing to the CheckBox in the **Condition** property. **ExpressionWhenTrue** contains four NumericTextBoxCalcItem objects. **ExpressionWhenFalse** contains two NumericTextBoxCalcItem objects. The expression would look like this:

```
Expression: IF (CheckBox1.Checked = True) THEN
            (TextBox1 + TextBox2 + TextBox3 + TextBox4) ELSE (TextBox1 + TextBox2)
```

The result is shown in a Label control called Label1.

```
<des:CalculationController id="CalculationController1" runat="server">
  <Expression>
    <des:ConditionCalcItem>
      <ConditionContainer>
        <des:CheckStateCondition ControlIDToEvaluate="CheckBox1" />
      </ConditionContainer>
      <ExpressionWhenTrue>
        <des:NumericTextBoxCalcItem TextBoxControlID="IntegerTextBox1" />
        <des:NumericTextBoxCalcItem TextBoxControlID="IntegerTextBox2" />
        <des:NumericTextBoxCalcItem TextBoxControlID="IntegerTextBox3" />
        <des:NumericTextBoxCalcItem TextBoxControlID="IntegerTextBox4" />
      </ExpressionWhenTrue>
      <ExpressionWhenFalse>
        <des:NumericTextBoxCalcItem TextBoxControlID="IntegerTextBox1" />
        <des:NumericTextBoxCalcItem TextBoxControlID="IntegerTextBox2" />
      </ExpressionWhenFalse>
    </des:ConditionCalcItem>
  </Expression>
</des:CalculationController>
```

EXAMPLES CONTINUE ON THE NEXT PAGE

Example 2

You have a screen with shipping charges determined by a DropDownList. Use SelectedIndexCondition to evaluate which index is selected in the DropDownList in the **Condition** property. The **ExpressionWhenTrue** property will have a ConstantCalcItem for shipping charges of the index selected. The **ExpressionWhenFalse** property will have another ConditionCalcItem to compare another index. The expression would look like this:

Expression: IF (ShippingCharges.SelectedIndex = 0) THEN 3.50 ELSE 6.00

```
<des:CalculationController id="CalculationController1" runat="server">
  <Expression>
    <des:ConditionCalcItem>
      <ConditionContainer>
        <des:SelectedIndexCondition ControlIDToEvaluate="ShippingCharges"
          Index="0" />
      </ConditionContainer>
      <ExpressionWhenTrue>
        <des:ConstantCalcItem Constant="3.5"/>
      </ExpressionWhenTrue>
      <ExpressionWhenFalse>
        <des:ConstantCalcItem Constant="6.0"/>
      </ExpressionWhenFalse>
    </des:ConditionCalcItem>
  </Expression>
</des:CalculationController>
```

Example 3

Test the total of 2 textboxes are within the range of 0 to 100. If they are in that range, use that value. Otherwise, use 0. To do this, you create a second CalculationController that totals the textboxes and use a RangeCondition to evaluate that second CalculationController is within the range:

Expression: IF (CalculationController2 is between 0 and 100) THEN
(use CalculationController2's value) ELSE (0)

```
<des:CalculationController id="CalculationController1" runat="server">
  <Expression>
    <des:ConditionCalcItem>
      <ConditionContainer>
        <des:RangeCondition ControlIDToEvaluate="CalculationController2"
          Minimum="0" Maximum="100" />
      </ConditionContainer>
      <ExpressionWhenTrue>
        <des:CalcControllerCalcItem ControlID="CalculationController2"/>
      </ExpressionWhenTrue>
      <ExpressionWhenFalse>
        <des:ConstantCalcItem Constant="0"/>
      </ExpressionWhenFalse>
    </des:ConditionCalcItem>
  </Expression>
</des:CalculationController>

<des:CalculationController id="CalculationController2" runat="server">
  <Expression>
    <des:NumericTextBoxCalcItem TextBoxControlID="DecimalTextBox1" />
    <des:NumericTextBoxCalcItem TextBoxControlID="DecimalTextBox2" />
  </Expression>
</des:CalculationController>
```

PeterBlum.DES.Web.WebControls.CalcControllerCalcItem

 [Online example](#)

Use another CalculationController for a part of the calculation. While you can reproduce a calculation several times in an expression, programmers prefer to write functions to encapsulate code that is reused. This has several benefits here:

- When using a ConditionCalcItem, you might want a [Condition](#) to look at the value of a calculation. You use the technique described above in the third example of ConditionCalcItem to create a second CalculationController. You reuse the value of the second CalculationController within this CalculationController with a CalcControllerCalcItem:

```
IF (CalculationController2 is between 0 and 100) THEN
  (CalculationController2) ELSE (0)
```

- Less JavaScript is generated, reducing the size of the page. While each CalculationController doesn't generate much JavaScript, this is an optimization.
- Less JavaScript is executed because the CalculationController is run once for each request. This is a speed optimization.

Set the ID of the other CalculationController in the [ControlID](#) property or a reference to CalculationController object in the [ControlInstance](#) property.

See "[Properties for the PeterBlum.DES.Web.WebControls.CalcControllerCalcItem Class](#)".

Note: If you are totaling a column of CalculationControllers in a ListView, GridView, DataList, or Repeater, consider using the [TotalingCalcItem](#) object.

Example

Test the total of 2 textboxes are within the range of 0 to 100. If they are in that range, use that value. Otherwise, use 0. To do this, you create a second CalculationController that totals the textboxes and use a RangeCondition to evaluate that second CalculationController is within the range:

```
Expression: IF (CalculationController2 is between 0 and 100) THEN
  (use CalculationController2's value) ELSE (0)
```

```
<des:CalculationController id="CalculationController1" runat="server">
  <Expression>
    <des:ConditionCalcItem>
      <ConditionContainer>
        <des:RangeCondition ControlIDToEvaluate="CalculationController2"
          Minimum="0" Maximum="100" />
      </ConditionContainer>
      <ExpressionWhenTrue>
        <des:CalcControllerCalcItem ControlID="CalculationController2"/>
      </ExpressionWhenTrue>
      <ExpressionWhenFalse>
        <des:ConstantCalcItem Constant="0"/>
      </ExpressionWhenFalse>
    </des:ConditionCalcItem>
  </Expression>
</des:CalculationController>

<des:CalculationController id="CalculationController2" runat="server">
  <Expression>
    <des:NumericTextBoxCalcItem TextBoxControlID="DecimalTextBox1" />
    <des:NumericTextBoxCalcItem TextBoxControlID="DecimalTextBox2" />
  </Expression>
</des:CalculationController>
```

PeterBlum.DES.Web.WebControls.TotalingCalcItem

 [Online example](#)

Totals a column of a ListView, GridView, DataList, and Repeater control. It needs to know of a control within that column, either a numeric textbox like IntegerTextBox or another CalculationController, which is hosting a value specific to that row.

TotalingCalcItem is effectively creating a ParenthesisCalcItem with NumericTextBoxCalcItems or CalcControllerCalcItems. You could do the same but with more work, involving adding an event handler to your list or grid control where you find the control in the row, create the CalcItem object and add it to the ParenthesisCalcItem.

When using a paged list, the rows shown create a sub total. If you want to calculate based on the grand total of all rows, you need to modify the values created by TotalingCalcItem to include the actual grand total minus the page's subtotal. The **GrandTotal** property helps you do this. Calculate the grand total. Then assign its value to the **GrandTotal** property. This will adjust the TotalingCalcItem's value, adding a constant that equals **GrandTotal** – the column's subtotal.

Set the list or grid control to the **ListControlID** property. The control in the row can be setup in two ways:

- Use **ControlIDToRow** to specify the ID of the control. Use it unless the next option makes more sense.
- Use the **GetColumnControl** event handler when the control is in a child naming container or you need to programmatically take some action to get the right control.

Since a textbox can be blank or contain an illegal (non-numeric) value, use **InvalidIsZero** property to determine if these states use the value of 0 or indicate an error.

See "[Properties for the PeterBlum.DES.Web.WebControls.TotalingCalcItem Class](#)".

ALERT: When using DES Dynamic Data, substitute the *DataFieldTotalingCalcItem* object for *TotalingCalcItem*. Replace *ControlIDToRow* with *DataField* identifying the column name.

Example

Expression: Sum of IntegerTextBox1 in the ListView control.

The result is shown in a Label control called Label1.

```
<asp:ListView id="ListView1" runat="server">
  <ItemTemplate>
    <des:IntegerTextBox id="IntegerTextBox1" runat="server" />
  </ItemTemplate>
</asp:ListView>

<des:CalculationController id="CalculationController1" runat="server"
  ShowValueControlID="Label1" >
  <Expression>
    <des:TotalingCalcItem ListControlID="ListView1"
      ControlIDToRow="IntegerTextBox1" />
  </Expression>
</des:CalculationController>
```

PeterBlum.DES.BLD.DataFieldTotalingCalcItem

This feature only applies when using Peter's Business Driven Logic UI ("BLD"). It replaces the PeterBlum.DES.Web.WebControls.TotalingCalcItem class.

Totals a column of a BLDListView control. It needs to know of a data field column name whose value will be totalled.

Set the BLDListView control to the **ListControlID** property. Set the column name in the **DataField** property.

Since a textbox can be blank or contain an illegal (non-numeric) value, use **InvalidIsZero** property to determine if these states use the value of 0 or indicate an error.

See "[Properties for the PeterBlum.DES.BLD.DataFieldTotalingCalcItem Class](#)".

Example

Expression: Sum of the Price column. The BLDListView is using automatic scaffolding here, so no data fields are listed.

The result is shown in a Label control called Label1.

```
<des:BLDListView id="BLDListView1" runat="server"
  PatternTemplateName="GridView" >
</des:BLDListView>

<des:CalculationController id="CalculationController1" runat="server"
  ShowValueControlID="Label1" >
  <Expression>
    <des:DataFieldTotalingCalcItem ListControlID="BLDListView1"
      DataField="Price" />
  </Expression>
</des:CalculationController>
```

General Guidelines for CalcItem objects

Each of the “CalcItem” objects described above provide an operator to add, subtract, multiply, and divide by the previous object in the expression. Set the operator in the **Operator** property. For the first element in an expression or subexpression, generally always use the Add operator.

The result of the expression may need to be rounded to an integer or a specific number of decimal places. Use the **RoundMode** and **DecimalPlaces** properties to establish rounding. **RoundMode** provides 5 common rounding rules for you to choose from.

Each CalcItem object allows you to define custom code to take the numeric value of the CalcItem object and further process it. You can change the value, return the original value, return a value when the CalcItem object reported an error, or indicate an error (such as the value was out of range). Use the **CustomCalcFunctionName** property for the client-side code and **CustomCalculation** property for the server-side code.

The DES Ordering page uses several CalculationControllers that evaluate radiobuttons, checkboxes, and an IntegerTextBox to determine the subtotal of your order and the per-unit value. It uses NumericTextBoxCalcItem, CheckStateCalcItem, ParenthesisCalcItem, and ConditionCalcItem objects.

See the live page at <http://www.peterblum.com/des/order.aspx>. To see its source code, view <http://www.peterblum.com/des/orderpagemarkup.aspx>.

Displaying The Result

You can display the result in any of these types of controls: Label, LocalizableLabel, IntegerTextBox, DecimalTextBox, and CurrencyTextBox. Set the ID to the control in the **ShowValueControlID** property or a reference to that control in **ShowValueInstance**. You are not required to display the value. If you leave these properties unassigned, the CalculationController can still be used by Validators, Conditions, and other CalculationControllers.

When displaying the result in IntegerTextBox, DecimalTextBox, or CurrencyTextBox, those controls will dictate the formatting. When the **DecimalPlaces** property is Auto, it uses the number of decimal places appropriate for the control. For example, an IntegerTextBox has 0 decimal places.

When displaying the result in a Label or LocalizableLabel, you have many formatting properties:

- Use the **DecimalPlaces** property to determine the number of decimal places.
- If you want to show thousands separators, set **LabelFormatThousandsSep** to true.
- If you want to show a currency symbol, set **LabelFormatCurrencySymbol** to true.
- You can either replace the entire text of the Label or replace a token. When using a token, you can have a sentence, like "The result is {TOKEN}". When using Tokens, multiple CalculationControllers can have their tokens in the same label. You define the token and assign it to the **LabelToken** property. Make sure the Label has that token embedded in its **Text** property.

Each time the server side code runs, it will update the control in **ShowValueControlID** based on the rules of the **AutoShowValue** property. If it updates the control, the page will show the value as it's loaded.

If there are any errors in the calculation, it will assign the text in **InvalidValueLabel** to the control, whether a Label or numeric TextBox. For example, you might set **InvalidValueLabel** to "0.00" or "-.--". If **InvalidValueLabel** is blank, the control will be blanked. Additionally, you can change the style sheet class name if you assign a new class to **InvalidValueCssClass**.

Using the Result in Validators and Conditions

 [Online example](#)

The following Validators and their Conditions can evaluate CalculationControllers by setting their **ControlIDToEvaluate** property to the ID of the CalculationController: RequiredTextValidator, CompareToValueValidator, CompareTwoFieldsValidator, RangeValidator, and DifferenceValidator. The RequiredTextValidator detects errors because the CalculationController returns an empty string when there is an error.

Validators can update each time the calculation runs or only on submit by using the ValidateOnCalc property.

Conditions appear throughout DES. With a CalculationController, your Enabler properties can enable controls based on a calculation. The MultiConditionValidator, CountTrueConditionsValidator, FieldStateController.Condition, and all other cases can evaluate CalculationControllers too.

Using the Result in Your Server-Side Code

You can access the result of the calculation with the **Value** property. It is a `System.Double` value. Always check the **IsValid** property first. If **IsValid** is false, the calculation failed and **Value** is 0.

JavaScript: Running CalculationControllers On Demand

DES provides methods and functions to run calculations on demand, both on the client and server side. On the server side, use the **Value** property. See “[Calculating The Value Properties](#)”. On the client-side, use these functions. They are available whenever you use the CalculationController:

- `DES_CalcAll()` – Runs all CalculationControllers on the page. They will update their controls to display.

```
DES_CalcAll();
```

- `DES_CalcOne(ID)` – Runs the calculation for the CalculationController supplied by its ID and returns the result. It does not update the value on the control to display.

The ID parameter must be the ClientID of the CalculationController.

It returns a decimal value or NaN. NaN is a special JavaScript symbol representing “not a number”. Here it means the calculation failed. To test for NaN, JavaScript provides the function `isNaN(value)`. It returns `true` if the value passed in is NaN.

```
var vResult = DES_CalcOne('CalculationController1');  
if (!isNaN(vResult)) // it's a valid decimal number  
    // use vResult
```

If you need to know how to add your JavaScript to the page, see “[Adding Your JavaScript to the Page](#)”.

Adding the CalculationController Control



These steps ask you to jump around the document using clicks on links. Adobe Reader offers a **Previous View** command to return to the link. Look for this in the Adobe Reader (shown v6.0)

1. Prepare the page for DES controls. See “Preparing a page for DES controls” in the **General Features Guide**. It covers issues like style sheets, AJAX, and localization.
2. Start by setting up all the Controls involved. Make sure the TextBoxes are IntegerTextBox, DecimalTextBox, CurrencyTextBox, or PercentTextBox. If you are using any conditional logic, be sure that controls that determine that logic are present too, like CheckBoxes and DropDownLists.
3. Add the CalculationController control to the page. Its location does not matter as it contributes no HTML to the page.

Visual Studio and Visual Web Developer Design Mode Users

Drag the CalculationController control from the Toolbox onto your web form. It will look like this:



Text Entry Users

Add the control (inside the <form> area):

```
<des:CalculationController id="[YourControlID]" runat="server" >
</des:CalculationController>
```

Programmatically Creating the Control

- Identify the control which you will add the CalculationController control to its **Controls** collection. Like all ASP.NET controls, the CalculationController control can be added to any control that supports child controls, like Panel, User Control, or TableCell. If you want to add it directly to the Page, first add a Placeholder and use the Placeholder.
- Create an instance of the CalculationController control class. The constructor takes no parameters.
- Assign the **ID** property.
- Add the CalculationController control to the **Controls** collection.

In this example, the CalculationController control is created with an **ID** of “CalculationController1”. It is added to Placeholder1.

[C#]

```
PeterBlum.DES.Web.WebControls.CalculationController vCalc =
    new PeterBlum.DES.Web.WebControls.CalculationController();
vCalc.ID = "CalculationController1";
Placeholder1.Controls.Add(vCalc);
```

[VB]

```
Dim vCalc As PeterBlum.DES.Web.WebControls.CalculationController = _
    New PeterBlum.DES.Web.WebControls.CalculationController()
vCalc.ID = "CalculationController1"
Placeholder1.Controls.Add(vCalc)
```

4. Write down your expression. For example, “(TextBox1 + TextBox2) * 25”. This will guide you as you build it in the CalculationController.

Guidelines for setting properties

- Design mode users can use the Properties Editor or the Expanded Properties Editor. (See “Expanded Properties Editor” in the **General Features Guide**.) The SmartTag  also offers some of the most important properties.
- Text entry users should add the properties into the `<des:ControlClass>` tag in this format:
propertyname="value"
- When setting a property programmatically, have a reference to the control’s object and set the property according to your language’s rules.

5. Define the expression in the **Expression** property. That property allows a list of CalcItem objects. Pick from this list. (Each topic includes examples.)

- ◆ [PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem](#)
- ◆ [PeterBlum.DES.Web.WebControls.ConstantCalcItem](#)
- ◆ [PeterBlum.DES.Web.WebControls.ListConstantsCalcItem](#)
- ◆ [PeterBlum.DES.Web.WebControls.CheckStateCalcItem](#)
- ◆ [PeterBlum.DES.Web.WebControls.ParenthesisCalcItem](#)
- ◆ [PeterBlum.DES.Web.WebControls.ConditionCalcItem](#)
- ◆ [PeterBlum.DES.Web.WebControls.CalcControllerCalcItem](#)
- ◆ [PeterBlum.DES.Web.WebControls.TotalingCalcItem](#)
- ◆ [PeterBlum.DES.BLD.DataFieldTotalingCalcItem](#) (BLD only)

For details on adding these objects in design mode, ASP.NET markup and programmatically, see the **Expression** property.

6. If you want to display the value, set the ID to the Label or numeric textbox in **ShowValueControlID** or programmatically assign the object to **ShowValueInstance**. Use the **InvalidValueLabel** and **InvalidValueCssClass** properties to customize the appearance of that control when there is a calculation error.
7. If you want to validate the value, add the appropriate Validator control. You can choose from RequiredTextValidator, CompareToValueValidator, CompareTwoFieldsValidator, RangeValidator and DifferenceValidator. The RequiredTextValidator can report when there was a calculation error. See the **Validation User’s Guide** for details on the Validators.

Then set the **ValidateOnCalc** property to `true`.

8. Here are some other considerations:

- If you are using an AJAX system to update this control, set the **InAJAXUpdate** property to `true`. Also make sure the PageManager control or AJAXManager object has been setup for AJAX. See “Using these Controls With AJAX” in the **General Features Guide**. *Failure to follow these directions can result in incorrect behavior and javascript errors.*
- This control does not preserve most of its properties in the ViewState, to limit its impact on the page. If you need to use the ViewState to retain the value of a property, see “The ViewState and Preserving Properties forPostBack” in the **General Features Guide**.
- If you encounter errors, see the “Troubleshooting” section for extensive topics based on several years of tech support’s experience with customers.
- See also “Additional Topics for Using These Controls”.

 [Online examples](#)

Properties on CalculationController

Click on any of these topics to jump to them:

- ◆ [Calculating The Value Properties](#)
- ◆ [Showing The Value Properties](#)
- ◆ [When to Use the Control Properties](#)
- ◆ [Behavior Properties](#)
- ◆ [Properties on CalcItem Classes](#)
 - [Properties Common To All CalcItem Classes](#)
 - [Properties for the PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem Class](#)
 - [Properties for the PeterBlum.DES.Web.WebControls.ListConstantsCalcItem Class](#)
 - [Properties for the PeterBlum.DES.Web.WebControls.CheckStateCalcItem Class](#)
 - [Properties for the PeterBlum.DES.Web.WebControls.ConstantCalcItem Class](#)
 - [Properties for the PeterBlum.DES.Web.WebControls.ParenthesisCalcItem Class](#)
 - [Properties for the PeterBlum.DES.Web.WebControls.ConditionCalcItem Class](#)
 - [Properties for the PeterBlum.DES.Web.WebControls.CalcControllerCalcItem Class](#)
 - [Properties for the PeterBlum.DES.Web.WebControls.TotalingCalcItem Class](#)
 - [Properties for the PeterBlum.DES.BLD.DataFieldTotalingCalcItem Class \(BLD only\)](#)
 - [Adding Custom Code to a CalcItem](#)

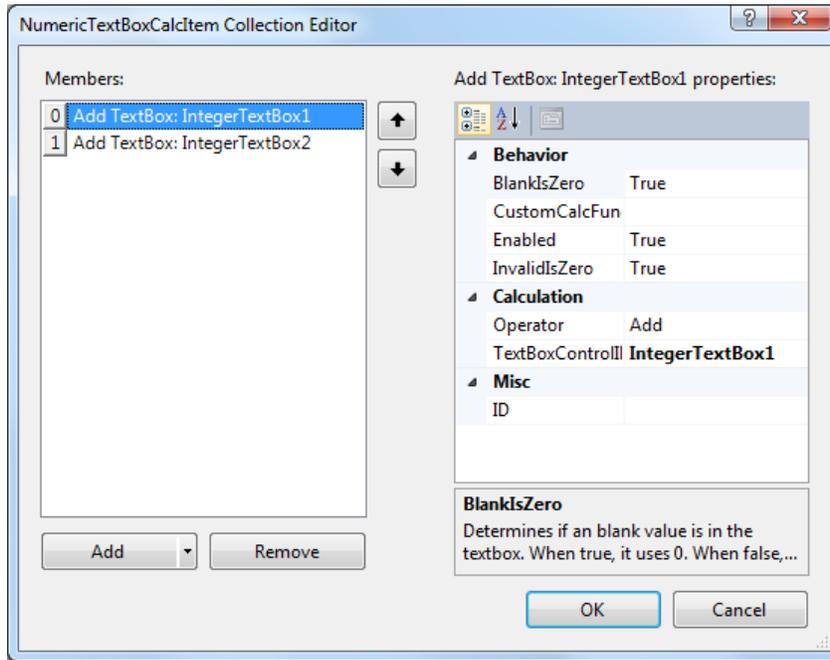
Calculating The Value Properties

The Properties Editor shows these properties in the “Calculation” category.

- **Expression** (PeterBlum.DES.BaseCalcExpression) – A list of CalcItem objects that define the calculation expression. See “Creating the Expression”.

Visual Studio and Visual Web Developer Design Mode Users

The Properties Editor for the **Expression** property provides a window where you can select CalcItem objects and establish their properties.



- The **Add** button contains a list the CalcItem classes. Select one to add to the end of the current list.
- Establish the properties in the Properties grid.
- When you see the following properties, they open to their own copy of the same Properties Editor: **Expression**, **ExpressionWhenTrue**, and **ExpressionWhenFalse**.
- Click **OK**.

ASP.NET Markup for the Expression Property

You add the **Expression** as child of the <Expression> tag.

The following example represents expression (IntegerTextBox1 + IntegerTextBox2) * 25:

```
<des:CalculationController id="CalculationController1" runat="server">
  <Expression>
    <des:ParenthesisCalcItem>
      <Expression>
        <des:NumericTextBoxCalcItem TextBoxControlID="IntegerTextBox1" />
        <des:NumericTextBoxCalcItem TextBoxControlID="IntegerTextBox2" />
      </Expression>
    </des:ParenthesisCalcItem>
    <des:ConstantCalcItem Constant="25" Operator="Multiply" />
  </Expression>
</des:CalculationController>
```

The following example represents expression DecimalTextBox1 + DecimalTextBox2 - If (CheckBox1 is checked) THEN DecimalTextBox3 ELSE 0:

```
<des:CalculationController id="CalculationController2" runat="server">
  <Expression>
    <des:NumericTextBoxCalcItem TextBoxControlID="DecimalTextBox1">
    </des:NumericTextBoxCalcItem>
    <des:NumericTextBoxCalcItem TextBoxControlID="DecimalTextBox2">
    </des:NumericTextBoxCalcItem>
    <des:ConditionCalcItem Operator="Subtract">

      <ConditionContainer>
        <des:CheckStateCondition ControlIDToEvaluate="CheckBox1" >
        </des:CheckStateCondition>
      </ConditionContainer>

      <ExpressionWhenTrue>
        <des:NumericTextBoxCalcItem TextBoxControlID="DecimalTextBox1">
        </des:NumericTextBoxCalcItem>
      </ExpressionWhenTrue>

      <ExpressionWhenFalse>
        <des:ConstantCalcItem Constant="0">
        </des:ConstantCalcItem>
      </ExpressionWhenFalse>

    </des:ConditionCalcItem>
  </Expression>
</des:CalculationController>
```

Notice that the **ConditionCalcItem.Condition** property name never appears in the attributes of the <des:ConditionCalcItem> tag. (It will be added when using the Properties Editor but it's completely meaningless.) Instead, the <ConditionContainer> tag is a child of the ConditionCalcItem control tag. That tag has no attributes. The child to <ConditionContainer> defines the class and all properties of the Condition:

```
<des:classname[all properties] />
```

- des:classname – Use any Condition class for the classname. If you create your own classes, you must declare the namespace using the <% @REGISTER %> tag at the top of the page.
- [all properties] – Enter the properties into the tag the same way you do for any other control.

Programmatically Adding To Expression

Here are the steps to set the **Expression**.

- Create an instance of the desired CalcItem. Here are the available constructors shown with property names in the parameters.

```

NumericTextBoxCalcItem()
NumericTextBoxCalcItem(Operator)
NumericTextBoxCalcItem(Operator, TextBoxControlID)
NumericTextBoxCalcItem(Operator, TextBoxInstance)

ConstantCalcItem()
ConstantCalcItem(Operator)
ConstantCalcItem(Operator, Constant)

ListConstantsCalcItem()
ListConstantsCalcItem(Operator)
ListConstantsCalcItem(Operator, ListControlID)
ListConstantsCalcItem(Operator, ListInstance)

CheckStateCalcItem()
CheckStateCalcItem(Operator)
CheckStateCalcItem(Operator, CheckStateControlID, ValueWhenChecked)
CheckStateCalcItem(Operator, CheckStateInstance, ValueWhenChecked)
CheckStateCalcItem(Operator, CheckStateControlID,
    ValueWhenChecked, ValueWhenUnchecked)
CheckStateCalcItem(Operator, CheckStateInstance,
    ValueWhenChecked, ValueWhenUnchecked)
CheckStateCalcItem(Operator, CheckStateControlID, Index, ValueWhenChecked)
CheckStateCalcItem(Operator, CheckStateInstance, Index, ValueWhenChecked)
CheckStateCalcItem(Operator, CheckStateControlID, Index,
    ValueWhenChecked, ValueWhenUnchecked)
CheckStateCalcItem(Operator, CheckStateInstance, Index,
    ValueWhenChecked, ValueWhenUnchecked)

ParenthesisCalcItem()
ParenthesisCalcItem(Operator)

ConditionCalcItem()
ConditionCalcItem(Operator)
ConditionCalcItem(Operator, Condition)

CalcControllerCalcItem()
CalcControllerCalcItem(Operator)
CalcControllerCalcItem(Operator, ControlID)
CalcControllerCalcItem(Operator, ControlInstance)

TotalingCalcItem()
TotalingCalcItem(Operator)
TotalingCalcItem(Operator, ListControlID)
TotalingCalcItem(Operator, ListInstance)
TotalingCalcItem(Operator, ListControlID, ControlIDInRow)
TotalingCalcItem(Operator, ListInstance, ControlIDInRow)

```

- Assign property values. For ParenthesisCalcItem, fill in its **Expression** property using these same steps. For ConditionCalcItem, fill in its **ExpressionWhenTrue** and **ExpressionWhenFalse** properties using these same steps.
- Assign the CalcItem object to the **Expression** property by passing it to the Add() method.

AN EXAMPLE FOLLOWS

Example

The following example represents expression (DecimalTextBox1 + DecimalTextBox2) * 25:

[C#]

```
PeterBlum.DES.Web.WebControls.ParenthesisCalcItem vSubExp1 =
    new PeterBlum.DES.Web.WebControls.ParenthesisCalcItem();
// DecimalTextBox1 inside the parenthesis
PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem vDTB1 =
    new PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem(
        PeterBlum.DES.CalcOperator.Add, DecimalTextBox1);
vSubExp1.Expression.Add(vDTB1);
// DecimalTextBox2 inside the parenthesis
PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem vDTB2 =
    new PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem(
        PeterBlum.DES.CalcOperator.Add, DecimalTextBox2);
vSubExp1.Expression.Add(vDTB2);
CalculationController1.Expression.Add(vSubExp1);
// Constant * 25
PeterBlum.DES.Web.WebControls.ConstantCalcItem vConst =
    new PeterBlum.DES.Web.WebControls.ConstantCalcItem(
        PeterBlum.DES.CalcOperator.Multiply, 25.0);
CalculationController1.Expression.Add(vConst);
```

[VB]

```
Dim vSubExp1 As PeterBlum.DES.Web.WebControls.ParenthesisCalcItem = _
    New PeterBlum.DES.Web.WebControls.ParenthesisCalcItem()
' DecimalTextBox1 inside the parenthesis
Dim vDTB1 As PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem = _
    New PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem( _
        PeterBlum.DES.CalcOperator.Add, DecimalTextBox1)
vSubExp1.Expression.Add(vDTB1)
' DecimalTextBox2 inside the parenthesis
Dim vDTB2 As PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem = _
    New PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem(
        PeterBlum.DES.CalcOperator.Add, DecimalTextBox2)
vSubExp1.Expression.Add(vDTB2)
CalculationController1.Expression.Add(vSubExp1)
' Constant * 25
Dim vConst As PeterBlum.DES.Web.WebControls.ConstantCalcItem = _
    New PeterBlum.DES.Web.WebControls.ConstantCalcItem( _
        PeterBlum.DES.CalcOperator.Multiply, 25.0)
CalculationController1.Expression.Add(vConst)
```

- **RoundMode** (enum PeterBlum.DES.CalcRoundMode) – Determines if the result of the calculation is rounded or truncated. It can round based on the number of decimal places defined in **DecimalPlaces**. When **DecimalPlaces** is Auto and **ShowValueControlID** is assigned to a numeric TextBox, it uses the number of decimal places determined by the numeric TextBox. For example, an IntegerTextBox uses 0 decimal places, which means it rounds to a whole number. For all other situations, when **DecimalPlaces** is Auto, there is no rounding or truncating.

The enumerated type PeterBlum.DES.CalcRoundMode has these values:

- Truncate – Truncate after the number of decimal digits specified by **DecimalPlaces**.
For example, if **DecimalPlaces** is 1, 2.9 is 2.9; 2.99 is 2.9; 3 is 3.
- Currency – Applies “Banker rules” where it rounds up after the digit after the number of decimal digits specified by **DecimalDigits** is 5 or higher but only when it will round up to an even number. All other cases round down (truncate). This is how the .net System.Math.Round() method works. On the server side, it in fact uses System.Math.Round().

For example, if **DecimalPlaces** is 1, 2.9 is 2.9; 2.99 is 3.0 (because it rounds the 9 up to 10), 2.89 is 2.8 (because 8 is already an even number.)

RoundMode defaults to Currency.

- **Point5** – Round up when the digit after the number of decimal digits specified by **DecimalDigits** is 5 or higher. All other cases round down (truncate). When the value is a negative number, rounding “up” makes a larger negative number.

For example, if **DecimalPlaces** is 1, 2.9 is 2.9; 2.99 is 3.0; 2.95 is 3.0; 2.94 is 2.9; -2.99 is -3.0; -2.94 is -2.9.

- **Ceiling** – Round up when the digits after the number of decimal digits specified by **DecimalDigits** is not zero. When the value is a negative number, it makes a smaller negative number. On the server-side, it uses `System.Math.Ceiling()`.

For example, if **DecimalPlaces** is 1, 2.9 is 2.9; 2.900000 is 2.9; 2.99 is 3.0; 2.90001 is 3.0; -2.900 is -2.9; -2.99 is -2.9.

- **NextWhole** – Round up when the digits after the number of decimal digits specified by **DecimalDigits** is not zero. When the value is a negative number, it makes a larger negative number. It is almost identical to `Ceiling` except it makes negative numbers larger.

For example, if **DecimalPlaces** is 1, 2.9 is 2.9; 2.900000 is 2.9; 2.99 is 3.0; 2.90001 is 3.0; -2.900 is -2.9; -2.99 is -3.0.

- **Value** (System.Double) – Gets the calculated value based on the current values in the controls associated with this expression. Always test **IsValid** is true first. If **IsValid** is false, this returns 0.0.

The first time this method is called, its value is cached. If you change any of the values in TextBoxes, call `Recalculate()` before using **Value** again. `Recalculate()` takes no parameters and returns no value. For example:

```
CalculationController1.Recalculate()
```

- **ValueText** (string) – Converts **Value** into a string. If **IsValid** is false, it returns "". This is used by Validators on the server side.
- **IsValid** (boolean) – When true, the calculated value is valid and reflected in **Value**. When false, the calculation had an error and **Value** is 0.0.

Showing The Value Properties

The Properties Editor shows these properties in the “Show Value” category.

- **ShowValueControlID** (string) – The ID to an IntegerTextBox, DecimalTextBox, CurrencyTextBox, PercentTextBox, Label, or LocalizableLabel that will display the result of the calculation.

This ID must be in the same or an ancestor naming container. If it is in another naming container, use **ShowValueInstance**.

If this and **ShowValueInstance** are unassigned, the calculation is not displayed.

Formatting, such as decimal character, currency symbol, and thousands separator are determined by [PeterBlum.DES.Globals.WebFormDirector.CultureInfo](#) and other properties in this section.

It defaults to "".

- **ShowValueInstance** (System.Web.UI.Control) – A reference to an IntegerTextBox, DecimalTextBox, CurrencyTextBox, PercentTextBox, Label, or LocalizableLabel that will display the result of the calculation. It is an alternative to **ShowValueControlID** that you must assign programmatically. It accepts controls in any naming container.

When programmatically assigning properties, if you have access to the control object that will be displayed, it is better to assign it here than assign its ID to the **ShowValueControlID** property because DES operates faster using **ShowValueInstance**.

- **InvalidValueLabel** (string) – When there is an error while calculating, this text is assigned to the control displaying the value.

If you assign this to a numeric textbox that has a DataTypeCheckValidator assigned, considering adding an [Enabler](#) to the DataTypeCheckValidator that disables it when this value is present. For the Enabler, use a CompareToValueCondition with **Operator** of `NotEqual` and **ValueToCompare** with the same text as this property.

It defaults to "".

- **InvalidValueCssClass** (string) – When there is an error while calculating, you can change the appearance of the control displaying the value by assigning a style sheet class name here.

It is not used when "".

It defaults to "".

- **DecimalPlaces** (enum PeterBlum.DES.CalcDecimalPlaces) – The number of decimal places to round or truncate the value. When the control to display is assigned to a Label control, this also determines the overall format of the value written into the Label.

The enumerated type `PeterBlum.DES.CalcDecimalPlaces` has these values:

- `Auto` – Depends on the type of control to display:
 - `IntegerTextBox` – 0 decimal places.
 - `DecimalTextBox` – Use the **DecimalTextBox.MaxDecimalPlaces** property. If **MaxDecimalPlaces** is 0, show as many decimal places as needed.
 - `CurrencyTextBox` – Use the number of decimal places for Currency defined in [PeterBlum.DES.Globals.WebFormDirector.CultureInfo](#).
 - `Labels` - Show as many decimal places as needed. Do not round or truncate. When you want to show a currency, always use `Currency`.

It defaults to `Auto`.

- `Integer` – 0 decimal places
- `Decimal1` – 1 decimal place
- `Decimal2` – 2 decimal places
- `Decimal3` – 3 decimal places

- Decimal4 – 4 decimal places
- Decimal5 – 5 decimal places
- Decimal6 – 6 decimal places
- Decimal7 – 7 decimal places
- Decimal8 – 8 decimal places
- Decimal9 – 9 decimal places
- Decimal10 – 10 decimal places
- Currency - Use the number of decimal places for Currency defined in [PeterBlum.DES.Globals.WebFormDirector.CultureInfo](#).
- **LabelFormatThousandsSep** (Boolean) – When the control to display is a Label and this is `true`, thousands separators are shown according to the rules of [PeterBlum.DES.Globals.WebFormDirector.CultureInfo](#). Numeric TextBoxes have their own properties to show thousands separators. It defaults to `false`.
- **LabelFormatCurrencySymbol** (Boolean) – When the control to display is a Label and this is `true`, the currency symbol are shown according to the rules of [PeterBlum.DES.Globals.WebFormDirector.CultureInfo](#). Numeric TextBoxes have their own properties to show the currency symbol. It defaults to `false`. This setting requires **DecimalPlaces** = Currency.
- **LabelToken** (string) – When the control to display is a label, the entire text or just a token within that text can be replaced. When this property is assigned, it is a token to be replaced. Otherwise, the entire text is replaced.
This allows the value to be inserted into a larger string, like a sentence. It also allows multiple CalculationControllers to update a common Label. For example, the Label.Text is “The result is {CALC}” and **LabelToken** is "{CALC}”.
To use it, add a token such as "{CALC}" or "{0}" into the Label's Text where the calculation belongs. There should only be one instance of this token per Label.
When it is "", the entire text of the Label is replaced.
When assigned, this exact text (case sensitive) is replaced with the calculation value where it appears in the Label's current text. You should only define one token in your Label's text.
It defaults to "".
- **AutoShowValue** (enum PeterBlum.DES.Web.AutoShowValueMode) – Determines if the control to display the value is assigned the result of the calculation on the server side. When it is used, the calculation will occur during the control's PreRender stage using the current values in the textboxes. Often users leave it off when the page is first created but have it calculated on post back because it now reflects the user's data.
You can always call the `CalculationController.ShowValue()` method to calculate and update the control with the value as an alternative to using this property.
The enumerated type `PeterBlum.DES.Web.AutoShowValueMode` has these values:
 - Off - It never displays the calculation result in PreRender. You can still use `ShowValue()` to set it.
 - PostBack - Display on postback. This is the default.
 - Always - Display when the page is first created and on postback.
- **ShowValue()** – This method will calculate and display the value in the control identified by **ShowValueControlID** or **ShowValueInstance**. It is an alternative to using [AutoShowValue](#). Usually you call it after setting valid values in your textboxes.

`ShowValue()` takes no parameters and returns no result. Here is an example on `CalculationController1`.

```
CalculationController1.ShowValue()
```

When to Use the Control Properties

The Properties Editor shows these properties in the “When To Use” category.

- **Visible** (Boolean) – Determines if the control is used or not. When `true`, it is used. When `false`, it is not. It defaults to `true`.
- **RunOnlyOnDemand** (Boolean) – Indicates that this calculation cannot run on the client-side unless you call the `DES_CalcOnDemand()` javascript function. You generally setup the `onchange` event to textboxes that will invoke this function.

When `false`, the calculation runs automatically.

When `true`, the calculation only runs if you call `DES_CalcOnDemand(ID)`. The function parameter is the `ClientID` value for the `CalculationController`.

It defaults to `false`.

For example, the field showing the calculation result is an `IntegerTextBox` where you don’t want it to be updated after the user edits that textbox. You could write a javascript function that knows the state of editing on the `IntegerTextBox` and calls `DES_CalcOnDemand()` when it has not been edited. Attach the `onchange` event of any textboxes that normally invoke this calculation to call your function.

```
<script type='text/javascript' >
var gResultWasEdited = false;
function UpdateResult(pSourceField)
{
    if (!gResultWasEdited)
        DES_CalcOnDemand("<% =CalculationController1.ClientID %>");
}
</script>
```

In `Page_Load()`, setup the `onchange` events using the

`PeterBlum.DES.Globals.WebFormDirector.AttachCodeToEvent()` function (which is required when using the DES numeric textboxes).

```
PeterBlum.DES.Globals.WebFormDirector.AttachCodeToEvent(
    EnterValueTextBox, "onchange", "UpdateResult(this);", false);
PeterBlum.DES.Globals.WebFormDirector.AttachCodeToEvent(
    ResultTextBox, "onchange", "gResultWasEdited = true", false);
```

- **Enabler** (`PeterBlum.DES.IBaseCondition`) – There are times when a `CalculationController` should be disabled. For example, don’t calculate because a textbox in the calculation is invisible. These rules are formed by `Condition` classes with the **Enabler** property on each `CalculationController`. By default, the **Enabler** property is set to “None”, where it doesn’t disable the control. You can set it to any `Condition`, including those you may create programmatically.

Consider these issues when using the **Enabler**:

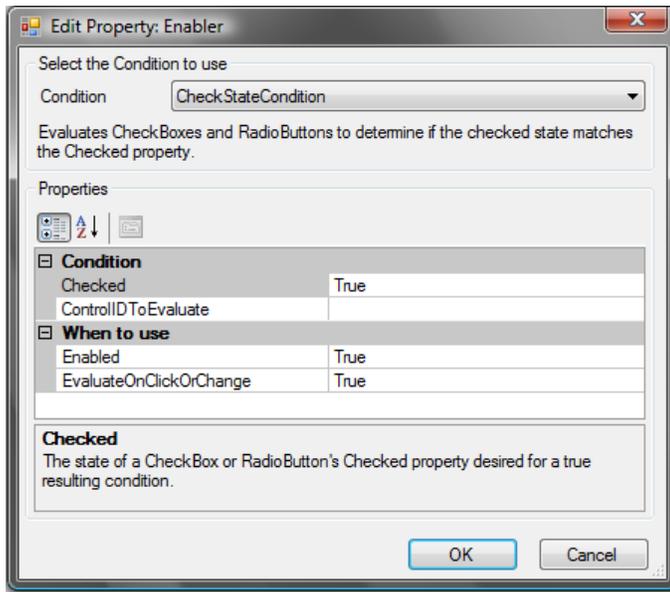
- Most `Conditions` have a property called **EvaluateOnClickOrChange** which defaults to `true`. Change it to `false` when using it in an **Enabler**.
- Do not use this to detect a control whose **Visible** property is set to `false`. Such a control does not create HTML for the client-side to use. Instead, set the **Enabled** property to `false` when the control is invisible.

 [Online example](#)

CONTINUED ON THE NEXT PAGE

Visual Studio and Visual Web Developer Design Mode Users

The Properties Editor offers this window to select a Condition and to edit its properties.



1. Select the Condition from the DropDownList. See “[The Condition classes](#)” for available Condition classes.
2. Establish the properties in the Properties grid.
3. Reminder: Be sure to set **EvaluateOnClickOrChange** to false on all Conditions within the Enabler as shown.
4. Click **OK**.

ASP.NET Markup for the Enabler Property

If you want to enter the **Enabler** property and its child properties into the web form using the HTML mode, there are special considerations. The format is very unusual, in part because the .Net framework doesn’t support changing the class of a property (polymorphism) without an interesting hack.

Here is the CalculationController with the **Enabler** set to the CheckStateCondition.

```
<des:CalculationController id="CalculationController1" runat="server" >
  <EnablerContainer>
    <des:CheckStateCondition ControlIDToEvaluate="CheckBox1"
      EvaluateOnClickOrChange="false" />
  </EnablerContainer>
</des:CalculationController >
```

*Reminder: Be sure to set **EvaluateOnClickOrChange** to false on all Conditions within the Enabler as shown.*

Notice that the **Enabler** property never appears in the attributes of the `<des:CalculationController>` tag. (*It will be added when using the Properties Editor but it’s completely meaningless.*) Instead, the `<EnablerContainer>` tag is a child of the CalculationController tag. That tag never has any attributes. The child to `<EnablerContainer>` defines the class and all properties of the [Condition](#):

```
<des:classname [all properties] />
```

- o `des:classname` – Use any Condition class for the classname. If you create your own classes, you must declare the namespace using the `<% @REGISTER %>` tag at the top of the page.
- o `[all properties]` – Enter the properties into the tag the same way you do for any other control.

Programmatically Setting The Enabler

Here are the steps to set the **Enabler**.

1. Create an instance of the desired Condition. There is a constructor that takes no parameters.

Note: There are also constructors that take parameters representing some of the control's properties. Each demands an "owner" in the first parameter. That value must be the FieldStateController object.

2. Assign property values.

Reminder: Be sure to set EvaluateOnClickOrChange to false on all Conditions within the Enabler.

3. Assign the Condition object to the **Enabler** property.

In this example, add the CheckStateCondition, which is checking CheckBox1, to CalculationController1.

[C#]

```
PeterBlum.DES.Web.WebControls.CheckStateCondition vCond =  
    new PeterBlum.DES.Web.WebControls.CheckStateCondition();  
vCond.ControlToEvaluate = CheckBox1;  
vCond.EvaluateOnClickOrChange = false;  
CalculationController1.Enabler = vCond;
```

[VB]

```
Dim vCond As PeterBlum.DES.Web.WebControls.CheckStateCondition = _  
    New PeterBlum.DES.Web.WebControls.CheckStateCondition()  
vCond.ControlToEvaluate = CheckBox1  
vCond.EvaluateOnClickOrChange = False  
CalculationController1.Enabler = vCond
```

Behavior Properties

The Properties Editor shows these properties in the “Behavior” category.

- **InAJAXUpdate** (Boolean) – When the page uses AJAX callbacks to add, update, or remove this control, set this to `true`. It defaults to `false`.

In addition, if any of these properties identify a control that participates in the AJAX callback, set this to `true`:

- **ShowValueControlID** and **ShowValueInstance**
- Any control identified inside of the **Expression** property, including textboxes, lists, and other CalculationControllers.
- **Enabler**. Look at the **ControlIDToEvaluate** and **SecondControlIDToEvaluate**.
- **ExtraControlsToRunThisAction**.

Note: This is only needed for non-DES controls. DES controls will tell the FieldStateController if their own IsAJAXUpdate property is true.

See “Using These Controls with AJAX” in the **General Features Guide**.

- **ValidateOnCalc** (boolean) – When `true`, client-side validation is applied to any Validators evaluating to this control after a numeric TextBox involved in this **Expression** is edited. When `false`, Validators only update when the page is submitted. It defaults to `false`.
- **ExtraControlsToRunThisAction** (PeterBlum.DES.Web.WebControls.ControlConnectionCollection) – Identifies additional controls and elements on the page that run this CalculationController when clicked or changed.

The **Expression** already identifies controls through its NumericTextBoxCalcItem, ConditionCalcItem, and ListConstantsCalcItem objects so this is rarely needed.

This property is a collection of `PeterBlum.DES.Web.WebControls.ControlConnection` objects. You can assign the control’s ID to the **ControlConnection.ControlID** property or a reference to the control in the **ControlConnection.ControlInstance** property. When using the **ControlID** property, the control must be in the same or an ancestor naming container. If it is in another naming container, use **ControlInstance**.

Here are some considerations:

- Be sure that the control assigned to this collection has the `runat=server` property.

ASP.NET Markup for the ExtraControlsToRunThisAction Property

ExtraControlsToRunThisAction is a type of collection. Therefore its ASP.NET Markup is nested as a series of child controls within the `<ExtraControlsToRunThisAction>` tag. Here is an example.

```
<des:CalculationController id="CalcController1" runat="server">
  <ExtraControlsToRunThisAction>
    <des:ControlConnection ControlID="TextBox1" />
    <des:ControlConnection ControlID="Label1" />
  </ExtraControlsToRunThisAction>
</des:CalculationController>
```

Programmatically adding to the ExtraControlsToRunThisAction Property

Use the `ExtraControlsToRunThisAction.Add()` method to add an entry. This overloaded method takes one parameter. Choose from the following:

- A reference to the control itself. This is the preferred form.
- A string giving the ID of the control. Do not use this when the control is not in the same naming container.
- An instance of the class `PeterBlum.DES.Web.WebControls.ControlConnection`.

This example shows how to update an existing `PeterBlum.DES.Web.WebControls.ControlConnection` and add a new entry. Suppose the ASP.NET code looks like the text above and the `Label1` control is not in the same or ancestor naming container. Also suppose the control referenced in the property `TextBox2` control must be added.

[C#]

```
uses PeterBlum.DES;
...
ControlConnection vConnection = (ControlConnection)
    CalcController1.ExtraControlsToRunThisAction[1];
vConnection.ControlInstance = Label1;
// add TextBox2. It can be either a control reference or its ID
CalcController1.ExtraControlsToRunThisAction.Add(TextBox2);
```

[VB]

```
Imports PeterBlum.DES
...
Dim vConnection As ControlConnection = _
    CType(CalcController1.ExtraControlsToRunThisAction(1), ControlConnection)
vConnection.ControlInstance = Label1
' add TextBox2. It can be either a control reference or its ID
CalcController1.ExtraControlsToRunThisAction.Add(TextBox2)
```

- **ViewStateMgr** (`PeterBlum.DES.Web.WebControls.ViewStateMgr`) – Enhances the ViewState on this control to provide more optimal storage and other benefits. Normally, the properties of this control and its segments are not preserved in the ViewState. When working in ASP.NET markup, define a pipe delimited string of properties in the **PropertiesToTrack** property. When working in code, call `ViewStateMgr.TrackProperty("propertyname")` to save the property. Individual segments have a similar method: `TrackPropertyInViewState("propertyname")`.

For more details, see “The ViewState and Preserving Properties forPostBack” in the **General Features User’s Guide**.

- **PropertiesToTrack** (string) – A pipe delimited list of properties to track. Designed for use in markup and the properties editor. The ViewState is not automatically used by most of these properties. To include a property, add it to this pipe delimited list.

For example, "Group|MayMoveOnClick".

When working programmatically, use `ViewStateMgr.TrackProperty("PropertyName")`.

Properties on CalcItem Classes

The CalcItem classes are used to build the expressions. See “[Creating the Expression](#)”. This section describes the properties of each class.

Click on any of these topics to jump to them:

- ◆ [Properties Common To All CalcItem Classes](#)
- ◆ [Properties for the PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem Class](#)
- ◆ [Properties for the PeterBlum.DES.Web.WebControls.ListConstantsCalcItem Class](#)
- ◆ [Properties for the PeterBlum.DES.Web.WebControls.CheckStateCalcItem Class](#)
- ◆ [Properties for the PeterBlum.DES.Web.WebControls.ConstantCalcItem Class](#)
- ◆ [Properties for the PeterBlum.DES.Web.WebControls.ParenthesisCalcItem Class](#)
- ◆ [Properties for the PeterBlum.DES.Web.WebControls.ConditionCalcItem Class](#)
- ◆ [Properties for the PeterBlum.DES.Web.WebControls.CalcControllerCalcItem Class](#)
- ◆ [Adding Custom Code to a CalcItem](#)
 - [The Client-Side Function and the CustomCalcFunctionName Property](#)
 - [The Server Side Event Handler and CustomCalculation Property](#)

Properties Common To All CalcItem Classes

- **Operator** (enum `PeterBlum.DES.CalcOperator`) – Determines if this value is added, subtracted, multiplied, or divided against values previously added to the expression. If there are no previous values, it calculates against 0. If this `CalcItem` object is the first in an expression, it is recommended that you leave this set to `Add`.

The enumerated type `PeterBlum.DES.CalcOperator` has these values:

- `Add` – This is the default.
 - `Subtract`
 - `Multiply`
 - `Divide`
- **Enabled** (Boolean) – Determines if this `CalcItem` is used in the calculation. When `true`, it is used. It defaults to `true`.
Often you use this when you define an expression on the ASP.NET Markup definition and need to customize it programmatically. When you do so, you usually also assign the **ID** property so you can search for this `CalcItem` object.
 - **ID** (string) – An optional ID used to allow a search for this element throughout the **Expression** or to let their custom functions identify the `CalcItem` object. (See [“Adding Custom Code to a CalcItem”](#).)

The user can leave it blank if not programmatically searching for this `CalcItem` object. When assigned, it should be unique amongst all `CalcItems` objects in this Expression. When "", it is not used. It defaults to "".

Use the `CalculationController.FindByID()` method to search for this `CalcItem` by ID. `FindByID()` takes one parameter, the ID to locate (as a string). It returns the `CalcItem` object as the interface `PeterBlum.DES.IBaseCalcItem` or null/nothing if no matching `CalcItem` is found. Usually you will typecast the result to the desired type. Here is an example that locates a `NumericTextBoxCalcItem` by the ID of “Rate1” and sets its **TextBoxInstance** property to the `DecimalTextBox1` in the field `DecimalTextBox1`.

[C#]

```
PeterBlum.DES.IBaseCalcItem vItem = CalculationController1.FindByID("Rate1");  
if (vItem != null)  
    ((PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem)vItem).TextBoxInstance =  
        DecimalTextBox1;
```

[VB]

```
Dim vItem As PeterBlum.DES.IBaseCalcItem = _  
    CalculationController1.FindByID("Rate1")  
If Not vItem Is Nothing Then  
    CType(vItem, _  
        PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem).TextBoxInstance = _  
        DecimalTextBox1  
End If
```

Properties for the PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem Class

For an overview, see “[PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem](#)”.

The `PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem` class retrieves its value from any of DES’s numeric TextBoxes – `IntegerTextBox`, `DecimalTextBox`, `CurrencyTextBox`, and `PercentTextBox`. It sets up the `CalculationController` to automatically calculate each time the textbox is changed and focus leaves the control.

The following are properties of this class:

- **Operator, Enabled, and ID** – See “[Properties Common To All CalcItem Classes](#)”.
- **CustomCalcFunctionName** and **CustomCalculation** – See “[Adding Custom Code to a CalcItem](#)”.
- **TextBoxControlID** (string) – The ID to an `IntegerTextBox`, `DecimalTextBox`, `CurrencyTextBox`, or `PercentTextBox`. This ID must be in the same or an ancestor naming container. If it is in another naming container, use **TextBoxInstance**.
If the control cannot be found in the current or any parent `NamingContainer`, an exception is thrown at runtime.
- **TextBoxInstance** (`PeterBlum.DES.INumberTextBox`) – A reference to an `IntegerTextBox`, `DecimalTextBox`, `CurrencyTextBox`, or `PercentTextBox`. It is an alternative to **TextBoxControlID** that you must assign programmatically. It accepts controls in any naming container.

When programmatically assigning properties, if you have access to the textbox control object, it is better to assign it here than assign its ID to the **TextBoxControlID** property because DES operates faster using **TextBoxInstance**.

- **InvalidIsZero** (Boolean) – Determines what to do when an invalid value is in the textbox.
When `true`, an invalid value becomes 0 in the calculation and the calculation continues.
When `false`, it stops the calculation and reports an error.
It defaults to `true`.
- **BlankIsZero** (Boolean) – Determines what to do when the textbox is empty.
When `true`, a blank textbox uses the value 0 in the calculation and the calculation continues.
When `false`, it stops the calculation and reports an error.
It defaults to `true`.

Properties for the PeterBlum.DES.Web.WebControls.ListConstantsCalcItem Class

For an overview, see “[PeterBlum.DES.Web.WebControls.ListConstantsCalcItem](#)”.

The `PeterBlum.DES.Web.WebControls.ListConstantsCalcItem` class associates the items in a `ListBox` or `DropDownList` with constants. You can define constants for each item or for a range. You can also report an error when a specific item is selected.

The following are properties of this class:

- **Operator**, **Enabled**, and **ID** – See “[Properties Common To All CalcItem Classes](#)”.
- **CustomCalcFunctionName** and **CustomCalculation** – See “[Adding Custom Code to a CalcItem](#)”.
- **ListControlID** (string) – The ID to a `ListBox`, `DropDownList`, or `System.Web.UI.HtmlControls.HtmlSelect` control. This ID must be in the same or an ancestor naming container. If it is in another naming container, use **ListInstance**.

If the control cannot be found in the current or any parent `NamingContainer`, an exception is thrown at runtime.

- **ListInstance** (`System.Web.UI.Control`) – A reference to a `ListBox`, `DropDownList`, or `System.Web.UI.HtmlControls.HtmlSelect` control. It is an alternative to **ListControlID** that you must assign programmatically. It accepts controls in any naming container.

When programmatically assigning properties, if you have access to the list control object, it is better to assign it here than assign its ID to the **ListControlID** property because DES operates faster using **ListInstance**.

- **ConstantWhenNoMatch** (double) – When the **SelectedIndex** of the list control does not find a match in the **ConstantsForSelectedIndexes** property, this value is used. It defaults to 0.
- **ErrorWhenNoMatch** (boolean) - When the **SelectedIndex** of the list control does not find a match in the **ConstantsForSelectedIndexes** property, set this to `true` to report an error to the `CalculationController`. When `false`, the value of **ConstantWhenNoMatch** is used. It defaults to `false`.

Reporting an error stops the calculation from continuing.

- **ConstantsForSelectedIndexes** (`PeterBlum.DES.ConstantsForSelectedIndexes`) – A list that defines how each item in the list control maps to a constant. This list should have at least one item.

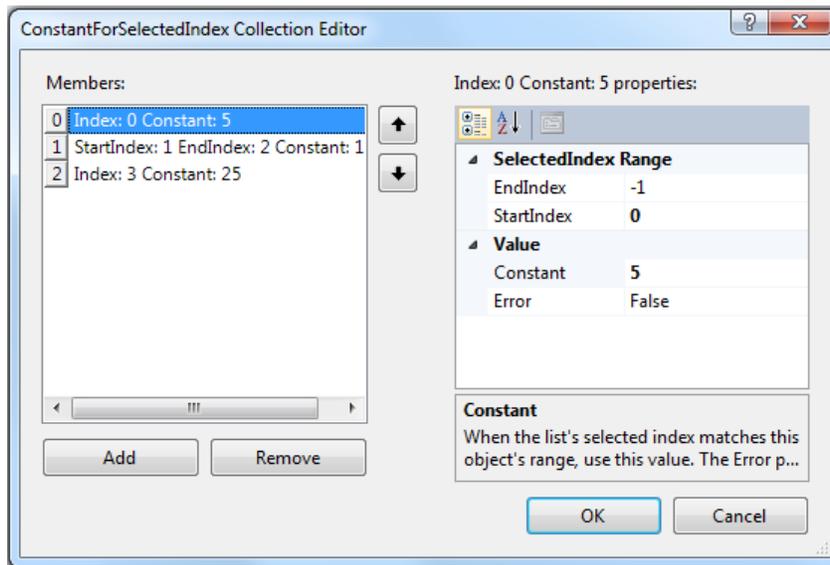
You add `PeterBlum.DES.ConstantForSelectedIndex` objects to this list. The `ConstantForSelectedIndex` class has these properties:

- **StartIndex** (integer) – The index of the item to map to the constant. If you are using a range, this is the lower index. It can be -1 to the highest position in the list control. -1 is used for no selection. 0 is for the first item shown in the list. It defaults to -1.
- **EndIndex** (integer) – When using a range, assign this to the upper index. Leave it at -1 if not using a range. It defaults to -1.
- **Constant** (double) – The numeric value that is used when this object matches the **SelectedIndex** of the list. It defaults to 0.
- **Error** (Boolean) – When `true`, report an error to the `CalculationController` instead of using the **Constant** property. When `false`, use the **Constant** property. It defaults to `false`.

Reporting an error stops the calculation from continuing.

Visual Studio and Visual Web Developer Design Mode Users

The Properties Editor for the **ConstantsForSelectedIndexes** property provides a window where you can select CalcItem objects and establish their properties.



- The Add button adds a new item. The Remove button removes the selected item.
- Establish the properties in the Properties grid.
- Click **OK**.

ASP.NET Markup for the ConstantsForSelectedIndexes Property

You add the **ConstantsForSelectedIndexes** as child of the `<ConstantsForSelectedIndexes>` tag.

The following example creates three ConstantForSelectedIndex objects that map to items in ListBox1:

```
<des:CalculationController id=CalculationController1 runat="server">
  <Expression>
    <des:ListConstantsCalcItem ListControlID="ListBox1" >
      <ConstantsForSelectedIndexes>
        <des:ConstantForSelectedIndex StartIndex="0" Constant="5" />
        <des:ConstantForSelectedIndex StartIndex="1" EndIndex="2"
          Constant="10" />
        <des:ConstantForSelectedIndex StartIndex="3" Constant="25" />
      </ConstantsForSelectedIndexes>
    </des:ListConstantsCalcItem>
  </Expression>
</des:CalculationController>
```

Programmatically Adding to the ConstantsForSelectedIndexes Property

Here are the steps to add PeterBlum.DES.ConstantForSelectedIndex objects to the ConstantsForSelectedIndexes Property.

1. Create an instance of the PeterBlum.DES.ConstantForSelectedIndex class. Here are the available constructors. Their parameters match to properties described in the class definition, above.

```
ConstantForSelectedIndex()
ConstantForSelectedIndex(Constant, StartIndex)
ConstantForSelectedIndex(Constant, StartIndex, EndIndex)
ConstantForSelectedIndex(Error, StartIndex)
ConstantForSelectedIndex(Error, StartIndex, EndIndex)
```

2. Assign property values.
3. Add the object to the **ConstantsForSelectedIndexes** property by passing it to the Add() method.

This example creates the three ConstantForSelectedIndex objects shown in the above HTML text. The ListConstantsCalcItem is assigned the ID of "ConstsForListBox1" elsewhere.

[C#]

```
PeterBlum.DES.Web.WebControls.ListConstantsCalcItem vCalcItem =
    (PeterBlum.DES.Web.WebControls.ListConstantsCalcItem) // typecast
    CalculationController1.FindByID("ConstsForListBox1");
// const=5 startindex=0
PeterBlum.DES.ConstantForSelectedIndex vCFSI =
    new PeterBlum.DES.ConstantForSelectedIndex(5.0, 0);
vCalcItem.ConstantsForSelectedIndexes.Add(vCalcItem);
// const=10, startindex=1, endindex=2
vCFSI = new PeterBlum.DES.ConstantForSelectedIndex(10.0, 1, 2);
vCalcItem.ConstantsForSelectedIndexes.Add(vCalcItem);
// Error=true, startindex=3
vCFSI = new PeterBlum.DES.ConstantForSelectedIndex(true, 3);
vCalcItem.ConstantsForSelectedIndexes.Add(vCalcItem);
```

[VB]

```
Dim vCalcItem As PeterBlum.DES.Web.WebControls.ListConstantsCalcItem = _
    CType(CalculationController1.FindByID("ConstsForListBox1"), _
    PeterBlum.DES.Web.WebControls.ListConstantsCalcItem)
' const=5 startindex=0
Dim vCFSI As PeterBlum.DES.ConstantForSelectedIndex = _
    New PeterBlum.DES.ConstantForSelectedIndex(5.0, 0)
vCalcItem.ConstantsForSelectedIndexes.Add(vCalcItem)
' const=10, startindex=1, endindex=2
vCFSI = New PeterBlum.DES.ConstantForSelectedIndex(10.0, 1, 2)
vCalcItem.ConstantsForSelectedIndexes.Add(vCalcItem)
' Error=true, startindex=3
vCFSI = New PeterBlum.DES.ConstantForSelectedIndex(true, 3)
vCalcItem.ConstantsForSelectedIndexes.Add(vCalcItem)
```

Properties for the PeterBlum.DES.Web.WebControls.CheckStateCalcItem Class

For an overview, see “[PeterBlum.DES.Web.WebControls.CheckStateCalcItem](#)”.

The `PeterBlum.DES.Web.WebControls.CheckStateCalcItem` class determines its value from one of two constants that are selected based on the check state of a `CheckBox` or `RadioButton`, including those in `CheckBoxLists` and `RadioButtonLists`.

One common usage is to add a series of checkboxes that are checked. Since this control also returns a value for the unchecked state, usually you use `ValueWhenUnchecked = 0` when adding or subtracting. You use `ValueWhenUnchecked = 1` when multiplying or dividing.

The following are properties of this class:

- **Operator**, **Enabled**, and **ID** – See “[Properties Common To All CalcItem Classes](#)”.
- **CustomCalcFunctionName** and **CustomCalculation** – See “[Adding Custom Code to a CalcItem](#)”.
- **CheckStateControlID** (string) – The ID to a `CheckBox`, `RadioButton`, `CheckBoxList`, or `RadioButtonList`. This ID must be in the same or an ancestor naming container. If it is in another naming container, use **CheckStateInstance**.
If the control cannot be found in the current or any parent `NamingContainer`, an exception is thrown at runtime.
When assigned to a `CheckBoxList` or `RadioButtonList`, set the **Index** property to the button within the list.
- **CheckStateInstance** (Control) – A reference to a `CheckBox`, `RadioButton`, `CheckBoxList`, or `RadioButtonList`. It is an alternative to **CheckStateControlID** that you must assign programmatically. It accepts controls in any naming container.
When programmatically assigning properties, if you have access to the textbox control object, it is better to assign it here than assign its ID to the **CheckStateControlID** property because DES operates faster using **CheckStateInstance**.
When assigned to a `CheckBoxList` or `RadioButtonList`, set the **Index** property to the button within the list.
- **Index** (integer) – Used with `RadioButtonList` and `CheckBoxList` controls to identify the specific button within the list whose state is evaluated.
Not used when using `RadioButton` or `CheckBox` controls.
Values start at 0 where 0 is the first button in the list.
It defaults to 0.
- **ValueWhenChecked** (double) – Gets and sets a number to use in the expression when the button is checked.
It defaults to 1.
- **ValueWhenUnchecked** (double) – Gets and sets a number to use in the expression when the button is not checked.
It defaults to 0.

Properties for the PeterBlum.DES.Web.WebControls.ConstantCalcItem Class

For an overview, see “[PeterBlum.DES.Web.WebControls.ConstantCalcItem](#)”.

The PeterBlum.DES.Web.WebControls.ConstantCalcItem class supplies a single number – a constant – into your expression.

The following are properties of this class:

- **Operator, Enabled, and ID** – See “[Properties Common To All CalcItem Classes](#)”.
- **CustomCalcFunctionName and CustomCalculation** – See “[Adding Custom Code to a CalcItem](#)”.
- **Constant** (double) – Gets and sets the number to use in the expression. It holds a decimal value using System.Double type but you can assign an integer to it too. It defaults to 1.

Properties for the PeterBlum.DES.Web.WebControls.ParenthesisCalcItem Class

For an overview, see “[PeterBlum.DES.Web.WebControls.ParenthesisCalcItem](#)”.

The `PeterBlum.DES.Web.WebControls.ParenthesisCalcItem` class creates a sub expression, where all of its elements are calculated together first before the result is used in the expression that contains this object. It's like using parenthesis in a mathematical expression.

The following are properties of this class:

- **Operator**, **Enabled**, and **ID** – See “[Properties Common To All CalcItem Classes](#)”.
- **CustomCalcFunctionName** and **CustomCalculation** – See “[Adding Custom Code to a CalcItem](#)”.
- **Expression** (`PeterBlum.DES.BaseCalcExpression`) – A list of any type of `CalcItem` objects that are calculated together. Calculations are always done from the first item in the list to the last. The first item's **Operator** property should always be `Add`.

See the [CalculationController.Expression](#) property for details on setting up this property as ASP.NET Text, in design mode, and programmatically.

Properties for the PeterBlum.DES.Web.WebControls.ConditionCalcItem Class

For an overview, see “[PeterBlum.DES.Web.WebControls.ConditionCalcItem](#)”.

The `PeterBlum.DES.Web.WebControls.ConditionCalcItem` class lets you select between two sub expressions using IF statement logic. You can select from any of DES’s `Condition` objects (the class that Validators use to evaluate data) See “[The Condition classes](#)” for more.

With IF statement logic, you can change the mathematical expression based on settings on the web form. For example, if a checkbox is used to enable a `NumericTextBox`, you will use this to determine if the checkbox is checked before using a `NumericTextBoxCalcItem` to include its value in the calculation.

The following are properties of this class:

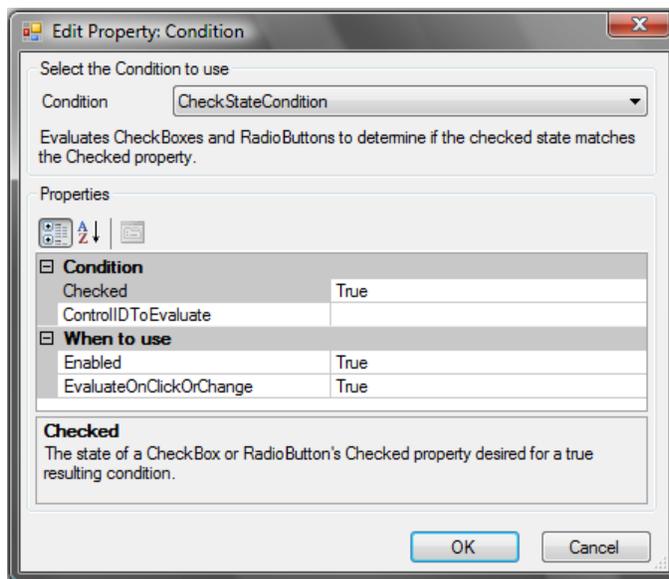
- **Operator**, **Enabled**, and **ID** – See “[Properties Common To All CalcItem Classes](#)”.
- **CustomCalcFunctionName** and **CustomCalculation** – See “[Adding Custom Code to a CalcItem](#)”.
- **Condition** (`PeterBlum.DES.IBaseCondition`) – The `Condition` object used to select between **ExpressionWhenTrue** and **ExpressionWhenFalse**. You can assign any Condition class. Initially its value is `null`; you must assign a Condition object or runtime exception will occur.

When Conditions are evaluated, they return one of three states: “success”, “failed”, and “cannot evaluate”:

- “success” calculates the expression defined in **ExpressionWhenTrue**.
- “failed” calculates the expression defined in `ExpressionWhenFalse`. If you set **InvalidWhenFalse** to true, it will stop the calculation and report an error. This is a common situation, where anything but the “success” state is considered an error.
- “cannot evaluate” uses the **CannotEvalMode** property to determine the action to take. It can select either **ExpressionWhenTrue** or **ExpressionWhenFalse**, report an error, or return 0.

Visual Studio and Visual Web Developer Design Mode Users

The Properties Editor offers this window to select a Condition and to edit its properties.



1. Select the Condition from the DropDownList. See “[The Condition classes](#)” for a list of available conditions.
2. Establish the properties in the Properties grid.
3. Click **OK**.

ASP.NET Markup for the Condition Property

If you want to enter the **Condition** property and its child properties into the web form using the HTML mode, there are special considerations. The format is very unusual, in part because the .Net framework doesn't support changing the class of a property (polymorphism) without an interesting hack.

Here is the CalculationController with a ConditionCalcItem whose Condition is set to the CheckStateCondition looking at CheckBox1.

```
<des:CalculationController id=CalculationController1 runat="server">
  <Expression>
    <des:ConditionCalcItem>

      <ConditionContainer>
        <des:CheckStateCondition ControlIDToEvaluate="CheckBox1" />
      </ConditionContainer>

      <ExpressionWhenTrue>
        <des:NumericTextBoxCalcItem TextBoxControlID="DecimalTextBox1" />
      </ExpressionWhenTrue>

      <ExpressionWhenFalse>
        <des:ConstantCalcItem Constant="0" />
      </ExpressionWhenFalse>

    </des:ConditionCalcItem>
  </Expression>
</des:CalculationController>
```

Notice that the **Condition** property never appears in the attributes of the <des:ConditionCalcItem> tag. (*It will be added when using the Properties Editor but it's completely meaningless.*) Instead, the <ConditionContainer> tag is a child of the ConditionCalcItem tag. That tag never has any attributes. The child to <ConditionContainer> defines the class and all properties of the Condition:

```
<des:classname [all properties] />
```

- o des:classname – Use any Condition class for the classname. If you create your own classes, you must declare the namespace using the <% @REGISTER %> tag at the top of the page.
- o [all properties] – Enter the properties into the tag the same way you do for any other control.

Programmatically Setting The Condition

Here are the steps to set the **Condition**.

1. Create an instance of the desired Condition class. There is a constructor that takes no parameters.
Note: There are also constructors that take parameters. Each demands an "owner" in the first parameter. That value must be the CalculationController object.
2. Assign property values.
3. Assign the **Condition** object to the **Condition** property.

In this example, add the CheckStateCondition, which is checking CheckBox1, to ConditionCalcItem object.

[C#]

```
PeterBlum.DES.Web.WebControls.ConditionCalcItem vConditionCalcItem =
    new PeterBlum.DES.Web.WebControls.ConditionCalcItem();
PeterBlum.DES.Web.WebControls.CheckStateCondition vCond =
    new PeterBlum.DES.Web.WebControls.CheckStateCondition();
vCond.ControlToEvaluate = CheckBox1;
vConditionCalcItem.Condition = vCond;
```

[VB]

```
Dim vConditionCalcItem As PeterBlum.DES.Web.WebControls.ConditionCalcItem = _
    New PeterBlum.DES.Web.WebControls.ConditionCalcItem()
Dim vCond As PeterBlum.DES.Web.WebControls.CheckStateCondition = _
    New PeterBlum.DES.Web.WebControls.CheckStateCondition()
vCond.ControlToEvaluate = CheckBox1
vConditionCalcItem.Condition = vCond
```

- **ExpressionWhenTrue** (PeterBlum.DES.BaseCalcExpression) – When the **Condition** evaluates as “success”, it evaluates this expression. If left empty, it returns a value of 0.

This is a list of any type of CalcItem objects that are calculated together. Calculations are always done from the first item in the list to the last. The first item’s **Operator** property should always be Add.

See the **CalculationController.Expression** property for details on setting up this property as ASP.NET Text, in design mode, and programmatically.

- **ExpressionWhenFalse** (PeterBlum.DES.BaseCalcExpression) – When the **Condition** evaluates as “failed”, it evaluates this expression. If left empty, it returns a value of 0.

This is a list of any type of CalcItem objects that are calculated together. Calculations are always done from the first item in the list to the last. The first item’s **Operator** property should always be Add.

Sometimes you need your IF statement to report an error when the Condition evaluates as “failed”. Use **InvalidWhenFalse = true** to ignore **ExpressionWhenFalse** and report an error instead.

See the **CalculationController.Expression** property for details on setting up this property as ASP.NET Text, in design mode, and programmatically.

- **CannotEvalMode** (enum PeterBlum.DES.CalcCondCannotEvalMode) – Determines how the calculation works when the Condition evaluates as "cannot evaluate". Some Conditions cannot evaluate data until certain values exist. For example, the RangeCondition cannot evaluate until the text in the textbox is formatted to match what is demanded by the **Data Type** property.

The enumerated type PeterBlum.DES.CalcCondCannotEvalMode has these values:

- Error - Stop the calculation. It’s an error. This is the default.
- Zero - Return 0.
- True - Use **ExpressionWhenTrue**.
- False - Use **ExpressionWhenFalse** (even when **InvalidWhenFalse** is true).

- **InvalidWhenFalse** (Boolean) – Sometimes you need your IF statement to report an error when the Condition evaluates as “failed”. Normally it uses the expression defined in **ExpressionWhenFalse**. If you set **InvalidWhenFalse** to true, it will stop the calculation and report an error.

When false, use **ExpressionWhenFalse**. When true, report an error. It defaults to true.

When the Condition cannot evaluate and **CannotEvalMode** is set to False, the **ExpressionWhenFalse** is still used. This only blocks when the Condition evaluates to "failed".

Properties for the PeterBlum.DES.Web.WebControls.CalcControllerCalcItem Class

For an overview, see “[PeterBlum.DES.Web.WebControls.CalcControllerCalcItem](#)”.

The `PeterBlum.DES.Web.WebControls.CalcControllerCalcItem` class lets you use the value of another `CalculationController` on the page. This reduces the size of your client-side code, makes it easier to set up by not managing duplicate expressions, and runs faster.

The following are properties of this class:

- **Operator**, **Enabled**, and **ID** – See “[Properties Common To All CalcItem Classes](#)”.
- **CustomCalcFunctionName** and **CustomCalculation** – See “[Adding Custom Code to a CalcItem](#)”.
- **ControlID** (string) – The ID to a `CalculationController`. This ID must be in the same or an ancestor naming container. If it is in another naming container, use **ControlInstance**.

If the control cannot be found in the current or any parent `NamingContainer`, an exception is thrown at runtime.

- **ControlInstance** (`PeterBlum.DES.INumberTextBox`) – A reference to a `CalculationController`. It is an alternative to **ControlID** that you must assign programmatically. It accepts controls in any naming container.

When programmatically assigning properties, if you have access to the textbox control object, it is better to assign it here than assign its ID to the **ControlID** property because DES operates faster using **ControlInstance**.

- **InvalidIsZero** (Boolean) – Determines what to do when an invalid value is in the other `CalculationController`.

When `true`, an invalid value becomes 0 in the calculation and the calculation continues.

When `false`, it stops the calculation and reports an error.

It defaults to `true`.

Properties for the PeterBlum.DES.Web.WebControls.TotalingCalcItem Class

For an overview, see “[PeterBlum.DES.Web.WebControls.TotalingCalcItem](#)”.

The `PeterBlum.DES.Web.WebControls.TotalingCalcItem` class lets you total a column of a `ListView`, `GridView`, `DataList`, and `Repeater` control. It needs to know of a control within that column, either a numeric textbox like `IntegerTextBox` or another `CalculationController`, which is hosting a value specific to that row.

The following are properties of this class:

- **Operator**, **Enabled**, and **ID** – See “[Properties Common To All CalcItem Classes](#)”. Note that the **Operator** is applied to the result of the `TotalingCalcItem` object, not to individual column values.
- **CustomCalcFunctionName** – See “[Adding Custom Code to a CalcItem](#)”.
- **GrandTotal** (double) – Reworks the **Expression** to consider paging issues, where the list of rows show a total for the page, but it lacks the rows on other pages. Calculate the grand total of the column and assign it here. The **Expression** will add a constant based on the grand total - the initial calculation of this page's total.

This effectively adds a `ConstantCalcItem` whose value is **GrandTotal** minus the page's column total.

When `null`, the calculation is not modified.

It defaults to `null`.

- **ListControlID** (string) – The ID of the list or grid control. The control can be one of these types: `ListView`, `GridView`, `DataList`, or `Repeater`. This ID must be in the same or an ancestor naming container. If it is in another naming container, use **ListInstance**.

If the control cannot be found in the current or any parent `NamingContainer`, an exception is thrown at runtime.

Recommendation: When programmatically assigning properties, if you have access to the control object, it is better to assign it to the **ListInstance** property than assign its ID to the **ListControlID** property because DES operates faster using **ListInstance**.

- **ListInstance** (object) – A reference to the list or grid control. It is an alternative to **ListControlID** that you must assign programmatically. It accepts controls in any naming container.
- **ControlIDInRow** (string) – The ID of the control within the column of the list/grid. It can specify the ID of any of these control types: `IntegerTextBox`, `DecimalTextBox`, `CurrencyTextBox`, `PercentTextBox`, and `CalculationController`.

You can also supply the control within the column programmatically through the **GetColumnControl** event handler.

- **InvalidIsZero** (Boolean) – Determines what to do when an invalid value is in the numeric textbox or `CalculationController`.

When `true`, an invalid value becomes 0 in the calculation and the calculation continues.

When `false`, it stops the calculation and reports an error.

It defaults to `true`.

The `GetColumnControl` event handler

An alternative to using the `ControlIDInRow` property that lets you programmatically find and return a reference to the numeric textbox or `CalculationController` within the current row of the list or grid.

It is especially when:

- The control to return varies based on the situation.
- The control is nested inside of another naming container, such as a `UserControl`, within the row.

Here is the delegate for this event handler:

[C#]

```
object TotalingCalcItemGetColumnControl(
    PeterBlum.DES.IBaseControlIDTotalingCalcItem source,
    object listRowContainer)
```

[VB]

```
Function TotalingCalcItemGetColumnControl(
    ByVal source As PeterBlum.DES.IBaseControlIDTotalingCalcItem,
    ByVal listRowContainer As Object) As Object
```

Parameters

source

The `TotalingCalcItem` object that is requesting this data.

listRowContainer

The object representing the row of the list or grid. While its untyped, it is actually one of these types, based on the list or grid control used.

Control	<code>listRowContainer</code> type
List View	System.Web.UI.WebControls.ListViewItem
Grid View	System.Web.UI.WebControls.GridViewRow
Data List	System.Web.UI.WebControls.DataListItem
Repeater	System.Web.UI.WebControls.RepeaterItem

Be sure to typecast to the type. Generally you call the [FindControl\(\)](#) method on the `listRowContainer` to get the desired control.

EXAMPLE IS ON THE NEXT PAGE

Example

Programmatically adds a TotalingCalcItem to CalculationController1. The list is a ListView control and the textbox in the column is IntegerTextBox1.

[C#]

```
using PeterBlum.DES;
using PeterBlum.DES.Web.WebControls;
...
protected void Page_Load(object sender, System.EventArgs e)
{
    TotalingCalcItem vTCI = new TotalingCalcItem();
    vTCI.ListInstance = ListView1;
    vTCI.GetColumnControl +=
        new PeterBlum.DES.TotalingCalcItemGetColumnControl(GetColumnControl);
    CalculationController1.Expression.Add(vTCI);
}

protected object GetColumnControl(IBaseControlIDTotalingCalcItem source,
    object listRowContainer)
{
    return ((ListViewItem)listRowContainer).FindControl("IntegerTextBox1");
}
```

[VB]

```
Protected Sub Page_Load(ByVal sender As object, ByVal e As System.EventArgs)
    ByVal vTCI As TotalingCalcItem = New TotalingCalcItem()
    vTCI.ListInstance = ListView1
    AddHandler vTCI.GetColumnControl AddressOf GetColumnControl
    CalculationController1.Expression.Add(vTCI)
End Sub

Protected Function GetColumnControl( _
    ByVal source As IBaseControlIDTotalingCalcItem,
    ByVal listRowContainer As Object) As Object
    Return CType(listRowContainer,
        ListViewItem).FindControl("IntegerTextBox1")
End Function
```

Properties for the PeterBlum.DES.BLD.DataFieldTotalingCalcItem Class

For an overview, see “[PeterBlum.DES.BLD.DataFieldTotalingCalcItem](#)”.

The `PeterBlum.DES.Web.WebControls.TotalingCalcItem` class lets you total a column of a `ListView`, `GridView`, `DataList`, and `Repeater` control. It needs to know of a control within that column, either a numeric textbox like `IntegerTextBox` or another `CalculationController`, which is hosting a value specific to that row.

The following are properties of this class:

- **Operator**, **Enabled**, and **ID** – See “[Properties Common To All CalcItem Classes](#)”. Note that the **Operator** is applied to the result of the `DataFieldTotalingCalcItem` object, not to individual column values.
- **CustomCalcFunctionName** – See “[Adding Custom Code to a CalcItem](#)”.
- **ListControlID** (string) – The ID of the `BLDListView` control. This ID must be in the same or an ancestor naming container. If it is in another naming container, use **ListInstance**.

If the control cannot be found in the current or any parent `NamingContainer`, an exception is thrown at runtime.

Recommendation: When programmatically assigning properties, if you have access to the control object, it is better to assign it to the **ListInstance** property than assign its ID to the **ListControlID** property because DES operates faster using **ListInstance**.

- **ListInstance** (object) – A reference to the `BLDListView` control. It is an alternative to **ListControlID** that you must assign programmatically. It accepts controls in any naming container.
- **DataField** (string) – The column name to total. This column must be included in the `BLDListView`.

You can also supply the control within the column programmatically through the **GetColumnControl** event handler.

- **InvalidIsZero** (Boolean) – Determines what to do when an invalid value is in the numeric textbox.

When `true`, an invalid value becomes 0 in the calculation and the calculation continues.

When `false`, it stops the calculation and reports an error.

It defaults to `true`.

Adding Custom Code to a CalcItem

All CalcItem classes support properties to extend them with your own code. You write code both for the client-side and server-side. Your code can have these objectives:

- Ignore the original value. Supply code to return your own value. When using this technique, you usually select a ConstantCalcItem object, leaving its **Constant** property at its default, and writing code that gets the value. It is common to get the value from another control on the page, although there are other scenarios. If you get a value from another field, be aware that it may start as a string. You must convert it into an integer or decimal for the CalculationController to process it.
- Detect something special about the original value and replace it with another number on a specific condition. A ConditionCalcItem can do the same thing, so you probably will use that instead of writing custom code.
- Handle errors reported by the CalcItem object. Many of the CalcItem classes can return an error. For example, NumericTextBoxCalcItem will return an error when the textbox is blank and the **BlankIsZero** property is `false`. You might fix the error by supplying a numeric value.
- Detect that the value is illegal and report an error. For example, if you demand all values are between 1 and 5 and the CalcItem returns 6, your function can declare it as an error, stopping the calculation. A ConditionCalcItem can do the same thing by using its **InvalidWhenFalse** property.

To create custom code, you will write a JavaScript function whose name is specified in the **CustomCalcFunctionName** property. You will also write a server side method that uses the delegate `PeterBlum.DES.CalcEventHandler` and assign that method to the **CustomCalculation** property.

Click on any of these topics to jump to them:

- ◆ [The Client-Side Function and the CustomCalcFunctionName Property](#)
- ◆ [The Server Side Event Handler and CustomCalculation Property](#)
 - [Hooking up the Method to the CalcItem.CustomCalculation Property](#)

The Client-Side Function and the CustomCalcFunctionName Property

Create a client-side function in JavaScript and assign its name to the **CustomCalcFunctionName** property.

Your client-side function will be passed the value already calculated on the CalcItem to which it's attached. It will get the numeric value or an indication that an error was detected by the CalcItem object. Your function returns a numeric value – the original one or a new one, or the JavaScript value NaN to report an error and stop further calculations.

Once defined, assign the name of the function to the **CustomCalcFunctionName** property.

Note: CustomCalcFunctionName must contain only the function name, no parenthesis or parameters. Since it reflects a JavaScript function, it must match the case of the function exactly.

Your function must take these three parameters in the order shown:

- pSender (string) – The ClientID of the CalculationController that is using this function.
- pCalcItem (object) - The client-side representation of the CalcItem object that calls this function. If you want a way to share the same function with several CalcItem objects, assign the **ID** property on each uniquely. Then look at the *pCalcItem.ID* property for the same value. (Note that ID is case sensitive.)

On NumericTextBoxCalcItem's the ClientID of the TextBox is pCalcItem.CID.

- pValue (double) - The value from the calculation already performed by *pCalcItem*. It is a double.

If the CalcItem object encountered an error, pValue is NaN (a special JavaScript name indicating "not a number").

You can test for NaN with the JavaScript function `isNaN(pValue)`.

The result of the function must be assigned either to a number or NaN. The number is used in the calculation instead of the original value. NaN lets you indicate an error occurred and stop the calculation.

If you need to know how to add your JavaScript to the page, see "[Adding Your JavaScript to the Page](#)".

Example 1

This function returns the original value for numbers between 1 and 5. All other positive numbers return 5. 0 and below return an error (NaN). If an error was passed in, it returns an error. The function name "MyCalcFunction" should be assigned to **CustomCalcFunctionName**.

```
function MyCalcFunction(pSender, pCalcItem, pValue)
{
    if (isNaN(pValue)) // if an error was passed, indicate error
        return NaN;
    else if (pValue < 1) // 0 and lower return NaN
        return NaN;
    else if (pValue > 5) // return 5
        return 5;
    else // values between 1-5 return pValue
        return pValue;
}
```

Example 2

This function returns the value from a hidden input control. It must convert the value from a string to a decimal value using the JavaScript function `parseFloat` (which is not culture sensitive so it demands only digits and the period character as the decimal separator.) The hidden input control has been assigned the id "Hidden1". The function name "MyCalcFunction2" should be assigned to **CustomCalcFunctionName**.

```
function MyCalcFunction2(pSender, pCalcItem, pValue)
{
    var vFld = DES_GetById('Hidden1');
    return parseFloat(vFld.value); // returns NaN if it conversion fails
}
```

The Server Side Event Handler and CustomCalculation Property

Your server-side code must hookup a method that matches the delegate `PeterBlum.DES.CalcEventHandler` to the **CustomCalculation** property.

Your method will be passed the value already calculated on the `CalcItem` to which it's attached. It will get the numeric value or an indication that an error was detected by the `CalcItem` object. Your function returns a numeric value and a boolean flag indicating an error to stop further calculations.

Once defined, assign the method to the **CustomCalculation** property.

*Note: The **CustomCalculation** property only handles one event handler and must be assigned programmatically (it does not appear in the Properties Editor.)*

The `CalcEventHandler` is defined here:

[C#]

```
public delegate double CalcEventHandler(  
    PeterBlum.DES.ICalculationController pSender,  
    PeterBlum.DES.IBaseCalcItem pCalcItem,  
    double pValue,  
    ref bool pValid);
```

[VB]

```
Public Delegate Function CalcEventHandler( _  
    ByVal pSender As PeterBlum.DES.ICalculationController, _  
    ByVal pCalcItem As PeterBlum.DES.IBaseCalcItem, _  
    ByVal pValue As Double, _  
    ByRef pValid As Boolean) As Double
```

Parameters

pSender

The CalculationController object that contains this CalcItem.

pCalcItem

The CalcItem object. When you have several CalcItems using the same method, this can help distinguish them. It also helps to assign the **ID** property on each CalcItem so your code can identify them. Be sure to typecast this object to the appropriate CalcItem class before getting its properties.

pValue

The numeric value already determined by the CalcItem object. If the CalcItem object determined there was an error, this is 0.0 and *pValid* is false.

pValid

Determines if there is an error. When true, an error is indicated and the calculation will stop processing. You can change this value, either to report an error (set it to true) or revoke an error (set it to false and return a value as the function result.)

Return value

Your method should return a Double and set the *pValid* property like this:

- If the value returned is valid and should be used in the calculation, return the number and set *pValid* to true.
- If the value is not valid and an error should be reported, return 0.0 and set *pValid* to false. (The value returned will be ignored.)

Hooking up the Method to the CalcItem.CustomCalculation Property

In Page_Load(), you attach your method to the **CustomCalculation** property. The syntax is shown here attaching to the method "MyCalcMethod". The CalcItem object was previously assigned an ID of "UseCalc1" so it can be retrieved into the variable vCalcItem.

[C#]

```
PeterBlum.DES.IBaseCalcItem vCalcItem = CalculationController1.FindByID("UseCalc1");
vCalcItem.CustomCalculation = new PeterBlum.DES.CalcEventHandler(MyCalcMethod);
```

[VB]

```
Dim vCalcItem As PeterBlum.DES.IBaseCalcItem = _
    CalculationController1.FindByID("UseCalc1")
vCalcItem.CustomCalculation = _
    New PeterBlum.DES.CalcEventHandler(AddressOf MyCalcMethod)
```

Example

This function returns the original value for numbers between 1 and 5. All other positive numbers returns 5. 0 and below return an error. If an error was passed in, an error is returned. The method MyCalcMethod should be assigned to **CustomCalculation**.

[C#]

```
public double MyCalcMethod(
    PeterBlum.DES.ICalculationController pSender,
    PeterBlum.DES.IBaseCalcItem pCalcItem, double pValue, ref bool pValid)
{
    if (!pValid)
        return 0.0; // pValid is already false
    else if (pValue < 1) // 0 and lower return an error
    {
        pValid = false;
        return 0.0;
    }
    else if (pValue > 5) // return 5
        return 5;
    else // values between 1-5 return pValue
        return pValue;
}
```

[VB]

```
Public Function MyCalcMethod( _
    ByVal pSender As PeterBlum.DES.ICalculationController, _
    ByVal pCalcItem As PeterBlum.DES.IBaseCalcItem, _
    ByVal pValue As Double, ByRef pValid As Boolean) As Double

    If Not pValid Then
        Return 0.0 ' pValid is already False
    ElseIf pValue < 1 Then ' 0 and lower return an error
        pValid = False
        Return 0.0
    ElseIf pValue > 5 Then ' return 5
        Return 5
    Else ' values between 1-5 return pValue
        Return pValue
    End If
End Function
```

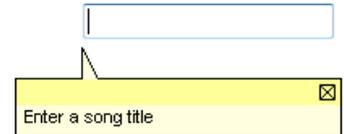
Interactive Hints

A hint on a field lets you offer guidance for data entry on the textbox. For example, if the textbox accepts integers between 1 and 5, say “Enter a number between 1 and 5.” It is displayed as the control gets focus and is hidden when focus is lost.

A tooltip is another kind of hint but it requires the mouse to point to the control to see it. Data entry fields need to show their hint as the user is editing, usually without the mouse pointing into the control. So the tooltip is ineffective during data entry. To solve this, developers sometimes provide labels with hints near a data entry field to assist the user. Due to the static nature of labels, they take up a lot of screen space. That often leads to reducing the information shown in the hint. Since hints are there to assist the user, you don’t want to be constrained by these space limitations. DES’s Interactive Hints feature solves this.

The Interactive Hints feature can display your hint in several ways:

- In a **PopupView**. A **PopupView** is similar to a **ToolTip**, created with **HTML** and **Javascript** to float near the control. It can be dragged and closed. It can be customized with style sheets, images, and settings using the **Global Settings Editor**. Shown here.
- In a **Label** on the page. As the data entry control gets focus, a hint is shown. As focus is lost, the hint is removed. Since only one control can have focus at a time, a single **Label** can show all of the hints. You can enhance the formatting by enclosing the **Label** in a **Panel**, which will be shown and hidden.
- In standard tooltips or **Enhanced ToolTips**. This lets the user point to the control at any time to read the hint.
- In the browser’s status bar as focus enters the control.



Click on any of these topics to jump to them:

- ◆ [Features](#)
- ◆ [Using Interactive Hints](#)
 - [Displaying Hints: The PeterBlum.DES.Web.WebControls.HintFormatter Class](#)
 - [Page-Level Hint Settings: The PeterBlum.DES.Globals.WebFormDirector.HintManager Property](#)
 - [Adding HintFormatters to the SharedHintFormatters Property](#)
 - [Defining PopupViews](#)
 - [Using PopupViews](#)
 - [Defining Hints shown on the Page](#)
 - [Using Hints shown on the Page](#)
 - [Customize the Text of the Hint: The Text Function](#)
 - [Javascript functions: Show and Hide the Hint On Demand](#)
- ◆ [Adding a Hint to any Control Programmatically: PeterBlum.DES.Globals.WebFormDirector.AddHintToControl Method](#)
- ◆ [Properties for the PeterBlum.DES.Web.WebControls.HintFormatter Class](#)
- ◆ [Properties on the PeterBlum.DES.Globals.WebFormDirector.HintManager Property](#)
- ◆ [Properties for the PeterBlum.DES.Web.WebControls.HintPopupView Class](#)

Features

The primary user interface of a hint is a Label control on the page or a PopupView, floating near the control.

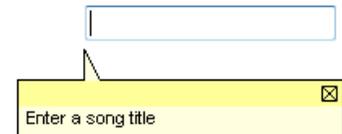
When using Labels

- The Label control uses screen real-estate but is optimized in three ways:
 - One Label can be used by several controls, such as those grouped together.
 - When hidden, the Label can optionally restore the screen space it uses much like validators do with their ErrorFormatter **Display** property is set to `Dynamic`.
 - Let the Label share the same location on the page as the control's validators. When there are validation error messages, DES can prevent the hint from showing.
- A Label can display formatted text, including HTML tags.
- The Label usually just contains the hint for the user. If you want it enclosed in a box or shown along with images and other controls, enclose the Label in the Panel. DES will show and hide the Panel while updating the Label with the correct hint text.
- Include a JavaScript function to customize what happens when DES shows and hides the Label or Panel. For example, you could use absolute positioning to move the Label nearer to the textbox with the hint.

When using PopupViews

A PopupView is similar to a ToolTip, created with HTML and Javascript to float near the control.

- Create as many PopupView definitions as needed in the **Global Settings Editor**.
- Style sheets control much of the appearance, including colors, borders, and fonts. There are predefined style sheets with yellow, red, blue and grey color schemes in the **DES\Appearance\Interactive Pages\Hints.css** style sheet file. The Global Settings Editor knows about these schemes so you only select a scheme instead of setting up numerous properties.
- The triangular extension shown at the top of the PopupView is called a Callout. It is a gif image file with transparency. The callout is optional. PopupViews can appear on any side of the textbox. When using callouts, there are images pointing left, up, down, and right.
- It has an optional titlebar. The title bar can have a label, including unique text for each control from the **HintHelp** property. It also has an optional close box.
- It can be dragged to expose other controls that it is covering. While dragging, its opacity decreases so the user can see other controls under it.
- Opacity changes in other ways. There is a default maximum opacity, so you can always see through it slightly if desired. If it is just shown or the mouse moves over it, it increases opacity to the maximum. After the mouse moves away, it reduces to a lessmore opaque state.
- It has a fixed width. Its height varies depending on the amount of text from the hint. There are predefined PopupViews for a variety of widths to choose the best width for the given text.
- It supports the **HintHelp** property from controls that use hints by showing a Help button (image or link). When clicked, there are a number of things you can do.
 - Switch the initial text to the text from the **HintHelp** property, offering the user expanded directions.
 - Run javascript that can use the text from **HintHelp** to customize it
 - Go to a URL with the text from the **HintHelp** containing part or all of the URL. This is great for opening a help page with a specific topic ID associated with the control.
- The hint text can contain HTML. The PopupView can define HTML that appears before and after the hint text, such as a for showing an image.



Other ways to display Hints

- Show the hint in standard tooltips or Enhanced ToolTips. This lets the user point to the control at any time to read the hint.
- Show the hint in the browser's status bar as focus enters the control.

Interactively Customizing the Hint Text

- Validation error messages are a very important part of data entry. They can automatically insert themselves into the hint text. They can either be shown first or completely replace the hint text. Provide a style sheet to distinguish the validator error messages from the other text.
- Include a JavaScript function to preprocess the hint's text. This allows you to customize the text based on the situation. For example, you define the token "{0}" in the hint text. Your JavaScript function replaces the token with the a value from the page.

Using Interactive Hints

There are several parts to the Interactive Hints feature:

- The `HintFormatter` class – Defines the display rules for a hint. The control that shows a hint needs this to know how to display its hint text.
- The controls that show a hint, such as textboxes. They need 4 key properties:
 - **Hint** – The hint text. By default, the **ToolTip** property can also supply the hint its text so long as the **Hint** property is blank. This rule can be overridden with the **HintManager.ToolTipAsHints** property.
 - **HintHelp** – If using a `PopupView`, this string’s usage is based on the **PopupView.HelpBehavior** property. It can be additional help text, the `PopupView`’s title, text to insert into a script, or text to insert into a URL for a hyperlink.
 - **LocalHintFormatter** and **SharedHintFormatterName** – Choose one of these to connect the `HintFormatter` to the control showing the hint. **SharedHintFormatterName** takes precedence over **LocalHintFormatter**. **LocalHintFormatter** is selected when **SharedHintFormatterName** is blank. **LocalHintFormatter** is also selected by **SharedHintFormatterName** is “{DEFAULT}” but the **HintManager.DefaultSharedHintFormatterName** property of the `PageManager` control and **PeterBlum.DES.Globals.WebFormDirector.HintManager** is blank.

Most of DES’s controls have these controls built in. For any other control, add the `NativeControlExtender`. It has these properties.

- When you want to display the hint as a popup, set up `PopupView` definitions. Set the **HintFormatter.DisplayMode** to `Popup` and **HintFormatter.PopupViewName** to the name of a defined `PopupView`.
- When showing the hint on the page, set up `Labels` and optionally `Panels` where the hint will appear. Set the **HintFormatter.DisplayMode** to either `Static` or `Dynamic` and **HintFormatter.HintControlID** to the ID of the `Label` or `Panel`.

Use demos here: <http://www.peterblum.com/DES/DemoHint.aspx>.

Click on any of these topics to jump to them:

- ◆ [Displaying Hints: The PeterBlum.DES.Web.WebControls.HintFormatter Class](#)
- ◆ [Page-Level Hint Settings: The PeterBlum.DES.Globals.WebFormDirector.HintManager Property](#)
 - [Showing Validation Errors In The Hints](#)
- ◆ [Adding HintFormatters to the SharedHintFormatters Property](#)
 - [When using a PopupView: AddSharedHintPopupView\(\)](#)
 - [When using a Label on the Page: AddSharedHintOnPage\(\)](#)
 - [Using Your Own HintFormatter definition: AddSharedHintFormatter\(\)](#)
- ◆ [Defining PopupViews](#)
 - [Creating your own Callouts](#)
 - [Adding your own Callouts to the PopupView Definition](#)
- ◆ [Using PopupViews](#)
- ◆ [Defining Hints shown on the Page](#)
 - [Using a Label](#)
 - [Using a Panel containing a Label](#)
 - [Customize How Hints Appear: The Formatter Function](#)
- ◆ [Using Hints shown on the Page](#)
- ◆ [Customize the Text of the Hint: The Text Function](#)
- ◆ [Javascript functions: Show and Hide the Hint On Demand](#)

Displaying Hints: The PeterBlum.DES.Web.WebControls.HintFormatter Class

The `PeterBlum.DES.Web.WebControls.HintFormatter` class describes how the hint text will be displayed. It provides its name, display mode - on the page or in a `PopupView`, if it's also in the tooltip and/or status bar, and more.

See [“Using PopupViews”](#), [“Using Hints shown on the Page”](#), and [“Properties for the PeterBlum.DES.Web.WebControls.HintFormatter Class”](#).

Page-Level Hint Settings: The PeterBlum.DES.Globals.WebFormDirector.HintManager Property

The **PeterBlum.DES.Globals.WebFormDirector.HintManager** property establishes a list of **HintFormatters** to be shared amongst the controls on the page in the **SharedHintFormatters** property. While you can create unique **HintFormatter** objects on each control, why not define them for each unique case, such as each unique label or **PopupView**? Additionally, these shared **HintFormatters** generate less JavaScript code embedded into your web form.

See “[Adding HintFormatters to the SharedHintFormatters Property](#)”, below.

To make it even easier:

- Establish one **HintFormatter** object as the default by setting its name to the **DefaultSharedHintFormatterName** property. All controls showing hints will use this default automatically, because their **SharedHintFormatterName** property defaults to the token “{DEFAULT}”.
- When using **PopupViews**, you can define the name of a **Hint PopupView Definition** (created in the **Global Settings Editor**) in the **SharedHintFormatterName** property of the control showing the hint. It will automatically create a shared **HintFormatter** using the same name.

Typically the **SharedHintFormatters** feature is used with **PopupViews** and when a **Label** is shared by several controls. It makes more sense to create **HintFormatters** on individual controls showing hints when they need their own **Labels** on the page. In that case, set up the **HintFormatter** in the controls’ **LocalHintFormatter** property and set that control’s **SharedHintFormatterName** property to “”.

Showing Validation Errors In The Hints

When using the DES Validation Framework, you can insert the error messages associated with a data entry control into its **Hint**. The error message is probably as important if not more important than the initial hint. If you feel it’s more important, you can have the hint text entirely replaced by the error messages. Otherwise, you can have them both appear with the error messages shown first. Use the **HintsShowErrors** property. It has these values:

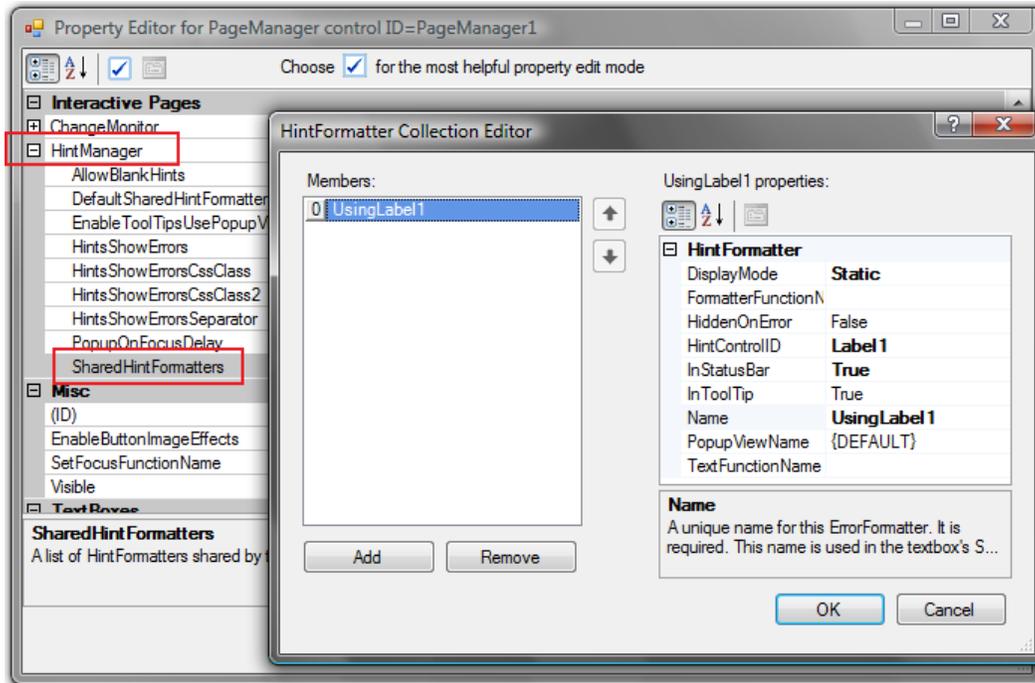
- **Hint** - Show the **Hint** text but not the validation errors.
- **OneErrorAndHint** - Show the first validation error and the **Hint** text.
- **AllErrorsAndHint** - Show all validation errors and the **Hint** text.
- **OneError** - Show the first validation error but not the **Hint** text.
- **AllErrors** - Show all validation errors but not the **Hint** text.

When error messages are displayed, you can use style sheets to change the appearance of the overall text (such as make a red background) using the **HintsShowErrorsCssClass** property and the font of just the error messages using the **HintsShowErrorsCssClass2** property. See “[Properties on the PeterBlum.DES.Globals.WebFormDirector.HintManager Property](#)”.

Adding HintFormatters to the SharedHintFormatters Property

Visual Studio and Visual Web Developer Design Mode Users

The easiest way to work with SharedHintFormatters is to add a PageManager control. In design mode, open the HintManager property to expose the SharedHintFormatters property. Then open its editor.



See [“Properties for the PeterBlum.DES.Web.WebControls.HintFormatter Class”](#) for details on its properties.

Text Entry Users

Add the PageManager control. Add <des:HintFormatter> tags into the <des:PageManager> control like this:

```
<des:PageManager ID="PageManager1" runat="server">
  <HintManager>
    <SharedHintFormatters>
      <des:HintFormatter DisplayMode="Static"
        HintControlID="Label1" InStatusBar="True" Name="UsingLabel1" />
    </SharedHintFormatters>
  </HintManager>
</des:PageManager>
```

See [“Properties for the PeterBlum.DES.Web.WebControls.HintFormatter Class”](#) for details on the properties.

Programmatically working with the HintManager

When working programmatically, you don't need the PageManager control's HintManager property. Instead, you use **PeterBlum.DES.Globals.WebFormDirector.HintManager** property with these methods:

AddSharedHintPopupView() – See [“When using a PopupView: AddSharedHintPopupView\(\)”](#).

AddSharedHintOnPage() – See [“When using a Label on the Page: AddSharedHintOnPage\(\)”](#)

AddSharedHintFormatter() – See [“Using Your Own HintFormatter definition: AddSharedHintFormatter\(\)”](#)

When using a PopupView: AddSharedHintPopupView()

Call `PeterBlum.DES.Globals.WebFormDirector.HintManager.AddSharedHintPopupView()` to add `HintFormatter` object that uses a `PopupView`.

[C#]

```
PeterBlum.DES.Web.WebControls.HintFormatter AddSharedHintPopupView(  
    string pName, bool pDefault,  
    string pPopupViewName);
```

```
PeterBlum.DES.Web.WebControls.HintFormatter AddSharedHintPopupView(  
    string pName, bool pDefault,  
    string pPopupViewName, bool pInStatusBar, bool pInTooltip);
```

[VB]

```
Function AddSharedHintPopupView(ByVal pName As String, _  
    ByVal pDefault As Boolean, ByVal pPopupViewName As String) _  
    As PeterBlum.DES.Web.WebControls.HintFormatter  
  
Function AddSharedHintPopupView(ByVal pName As String, _  
    ByVal pDefault As Boolean, ByVal pPopupViewName As String, _  
    ByVal pInStatusBar As Boolean, ByVal pInTooltip As Boolean) _  
    As PeterBlum.DES.Web.WebControls.HintFormatter
```

Parameters

pName

A unique name for this `HintFormatter`. If unassigned, it will get the *pPopupViewName*.

pDefault

When `true`, use *pName* as the default for any **SharedHintFormatterName** property that is "{DEFAULT}" on the controls using hints.

It sets **PeterBlum.DES.Globals.WebFormDirector.HintManager.DefaultSharedHintFormatterName**. It has no affect when **HintManager.DefaultSharedHintFormatterName** is already assigned.

pPopupViewName

The name of a `PopupView` definition from the `PopupView` Hint definitions defined in the **Global Settings Editor**.

pInStatusBar

When `true`, show the hint in the status bar of the browser. When not supplied, **HintFormatter.InStatusBar** is set to `false`.

pInTooltip

When `true`, the control's tooltip is assigned the hint when there is nothing already assigned to the tooltip. The tooltip does not support the merger of validation error messages. When not supplied, **HintFormatter.InToolTip** is set to `true`.

Return value

The `HintFormatter` object that was defined.

Example

Adds a PopupView named “YellowToolTip”. It uses that same name for the HintFormatter.Name by passing "" for the pName parameter. It uses the tooltip and status bar features.

[C#]

```
PeterBlum.DES.Web.WebControls.HintFormatter vHF =  
    PeterBlum.DES.Globals.WebFormDirector.HintManager.AddSharedHintPopupView(  
        "", false, "YellowToolTip", true, true);
```

[VB]

```
Dim vHF As PeterBlum.DES.Web.WebControls.HintFormatter = _  
    PeterBlum.DES.Globals.WebFormDirector.HintManager.AddSharedHintPopupView( _  
        "", False, "YellowToolTip", True, True)
```

When using a Label on the Page: AddSharedHintOnPage()

Call `PeterBlum.DES.Globals.WebFormDirector.HintManager.AddSharedHintOnPage()` to add `HintFormatter` object that uses a control on the page, such as a `Label` or `Panel`.

[C#]

```
PeterBlum.DES.Web.WebControls.HintFormatter AddSharedHintOnPage(
    string pName, bool pDefault,
    Control pHintControl,
    PeterBlum.DES.HintDisplayMode pDisplayMode,
    bool pHiddenOnError);
```

```
PeterBlum.DES.Web.WebControls.HintFormatter AddSharedHintOnPage(
    string pName, bool pDefault,
    Control pHintControl,
    PeterBlum.DES.HintDisplayMode pDisplayMode,
    bool pInStatusBar, bool pInTooltip, bool pHiddenOnError);
```

[VB]

```
Function AddSharedHintOnPage(ByVal pName As String, _
    ByVal pDefault As Boolean,
    ByVal pHintControl As Control, _
    ByVal pDisplayMode As PeterBlum.DES.HintDisplayMode _
    ByVal pHiddenOnError As Boolean) _
    As PeterBlum.DES.Web.WebControls.HintFormatter
```

```
Function AddSharedHintOnPage(ByVal pName As String, _
    ByVal pDefault As Boolean, _
    ByVal pHintControl As Control, _
    ByVal pDisplayMode As PeterBlum.DES.HintDisplayMode _
    ByVal pInStatusBar As Boolean, ByVal pInTooltip As Boolean, _
    ByVal pHiddenOnError As Boolean) _
    As PeterBlum.DES.Web.WebControls.HintFormatter
```

Parameters*pName*

A unique name for this `HintFormatter`. Required.

pDefault

When `true`, use *pName* as the default for any `SharedHintFormatterName` property that is "{DEFAULT}" on the controls using hints.

It sets `PeterBlum.DES.Globals.WebFormDirector.HintManager.DefaultSharedHintFormatterName`. It has no affect when `HintManager.DefaultSharedHintFormatterName` is already assigned.

pHintControl

This points to a control where the Hint will appear.

Use a `Panel`, `Label`, any control that can have its innerHTML replaced, or any control containing a `Label` where the Hint will appear.

This control has its visibility changed as focus moves in and out of the control with the hint.

The Hint text will be assigned as follows:

If this is a containing control with the `Label` that shows the hint, make sure that `Label` has the ID = `pHintControl.ID+"_Text"`. Otherwise, *pHintControl* itself will show the hint in its innerHTML.

pDisplayMode

Pass only `HintDisplayMode.Static` or `HintDisplayMode.Dynamic`. When `Static`, the `Label` preserves its space on the page when hidden. When `Dynamic`, it uses no space on the page when hidden.

pInStatusBar

When `true`, show the hint in the status bar of the browser. When not supplied, **HintFormatter.InStatusBar** is set to `false`.

pInTooltip

When `true`, the control's tooltip is assigned the hint when there is nothing already assigned to the tooltip. The tooltip does not support the merger of validation error messages. When not supplied, **HintFormatter.InToolTip** is set to `true`.

pHiddenOnError

When `true`, do not show the hint when any validator attached to this control reports an error. This allows you to place the Label in the same location as the validator.

Return value

The HintFormatter object that was defined.

Example

Uses the Label "HintLabel". That control's ID is also used as the name of this HintFormatter. It uses the tooltip and status bar features.

[C#]

```
PeterBlum.DES.Web.WebControls.HintFormatter vHF =  
    PeterBlum.DES.Globals.WebFormDirector.HintManager.AddSharedHintOnPage(  
        HintLabel1.ID, HintLabel1, PeterBlum.DES.HintDisplayMode.Dynamic,  
        true, true, false);
```

[VB]

```
Dim vHF As PeterBlum.DES.Web.WebControls.HintFormatter = _  
    PeterBlum.DES.Globals.WebFormDirector.HintManager.AddSharedHintPopupView( _  
        HintLabel1.ID, HintLabel1, PeterBlum.DES.HintDisplayMode.Dynamic, _  
        True, True, False)
```

Using Your Own HintFormatter definition: AddSharedHintFormatter()

Call `PeterBlum.DES.Globals.WebFormDirector.HintManager.AddSharedHintFormatter()` to add `HintFormatter` object that uses the exact properties you set up. This is often used when you are providing a custom Formatting Function (**`HintFormatter.FormattingFunctionName`**) or Text Function (**`HintFormatter.TextFunctionName`**).

[C#]

```
void AddSharedHintFormatter(PeterBlum.DES.Web.WebControls.HintFormatter
pHintFormatter,
    bool pDefault);
```

[VB]

```
Sub AddSharedHintFormatter(ByVal pHintFormatter As
PeterBlum.DES.Web.WebControls.HintFormatter, _
    ByVal pDefault As Boolean)
```

Parameters

pHintFormatter

The `HintFormatter` object, with its properties fully assigned. The `HintFormatter.Name` must be assigned. See [“Properties for the PeterBlum.DES.Web.WebControls.HintFormatter Class”](#).

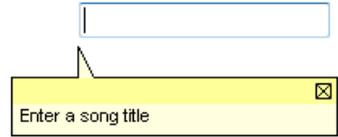
pDefault

When `true`, use *pName* as the default for any **`SharedHintFormatterName`** property that is "{DEFAULT}" on the controls using hints.

It sets **`PeterBlum.DES.Globals.WebFormDirector.HintManager.DefaultSharedHintFormatterName`**. It has no affect when **`HintManager.DefaultSharedHintFormatterName`** is already assigned.

Defining PopupViews

A PopupView is similar to a ToolTip, created with HTML and Javascript to float near the control. For an overview of its features, see [“When using PopupViews”](#).



Click on any of these topics to jump to them:

- ◆ [View an existing definition](#)
- ◆ [Edit a definition](#)
- ◆ [Add a definition](#)
- ◆ [Rename a definition](#)
- ◆ [Delete a definition](#)
- ◆ [Creating your own Callouts](#)
- ◆ [Adding your own Callouts to the PopupView Definition](#)

PopupView definitions are created within the **Global Settings Editor** and stored in the **custom.des.config** file. Within the **Global Settings Editor**, you can add, edit, rename, and delete definitions. In addition, you can choose one of your PopupView definitions to be the default used when the token “{DEFAULT}” appears in a **HintFormatter.PopupViewName** property by setting its name in the **DefaultHintPopupViewName** property.



Here is how to define PopupViews.

1. Open the **Global Settings Editor**.

It is available from the Windows Start menu, the Context menu and SmartTag on the PageManager control, and in the **[DES Product Folder]**.

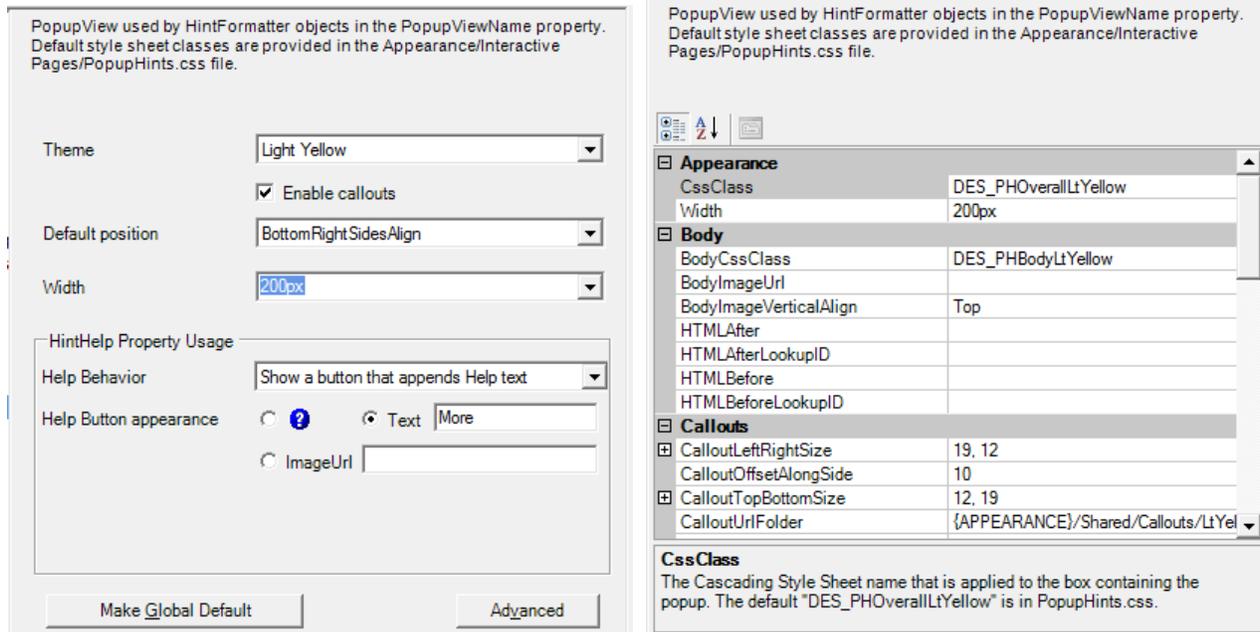
2. Confirm that the **custom.des.config** file for your web application is loaded. If it is not, click the **Open**  button and select it.



3. Select the **PopupView definitions used by Hint Formatters** topic in the list on the left.
4. [View](#), [add](#), [edit](#), [rename](#) or [delete](#) a PopupView definition, using the topics below. See also [“Properties for the PeterBlum.DES.Web.WebControls.HintPopupView Class”](#).
5. Save the changes using the **Save**  button.
6. If you have changed the name of an existing PopupView, review your web forms in case the old name is in use. Correct those that need it.

View an existing definition

To view an existing definition, click on its name in the items below the **PopupView definitions used by HintFormatters** topic heading. There are two views. The initial display shows the most common properties and combines a number of properties for style sheets and callout images into a single **Theme**. Click the **Advanced** button to see a Properties Editor with all available properties.



Fields on the Initial View

For details, see “[Properties for the PeterBlum.DES.Web.WebControls.HintPopupView Class](#)”.

- **Theme** – DES predefines style sheets and images that correspond to these colors: Light Red, Light Blue, Alice Blue, Light Yellow, and Light Gray. When you pick one of these, the following PopupView class properties are changed: **CssClass**, **HeaderCssClass**, **BodyCssClass**, **FooterCssClass**, **CloseButtonCssClass**, **HelpButtonCssClass**, and **CalloutUrlFolder**. If it says Custom, then you have modified at least one of these properties in the Advanced view.
- **Enable Callouts** – Sets the **PopupView.EnableCallouts** property. Callouts are the triangles projecting out of the PopupView to point it to the control with the hint. They are gif images stored in the folder defined by **PopupView.CalloutUrlFolder**. *Use the Advanced View to edit the CalloutUrlFolder property.*
- **Default position** – Sets the **PopupView.DefaultPosition** property. Determines the default position when the popup view appears. If there is not enough screen space to appear in the default position, DES will reposition it.
- **Width** – The width of the definition in pixels. Each definition has a fixed width (although its height can change). As a result, you usually define several definitions with the same features, but varying the width.
- **Help Behavior** – Sets the **PopupView.HelpBehavior** property. Determines how the **HintHelp** property on each control behaves. In most, cases it adds the Help button and determines how it behaves. Here are its values:
 - Not used – Do not use **HintHelp**. Do not show a Help Button.
 - Show a button that appends the help text – Use the Help Button. When clicked, redraw with the **HintHelp** text appended to the current text. The value of **PopupView.AppendHelpSeparator** is inserted between the original hint and the text of **HintHelp**.
 - Show a button that replaces the help text – Use the Help Button. When clicked, redraw with the **HintHelp** text replacing the current text.
 - Show the help text in the titlebar – The **HintHelp** text appears in the header as the title. It is used instead of the **PopupView.HeaderText** property value. There is no Help Button.

- Show a button that hyperlinks – Use the Help Button that acts as a hyperlink. Define the URL in the **URL** field. The **HintHelp** text will appear in the “{0}” token.
- Show a button that hyperlinks using another window – Use the Help Button that acts as a hyperlink which opens in a new window. Define the URL in the URL field. The **HintHelp** text will appear in the “{0}” token.
- Show a button that runs a script – Runs the script supplied in the **Script** field. The **HintHelp** text will replace the token “{0}” in that script.
- **Help Button appearance** – When Help Behavior specifies a Help button, you can use an image or text for that button. DES predefines the  image and makes it available as the first radio button. Otherwise, specify the text in the Text field or the image’s URL in the ImageUrl field.



Help Button appearance  Text
 ImageUrl

- **Make Global Default** – When clicked, this PopupView definition will become the default for all HintFormatters whose **PopupViewName** property is “{DEFAULT}”. It updates the setting **DefaultHintPopupViewName** in the topic “HintManager Defaults” of the **Global Settings Editor**.
- **Advanced** – Switch to the Advanced view, where you have access to every PopupView property using a Properties Editor.

Edit a definition

To edit a definition, click on its name in the items below the **PopupView definitions used by Hint Formatters** topic heading. Change the properties on either the default or Advanced view. You do not need to click anything to save your edits. (You also cannot undo your edits without reloading the **custom.des.config** file.)

See “[Fields on the Initial View](#)”, above, and “[Properties for the PeterBlum.DES.Web.WebControls.HintPopupView Class](#)”.

Add a definition

To add a definition, click the **Add** button in the lower right corner of the window or right click on the **PopupView definitions used by Hint Formatters** topic and chose **Add**.

Use “[Fields on the Initial View](#)”, above, to fill in the window. When done, click **Save**.

It will automatically create a name for you based on the **Theme** and **Width**. See below to rename it.

Rename a definition

Click on the name of the PopupView definition in the list, or click the **Rename** button when viewing the definition.

Note: The Global Settings Editor will automatically rename definitions if it created the original name and you change either the Theme or Width field.

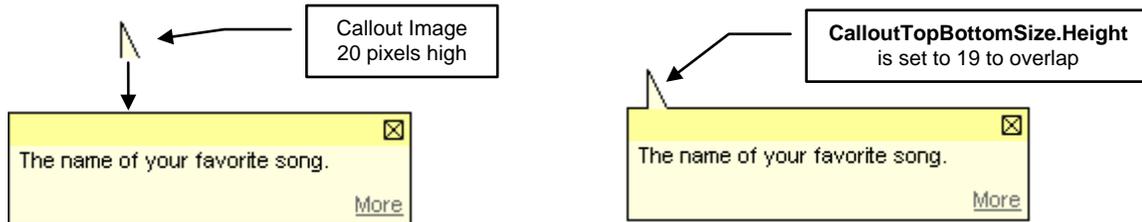
Delete a definition

Click on the name of the PopupView definition in the list and click the **Delete** button at the bottom of the window.

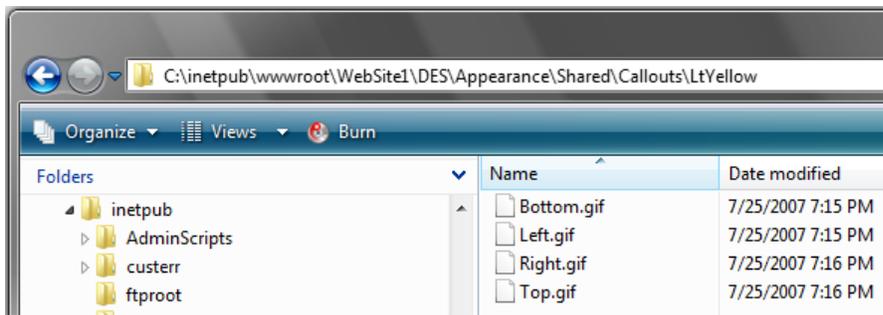
Creating your own Callouts

The Callout is an image file. In fact, there are 4 of these image files, one for each direction: left, right, top, and bottom. They have these characteristics:

- Use a gif file. Make all pixels “outside” of your image transparent.
- To make their borders merge with the box of the popup view, do not use a border where it intersects with the box. You will also make the image slightly overlap the box with the **CalloutLeftRightSize** and **CalloutTopBottomSize** properties.



- The Callout image is inset along the box as determined by the **CalloutOffsetAlongSide** property. In the above image, **CalloutOffsetAlongSide** is set to 10 pixels.
- All four image files have a specific name: Left.gif, Right.gif, Top.gif, Bottom.gif. They all go into single folder whose Url is specified in the **CalloutUrlFolder**.



Adding your own Callouts to the PopupView Definition

1. Create 4 callout image files, one for each direction.
 - They must be named Left.gif, Right.gif, Top.gif, and Bottom.gif. Each *points* in the direction taken from the filename.
 - They all go in a single folder. A suggesting containing folder is **[Web application root]/DES/Appearance/Shared/Callouts**. However, any folder that is accessible to your web application through a URL is acceptable.
 - The Top and Bottom images should have identical dimensions to each other.
 - The Left and Right images should have identical dimensions to each other.
2. In the PopupView definition, assign these properties:
 - **EnableCallout** = true
 - **CalloutUrlFolder** = the URL to the folder. If using the suggested path: “{APPEARANCE}/Shared/Callouts/*your foldername*”.

The “{APPEARANCE}” token will be replaced by the default path to the **Appearance** folder, which you defined as you set up the web site.
 - **CalloutLeftRightSize.Width** = The width of the Left and Right images. To overlap the PopupView box, subtract 1.
 - **CalloutLeftRightSize.Height** = The height of the Left and Right images.
 - **CalloutTopBottomSize.Width** = The width of the Top and Bottom images.
 - **CalloutTopBottomSize.Height** = The height of the Top and Bottom images. To overlap the PopupView box, subtract 1.
 - **CalloutOffsetAlongSide** = How many pixels to offset the image along the side of the PopupView box. It defaults to 10 pixels.

See “[Callout Properties](#)” for details.

Using PopupViews

For each control that needs a hint, it must have these four properties: **Hint**, **HintHelp**, **SharedHintFormatterName**, and **LocalHintFormatter**. Most DES controls have them. For any other control, add a NativeControlExtender control. It has these properties. (See the “General Features Guide” for this control.)

Here is how to use these properties:



These steps ask you to jump around the document using clicks on links. Adobe Reader offers a **Previous View** command to return to the link. Look for this in the Adobe Reader (shown v6.0)

1. Set the text of the hint in the **Hint** property. It can contain HTML tags if desired. If you are using the same text in the **ToolTip** property, you do not need to assign anything to **Hint**. It uses the **ToolTip** property when **Hint** is "" unless you set the **HintManager.ToolTipAsHints** property to `False`.

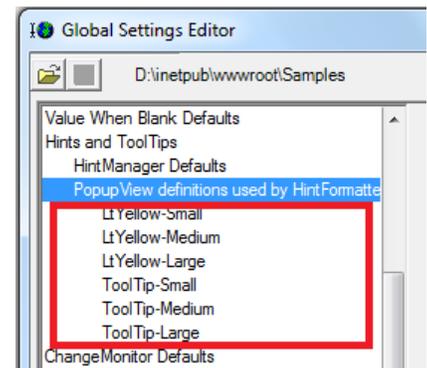
```
<des:control id="id" runat="server" Hint="your hint" />
```

2. Pick a PopupView definition from those defined with the **Global Settings Editor**. See “Defining PopupViews”.

Here are the predefined values:

LtYellow-Small, LtYellow-Medium, LtYellow-Large, ToolTip-Small, ToolTip-Medium, and ToolTip-Large.

All of these are light yellow. Their widths vary from 200px to 600px. The ToolTip definitions do not have the callout feature enabled.



3. Define a HintFormatter that uses the PopupView using one of these three approaches:

- Set the web control’s **SharedHintFormatterName** to the name of the Popup View. DES automatically creates a HintFormatter for you with `HintFormatter.DisplayMode` and `HintFormatter.PopupViewName` correctly set. It will show the hint in the tooltip and status bar too. If you want to avoid it in the tooltip or status bar, use one of these next two techniques.

```
<des:control id="id" runat="server" Hint="your hint"
  SharedHintFormatterName="PopupView name" />
```

- Define a HintFormatter object that will be shared amongst several controls on the page by adding it to the **HintManager.SharedHintFormatters** property. This can be done in the PageManager control or programmatically. See “Adding HintFormatters to the SharedHintFormatters Property” and “Properties for the PeterBlum.DES.Web.WebControls.HintFormatter Class”.

```
<des:PageManager ID="PageManager1" runat="server">
  <HintManager>
    <SharedHintFormatters>
      <des:HintFormatter Name="YellowMedium"
        PopupViewName="LtYellow-Medium"
        InToolTip="False" InStatusBar="False" />
    </SharedHintFormatters>
  </HintManager>
</des:PageManager>

<des:control id="id" runat="server" Hint="your hint"
  SharedHintFormatterName="YellowMedium" />
```

- Otherwise, use the **LocalHintFormatter** property on the web control. It is a HintFormatter (see “Properties for the PeterBlum.DES.Web.WebControls.HintFormatter Class”). It defaults to **DisplayMode**=Popup, **InToolTip**=true, and **InStatusBar**=true. Take these actions
 - Set the **SharedHintFormatterName** property to "". This enables the LocalHintFormatter.
 - Set the **HintFormatter.PopupViewName** to the name of the PopupView.
 - Determine if a mouse over and/or focus action should pop it up and set that in **HintFormatter.PopupAction**.

- o Consider if these properties apply: **InToolTip**, **InStatus** and **TextFunctionName**. (All others are used when **Display** mode is not set to **Popup**.)

```
<des:control id="id" runat="server" Hint="your hint"
  SharedHintFormatterName=""
  LocalHintFormatter-PopupViewName="PopupViewName"
  LocalHintFormatter-InToolTip="false" />
```

4. If you are using the **PopupView.HelpBehavior** property, set your web control's **HintHelp** property to the appropriate text, whether it is a more detailed description, a title, a URL, or a script. *If left blank and HelpBehavior is ButtonAppends or ButtonReplaces, the Help button is not shown.*

```
<des:control id="id" runat="server" Hint="your hint" HintHelp="help text" />
```

5. If you also want to show validation error messages (from the DES Validation Framework) in the **PopupView**, use the **HintManager.HintsShowErrors** property.

```
<des:PageManager ID="PageManager1" runat="server">
  <HintManager HintsShowErrors="OneError" />
</des:PageManager>
```

Defining Hints shown on the Page

The **hint control** is the HTML that displays the hint. It is usually a Label control or a Panel containing a Label.

DES performs two actions on your Panel and Label controls.

- It shows and hides the control assigned to the **HintFormatter.HintControlID** property.
- It updates the text of the Label. (Specifically, it updates the innerHTML so it supports HTML.)

You can customize its behavior by creating a client-side function that does anything you want. It can replace DES's ability to change visibility and the text or it can let DES continue to do these actions while your function does other things. See "[Customize How Hints Appear: The Formatter Function](#)".

Click on any of these topics to jump to them:

- ◆ [Using a Label](#)
- ◆ [Using a Panel containing a Label](#)
- ◆ [Customize How Hints Appear: The Formatter Function](#)

Using a Label

Add a Label to the page where you want the hint's text to appear. You can have several of them, one for each control or for a group of controls that share the hint's location on the page.

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

The Label can be substituted with any tag that allows setting its "innerHTML", including ``, `<p runat="server">`, `<td runat="server">`, and `<div runat="server">`. The innerHTML will always be replaced by the hint text.

Using a Panel containing a Label

You can put the Label inside a Panel if you want it surrounded by some formatting, like a border and title. When you do so, assign the ID of the Label to the ID of the Panel plus the text "_Text".

When used, the Panel will be shown and hidden (along with the Label it contains). When not used, the Label will be shown and hidden.

Here is an example of the Panel with a Label:

```
<asp:Panel id="HintPanel" runat="server" width="20px">
  <asp:Label id="HintPanel_Text" runat="server"></asp:Label>
</asp:Panel>
```

It is a good idea to set the Panel's width. Here is an example with formatting that establishes a yellow background, border, and centers the text.

```
<asp:Panel id="HintPanel" runat="server" width="200px"
  style="BORDER-RIGHT:gray thin outset; BORDER-TOP:gray thin outset;
  BORDER-LEFT:gray thin outset; BORDER-BOTTOM:gray thin outset;
  BACKGROUND-COLOR:lightyellow; TEXT-ALIGN:center">
  <asp:Label id="HintPanel_Text" runat="server"></asp:Label>
</asp:Panel>
```

The Panel can be substituted with almost any control that can contain child control tags, such as a Table control, TableCell control, <table runat="server">, <td runat="server">, <div runat="server">, , and <p runat="server"> tag.

The Label can be substituted with any tag that allows setting its "innerHTML", including , <p runat="server">, <td runat="server">, and <div runat="server">. The innerHTML will always be replaced by the hint text.

Customize How Hints Appear: The Formatter Function

If you want additional control over the appearance when the hint is shown or hidden, you can provide your own JavaScript function. You can use it to assign the hint or change appearance of the Hint control or any element on the page. Your function returns a flag indicating if DES should still change visibility and the hint text or not.

Your function takes these parameters in the order show:

- *pFld* (object) – The DHTML element to which the hint is attached. The `pFld.id` attribute is often used to determine which textbox is passed to your function. The `pFld.value` contains the text currently in the textbox. For other attributes, see “[DHTML Reference for <input type='text'>](#)”.
- *pSH* (boolean) - When `true`, show the hint. When `false`, hide the hint. If you plan to change visibility, here are guidelines:
 - Show in Static mode: `pCFld.style.visibility = "inherit";`
 - Show in Dynamic mode: `pCFld.style.visibility = "inherit"; pCFld.style.display = "";`
 - Hide in Static mode: `pCFld.style.visibility = "hidden";`
 - Hide in Dynamic mode: `pCFld.style.visibility = "hidden"; pCFld.style.display = "none";`
- *pHint* (string) - Text of the hint. It may contain HTML tags.
- *pCFld* (element) – The element that is the hint control. This is what you will be modifying. If it is a Panel, your Label is available by using this function:

```
vFld = DES_GetById(pCFld.id + "_Text");
```

It is `null` when you have nothing assigned to the **HintFormatter.HintControlID** property. In that case, your function must internally know the ID of an element.

Your function must return `true` if it has changed visibility and the text; return `false` if it needs DES to change visibility and the text.

Your function can use `DES_SetInnerHTML(ID, pHint)` to change the innerHTML with the hint.

See “[Adding Your JavaScript to the Page](#)” for instructions on adding JavaScript to the page.

Example

Makes a popup hint that appears below the textbox by setting the panel with absolute positioning and establishing its top and left positions. It lets DES handle visibility and assigning the text. **HintFormatter.FormatterFunctionName** is assigned to “MyCstmHint”.

Note: This popup hint technique works well only when the data entry control is not inside a “container” tag like a <div> or <table>. Once in those, better positioning calculations are needed. The PopUpView feature will handle this automatically.

```
<script type="text/javascript" language="javascript">
function MyCstmHint(pFld, pSH, pHint, pCFld)
{
  if (pSH)
  {
    pCFld.style.position = "absolute";
    pCFld.style.posLeft = pFld.offsetLeft - 5;
    pCFld.style.posTop = pFld.offsetTop + pFld.clientHeight + 5;
  }
  return false; // let the normal processing change the hint and visibility
}
</script>
```

Using Hints shown on the Page

For each control that needs a hint, it must have these four properties: **Hint**, **HintHelp**, **SharedHintFormatterName**, and **LocalHintFormatter**. Most DES controls have them. For any other control, add a NativeControlExtender control. It has these properties. (See the “General Features Guide” for this control.)

Here is how to use these properties:



These steps ask you to jump around the document using clicks on links. Adobe Reader offers a **Previous View** command to return to the link. Look for this in the Adobe Reader (shown v6.0)

1. Set the text of the hint in the **Hint** property. It can contain HTML tags if desired. If you are using the same text in the **ToolTip** property, you do not need to assign anything to **Hint**. It uses the **ToolTip** property when **Hint** is "" unless you set the [HintManager.ToolTipAsHints](#) property to `False`.
2. If you also want to show validation error messages (from the DES Validation Framework) in the `PopupView`, use the [HintManager.HintsShowErrors](#) property.
3. Determine what kind of appearance that you want for your hint. It can be simply a `Label` or a `Panel` whose formatting encloses a `Label` and is fully hidden when there is no hint text to show. See “[Defining Hints shown on the Page](#)”.
4. Determine the locations for hints. You can have one on the page, one for each group of controls, or even one for each control. When you put one next to a control, it can be located where `Validators` appear as there is a feature to prevent the hint from showing when a `Validator` is shown.
5. Add the controls for hints to the page. Remember that they will be hidden until focus is set to them.

If you are using a `Panel` that contains a `Label`, make sure the `Label`’s ID is `Panel.ID + "_Text"`.

6. Define a `HintFormatter` using one of these three approaches:
 - If several controls will share a `HintFormatter`, add a `HintFormatter` object to the [HintManager.SharedHintFormatters](#) property. This can be done in the `PageManager` control or programmatically.
 - Set the **HintFormatter.HintControlID** to the `Label` or `Panel` control where the hint is shown. When using a `Panel` or other containing control, the ID must be to the `Panel`, not the `Label`.
 - Set the **HintFormatter.DisplayMode** to `Static`, if you want to preserve the space used by the `Panel` or `Label` when the hint is not shown. Use `Dynamic` to avoid using that space.
 - If you positioned the `Label` or `Panel` in the same space as a `validator`, set **HintFormatter.HiddenOnError** to `true`.
 - Consider if these properties apply: **InToolTip**, **InStatus**, **FormatterFunctionName**, and **TextFunctionName**. See “[Properties for the PeterBlum.DES.Web.WebControls.HintFormatter Class](#)”.
 - Otherwise, use the **LocalHintFormatter** property on the control:
 - Set the **HintFormatter.HintControlID** to the `Label` or `Panel` control where the hint is shown. When using a `Panel` or other containing control, the ID must be to the `Panel`, not the `Label`.
 - Set the **HintFormatter.DisplayMode** to `Static`, if you want to preserve the space used by the `Panel` or `Label` when the hint is not shown. Use `Dynamic` to avoid using that space.
 - If you positioned the `Label` or `Panel` in the same space as a `validator`, set **HintFormatter.HiddenOnError** to `true`.
 - Consider if these properties apply: **InToolTip**, **InStatus**, **FormatterFunctionName**, and **TextFunctionName**. See “[Properties for the PeterBlum.DES.Web.WebControls.HintFormatter Class](#)”.

Customize the Text of the Hint: The Text Function

You can modify or replace the text of the hint prior to it being displayed. Typically this is used when you want to retrieve some value from the page and insert it into the text. When replacing a part of the text, a good technique is to use a token, like "{0}". Your function can use `DES_RERpl()` function to replace tokens.

You can also disable hint from showing by returning `null` from your function.

If you want to fully create the hint text on the client side, you need to set `HintManager.AllowBlankHints` to `true` so a control whose `Hint` property is "" still gets attached to the hint system.

Your function takes these parameters in the order shown here:

- *pFld* (object) – The DHTML element to which the hint is attached. The `pFld.id` attribute is often used to determine which control is passed to your function. It is the **ClientID** value of that Control. If this is an `<input>`, `<textarea>` or `<select>` element, the `pFld.value` contains its current value.
- *pHint* (string) - Text of the hint from the **Hint** or **HintLookupID** property on the control with the hint. It may contain HTML tags. It may be blank.
- *pErr* (Boolean) – When `true`, there is a validation error on this control. This helps determine how to prepare the hint.

Your function must return one of these values:

- The string used for the hint
- "" if the hint text is not shown but the validation error messages are shown based `HintManager.HintsShowErrors`.
- `null` to prevent showing any hint.

See “[Adding Your JavaScript to the Page](#)” for instructions on adding JavaScript to the page.

Example

Assumes the hint text is “{0} characters”. Assumes that control is a textbox.

Replaces the token “{0}” with the text length of the control with the hint. If the textbox is blank, it uses an alternative string. If text length is 1, it returns “1 character”. `HintFormatter.TextFunctionName` is assigned to “MyHintText”.

```
<script type="text/javascript" >
function MyHintText(pFld, pHint, pErr)
{
    if (pFld.value != "")
    {
        if (pFld.value.length > 1)
            return DES_RERpl(pHint, "{0}", pFld.value.length.toString());
        else
            return "1 character";
    }
    else
        return "Please enter text.";
}
</script>
```

Javascript functions: Show and Hide the Hint On Demand

Sometimes a third party control has an alternative way to handle the “onfocus” and “onblur” events. (onfocus is called when focus is set to the control; onblur is called when focus leaves the control.) You can call the `DES_ShowHint()` and `DES_HideHint()` JavaScript functions from within the alternative event handlers of the control.

function `DES_ShowHint(pID)`

Displays the hint associated with the ID of the control passed. Call it when focus is established, usually in the onfocus event handler. If that ID is unknown, nothing happens.

Parameters

pID

The **ClientID** property value from the server side control. It is the value written into the `id=` attribute of the HTML element. See “[Embedding the ClientID into your Script](#)”.

Example

```
DES_ShowHint('TextBox1');
```

function `DES_HideHint(pID)`

Hides the hint associated with the ID of the control passed. Call it when focus is lost, usually in the onblur event handler. If that ID is unknown, nothing happens. You can also pass null and have it detect if a hint is open on any control and close it.

Parameters

pID

The **ClientID** property value from the server side control. It is the value written into the `id=` attribute of the HTML element. See “[Embedding the ClientID into your Script](#)”.

Also pass null to have it close any open hint.

Example

```
DES_HideHint(null); // closes whichever is open  
DES_HideHint('TextBox1');
```

Providing an Initialization Function

The HintFormatter object provides the **InitFunctionName** property for you to establish a function that is called as the page is setup. Your function will hookup the control's own onfocus and onblur handlers to call `DES_ShowHint()` and `DES_HideHint()`. The function also returns a boolean value to tell DES whether it should also hookup these functions to the standard DHTML onfocus and onblur events.

Your function takes one parameter:

- *pHO* (object) – The “Hint Object”. Its properties associated the hint text with the control and formatters. You will use one or more of its properties as shown below.

Your function must return true to allow DES to also attach to the standard onfocus/onblur events or false to avoid attaching to the standard events.

See “[Adding Your JavaScript to the Page](#)” for instructions on adding JavaScript to the page.

Key Properties on the Hint Object (pHO parameter)

- CID (string) – The ID of the control on the page. Call `DES_GetById(pHO.CID)` to get a reference to its DHTML object. Pass this value as the parameter of `DES_ShowHint` and `DES_HideHint`.
- Fmt (object) – Client-side representation of the HintFormatter object. Its ID property is the ID of the Panel or Label control on the page where the hint is displayed.
- H (string) – the Hint text. This can also be customized by the TextFunction.
- Hlp (string) - the Hint Help text.

Example

In this fictitious control, Dial1, it expects its property Events.onfocus to contain a string of javascript for the onfocus event. It expects Events.onblur to contain a string of javascript for the onblur event. **HintFormatter.InitFunctionName** is assigned to “InitDial1”.

```
<script type="text/javascript" language="javascript">
function InitDial1(pHO)
{
    var vDial1 = <% =Dial1.ClientID %>;
    vDial1.Events.onfocus = "DES_ShowHint(' " + pHO.CID + " ');";
    vDial1.Events.onblur = "DES_HideHint(' " + pHO.CID + " ');";
    return false; // DES does not need to setup the onfocus/onblur events
}
</script>
```

Adding a Hint to any Control Programmatically: *PeterBlum.DES.Globals.WebFormDirector.AddHintToControl Method*

The `PeterBlum.DES.Globals.WebFormDirector.HintManager.AddHintToControl()` method should be called from `Page_Load()`. It takes one data entry control and assigns it to a hint. Use it when the control does not have its own **Hint**, **HintHelp**, **SharedHintFormatterName** or **LocalHintFormatter** properties and you prefer to work programmatically instead of using the `NativeControlExtender` control.

This method is overloaded. Use one when you have a `HintFormatter` object. Use the other when you are using the **HintManager.SharedHintFormatters** list.

[C#]

```
void AddHintToControl(Control pFocusControl,
    string pHintText, string pHintHelp,
    PeterBlum.DES.Web.WebControls.HintFormatter pHintFormatter,
    Control pOverrideHintControl)

void AddHintToControl(Control pFocusControl,
    string pHintText, string pHintHelp,
    string pSharedHintFormatterName,
    Control pOverrideHintControl)
```

[VB]

```
Sub AddHintToControl(ByVal pFocusControl As Control,
    ByVal pHintText As String, ByVal pHintHelp As String,
    ByVal pHintFormatter As PeterBlum.DES.Web.WebControls.HintFormatter,
    ByVal pOverrideHintControl As Control)

Sub AddHintToControl(ByVal pFocusControl As Control,
    ByVal pHintText As String, ByVal pHintHelp As String,
    ByVal pSharedHintFormatterName As String,
    ByVal pOverrideHintControl As Control)
```

Parameters

pFocusControl

The control that is assigned to the hint. It will activate the hint when it receives focus and optionally in a tooltip.

pHintText

The text of the hint. It can contain HTML tags. ENTER and LINEFEED characters are not permitted.

When this text is used in the status bar (**InStatusBar** property), all HTML tags are stripped.

When "", the control does not show a hint. However, it sets up the hint system in case you are using the **HintFormatter.FormatterFunctionName** or **HintFormatter.TextFunctionName** properties to establish the text.

pHintHelp

When the `HintFormatter` uses a `PopupView`, this parameter provides data for use by the **PopupView.HelpBehavior** property.

pHintFormatter

The `HintFormatter` object that describes the appearance of the hint. See "[Properties for the PeterBlum.DES.Web.WebControls.HintFormatter Class](#)".

pSharedHintFormatterName

When using a `HintFormatter` defined in the `HintManager.SharedHintFormatters` property, this is the name of that `HintFormatter` object.

pOverrideHintControl

The HintFormatter specifies a **HintControlID** where the hint appears on the page when **DisplayMode** is `Static` or `Dynamic`. You can override it with a control specified here. Overriding allows you to share a HintFormatter object with the exception of its HintControl. (Create one HintFormatter and pass it to this method multiple times.) When not used, pass `null`.

Properties for the `PeterBlum.DES.Web.WebControls.HintFormatter` Class

The `PeterBlum.DES.Web.WebControls.HintFormatter` class defines the appearance of a hint. See “[Using Interactive Hints](#)”.

- **Name** (string) – When adding the `HintFormatter` to the `HintManager.SharedHintFormatters` collection, this must be assigned with a unique name (amongst those in the collection). Controls that need hints will assign this name in their `SharedHintFormatterName` property to retrieve the `HintFormatter`.
- **DisplayMode** (enum `PeterBlum.DES.HintDisplayMode`) – Determines how the hint is displayed: on screen or in a popup.

Use `Static` or `Dynamic` if you have a control shown on the page to output the hint. Specify the control on the page with the `HintControlID` or `HintControl` properties. See “[Defining Hints shown on the Page](#)”.

Use `Popup` if you want the hint to popup. It uses the `PopupView` definition in the `PopupViewName` property. See “[Using PopupViews](#)”.

Only use `None` when the hint appears in the tooltip and/or status bar, but not in a control on the page or a popup hint.

The enumerated type `PeterBlum.DES.HintDisplayMode` has these values:

- `None` – No hint is shown on the page or as a `PopupView`.
- `Static` – The control appears on the page in a `Label` or `Panel`. When hidden, space is preserved. Assign the `Label` or `Panel` to `HintControlID`.
- `Dynamic` – The control appears on the page in a `Label` or `Panel`. When hidden, space is not used. Assign the `Label` or `Panel` to `HintControlID`.
- `Popup` – A `PopupView` is used. The `PopupViewName` property must specify the name of the `PopupView` definition.

It defaults to `HintDisplayMode.Popup`.

- **HintControlID** (string) – The control where the hint will be shown on the page. It must be assigned when `DisplayMode` is `Static` or `Dynamic`.

It can either be a `Label` or a `Panel` that contains a `Label` whose ID is the `Panel.ID + "_Text"`. See “[Defining Hints shown on the Page](#)”.

When `FormatterFunctionName` is assigned, this control is passed into your formatter function and it decides how to prepare the hint control.

`HintControlID` must be assigned to a control in the same or a parent naming container. For any other naming container, use `HintControl`.

It defaults to "".

- **HintControl** (Control) – This is an alternative to `HintControlID`. It has the same features as `HintControlID`. It is assigned a reference to a control instead of an ID. As a result, it supports controls in any naming container. It must be assigned programmatically.

When programmatically assigning properties to a `HintFormatter`, it is better to use `HintControl` instead of `HintControlID` because DES operates faster using `HintControl`.

- **FormatterFunctionName** (string) – Assign to the name of a JavaScript function that will be called as the hint control is shown or hidden. It allows you to customize the `HintControl` based on conditions at the time the hint is requested. See “[Customize How Hints Appear: The Formatter Function](#)”.

When "", no formatter function is set up. It defaults to "".

ALERT: Many users make the mistake of assigning JavaScript code to this property. This will cause JavaScript errors. **GOOD:** “`MyFunction`”. **BAD:** “`MyFunction();`” and “`alert('stop it')`”.

Note: JavaScript is case sensitive. Be sure the value of this property exactly matches the function definition.

- **PopupViewName** (string) – Determines which globally defined PopupView is used. Specify the name or use "{DEFAULT}" to select the name defined globally. See [“Using PopupViews”](#).

Use the **Global Settings Editor** to edit PopupView definitions. See [“Defining PopupViews”](#).

Here are the predefined values:

LtYellow-Small, LtYellow-Medium, LtYellow-Large, ToolTip-Small, ToolTip-Medium, and ToolTip-Large.

All of these are light yellow. Their widths vary from 200px to 600px. The ToolTip definitions do not have the callout feature enabled.

When "{DEFAULT}", it selects the name from the global setting **DefaultHintPopupViewName**, which is defined in the **Global Settings Editor**. For ToolTips on controls that don't have an associate Hint, it selects the name from the global setting **DefaultToolTipPopupViewName**, which is defined in the **Global Settings Editor**.

When "", it uses the factory default PopupView, which is a light yellow style, Width=200px, **PopupView.HelpBehavior**=ButtonAppends, and **PopupView.DefaultPosition**=BottomRightSidesAlign.

When the name is specified here is unknown, it also uses the factory default. This allows the software to operate if the you change the name of a global value and forget to change the name in this property.

It defaults to "{DEFAULT}".

- **OverriddenPopupView** (PeterBlum.DES.Web.WebControls.HintPopupView) – Overrides the value in **PopupViewName** with an instance of your own PeterBlum.DES.Web.WebControls.HintPopupView class to establish the appearance of the popup hint box.

When null, **PopupViewName** is used.

When assigned, this property is used to establish the appearance of the popup hint message box. See [“Properties for the PeterBlum.DES.Web.WebControls.HintPopupView Class”](#).

It defaults to null.

If you want to start with one of the Hint PopupViews defined in the **Global Settings Editor**, use the `GlobalToOverriddenPopupView()` method to set up **OverriddenPopupView**. Then edit the properties of **OverridePopupView** to customize it. See the example below.

Example: Creating a PopupView

[C#]

```
PeterBlum.DES.Web.WebControls.HintPopupView vPV = new HintPopupView();
vPV.HelpBehavior = PeterBlum.DES.HelpBehavior.ButtonReplaces;
vPV.Width = new Unit("350px");
vPV.DefaultPosition = PeterBlum.DES.DefaultViewPosition.BottomCentered;
vHintFormatter.OverriddenPopupView = vPV;
```

[VB]

```
Dim vPV As PeterBlum.DES.Web.WebControls.HintPopupView = New HintPopupView()
vPV.HelpBehavior = PeterBlum.DES.HelpBehavior.ButtonReplaces
vPV.Width = New Unit("350px")
vPV.DefaultPosition = PeterBlum.DES.DefaultViewPosition.BottomCentered
vHintFormatter.OverriddenPopupView = vPV
```

Example: Using GlobalToOverriddenPopupView() method

The `HintFormatter.GlobalToOverriddenPopupView()` method populates the **OverriddenPopupView** based on a `PopupView` defined in the **Global Settings Editor**. It has one parameter, the name of the `PopupView`. It returns an instance of the `PopupView`, which you can edit. You don't need to assign it to **OverriddenPopupView**.

[C#]

```
PeterBlum.DES.Web.WebControls.HintPopupView vPV =
    vHintFormatter.GlobalToOverriddenPopupView("MyPopupView");
vPV.HelpBehavior = PeterBlum.DES.HelpBehavior.ButtonReplaces;
vPV.Width = new Unit("350px");
vPV.DefaultPosition = PeterBlum.DES.DefaultViewPosition.BottomCentered;
```

[VB]

```
Dim vPV As PeterBlum.DES.Web.WebControls.HintPopupView = _
    vHintFormatter.GlobalToOverriddenPopupView("MyPopupView")
vPV.HelpBehavior = PeterBlum.DES.HelpBehavior.ButtonReplaces
vPV.Width = New Unit("350px")
vPV.DefaultPosition = PeterBlum.DES.DefaultViewPosition.BottomCentered
```

- **PopupAction** (enum `PeterBlum.DES.HintPopupAction`) – Used when `DisplayMode=Popup` to determine the events that display the `PopupView`.

The enumerated type `PeterBlum.DES.HintPopupAction` has these values:

- **Focus** – The control has focus.
- **MouseOver** – The mouse passes over the control, like a tooltip.
- **Both** – The control has focus or the mouse passes over it.
- **Default** – Get the value from the **PageManager.HintManager.DefaultPopupAction** property. If you don't use the `PageManager`, its set in **PeterBlum.DES.Globals.GetWebFormDirector().HintManager.DefaultPopupAction**. This is the default.

- **InStatusBar** (Boolean) – When `true`, the hint text appears in the browser's status bar. When `false`, it does not.

HTML tags in the hint text are stripped before showing it in the status bar.

It defaults to `false`.

- **InToolTip** (Boolean) – When `true`, show the Hint as the tooltip, but only if the **ToolTip** (and **ToolTipLookupID**) property on the control is empty. It defaults to `true`.

Since the tooltip is not activated by focus on the control, its text is static, not influenced by **HiddenOnError**. It will strip out HTML tags found in the **Hint** property automatically.

- **HiddenOnError** (Boolean) – When `true`, do not show the hint in the hint control when any `Validator` attached to this `TextBox` reports an error "inline" or is showing its **NoErrorFormatter**. It defaults to `false`.

This allows the user to place the hint control in the same location as a `Validator`. Recommendation: Set **DisplayMode** to `Dynamic`.

This property has no effect on showing the hint in the status bar because it never conflicts with a `Validator` on the page. It has no effect if the `validator` has its **ErrorFormatter.Display** property set to `None` and is not using the **NoErrorFormatter**.

Validation errors can also be blended into the Hint Control using the **HintManager.HintsShowErrors** property. (See the next section.) When **HiddenOnError** is `true`, it overrides **HintManager.HintsShowErrors**.

- **TextFunctionName** (string) – Assign to the name of a JavaScript function that customizes the hint text before the hint is shown. See "[Customize the Text of the Hint: The Text Function](#)".

When "", no text function is set up. It defaults to "".

ALERT: Many users make the mistake of assigning JavaScript code to this property. This will cause JavaScript errors.

GOOD: "MyFunction". BAD: "MyFunction();" and "alert('stop it')".

Note: JavaScript is case sensitive. Be sure the value of this property exactly matches the function definition.

- **InitFunctionName** (string) – Assign to the name of a JavaScript function that establishes the onfocus and onblur events of a custom control so they call `DES_ShowHint()` and `DES_HideHint()`. See "[Javascript functions: Show and Hide the Hint On Demand](#)".

When "", no initialization function is set up. It defaults to "".

ALERT: Many users make the mistake of assigning JavaScript code to this property. This will cause JavaScript errors.

GOOD: "MyFunction". BAD: "MyFunction();" and "alert('stop it')".

Note: JavaScript is case sensitive. Be sure the value of this property exactly matches the function definition.

Properties on the *PeterBlum.DES.Globals.WebFormDirector.HintManager* Property

The following properties are on the **HintManager** property of the PageManager control and **PeterBlum.DES.Globals.WebFormDirector** object. You set them in the `Page_Load()` method. You can also set global default values for these properties using the **Global Settings Editor**. Each property will identify the name to set in the **Global Settings Editor**.

- **SharedHintFormatters** (*PeterBlum.DES.Web.WebControls.HintFormattersList*) – A collection of *HintFormatters* shared by the controls on this page.

This list is optional. Controls specify the name of a *HintFormatter* from this list in their *SharedHintFormatterName* property.

Each *HintFormatter* must have its *Name* property assigned and each must have a unique name within the list.

This list can optimize a page because it reduces the amount of javascript written. Instead of one *HintFormatter* per control, there is one for a group of controls.

It also makes a centralized place for the formatting definitions, so you can make a change in one place and affect all associated controls.

See “[Adding HintFormatters to the SharedHintFormatters Property](#)”.
- **DefaultSharedHintFormatterName** (string) – When the **SharedHintFormatterName** is the text "{DEFAULT}", it is replaced by this value. It allows a page-level default.

When "", *TextBoxes* and the *MultiSegmentDataEntry* control will automatically use the *HintFormatter* defined in their own *LocalHintFormatter* property.

It defaults to "".
- **HintsShowErrors** (enum *PeterBlum.DES.Web.WebControls.HintsShowErrors*) – Determines if error messages are shown in the Hint Control. The enumerated type *PeterBlum.DES.Web.WebControls.HintsShowErrors* has these values:

 - *Hint* – Show only the hint text. If there is no hint text, nothing is shown.
 - *OneErrorAndHint* – Show the error message of first validator reporting an error and the hint text.
 - *AllErrorsAndHint* – Show all error messages of all validators reporting an error and the hint text.
 - *OneError* – Show the error message of first validator reporting an error. The hint text is not shown.
 - *AllErrors* – Show all error messages of all validators reporting an error. The hint text is not shown.

It defaults to the **DefaultHintsShowErrors** property in the **Global Settings Editor**, which defaults to *HintsShowErrors.Hint*.
- **HintsShowErrorsCssClass** (string) – When showing error messages in the Hint Control, use this to change the style sheet class name of the entire Hint Control. For example, change the background color to make it obvious that it’s showing an error.

When "", it is not used.

It defaults to the **DefaultHintsShowErrorsCssClass** property in the **Global Settings Editor**, which defaults to "".
- **HintsShowErrorsCssClass2** (string) – When showing error messages in the Hint Control, use this to change the style sheet class name of the text for error messages. This does not effect the overall Hint Control’s style nor the hint text, if not shown. It helps make the error messages stand out from the hint text.

When "", it is not used.

It defaults to the **DefaultHintsShowErrorsCssClass2** property in the **Global Settings Editor**, which defaults to "".
- **HintsShowErrorsSeparator** (string) – When showing error messages in the Hint Control, there may be several pieces of text joined together. This provides text that goes between the text. Since this will appear on a web page, use HTML for formatting like spaces (“`nbsp;`”) and newline (“`
`”).

It defaults to the **DefaultHintsShowErrorsSeparator** property in the **Global Settings Editor**, which defaults to " ".

- **ToolTipsAsHints** (enum PeterBlum.DES.TrueFalseDefault) – When a control has its **ToolTip** property assigned and its **Hint** property unassigned, setup a hint with the same text as the tooltip. By default, this feature is enabled.

This feature does not apply to every control. It is limited to data entry controls, where focus can be established. For all other controls, you must explicitly setup the **Hint** property. *Supported control types: TextBox, ListBox, DropDownList, RadioButtonList, CheckBoxLayout, RadioButton, CheckBox, and any control that supports the ValidationPropertyAttribute (which allows many third party controls to be included.)*

The enumerated type PeterBlum.DES.TrueFalseDefault has these values:

- True – ToolTips are used as hints.
- False – Do not use ToolTips as hints.
- Default – Determine if ToolTips are used as Hints from the global setting **DefaultToolTipsAsHints**, defined in the “HintManager Defaults” section of the **Global Settings Editor**. The global default is true.

It defaults to TrueFalseDefault.Default.

- **AllowBlankHints** (Boolean) – Normally hints are not created if the hint text is blank. This sets up the control to use a hint, when its hint text is blank. Used when you use the **HintFormatter.TextFunctionName** property.

It defaults to false.

- **DefaultPopupAction** (enum PeterBlum.DES.HintPopupAction) – Used when HintFormatter.DisplayMode=Popup to determine the events that display the PopupView.

The enumerated type PeterBlum.DES.HintPopupAction has these values:

- Focus – The control has focus. This is the default.
- MouseOver – The mouse passes over the control, like a tooltip.
- Both – The control has focus or the mouse passes over it.
- Default – Same as Focus.

- **PopupOnFocusDelay** (Integer) - When using a PopupView for a hint, this is the time delay between when the focus enters the control until it pops up.

The value is in milliseconds.

If 0, it pops up immediately.

It defaults to 350 (>1/3 second).

- **EnableToolTipsUsePopupViews** (enum PeterBlum.DES.TrueFalseDefault) – Enables displaying tooltips in PopupViews. See “[Enhanced ToolTips](#)”. By default, this feature is disabled.

When enabled, controls that are using hints, tooltips or the NativeControlExtender can switch from the standard browser tooltip to a PopupView defined in the Hints system.

The enumerated type PeterBlum.DES.TrueFalseDefault has these values:

- True – Enable the Enhanced ToolTips feature.
- False – Do not use the Enhanced ToolTips feature. ToolTips are defined by the browser’s standard tooltip mechanism.
- Default – Determine if the feature is enabled from the global setting **DefaultEnableToolTipsUsePopupViews**, defined in the “HintManager Defaults” section of the **Global Settings Editor**. The global default is false.

It defaults to TrueFalseDefault.Default.

- **HintShowTextCounterSeparator** (string) – Used with by TextCounter control to establish a separator between the actual hint text and the text from the TextCounter message.

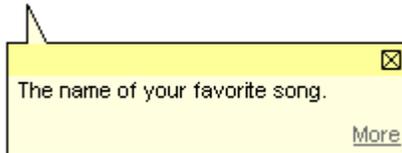
It defaults to the **DefaultHintsShowTextCounterSeparator** property in the **Global Settings Editor**, which defaults to “
”.

If you want the TextCounter message to appear first, use the token “{~}” as the first element of the **HintShowTextCounterSeparator** property.

Properties for the PeterBlum.DES.Web.WebControls.HintPopupView Class

The PeterBlum.DES.Web.WebControls.HintPopupView class contains a PopupView definition. You normally edit these in the Global Settings Editor. You can also create them for use with the **HintFormatter.OverriddenPopupView** property.

See “[Defining PopupViews](#)”.



Click on any of these topics to jump to them:

- ◆ [Overall Appearance Properties](#)
- ◆ [Header Properties](#)
- ◆ [Body Properties](#)
- ◆ [Footer Properties](#)
- ◆ [Callout Properties](#)
- ◆ [Positioning Properties](#)
- ◆ [Other Properties](#)

Overall Appearance Properties

- **CssClass** (string) – The Cascading Style Sheet name that is applied to the overall control. Use to define the background and border.

It defaults to “DES_PHOverallLtYellow”.

These styles are declared in **DES/Appearance/Interactive Pages/PopupHints.css**:

```
.DES_PHOverallLtYellow
{
    border-right: black 1px solid;
    border-top: black 1px solid;
    border-left: black 1px solid;
    border-bottom: black 1px solid;
    font-family: Arial;
    font-size: 8pt;
    color: Black;
    background-color: #ffffe0; /* lightyellow */
}
/* default font for all nested tables in the control */
.DES_PHOverallLtYellow TABLE
{
    font-family: Arial;
    font-size: 8pt;
}

/* prevent external img styles from affecting these styles */
.DES_PHOverallLtYellow img
{
    background-color:transparent;
    margin-left: 0px;
    margin-top: 0px;
    margin-bottom:0px;
    margin-right:0px;
}
```

In addition, the style sheet file contains these alternatives which just change the background-color attribute:

DES_PHOverallLtBlue

DES_PHOverallLtGray

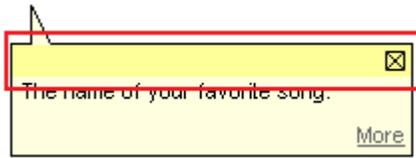
DES_PHOverallLtRed

- **Width** (System.Web.UI.WebControls.Unit) – The width of the PopupView (excluding any callouts). The width is a fixed value. The height varies based on hint text.

Create different width PopupView definitions for any appearance you want.

It defaults to 200px.

Header Properties



- **HeaderTitle** (string) – Optional text shown in the header. It supports HTML.
 When "", no title is offered. The header is hidden if also **ShowCloseButton** is *false*.
 It defaults to "".
- **HeaderTitleLookupID** (string) – Gets the value for **HeaderTitle** through the String Lookup System. (See “String Lookup System” in the **General Features Guide**.) The LookupID and its value should be defined within the String Group of **PopupViews**. If no match is found OR this is blank, **HeaderTitle** will be used.
 The String Lookup System lets you define a common set of terms so the programmer doesn't uniquely define them each time. It also provides localization based on the current culture.
 To use it, define a LookupID and associated textual value in your data source (resource, database, etc). Assign the same LookupID to this property.
 It defaults to "".
- **HeaderHorizontalAlign** (enum [System.Web.UI.WebControls.HorizontalAlign](#)) – The alignment of contents of the header.
 It defaults to `HorizontalAlign.Left`.
- **HeaderCssClass** (string) – The Cascading Style Sheet name that is applied to the header.
 It defaults to “DES_PHHeaderLtYellow”.
 This style is declared in **DES/Appearance/Interactive Pages/PopupHints.css**:


```
.DES_PHHeaderLtYellow
{
    background-color: #ffff99; /* darker version of LightYellow */
    font-size: 8pt;
    /* add this if you allow dragging and want to emphasize that fact
    cursor: move;
*/
}
```

 In addition, the style sheet file contains these alternatives which just change the background-color attribute:
 DES_PHHeaderLtBlue
 DES_PHHeaderLtGray
 DES_PHHeaderLtRed
- **ShowCloseButton** (Boolean) – Show the Close button in the header, on the right side. It will use **CloseButtonImageUrl** or **CloseButtonText** to determine its appearance. If **CloseButtonImageUrl** is assigned, an image is shown. If **CloseButtonImageUrl** is "", a hyperlink is shown using the **CloseButtonText**.
 It defaults to `true`.
- **CloseButtonImageUrl** (string) – The Url to an image for the Close Button.
 If supplied, an image is shown with the tooltip and Image Alt= text from **CloseButtonText**.
 It defaults to "{APPEARANCE}/Shared/CloseCmd.gif" (☒).
 DES also includes this image: ☒ To use it, assign **CloseButtonImageUrl** to "{APPEARANCE}/Shared/CloseCmd2.gif".

Special Symbols for URLs

The “{APPEARANCE}” token will be replaced by the default path to the **Appearance** folder, which you defined as you set up the web site.

Supports the use of the tilde (~) as the first character to be replaced by the virtual path to the web application.

Images for Pressed and MouseOver Effects

You can have images for pressed and mouseover effects as well as the normal image. The names of the image files determine their purpose. Define the name of the normal image. For example, “myimage.gif”. Create the pressed version and give it the same name, with “Pressed” added before the extension. For example, “myimagepressed.gif”. Create the mouseover version and give it the same name, with “MouseOver” added before the extension. For example, myimagemouseover.gif.

The **CloseButtonImageUrl** property should refer to the normal image. DES will detect the presence of the other two files. If any are missing, DES continues to use the normal image for that case. *Note: Auto detection only works when the URL is a virtual path to a file. You can manage this capability with the [PeterBlum.DES.Globals.WebFormDirector.ButtonEffectsManager.EnableButtonImageEffects](#).*

If you need more control over paths for pressed and mouseover images, you can embed up to 3 URLs into this property using a pipe (|) delimited list. The order is important: normal | pressed | mouseover. If you want to omit the pressed image, use: normal | | mouseover. If you want to omit the mouseover image, use: normal | pressed.

- **CloseButtonText** (string) – The text for the Close Button. When **CloseButtonImageUrl** is used, this is the alternative text for the image.

When **CloseButtonImageUrl** is "", this is the text of a hyperlink.

It defaults to "[x]".

- **CloseButtonTextLookupID** (string) – Gets the value for **CloseButtonText** through the String Lookup System. (See “String Lookup System” in the **General Features Guide**.) The LookupID and its value should be defined within the String Group of [PopupViews](#). If no match is found OR this is blank, **CloseButtonText** will be used.

The String Lookup System lets you define a common set of terms so the programmer doesn't uniquely define them each time. It also provides localization based on the current culture.

To use it, define a LookupID and associated textual value in your data source (resource, database, etc). Assign the same LookupID to this property.

It defaults to "".

- **CloseButtonCssClass** (string) – The Cascading Style Sheet name that is applied to the Close Button in the header.

You can define pressed and mouseover styles by using the same style sheet class name plus the text “Pressed” or “MouseOver”. These styles will merge with the style sheet class defined here. So any properties in the pressed and mouseover classes will override properties in this, but not the entire style.

If blank, it is not used.

It defaults to “DES_CloseButtonLtYellow”.

These styles are declared in **DES/Appearance/Interactive Pages/PopupHints.css**:

```
.DES_CloseButtonLtYellow
{
  cursor: default;
  color: #696969; /* dimgray */
  font-size:8pt;
  background-color:White;
}
.DEs_CloseButtonLtYellowPressed
{
  color: black;
}
.DEs_CloseButtonLtYellowMouseOver
{
  color: #a9a9a9; /* darkgray */
}
```

In addition, the style sheet file contains these alternatives (which have identical attributes but are selected depending to the desired color scheme):

```
DES_PHCloseButtonLtBlue
DES_PHCloseButtonLtGray
DES_PHCloseButtonLtRed
```

- **CloseButtonToolTip** (string) – The ToolTip for the Close button.

It defaults to “Close”.

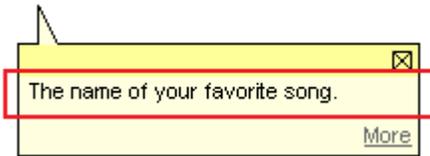
- **CloseButtonToolTipLookupID** (string) – Gets the value for **CloseButtonToolTip** through the String Lookup System. (See “String Lookup System” in the **General Features Guide**.) The LookupID and its value should be defined within the String Group of PopupViews. If no match is found OR this is blank, **CloseButtonToolTip** will be used.

The String Lookup System lets you define a common set of terms so the programmer doesn't uniquely define them each time. It also provides localization based on the current culture.

To use it, define a LookupID and associated textual value in your data source (resource, database, etc). Assign the same LookupID to this property.

It defaults to "".

Body Properties



- **BodyCssClass** (string) – The style sheet class name used for the body. It contains the text of the hint, so use it to establish the font of the hint and margins around that text.

It defaults to “DES_PHBodyLtYellow”.

These styles are declared in **DES/Appearance/Interactive Pages/PopupHints.css**:

```
.DES_PHBodyLtYellow
{
    cursor: default;
    margin-left: 5px;
    margin-right: 5px;
    margin-bottom: 5px;
}

/* when using HelpBehavior=ButtonAppend, the HelpSeparator
may contain an <hr> tag. This helps set its style. */
.DES_PHBodyLtYellow hr
{
}
```

In addition, the style sheet file contains these alternatives (which have identical attributes but are selected depending to the desired color scheme):

DES_PHBodyLtBlue

DES_PHBodyLtGray

DES_PHBodyLtRed

- **BodyImageUrl** (string) – The Url to an image that appears to the left of the message text in the body.

If supplied, it appears to the left of the message text using a two column table. Use **BodyImageVerticalAlign** to determine how the image is positioned within its table cell.

There is a global default in the **DefaultHintPopupViewBodyImageUrl** property of the **Global Settings Editor**. Assign **BodyImageUrl** to “{DEFAULT}” to use the global default. It is unassigned by default.

It defaults to “{DEFAULT}”.

Special Symbols for URLs

The “{APPEARANCE}” token will be replaced by the default path to the **Appearance** folder, which you defined as you set up the web site.

Supports the use of the tilde (~) as the first character to be replaced by the virtual path to the web application.

Images for Pressed and MouseOver Effects

You can have images for pressed and mouseover effects as well as the normal image. The names of the image files determine their purpose. Define the name of the normal image. For example, “myimage.gif”. Create the pressed version and give it the same name, with “Pressed” added before the extension. For example, “myimagepressed.gif”. Create the mouseover version and give it the same name, with “MouseOver” added before the extension. For example, myimagemouseover.gif.

The **BodyImageUrl** property should refer to the normal image. DES will detect the presence of the other two files. If any are missing, DES continues to use the normal image for that case. *Note: Auto detection only works when the URL is a virtual path to a file. You can manage this capability with the [PeterBlum.DES.Globals.WebFormDirector.ButtonEffectsManager.EnableButtonImageEffects](#).*

If you need more control over paths for pressed and mouseover images, you can embed up to 3 URLs into this property using a pipe (|) delimited list. The order is important: `normal | pressed | mouseover`. If you want to omit the pressed image, use: `normal | | mouseover`. If you want to omit the mouseover image, use: `normal | pressed`.

- **BodyImageVerticalAlign** (enum `System.Web.UI.WebControls.VerticalAlign`) – The vertical alignment of the image identified by **BodyImageUrl**.

It defaults to `VerticalAlign.Top`

- **HTMLBefore** (string) – Include HTML that appears before the hint text.

It defaults to ""

- **HTMLBeforeLookupID** (string) – Gets the value for **HTMLBefore** through the String Lookup System. (See “String Lookup System” in the **General Features Guide**.) The LookupID and its value should be defined within the String Group of **PopupViews**. If no match is found OR this is blank, **HTMLBefore** will be used.

The String Lookup System lets you define a common set of terms so the programmer doesn't uniquely define them each time. It also provides localization based on the current culture.

To use it, define a LookupID and associated textual value in your data source (resource, database, etc). Assign the same LookupID to this property.

It defaults to "".

- **HTMLAfter** (string) – Include HTML that appears after the hint text.

It defaults to ""

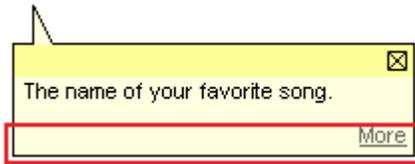
- **HTMLAfterLookupID** (string) – Gets the value for **HTMLAfter** through the String Lookup System. (See “String Lookup System” in the **General Features Guide**.) The LookupID and its value should be defined within the String Group of **PopupViews**. If no match is found OR this is blank, **HTMLAfter** will be used.

The String Lookup System lets you define a common set of terms so the programmer doesn't uniquely define them each time. It also provides localization based on the current culture.

To use it, define a LookupID and associated textual value in your data source (resource, database, etc). Assign the same LookupID to this property.

It defaults to "".

Footer Properties



- **HelpButtonImageUrl** (string) – The Url to an image for the Help Button.

If supplied, an image is shown with the image's Alt= text from **HelpButtonText**.

It defaults to "". DES includes a Help button image in "{APPEARANCE}/Shared/HelpCmd.gif" (?).

Special Symbols for URLs

The "{APPEARANCE}" token will be replaced by the default path to the **Appearance** folder, which you defined as you set up the web site.

Supports the use of the tilde (~) as the first character to be replaced by the virtual path to the web application.

Images for Pressed and MouseOver Effects

You can have images for pressed and mouseover effects as well as the normal image. The names of the image files determine their purpose. Define the name of the normal image. For example, "myimage.gif". Create the pressed version and give it the same name, with "Pressed" added before the extension. For example, "myimagepressed.gif". Create the mouseover version and give it the same name, with "MouseOver" added before the extension. For example, myimagemouseover.gif.

If you need more control over paths for pressed and mouseover images, you can embed up to 3 URLs into this property using a pipe (|) delimited list. The order is important: normal | pressed | mouseover. If you want to omit the pressed image, use: normal | | mouseover. If you want to omit the mouseover image, use: normal | pressed.

- The **HelpButtonImageUrl** property should refer to the normal image. DES will detect the presence of the other two files. If any are missing, DES continues to use the normal image for that case. *Note: Auto detection only works when the URL is a virtual path to a file. You can manage this capability with the [PeterBlum.DES.Globals.WebFormDirector.ButtonEffectsManager.EnableButtonImageEffects](#).*
- **HelpButtonText** (string) – The text for the Help Button. When **HelpButtonImageUrl** is used, this is the alternative text for the image.

When **HelpButtonImageUrl** is "", this is the text of a hyperlink.

It defaults to "More".

- **HelpButtonTextLookupID** (string) – Gets the value for **HelpButtonText** through the String Lookup System. (See "String Lookup System" in the **General Features Guide**.) The LookupID and its value should be defined within the String Group of **PopupViews**. If no match is found OR this is blank, **HelpButtonText** will be used.

The String Lookup System lets you define a common set of terms so the programmer doesn't uniquely define them each time. It also provides localization based on the current culture.

To use it, define a LookupID and associated textual value in your data source (resource, database, etc). Assign the same LookupID to this property.

It defaults to "".

- **HelpButtonCssClass** (string) – The Cascading Style Sheet name that is applied to the Help Button in the footer.

You can define pressed and mouseover styles by using the same style sheet class name plus the text "Pressed" or "MouseOver". These styles will merge with the style sheet class defined here. So any properties in the pressed and mouseover classes will override properties in this, but not the entire style.

If blank, it is not used.

It defaults to "DES_PHHelpButtonLtYellow".

These styles are declared in **DES/Appearance/Interactive Pages/PopupHints.css**:

```
.DES_PHHelpButtonLtYellow
{
    cursor: default;
    color: #696969; /* dimgray */
    font-size: 8pt;
    text-decoration: underline;
}
.DE_S_PHHelpButtonLtYellowPressed
{
    color: black;
    text-decoration: underline;
}
.DE_S_PHHelpButtonLtYellowMouseOver
{
    color: #a9a9a9; /* darkgray */
    text-decoration: underline;
}
```

In addition, the style sheet file contains these alternatives (which have identical attributes but are selected depending to the desired color scheme):

```
DES_PHHelpButtonLtBlue
DES_PHHelpButtonLtGray
DES_PHHelpButtonLtRed
```

- **FooterCssClass** (string) – The Cascading Style Sheet name that is applied to the footer.

It defaults to “DES_PHFooterLtYellow”.

This style is declared in **DES/Appearance/Interactive Pages/PopupHints.css**:

```
.DES_PHFooterLtYellow
{
}
```

In addition, the style sheet file contains these alternatives which just change the background-color attribute:

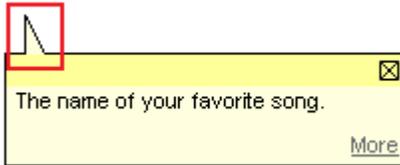
```
DES_PHFooterLtBlue
DES_PHFooterLtGray
DES_PHFooterLtRed
```

- **FooterHorizontalAlign** (string) – (enum [System.Web.UI.WebControls.HorizontalAlign](#)) – The alignment of contents of the footer.

It defaults to `HorizontalAlign.Right`.

Callout Properties

See “[Creating your own Callouts](#)” and “[Adding your own Callouts to the PopupView Definition](#)”.



- **EnableCallouts** (Boolean) – When `true`, the callout graphics are added. Only one appears at a time, based on the positioning of the message box.

A callout is a graphic inserted between the positioning control and the `PopupView` to make the entire presentation look like a callout in a cartoon.

Requires **CalloutUrls** and **CalloutOffsets** to be defined.

NOTE: When set, the `UseShadowEffect` property is ignored because it generates a poor appearance with callouts.

It defaults to `true`.

- **CalloutUrlFolder** (string) – The URL to folder that contains four image files for the callouts. The files must be transparent gifs with the names: `Left.gif`, `Top.gif`, `Right.gif`, and `Bottom.gif`.

There are several predefined callout folders, each with a set of images that work together with the predefined style sheets in **DES/Appearance/Interactive Pages/PopupHints.css**. They are:

```
{APPEARANCE}/Shared/Callouts/AliceBlue
{APPEARANCE}/Shared/Callouts/LtRed
{APPEARANCE}/Shared/Callouts/LtBlue
{APPEARANCE}/Shared/Callouts/LtYellow
{APPEARANCE}/Shared/Callouts/LtGray
{APPEARANCE}/Shared/Callouts/Mistyrose
```

If you define your own, images use transparency and must be a gif file format. See “[Creating your own Callouts](#)”.

Always define the size of these images using **CalloutTopBottomSize** and **CalloutLeftRightSize**. The sizes of the predefined callout files predefined in these properties: 20 tall and 12 wide.

It defaults to “`{APPEARANCE}/Shared/Callouts/LtYellow`” which has these images:



Special Symbols for URLs

The “`{APPEARANCE}`” token will be replaced by the default path to the **Appearance** folder, which you defined as you set up the web site.

Supports the use of the tilde (`~`) as the first character to be replaced by the virtual path to the web application.

- **CalloutLeftRightSize** ([System.Drawing.Size](#)) – The actual width and height of the `Left.gif` and `Right.gif` images defined in **CalloutUrlFolder**. It is used in positioning the `PopupView` box. As a result, if it’s slightly larger, the entire callout will be moved away from the target. If it’s smaller, it will overlap the `PopupView` box.

If you have a border around the `PopupView` box and the outside edges of the callout, subtract the number of pixels used to make the border.

It defaults to `Width=19` and `Height=12`.

- **CalloutTopBottomSize** (System.Drawing.Size) – The actual width and height of the Top.gif and Bottom.gif images defined in **CalloutUrlFolder**. It is used in positioning the PopupView box. As a result, if it's slightly larger, the entire callout will be moved away from the target. If it's smaller, it will overlap the PopupView box.
If you have a border around the PopupView box and the outside edges of the callout, subtract the number of pixels used to make the border.
It defaults to Width=12 and Height=19.
- **CalloutOffsetIntoAnchorPercent** (integer) – Determines how much to offset the callout into the body of the anchor control - the control that the callout points to. It is a percentage where 0 is the top or left and 100 is the bottom or right. Generally avoid using values near 100 as the callout may exceed the boundaries of the PopupView.
It defaults to 50 (percent).
- **CalloutOffsetAlongSide** (integer) – Determines the minimum offset for the callout from the nearest corner so it is not flush with that corner. The value is in pixels.
It defaults to 10 (pixels).

Positioning Properties

- **DefaultPosition** (enum PeterBlum.DES.PopupViewPosition) – Positions the PopupView relative to the target control. At runtime, the position may change if the PopupView either overlaps the target control or the limits of the viewable space.

Use **HorizPositionOffset** and **VerticalPositionOffset** to offset from the selected position by a specific number of pixels.

The enumerated type `PeterBlum.DES.PopupViewPosition` has these values:

- `LeftCentered` - Horizontal alignment: Left of the target. Vertical alignment: Centered
- `LeftTopsAlign` - Horizontal alignment: Left of the target. Vertical alignment: top of target aligns with top of popup view
- `RightCentered` - Horizontal alignment: Right of the target. Vertical alignment: Centered
- `RightTopsAlign` - Horizontal alignment: Right of the target. Vertical alignment: top of target aligns with top of popup view
- `BottomCentered` - Horizontal alignment: Centered. Vertical alignment: Below the target
- `BottomLeftSidesAlign` - Horizontal alignment: Left sides of popup and target align. Vertical alignment: Below the target
- `BottomRightSidesAlign` - Horizontal alignment: Right sides of popup and target align. Vertical alignment: Below the target
- `TopCentered` - Horizontal alignment: Centered. Vertical alignment: Above the target
- `TopLeftSidesAlign` - Horizontal alignment: Left sides of popup and target align. Vertical alignment: Above the target
- `TopRightSidesAlign` - Horizontal alignment: Right sides of popup and target align. Vertical alignment: Above the target

It defaults to `PopupViewPosition.BottomRightSidesAlign`.

- **HorizPositionOffset** (short) – Adjusts the Horizontal position of the popup by a number of pixels to allow more precise positioning for **DefaultPosition**.

If negative, the popup panel moves left. Positive moves right. Zero does nothing.

It defaults to 5.

- **VertPositionOffset** (short) – Adjusts the vertical position of the popup by a number of pixels to allow more precise positioning for **DefaultPosition**.

If negative, the popup panel moves up. Positive moves down. Zero does nothing.

It defaults to 5.

Other Properties

- **HelpBehavior** (enum PeterBlum.DES.HelpBehavior) – Determines how the **HintHelp** property (on the control with the hint) is used.

The enumerated type **PeterBlum.DES.HelpBehavior** has these values:

- **None** – Do not use **HintHelp**. Do not show a Help Button.
- **ButtonAppends** – Use the Help Button. When clicked, redraw with the **HintHelp** text appended to the current text. The value of **PopupView.AppendHelpSeparator** is inserted between the original hint and the text of **HintHelp**.
- **ButtonReplaces** – Use the Help Button. When clicked, redraw with the **HintHelp** text replacing the current text.
- **Title** – The **HintHelp** text appears in the header as the title. It is used instead of the **PopupView.HeaderText** property value. There is no Help Button.
- **Hyperlink** – Use the Help Button that acts as a hyperlink. Define the URL in the **HyperlinkUrlForHelpButton** property. The **HintHelp** text will appear in the "{0}" token.
- **HyperlinkNewWindow** – Use the Help Button that acts as a hyperlink which opens in a new window. Define the URL in the **HyperlinkUrlForHelpButton** property. The **HintHelp** text will appear in the "{0}" token.
- **Script** – Runs the script supplied in the **ScriptForHelpButton** property. The **HintHelp** text will replace the token "{0}" in that script.

It defaults to `HelpBehavior.ButtonAppends`.

- **HyperlinkUrlForHelpButton** (string) – Used when **HelpBehavior** is **Hyperlink** or **HyperlinkNewWindow**. It defines the URL of the Hyperlink.

Create a full URL that will be used in the href= attribute of the A tag. It can contain the token "{0}" to be replaced by the **HintHelp** value of the control requesting your **PopupView**. That token is used to differentiate elements of URLs, such as the page or querystring parameter. For example:

```
http://www.mywebsite.com/help?helpid={0}
```

The entire value can be "{0}" if the **HintHelp** value contains the complete URL.

It defaults to "{0}".

- **ScriptForHelpButton** (string) – Used when **HelpBehavior** is **Script**. It defines the script to invoke when the button is clicked.

The token "{0}" is replaced by the **HintHelp** text. Use it to customize the script. For example:

```
alert('{0}');
```

WARNING: When the token is inside a string, like in the above example, the **HintHelp** property should not contain the same quote characters that enclose the string. For example, the text "Peter's Software" is illegal. It will cause a JavaScript error.

This script should be valid javascript. It should not start with "javascript:".

It defaults to "".

- **AppendHelpSeparator** (string) – Used when **HelpBehavior** is **ButtonAppends**. It is inserted between the initial text and the help text.

It supports HTML.

Typical separators are `
` and `<hr />`.

It defaults to "`<hr />`".

- **Draggable** (enum PeterBlum.DES.PopupViewDraggable) – Determines if the user can drag the popupview.

The enumerated type `PeterBlum.DES.PopupViewDraggable` has these values:

- No - It is not supported.
- Header - Only by dragging the header area, like a title bar
- All - All elements, except buttons, are draggable.

It defaults to `PopupViewDraggable.Header`.

- **UseOpaqueEffect** (Boolean) – When `true` and on a browser that supports Opacity, the entire view is slightly opaque to show its underlying info at various times.

Rules for opacity:

- When the mouse moves over the `PopupView`, it immediately brightens in about .5 second.
- When the mouse leaves the `PopupView`, it starts to dim after 2 seconds and finishes .5 seconds later.
- When focus is placed into a control that shows the `PopupView`, it brightens in about .5 second and stays that way until focus changes. Usually the only way to do that is clicking the Help button.

It defaults to `true`.

Changing the Opacity Behaviors

Opacity behaviors can be adjusted in the “Visual Effects” topic of the **Global Settings Editor** with these properties:

- **MinimumOpaquePercent** (integer) – When **UseOpaqueEffect** is `true` on a `PopupView` definition, this is the value of opacity used as the minimum opacity.

Opacity has a range between 10 and 99, which represents a percentage of opacity. 100 is solid. 0 is transparent.

It defaults to 90.

- **MaximumOpaquePercent** (integer) – When **UseOpaqueEffect** is `true` on a `PopupView` definition, this is the value of opacity used as the maximum opacity.

Opacity has a range between 10 and 99, which represents a percentage of opacity. 100 is solid. 0 is transparent.

It defaults to 100.

- **OpaqueFadeDelay** (integer) – When **UseOpaqueEffect** is `true` on a `PopupView` definition, this is the number of milliseconds before fading begins.

The value is in milliseconds.

If 0, it fades immediately.

It defaults to 2000 (2 seconds).

- **UsePopupEffect** (Boolean) – When using Internet Explorer for Windows 5 and higher, this applies an animation to the opening and closing of the popup. It uses the Filters feature which are set up globally.

When `true` and Internet Explorer is in use, filters are applied.

It defaults to `true`.

- **UseShadowEffect** (Boolean) – When `true` and on Internet Explorer 5.5+ for Windows, a shadow effect is applied.

If Callouts are used, this property is ignored because Callouts perform poorly with shadows.

It defaults to `true`.

- **IEFixPopupOverList** (Boolean) – Internet Explorer for Windows versions 5.0 through 6. have a problem allowing absolutely positioned objects appearing over ListBox and DropDownLists. There is a special hack that uses an IFrame and filter style sheet to make it appear like it's over these controls. This property enables that hack on IE versions 5.5-6. (IE 5 doesn't support the hack; IE 7 doesn't require the hack.)

The hack is imperfect. It breaks when another IFrame is in the same area of the page. By "breaks", this means the popup usually looks incorrect including being transparent.

If the problem is affecting the PopupView, set the **UseShadowEffect** property to `false`.

Turn off the hack to work around this problem. Set this property to `false`. But you should only do this when the popup does not overlap any listboxes or dropdownlists. If there is overlap, you have to make a design decision to change your positioning or avoid using the IFrame.

When `true`, the hack is used when the browser is Internet Explorer for Windows versions 5.5 through 6..

When `false`, the hack not used. Choose this when the hack causes visual problems such as a transparent popup.

It defaults to `true`.

Enhanced ToolTips

The browser provides the ToolTip to describe almost any field as the mouse passes over it. That tooltip is very limited. For most browsers, it cannot be multiline. It has one style (yellow). It cannot support HTML.

Using the same PopupView feature found in DES's [Interactive Hints](#) and the DES Validator's [PopupErrorFormatter](#), DES gives you a better tooltip. You control its appearance and supply it with HTML to convey the information better.

See "[Defining PopupViews](#)".

Note: The terms "Hint" and "ToolTip" both describe ways to provide documentation to the user. A Hint displays the message when focus enters the field and is best for data entry controls. A ToolTip displays the message when the mouse points to the control. It can be used on almost any type of control.

Click on any of these topics to jump to them:

- ◆ [Features](#)
- ◆ [Using Enhanced ToolTips](#)
 - [HintManager.AddToolTipPopupViewToControl\(\) method](#)

Features

- Can be attached to almost any control. DES controls automatically use them when the feature is activated. Non-DES controls get them through the NativeControlExtender or programmatically.
- Appears as the mouse passes over a control. Is removed as the mouse leaves (after a short delay). One difference from the browser's tooltip is that the user can move the mouse onto the tooltip and it will remain visible even though the mouse is outside the control. This lets the text be visible without the mouse hiding a part of the control.
- The PopupView element containing the tooltip text will does not overlap the control (except in extreme circumstances). It positions itself to one side. If there isn't enough screen space for your preferred side, it chooses another side.
- Uses the PopupView feature from Interactive Hints which means:
 - Style-sheet driven, allowing color and other appearance changes
 - The text of the tooltip supports HTML formatting
 - It's easy to add an image to the left of the message with the `PopupView.BodyImageUrl` property
 - Supports Callouts.
 - Draggable
 - Optional title bar
 - Optional close box
 - The same PopupView definitions can be used for both Interactive Hints and ToolTips
- With a single property setting, all DES controls and controls using the NativeControlExtender can be switched to using PopupViews. Set `HintManager.EnableToolTipsUsePopupViews` to `TrueFalseDefault.True` or set it globally in the **Global Settings Editor** with the `DefaultEnableToolTipsUsePopupViews` property.
- When using Interactive Hints, that feature optionally sets up the hint as a tooltip. If the Popup ToolTip feature is enabled, that hint uses the same PopupView definition as the hint (in `HintFormatter.PopupViewName`).
- If the browser does not support the scripts needed for the Popup ToolTip feature or javascript is disabled, it scales down gracefully to using the standard browser tooltip.

Use demos here: <http://www.peterblum.com/DES/DemoToolTips.aspx>.

Using Enhanced ToolTips

Use demos here: <http://www.peterblum.com/DES/DemoToolTips.aspx>.

The Popup ToolTip feature gets the text for a tooltip from either a control’s **ToolTip** property or its **Hint** property (if the Interactive Hints feature is in use and the **HintFormatter** for that control has **InToolTip** set to true).

The setup is easy:

- Assign text for your tooltip messages
- Activate the Enhanced ToolTips feature
- Determine the desired appearance for ToolTips

Here are the details:



These steps ask you to jump around the document using clicks on links. Adobe Reader offers a **Previous View** command to return to the link. Look for this in the Adobe Reader (shown v6.0)

1. DES controls are “Enhanced ToolTips Ready”. Add the `NativeControlExtender` control to any non-DES control that needs a Popup ToolTip. See the **General Features Guide** for the `NativeControlExtender` control.

Alternatively, use the `HintManager.AddToolTipPopupViewToControl()` method. See below.

2. Assign the text for your tooltip messages. There are three possible cases:
 - When using Interactive Hints, use the text from the control’s **Hint** property. Be sure the control’s **HintFormatter.InToolTip** property is true.

Note: The `HintHelp` property is not used on Enhanced ToolTips. `PopupView.HelpBehavior` is ignored.
 - Assign the text to the **ToolTip** property of the control.
 - If the control does not have a **ToolTip** property (such as an `HtmlControl` like ``), use the **ToolTip** property on the `NativeControlExtender`.
3. Activate the Enhanced ToolTips feature either for the page or globally.
 - For the page, set **HintManager.EnableToolTipsUsePopupViews** to `TrueFalseDefault.True` on either **PeterBlum.DES.Globals.WebFormDirector** or the `PageManager` control.
 - Globally, set the **DefaultEnableToolTipsUsePopupViews** property to `true` in the “HintManager Defaults” section of the **Global Settings Editor**.
4. There are several sources that determine which `PopupView` is used for your ToolTip.

- The Interactive Hints feature determines the `PopupView` definition for any control using `PopupViews` as Hints.
- Many DES controls provide the **ToolTipUsesPopupViewName** property to assign the `PopupView` definition name.
- There is a global default. Specify a `PopupView` definition name in the **DefaultToolTipPopupViewName** property in the “HintManager Defaults” section of the **Global Settings Editor**. See “[Defining PopupViews](#)”.

It is used by controls that do not have the **ToolTipUsesPopupViewName** property and those whose **ToolTipUsesPopupViewName** property is set to the “{DEFAULT}” token.

HintManager Defaults	
DefaultHintPopupViewBodyImageUrl	
DefaultHintPopupViewName	LtYellow-Small
DefaultHintsShowErrors	Hint
DefaultHintsShowErrorsCssClass	
DefaultHintsShowErrorsCssClass2	
DefaultHintsShowErrorsSeparator	
DefaultHintsShowTextCounterSeparato	
DefaultToolTipsAsHints	True
Tool Tips Replaced by PopupViews	
DefaultEnableToolTipsUsePopupViews	False
DefaultToolTipPopupViewName	ToolTip-Small
ToolTipHideDelay	300
ToolTipHideOnClick	False
ToolTipHideOnTyping	False
ToolTipShowDelay	500

DefaultToolTipPopupViewName
The default `PopupView` when tooltips are replaced by `Popup` (`HintManager.EnableToolTipsUsePopupViews` is true). Ho

HintManager.AddToolTipPopupViewToControl() method

Use with any non-DES control to convert its standard tooltip to a PopupView. It is an alternative to using the NativeControlExtender control. If this function is called while HintManager.EnableToolTipsUsePopupViews is false, nothing happens.

This method is overloaded.

[C#]

```
void AddToolTipPopupViewToControl(Control pControlWithToolTip,  
    string pPopupViewName)
```

```
void AddToolTipPopupViewToControl(Control pControlWithToolTip,  
    PeterBlum.DES.Web.WebControls.HintPopupView pPopupView)
```

[VB]

```
Sub AddToolTipPopupViewToControl(ByVal pControlWithToolTip As Control,  
    ByVal pPopupViewName As String)
```

```
Sub AddToolTipPopupViewToControl(ByVal pControlWithToolTip As Control,  
    ByVal pPopupView As PeterBlum.DES.Web.WebControls.HintPopupView)
```

Parameters

pControlWithToolTip

The control whose tooltip will be replaced. If this control is **Visible**=false or its **ToolTip** property is unassigned, nothing happens.

pPopupViewName

The name of a Hint PopupView defined globally and will be the PopupView for the tooltip. If "", it uses the global property **DefaultToolTipPopupViewName** which is set in the "HintManager Defaults" topic of the **Global Settings Editor**.

pPopupView

A `PeterBlum.DES.Web.WebControls.HintPopupView` object that defines the PopupView. See "[Properties for the PeterBlum.DES.Web.WebControls.HintPopupView Class](#)". If you assign its **Name** property, it will be used more efficiently so all that have this name will use a common definition.

TextCounter Control

The TextCounter control displays the number of characters or words within a textbox. It assists users when there are limits to the size of text they can enter. It compliments, but does not replace the TextLengthValidator/WordCountValidator, because it does not impose a limit. It merely communicates the count and if a limit is exceeded.

10 characters

The user interface of the TextCounter can be like an interactive label control. It also can present itself in the Hint feature of DES TextBoxes.

Click on any of these topics to jump to them:

- ◆ [Features](#)
- ◆ [Using the TextCounter Control](#)
 - [Connecting To a TextBox](#)
 - [Establishing the Limits](#)
 - [Setting the Text and Style Sheets](#)
- ◆ [Adding a TextCounter Control](#)
- ◆ [Properties of the TextCounter Control](#)
- ◆ [Online examples](#)

Features

- Evaluates the size of the text as compared to a maximum and possibly a minimum.
- Evaluates either the number of characters or the number of words
- Its text and optionally style sheet changes as the text size changes:
 - Below the minimum
 - Between the minimum and the next milestone
 - At or above a milestone prior to the maximum, such as 20 characters left
 - At or above a second milestone prior to the maximum, such as 10 characters left, to allow further emphasis that the user is reaching the maximum
 - Above the maximum
- The text shown supports tokens that can be replaced by the current count, minimum, maximum, and how much it exceeds the maximum.
- It can be shown on the page like a label and/or in the [Interactive Hints](#) feature. When in a Hint, it is hidden except when focus is on the textbox.
- It can change the style sheet of the textbox as the length crosses the maximum. It uses the same style sheet feature as validators do for their “Change Style on Control with Error” feature. It will also hide the TextLengthValidator error message if the length is below the maximum.

Using the TextCounter Control

Click on any of these topics to jump to them:

- ◆ [Connecting To a TextBox](#)
- ◆ [Establishing the Limits](#)
- ◆ [Setting the Text and Style Sheets](#)
 - [Tokens in Messages](#)
- 📄 [Online examples](#)

Connecting To a TextBox

Attach the TextCounter control to a textbox by specifying the textbox's ID in the **TextBoxControlID** property.

Determine where you want the TextCounter to display its information with the **DisplayMode** property. It can act like a Label control, where it displays the current count text in the location of the TextCounter control. It can also integrate itself with the [Interactive Hints](#) feature on the TextBox. The Interactive Hints feature can display the text in a label on the page or in a PopupView. In both cases, the text is not shown unless the textbox has focus.

When **DisplayMode** uses the Interactive Hint feature, use the **HintManager.HintShowTextCounterSeparator** property to describe the HTML that separates the hint text from the textcounter message. It defaults to "
". If you want the textcounter message to appear first, use the token "{~}" as the first element of the **HintManager.HintShowTextCounterSeparator** property. The HintManager is available on the PageManager control and on the **PeterBlum.DES.Globals.WebFormDirector** object.

Determine if it counts characters or words with the **CountType** property. When counting words, a word is considered any sequence of letters, digits, underscores, and single quotes (handles contractions and possessive nouns). Everything else is considered whitespace.

Establishing the Limits

While you can establish a minimum and maximum within the TextCounter's **Minimum** and **Maximum** properties, the TextCounter can get these values for you by looking in two places:

- If you have a TextLengthValidator or WordCountValidator attached to the textbox (and you should!), the validator supplies the limits.
- The **MaxLength** property on the TextBox control.

You can also establish two milestones before reaching the Maximum in **Milestone1** and **Milestone2**. When reached, the message and/or the style sheet class can switch. This allows a visual escalation as the user nears the Maximum. Milestones are the number of characters before the end. Milestone2 defaults to 10, so it will change the message and/or style sheet from 10 characters before the maximum until the maximum is reached. **Milestone1** is not setup by default.

Setting the Text and Style Sheets

This control's job is to communicate to the user that they are in a textbox that has a size limit, and how their entry is affected by that limit. To do this job, it can change the text and style sheet as the user gets close to the maximum. There are 6 cases:

- No maximum defined. This eliminates most of the remaining cases.
- Below the minimum. Only used when there is a minimum defined.
- Between the minimum and Milestone1
- Milestone1: At or above a milestone prior to the maximum, such as 20 characters left. Milestone1 is defined in the **Milestone1** property as the number of characters or words BEFORE the maximum.

For example, if the maximum is 100 and you want to change the message or style sheet at 60 characters, set Milestone1 to 40 (100 – 60).

- Milestone2: At or above a second milestone prior to the maximum, such as 10 characters left, to allow further emphasis that the user is reaching the maximum. Milestone2 is defined in the **Milestone2** property as the number of characters or words BEFORE the maximum.

For example, if the maximum is 100 and you want to change the message or style sheet at 90 characters, set Milestone1 to 10 (100 – 90).

- Above the maximum

Each has its own text, style sheet class, and second style sheet class to differentiate the count (a token) from the rest of the text. The style sheet classes are in **DES\Appearance\Interactive Pages\TextCounter.css**. Here is where to edit the text and style sheet classes for each case:

Case	Property for the Text	Style Sheet Class	Tokens Style Sheet Class
No maximum defined	NoMaximumMessage	DES_TCCNormal	DES_TCCNormalToken
Below the minimum	BelowMinimumMessage	DES_TCCBelowMinimum	DES_TCCBelowMinimumToken
Between the minimum and Milestone1	NormalMessage	DES_TCCNormal	DES_TCCNormalToken
Between Milestone1 and Milestone2	Milestone1Message	DES_TCCMilestone1	DES_TCCMilestone1Token
Between Milestone2 and the maximum	Milestone2Message	DES_TCCMilestone2	DES_TCCMilestone2Token
Above the maximum	AboveMaximumMessage	DES_TCCAboveMaximum	DES_TCCAboveMaximumToken

Tokens in Messages

Each of these messages can display live data by using tokens. Here are the tokens:

{COUNT}

Number of characters or words in the textbox. It is updated as the user types.

Example

You have entered {COUNT}

{COUNT:singular:plural}

Helps build sentences where singular and plural forms are needed when you use the {COUNT} token. For example, “There is 1 item.” and “There are 2 items.”

You replace the term “singular” with the singular form of the word. You replace the term “plural” with the plural form of the word.

It is updated as the user types.

Examples

You entered {COUNT} {COUNT:character:characters}.

You entered {COUNT} character{COUNT::s}

{NEARNESS}

How close the count is to the maximum or minimum. Once the count reaches the minimum, it evaluates the count to the maximum. It is updated as the user types.

Example

You are over the maximum by {NEARNESS}

{NEARNESS: singular:plural}

Helps build sentences where singular and plural forms are needed when you use the {NEARNESS} token.

You replace the term “singular” with the singular form of the word. You replace the term “plural” with the plural form of the word.

It is updated as the user types.

Examples

You are over the maximum by {NEARNESS:character:characters}

You are over the maximum by {NEARNESS} character{NEARNESS::s}

{MINIMUM}

The value of the **Minimum** property.

Example

You are below the minimum of {MINIMUM}

{MAXIMUM}

The value of the **Maximum** property.

Example

You are above the maximum of {MAXIMUM}

Adding a TextCounter Control



These steps ask you to jump around the document using clicks on links. Adobe Reader offers a **Previous View** command to return to the link. Look for this in the Adobe Reader (shown v6.0)

1. Prepare the page for DES controls. See “Preparing a page for DES controls” in the **General Features Guide**. It covers issues like style sheets, AJAX, and localization.
2. Add a TextBox control to the page. See the **TextBoxes User’s Guide** for details. Add the TextLengthValidator or WordCountValidator to the textbox (only available when using the DES Validation Framework.)
3. Add a TextCounter control to the page.

Visual Studio and Visual Web Developer Design Mode Users

Drag the TextCounter control from the Toolbox onto your web form.

Text Entry Users

Add the control (inside the <form> area):

```
<des:TextCounter id="[YourControlID]" runat="server" />
```

Programmatically creating the TextCounter control

- Identify the control which you will add the TextCounter control to its **Controls** collection. Like all ASP.NET controls, the TextCounter can be added to any control that supports child controls, like Panel, User Control, or TableCell. If you want to add it directly to the Page, first add a Placeholder at the desired location and use the Placeholder.
- Create an instance of the TextCounter control class. The constructor takes no parameters.
- Assign the **ID** property.
- Add the TextCounter control to the **Controls** collection.

In this example, the TextCounter is created with an **ID** of “TextCounter1”. It is added to Placeholder1.

[C#]

```
PeterBlum.DES.Web.WebControls.TextCounter vTextCounter =  
    new PeterBlum.DES.Web.WebControls.TextCounter();  
vTextCounter.ID = "TextCounter1";  
Placeholder1.Controls.Add(vTextCounter);
```

*Note: The namespace for these controls is PeterBlum.DES. If you prefer, add a **using** clause to that namespace on your form.*

[VB]

```
Dim vTextCounter As PeterBlum.DES.Web.WebControls.TextCounter = _  
    New PeterBlum.DES.Web.WebControls.TextCounter()  
vTextCounter.ID = "TextCounter1"  
Placeholder1.Controls.Add(vTextCounter)
```

*Note: The namespace for these controls is PeterBlum.DES. If you prefer, add an **Imports** clause to that namespace on your form.*

Guidelines for setting properties

- Design mode users can use the Properties Editor or the Expanded Properties Editor. (See “Expanded Properties Editor” in the **General Features Guide**.) The SmartTag  also offers some of the most important properties.
- Text entry users should add the properties into the `<des:ControlClass>` tag in this format:
propertyname="value"
- When setting a property programmatically, have a reference to the control’s object and set the property according to your language’s rules.

4. Assign the TextBox control to the **TextBoxControlID** property.
5. If counting words, set **CountType** to Words.
6. Establish the limits, either in the textbox’s **MaxLength** property, the TextLengthValidator or WordCountValidator’s **Minimum** and **Maximum** properties, or the TextCounter’s **Minimum** and **Maximum** properties.
7. If you want to use **Milestone1** and **Milestone2**, assign them. Remember that they are the number of characters or words before the maximum is reached. *Note: Milestone2 defaults to 10. To turn it off, set it to 0.*
8. Edit any of the messages and style sheets. See “[Setting the Text and Style Sheets](#)”.
9. For any other property, see “[Properties of the TextCounter Control](#)”.
10. Here are some other considerations:
 - If you are using an AJAX system to update this control, set the **InAJAXUpdate** property to true. Also make sure the PageManager control or AJAXManager object has been setup for AJAX. See “Using these Controls With AJAX” in the **General Features Guide**. *Failure to follow these directions can result in incorrect behavior and javascript errors.*
 - This control does not preserve most of its properties in the ViewState, to limit its impact on the page. If you need to use the ViewState to retain the value of a property, see “The ViewState and Preserving Properties forPostBack” in the **General Features Guide**.
 - If you encounter errors, see the “[Troubleshooting](#)” section for extensive topics based on several years of tech support’s experience with customers.
 - See also “[Additional Topics for Using These Controls](#)”.

Properties of the TextCounter Control

For an overview, see “[Using the TextCounter Control](#)”.

Click on any of these topics to jump to them:

- ◆ [TextBox Properties](#)
- ◆ [Message Properties](#)
- ◆ [Appearance Properties](#)
- ◆ [Behavior Properties](#)

TextBox Properties

The Properties Editor shows these properties in the TextBox category.

- **TextBoxControlID** (string) – The ID to the TextBox control whose text will be evaluated. Supports the `System.Web.UI.WebControls.TextBox` and its subclasses, including DES’s textboxes. It also supports controls registered in the `<ThirdPartyControls>` section of the **custom.des.config** file whose `sameas=` attribute is `textbox`.

When the TextBox control is not in the same naming container, assign the control reference programmatically to the **TextBoxControl** property (below).

- **TextBoxControl** (Control) – A reference to the TextBox control attached to this TextCounter control. It is an alternative to **TextBoxControlID** that allows the control to be anywhere on the page instead of the same naming container as the TextCounter control. You must assign it programmatically.

When assigned, it overrides the value of **TextBoxControlID**.

- **Minimum** (int) – The minimum number of characters or words required. The page should not be saved when the text is below this limit.

When below this number, the `TextLengthValidator` or `WordCountValidator` will report an error (based on its own **Minimum** property).

When below, the control displays the text from the **BelowMinimumMessage** property and uses the style sheet class defined in **BelowMinimumCssClass**.

When reached, the control changes the text to **NormalMessage**.

When 0, it first looks for a **Minimum** property on `TextLengthValidator` or `WordCountValidator` assigned to the textbox. If not found, no minimum is used.

It defaults to 0.

- **Maximum** (int) – The maximum number of characters or words permitted. The page should not be saved when the text is above this limit. This is a boundary but it doesn't stop the typing. It switches the text displayed.

When above this number, the `TextLengthValidator` or `WordCountValidator` reports an error. When reached, the control changes the text it displays to the **AboveMaximumMessage** and uses the style sheet class defined in **AboveMaximumCssClass**.

When 0, it first looks for a **Maximum** property on `TextLengthValidator` or `WordCountValidator` assigned to the textbox. If not found, it looks at the **MaxLength** property on the `TextBox` (even if the textbox is `MultiLine`).

If both sources also are 0, then no maximum is used. The text displayed will come from **NoMaximumMessage** instead of **NormalMessage**.

It defaults to 0.

- **Milestone1** (int) – The number of characters or words before **Maximum** when **Milestone1** is used. When **Maximum** minus **Milestone1** is reached, **Milestone1Message** is displayed and the style sheet class in **Milestone1CssClass** is applied. For example, when **Maximum** is 100 and you want the **Milestone1** to occur at the 80th character, use 20 (100 – 80) in **Milestone1**.

There are two milestones prior to **Maximum**. When each is hit, its own is displayed, replacing the current message and style sheet class. This provides a way to escalate the message as the text count nears the limit.

When 0, milestone1 is not used.

It defaults to 0.

- **Milestone2** (int) – The number of characters or words before **Maximum** when **Milestone2** is used. When **Maximum** minus **Milestone2** is reached, **Milestone2Message** is displayed and the style sheet class in **Milestone2CssClass** is applied. For example, when **Maximum** is 100 and you want the **Milestone2** to occur at the 90th character, use 10 (100 – 90) in **Milestone2**.

There are two milestones prior to **Maximum**. When each is hit, its own is displayed, replacing the current message and style sheet class. This provides a way to escalate the message as the text count nears the limit.

When 0, milestone2 is not used.

It defaults to 10.

- **CountType** (enum PeterBlum.DES.CountType) - Determines if it counts by characters or words.

The enumerated type `PeterBlum.DES.CountType` has these values:

- Characters
- Words

It defaults to `CountType.Characters`.

Note: Messages have terms like “character” and “word” embedded. As you switch this property from Characters to Words, it automatically switches “character” to “word”. It does the reverse when you switch from Words to Characters.

Message Properties

The Properties Editor shows these properties in the Messages category.

- **NoLimitMessage** (string) - The message when there is no maximum. If a maximum is used, **NormalMessage** and several others below are used.

*Note: By having **NoLimitMessage** and **NormalMessage** properties defined, the TextCounter can have default text for two cases without forcing you to edit a property.*

The style sheet class used for this message is assigned to **CssClass**, which defaults to “DES_TCCNormal” in the **DES\Appearance\Interactive Pages\TextCounter.css** file.

Use tokens to display dynamic information. See “[Tokens in Messages](#)”.

If “”, it will still show the count.

It defaults to “{COUNT} {COUNT:character:characters}”.

- **NoLimitMessageLookupID** (string) – Gets the value for **NoLimitMessage** through the String Lookup System. (See “String Lookup System” in the **General Features Guide**.) The LookupID and its value should be defined within the String Group of **TextCounter**. If no match is found OR this is blank, **NoLimitMessage** will be used.

The String Lookup System lets you define a common set of terms so the programmer doesn't uniquely define them each time. It also provides localization based on the current culture.

To use it, define a LookupID and associated textual value in your data source (resource, database, etc). Assign the same LookupID to this property.

It defaults to “”.

- **NormalMessage** (string) - The message when the count is between the **Minimum** and **Maximum**. If any milestones are used, it must be less than the milestone. If there is no maximum, **NoLimitMessage** is used instead.

*Note: By having **NoLimitMessage** and **NormalMessage** properties defined, the TextCounter can have default text for two cases without forcing you to edit a property.*

The style sheet class used for this message is assigned to **CssClass**, which defaults to “DES_TCCNormal” in the **DES\Appearance\Interactive Pages\TextCounter.css** file.

Use tokens to display dynamic information. See “[Tokens in Messages](#)”.

If “”, it will still show the count.

It defaults to “{COUNT} of {MAXIMUM} characters”.

- **NormalMessageLookupID** (string) – Gets the value for **NormalMessage** through the String Lookup System. (See “String Lookup System” in the **General Features Guide**.) The LookupID and its value should be defined within the String Group of **TextCounter**. If no match is found OR this is blank, **NormalMessage** will be used.

The String Lookup System lets you define a common set of terms so the programmer doesn't uniquely define them each time. It also provides localization based on the current culture.

To use it, define a LookupID and associated textual value in your data source (resource, database, etc). Assign the same LookupID to this property.

It defaults to “”.

- **Milestone1Message** (string) - The message when the count reaches **Milestone1**.

The style sheet class used for this message is assigned to **Milestone1CssClass**, which defaults to “DES_TCCMilestone1” in the **DES\Appearance\Interactive Pages\TextCounter.css** file.

Use tokens to display dynamic information. See “[Tokens in Messages](#)”.

If “”, it uses the text from **NormalMessage**.

It defaults to “{COUNT} {COUNT:character:characters} - {NEARNESS} {NEARNESS:character remains:characters remain}”.

- Milestone1MessageLookupID** (string) – Gets the value for **Milestone1Message** through the String Lookup System. (See “String Lookup System” in the **General Features Guide**.) The LookupID and its value should be defined within the String Group of **TextCounter**. If no match is found OR this is blank, **Milestone1Message** will be used.

The String Lookup System lets you define a common set of terms so the programmer doesn't uniquely define them each time. It also provides localization based on the current culture.

To use it, define a LookupID and associated textual value in your data source (resource, database, etc). Assign the same LookupID to this property.

It defaults to "".
- Milestone2Message** (string) - The message when the count reaches **Milestone2**.

The style sheet class used for this message is assigned to **Milestone2CssClass**, which defaults to “DES_TCCmilestone2” in the **DES\Appearance\Interactive Pages\TextCounter.css** file.

Use tokens to display dynamic information. See “[Tokens in Messages](#)”.

If "", it uses the text from **Milestone1Message** or if that is blank, **NormalMessage**.

It defaults to “{COUNT} {COUNT:character:characters} - {NEARNESS} {NEARNESS:character remains:characters remain}”.
- Milestone2MessageLookupID** (string) – Gets the value for **Milestone2Message** through the String Lookup System. (See “String Lookup System” in the **General Features Guide**.) The LookupID and its value should be defined within the String Group of **TextCounter**. If no match is found OR this is blank, **Milestone2Message** will be used.

The String Lookup System lets you define a common set of terms so the programmer doesn't uniquely define them each time. It also provides localization based on the current culture.

To use it, define a LookupID and associated textual value in your data source (resource, database, etc). Assign the same LookupID to this property.

It defaults to "".
- AboveMaximumMessage** (string) - The message when the count exceeds **Maximum**.

The style sheet class used for this message is assigned to **AboveMaximumCssClass**, which defaults to “DES_TCCAboveMax” in the **DES\Appearance\Interactive Pages\TextCounter.css** file.

Use tokens to display dynamic information. See “[Tokens in Messages](#)”.

If "", it uses the text from **NormalMessage**.

It defaults to “{COUNT} {COUNT:character:characters} - Exceeded by {NEARNESS} {NEARNESS:character:characters}”.
- AboveMaximumMessageLookupID** (string) – Gets the value for **AboveMaximumMessage** through the String Lookup System. (See “String Lookup System” in the **General Features Guide**.) The LookupID and its value should be defined within the String Group of **TextCounter**. If no match is found OR this is blank, **AboveMaximumMessage** will be used.

The String Lookup System lets you define a common set of terms so the programmer doesn't uniquely define them each time. It also provides localization based on the current culture.

To use it, define a LookupID and associated textual value in your data source (resource, database, etc). Assign the same LookupID to this property.

It defaults to "".

- **BelowMinimumMessage** (string) - The message when the count is below **Minimum**.

The style sheet class used for this message is assigned to **BelowMinimumCssClass**, which defaults to "DES_TCCBelowMin" in the **DES\Appearance\Interactive Pages\TextCounter.css** file.

Use tokens to display dynamic information. See "[Tokens in Messages](#)".

If "", it uses the text from **NormalMessage**.

It defaults to "{COUNT} {COUNT:character:characters} - Requires at least {MINIMUM}".

- **BelowMinimumMessageLookupID** (string) – Gets the value for **BelowMinimumMessage** through the String Lookup System. (See "String Lookup System" in the **General Features Guide**.) The LookupID and its value should be defined within the String Group of **TextCounter**. If no match is found OR this is blank, **BelowMinimumMessage** will be used.

The String Lookup System lets you define a common set of terms so the programmer doesn't uniquely define them each time. It also provides localization based on the current culture.

To use it, define a LookupID and associated textual value in your data source (resource, database, etc). Assign the same LookupID to this property.

It defaults to "".

Appearance Properties

The Properties Editor shows these properties in the Appearance category.

- **DisplayMode** (enum `PeterBlum.DES.Web.WebControls.TextCounterDisplayMode`) – Determines where the messages are shown. They can appear within this control and in the textbox's Hint feature.

The enumerated type `PeterBlum.DES.Web.WebControls.TextCounterDisplayMode` has these values:

- `Here` - Use the location of the `TextCounter`.
- `Hint` - Use the textbox's Hint feature if it is set up. It will append its message to the **Hint** text. Because it appends, you may want some separator between the Hint and the textcounter's message.
- `Both` - Use both the `TextCounter` and Hint feature.

The default value is `TextCounterDisplayMode.Here`.

When **DisplayMode** is `Hint` or `Both`, use the **HintManager.HintShowTextCounterSeparator** to define the HTML that separates the hint text from the `TextCounter` message. It defaults to “`
`”. If you want the `TextCounter` message to appear first, use the token “`{~}`” as the first element of the **HintManager.HintShowTextCounterSeparator** property. The `HintManager` is available on the `PageManager` control and on the **PeterBlum.DES.Globals.WebFormDirector** object.

- **CssClass** (string) – The Cascading Style Sheet applied when text count is between the **Minimum** and **Maximum**, and has not reached any milestones.

The style applies to the entire message. If you want the `{COUNT}` and `{NEARNESS}` tokens to be a different style, use the **NormalTokenCssClass** property, which defaults to “`DES_TCCNormalToken`”.

It defaults to “`DES_TCCNormal`”.

This style is declared in **DES\Appearance\Interactive Pages\TextCounter.css**.

```
.DES_TCCNormal
{
}
```

- **NormalTokenCssClass** (string) – The Cascading Style Sheet applied to the `{COUNT}` and `{NEARNESS}` tokens when text count is between the **Minimum** and **Maximum**, and has not reached any milestones.

To apply a style to the overall control, use the **CssClass** property.

It defaults to “`DES_TCCNormalToken`”.

This style is declared in **DES\Appearance\Interactive Pages\TextCounter.css**.

```
.DES_TCCNormalToken
{
}
```

- **Milestone1CssClass** (string) – The Cascading Style Sheet applied when text count has reached **Milestone1**.

The style applies to the entire message. If you want the `{COUNT}` and `{NEARNESS}` tokens to be a different style, use the **Milestone1TokenCssClass** property, which defaults to “`DES_TCCMilestone1Token`”.

It defaults to “`DES_TCCMilestone1`”.

This style is declared in **DES\Appearance\Interactive Pages\TextCounter.css**.

```
.DES_TCCMilestone1
{
}
```

- **Milestone1TokenCssClass** (string) – The Cascading Style Sheet applied to the `{COUNT}` and `{NEARNESS}` tokens when text count has reached **Milestone1**.

To apply a style to the overall control, use the **Milestone1CssClass** property.

It defaults to “DES_TCCMilestone1Token”.

This style is declared in **DES\Appearance\Interactive Pages\TextCounter.css**.

```
.DES_TCCMilestone1Token
{
}
```

- **Milestone2CssClass** (string) – The Cascading Style Sheet applied when text count has reached Milestone2.

The style applies to the entire message. If you want the {COUNT} and {NEARNESS} tokens to be a different style, use the **Milestone2TokenCssClass** property, which defaults to “DES_TCCMilestone2Token”.

It defaults to “DES_TCCMilestone2”.

This style is declared in **DES\Appearance\Interactive Pages\TextCounter.css**.

```
.DES_TCCMilestone2
{
}
```

- **Milestone2TokenCssClass** (string) – The Cascading Style Sheet applied to the {COUNT} and {NEARNESS} tokens when text count has reached **Milestone2**.

To apply a style to the overall control, use the **Milestone2CssClass** property.

It defaults to “DES_TCCMilestone2Token”.

This style is declared in **DES\Appearance\Interactive Pages\TextCounter.css**.

```
.DES_TCCMilestone2Token
{
    color: Red;
}
```

- **AboveMaximumCssClass** (string) – The Cascading Style Sheet applied when text count has exceeded the **Maximum**.

The style applies to the entire message. If you want the {COUNT} and {NEARNESS} tokens to be a different style, use the **AboveMaximumTokenCssClass** property, which defaults to “DES_TCCAboveMaxToken”.

It defaults to “DES_TCCAboveMax”.

This style is declared in **DES\Appearance\Interactive Pages\TextCounter.css**.

```
.DES_TCCAboveMax
{
    color: Red;
}
```

- **AboveMaximumTokenCssClass** (string) – The Cascading Style Sheet applied to the {COUNT} and {NEARNESS} tokens when text count has exceeded **Maximum**.

To apply a style to the overall control, use the **AboveMaximumCssClass** property.

It defaults to “DES_TCCAboveMaxToken”.

This style is declared in **DES\Appearance\Interactive Pages\TextCounter.css**.

```
.DES_TCCAboveMaxToken
{
    color: Red;
}
```

- **BelowMinimumCssClass** (string) – The Cascading Style Sheet applied when text count is below the **Minimum**.

The style applies to the entire message. If you want the {COUNT} and {NEARNESS} tokens to be a different style, use the **BelowMinimumTokenCssClass** property, which defaults to “DES_TCCBelowMinToken”.

It defaults to “DES_TCCBelowMin”.

This style is declared in **DES\Appearance\Interactive Pages\TextCounter.css**.

```
.DES_TCCBelowMin
{
    color: Red;
}
```

- **BelowMinimumTokenCssClass** (string) – The Cascading Style Sheet applied to the {COUNT} and {NEARNESS} tokens when text count is below the **Minimum**.

To apply a style to the overall control, use the **BelowMinimumCssClass** property.

It defaults to “DES_TCCBelowMinToken”.

This style is declared in **DES\Appearance\Interactive Pages\TextCounter.css**.

```
.DES_TCCBelowMinToken
{
    color: Red;
}
```

- **BackColor, BorderColor, BorderStyle, BorderWidth, Columns, Font, ForeColor, Height, and Style** – These properties are described in [System.Web.UI.WebControls.WebControl Members](#).

Recommendation: Use style sheets class with the **CssClass** property. If any of these properties are applied, they will override the corresponding attribute in any style sheet class used on this control.

Behavior Properties

The Properties Editor shows these properties in the Behavior category.

- **InAJAXUpdate** (Boolean) – When using AJAX on this page, set this to `true` if the control is involved in an AJAX update. See “Using These Controls with AJAX” in the **General Features Guide**. It defaults to `false`.
- **Visible** (Boolean) – When `false`, this control is not used. It defaults to `true`.
- **ViewStateMgr** (PeterBlum.DES.Web.WebControls.ViewStateMgr) – Enhances the ViewState on this control to provide more optimal storage and other benefits. Normally, the properties of this control and its segments are not preserved in the ViewState. When working in ASP.NET markup, define a pipe delimited string of properties in the **PropertiesToTrack** property. When working in code, call `ViewStateMgr.TrackProperty("propertyname")` to save the property. Individual segments have a similar method: `TrackPropertyInViewState("propertyname")`.

For more details, see “The ViewState and Preserving Properties forPostBack” in the **General Features User’s Guide**.

- **PropertiesToTrack** (string) – A pipe delimited list of properties to track. Designed for use in markup and the properties editor. The ViewState is not automatically used by most of these properties. To include a property, add it to this pipe delimited list.

For example, "Group|MayMoveOnClick".

When working programmatically, use `ViewStateMgr.TrackProperty("PropertyName")`.

Context Menu and DropDownMenu Controls

The Context Menu control provides a client-side popup menu that looks like the browser's right-click context menus. It is designed to show a list of commands and optionally their keystroke equivalents. When the mouse passes over a command, the row is highlighted, the way context menus do. When the mouse is clicked on a row, the popup closes and associated JavaScript code is executed. You supply the JavaScript code for each command.

The DropDownMenu control is a button with built-in ContextMenu, providing an easy way to launch a context menu at a specific location on the page.

Click on any of these topics to jump to them:

- ◆ [Features](#)
- ◆ [Using the Context Menu](#)
 - [Overall Appearance](#)
 - [Menu Commands: PeterBlum.DES.Web.WebControls.CommandMenuItem class](#)
 - [Menu Separators: PeterBlum.DES.Web.WebControls.SeparatorMenuItem class](#)
 - [Hint Rows: PeterBlum.DES.Web.WebControls.HintMenuItem class](#)
 - [Popup Controls: PeterBlum.DES.Web.WebControls.MenuActivator class](#)
- ◆ [Using the DropDownMenu control](#)
- ◆ [Adding a ContextMenu](#)
 - [Complete Example](#)
- ◆ [Adding a DropDownMenu](#)
 - [Complete Example](#)
- ◆ [Properties of the ContextMenu](#)
- ◆ [Properties of the DropDownMenu](#)
- ◆ [Online examples](#)

The Context Menu control offers an interesting extension, where you can put non-command information in for hints. The entire menu can consist of hints and work as popup help, or it can be mixed with commands. The hint area can have a different background and font.

This control can popup in several ways:

- You select a list of one or more elements on the page that respond to the mouse click to popup. This allows you to have numerous surfaces, perhaps a group of related fields, all which offer the same commands.
- The browser page can be a popup. This would provide a menu for the entire page.
- You select whether the left, right or either button pops up. The left button causes the context menu to drop down from the associated element clicked. The right button causes the context menu to drop down from the current mouse point. The document.body always pops up from the current mouse point.

It pops down by clicking on any row except those with a separator bar and clicking outside the menu frame.

Features

WARNING: *Not all browsers support the ContextMenu. It requires the browser implements the `oncontextmenu` event. Internet Explorer for Windows, FireFox, Netscape 7+, Safari and Chrome do. Opera does not. If you want a menu to popup from a button or label, use the DropDownMenu. It works with Opera.*

- By default, it takes on the appearance of a standard context menu. (The default style sheets are designed to look much like the Context menu of Windows XP under IE 6.)
- Each “command” row can show a label and keyboard equivalent.
- The keyboard equivalent will actually operate the menu without requiring a popup.
- The command is any JavaScript you want to write. It also provides some built in scripts to show a confirmation message and post back, much like a DES button can.
- It can popup either with a left or right click. Right click is the tradition for a context menu. Left click is excellent for putting a Help button on the page which opens the menu. The left button causes the context menu to drop down from the associated element clicked. The right button causes the context menu to drop down from the current mouse point.
- It can be attached to a single control, a list of controls, or the window. When attached to the window, it overrides the standard browser’s context menu.
- In addition to commands, you can add hints, which are rows of text (or HTML). They don’t fire commands.
- You can also insert menu separators.
- If you establish a maximum height and the menu items exceed that height, it installs a scrollbar to access all commands.
- When you have a license for this module, many controls in the **Peter’s Date and Time** module will offer a context menu with their own commands.

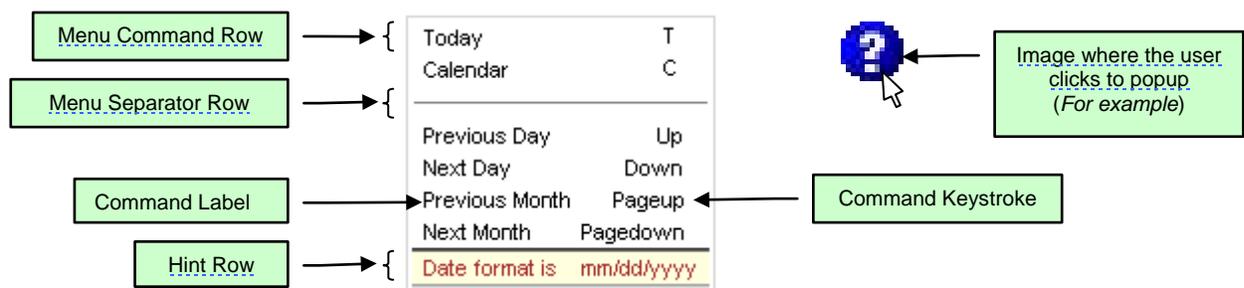
Using the Context Menu

The Context Menu contains rows that show menu commands, separators, and hint rows in its **Items** property. It is attached to controls or the window itself using **MenuActivator** objects in its **MenuActivators** property.

Click on any of these topics to jump to them:

- ◆ [Overall Appearance](#)
 - ◆ [Menu Commands: PeterBlum.DES.Web.WebControls.CommandMenuItem class](#)
 - [Providing a Script for your Command](#)
 - [Appearance of Menu Command Rows](#)
 - [Adding a PeterBlum.DES.Web.WebControls.CommandMenuItem to the ContextMenu](#)
 - [Properties for PeterBlum.DES.Web.WebControls.CommandMenuItem](#)
 - ◆ [Menu Separators: PeterBlum.DES.Web.WebControls.SeparatorMenuItem class](#)
 - [Appearance of Menu Separator Rows](#)
 - [Adding a PeterBlum.DES.Web.WebControls.SeparatorMenuItem to the ContextMenu](#)
 - [Properties for PeterBlum.DES.Web.WebControls.SeparatorMenuItem](#)
 - ◆ [Hint Rows: PeterBlum.DES.Web.WebControls.HintMenuItem class](#)
 - [Appearance of Menu Hint Rows](#)
 - [Adding a PeterBlum.DES.Web.WebControls.HintMenuItem to the ContextMenu](#)
 - [Properties for PeterBlum.DES.Web.WebControls.HintMenuItem](#)
 - ◆ [Popup Controls: PeterBlum.DES.Web.WebControls.MenuActivator class](#)
 - [Inserting Variables Into Your Scripts](#)
 - [Adding a PeterBlum.DES.Web.WebControls.MenuActivator to the ContextMenu](#)
 - [Properties for PeterBlum.DES.Web.WebControls.MenuActivator](#)
-  [Online examples](#)

Here is a sample Context Menu.



Overall Appearance

The appearance is style sheet driven, using style sheet classes defined in `\DES\Appearance\Interactive PagesMenu.css`. This topic discusses the classes for the overall appearance. See other topics for their respective style sheet classes.

Here is the default overall appearance style sheet class. It uses padding attributes to establish the gaps around the menu as traditionally seen in context menus.

```
.DESMenu
{
    background-color: white;
    color: black;
    font-size: 8pt;
    font-family: Arial;
    line-height: normal;
    border-right: #a9a9a9 1px solid; /* dark grey */
    border-top: #a9a9a9 1px solid;
    border-left: #a9a9a9 1px solid;
    border-bottom: #a9a9a9 1px solid;
    padding-top: 2px;
    padding-left: 2px;
    padding-right: 2px;
    padding-bottom: 2px;
}
```

While you can add the width here, you may have several menus throughout your web application requiring differing widths. So set the width in the control's **Width** property. It defaults to 200px.

The menu height is normally a function of the number of menu items you add. If you have a large number of items, set a maximum height in the control's **Height** property. It will establish a scrollbar to allow viewing the commands that exceed the height.

On Internet Explorer, you can take advantage of two visual effects. For a shadow, set the **UseShadowEffect** property to true. To fade in and out as it pops up and down, set the **UsePopupEffect** property to true.

The menu appears relative to either the control that was clicked or to the mouse, depending on whether the using a left button click or right button click. The menu positions itself using the **HorizPosition** and **VertPosition** properties, which default to `RightSidesAlign` and `PopupBelow`, respectively.

Menu Commands: PeterBlum.DES.Web.WebControls.CommandMenuItem class

The primary type of “MenuItem” is a menu command, which provides a label, optional keystroke, and JavaScript to run when the command is selected. The `PeterBlum.DES.Web.WebControls.CommandMenuItem` class describes a single menu command.

Set the label in the **CommandLabel** property of the `CommandMenuItem`.

If you want to use a keystroke, set it in the **CommandKey** property. The keystroke will only work on HTML elements that have focus. They can also override the browser’s menu command keystrokes so make judicious choices.

Each `CommandMenuItem` must have its **CommandID** property assigned to a unique number. This number is used to let you find it programmatically for modification or deletion. It is also used by the script you write to determine the command that invoked the script.

Click on any of these topics to jump to them:

- ◆ [Providing a Script for your Command](#)
 - [OnClickScript Property](#)
 - [ProcessCommandFunctionName property](#)
 - [Validating, Showing A Confirmation Message, and Posting Back](#)
 - [Order of the Actions](#)
- ◆ [Appearance of Menu Command Rows](#)
 - [Command Row](#)
 - [Hiliting on MouseOver](#)
 - [Label](#)
 - [Command Key](#)
- ◆ [Adding a PeterBlum.DES.Web.WebControls.CommandMenuItem to the ContextMenu](#)
- ◆ [Properties for PeterBlum.DES.Web.WebControls.CommandMenuItem](#)

Providing a Script for your Command

Menu commands run JavaScript. You determine the actions of the script. Often your script takes a client-side action like changing the value of a textbox (for example, a Clear command could remove the text of the textbox using the context menu). Sometimes you want it to post back to the server. You may want to validate or show a confirmation message too. The `CommandMenuItem` supports all of this with several properties.

There are two places to write script that takes a client-side action: **OnClickScript** property and **ProcessCommandFunctionName** property.

OnClickScript Property

The **OnClickScript** property of the `CommandMenuItem`. Just add the desired script and it will be run.

For example, this `CommandMenuItem` shows an alert:

```
<des:CommandMenuItem CommandID="10" CommandLabel="Say hello, Ollie"
    OnClickScript="alert('Hello Ollie!');" />
```

It is effective when you encapsolate all data needed by your script into the code. You can embed the **CommandID** and data specific to the control in the script, like this:

```
OnClickScript="MyCmdFunction(10, new Date(2008, 8, 1));"
```

However, there is a better way to call a function and supply it with data, by using the **ProcessCommandFunctionName** property.

ProcessCommandFunctionName property

The **ProcessCommandFunctionName** property on the `ContextMenu` control. Use this to specific a function that is called by all `CommandMenuItems`. The function is passed the **CommandID** so your function can determine the command that invoked it. In its *Args* parameter, it is also passed two values that come from the `MenuActivator` to let you know about the element that opened the context menu. These values are very powerful as they let you have a single context menu shared by many controls, while letting your scripts to know enough about each control to take specialized actions.

This example is a snippet from the DES `DateTextBox`'s context menu function. It assumes the first token is the **id** of the `DateTextBox` so it knows how to update that control's value. The **ProcessCommandFunctionName** property is assigned "DES_DTBMMenuCmd". The script functions are documented in the **Date and Time User's Guide**.

```
function DES_DTBMMenuCmd(pCmdID, pArgs)
{
    var vDTB = DES_GetById(pArgs.Token1);
    switch (pCmdID)
    {
        case 10:    // next day
            DES_DTBAAddDays(vDTB, -1);
            break;
        case 11:    // previous day
            DES_DTBAAddDays(vDTB, 1);
            break;
        case 2:     // today
            DES_DTBTTodayCmd(vDTB);
            break;
    }
    return true;
}
```

See also "[Adding Your JavaScript to the Page](#)".

Validating, Showing A Confirmation Message, and Posting Back

To validate, set **CausesValidation** to `true` and **ValidationGroup** to the appropriate validation group name. These properties are on the `CommandMenuItem` class. *This feature supports both the DES and Native Validation Frameworks.*

To show a confirmation message, set the **ConfirmMessage** property to the text of your message on the `CommandMenuItem` class.

To post back, set the **PostBack** property to `true` on the `CommandMenuItem` class. Use the Context Menu's **MenuSelected** event to process the post back. That event is passed the **CommandID** value so you can determine which command was invoked.

Alternatively, you can have the command point the browser to another URL by setting the **NavigateUrl** property on the `CommandMenuItem` class.

Order of the Actions

With so many ways to set up your scripts, it is important to know the order of the actions.

1. Validate when **CausesValidation**=`true`. If there is a validation error, stop.
2. Show the Confirm Message when **ConfirmMessage** is assigned. If the user clicks Cancel, stop.
3. Run the script in the **OnClickScript** property.
4. Run the script through the function assigned to the **ProcessCommandFunctionName** property. If your function returns `false`, stop.
5. Either post back when **PostBack** is `true` or navigate to another page when **NavigateUrl** is assigned.

Appearance of Menu Command Rows

Style sheet classes determine the appearance and position of the label and command key. They also provide the hilite effect when the mouse is over a menu command. The default classes are defined in `\DES\Appearance\Interactive Pages\Menu.css`.

Command Row

The overall row has a style sheet. It is generally used to establish row height and padding. By default, it uses the class “`DESMenuCommand`” but you can change it in the `CommandRowCssClass` property on the `ContextMenu` control.

Here is the default style sheet:

```
.DESMenuCommand
{
    height: 20px;
    padding-top: 4px;
}
```

Hiliting on MouseOver

To provide a mouse over effect, the `ContextMenu` merges the `DESMenuCommand` class with the `DESMenuMouseOver` class (or whatever is assigned to the `MouseOverCssClass` property on the `ContextMenu` control).

As a result, the `DESMenuMouseOver` class should only contain the changes that occur when the mouse is over, such as the background color.

Here is the default style sheet:

```
.DESMenuMouseOver
{
    background-color: #3366cc;
    color: White;
}
```

Label

The label has it's own style sheet. It should always use the float and position styles to retain the correct positioning. By default, it uses the class “`DESMenuLabel`” but you can change it in the `CommandLabelCssClass` property on the `ContextMenu` control.

Here is the default style sheet:

```
.DESMenuLabel
{
    text-align: left;
    /* float and position here allow varying width labels to be complimented by
    varying width keystrokes. If these two elements overlap, consider changing
    the overall width of the ContextMenu control in its Width property. */
    float: left;
    position: relative;
    left: 16px;
}
```

Command Key

The command key has its own style sheet. It should always use the float and position styles to retain the correct positioning. By default, it uses the class “DESMenuKey” but you can change it in the **CommandKeyCssClass** property on the ContextMenu control.

Here is the default style sheet:

```
.DESMenuKey
{
    text-align: right;
    /* float and position here allow varying width labels to be complimented by
    varying width keystrokes. If these two elements overlap, consider changing
    the overall width of the ContextMenu control in its Width property. */
    float:right;
    position:relative;
    left:-16px;
}
```

Adding a PeterBlum.DES.Web.WebControls.CommandMenuItem to the ContextMenu

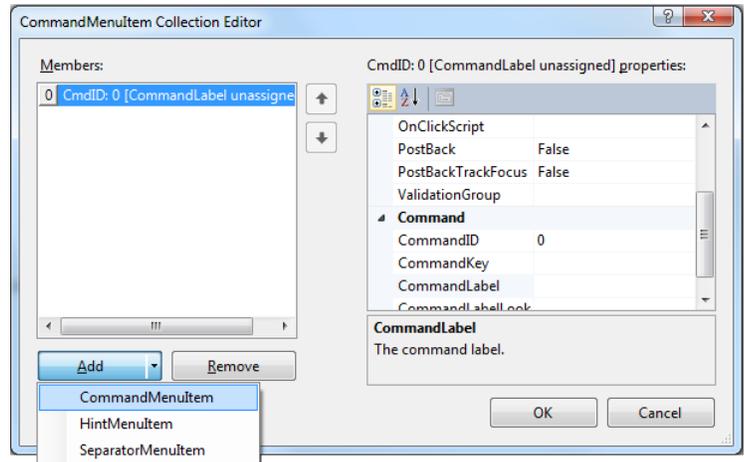
Add CommandMenuItem objects to the ContextMenu's **Items** property. It is a collection type. Items are shown in the order they are added.

Visual Studio and Visual Web Developer Design Mode

In the Properties Editor, click on the button to the right of the **Items** property or in the SmartTag , select **Define Menu Items**.

Click the **Add** button and select **CommandMenuItem**. Fill in the properties.

Always assign **CommandID** and **CommandLabel**.



ASP.NET Markup

Define each CommandMenuItem object as a child object of the Context Menu control. Always supply the **CommandID** and **CommandLabel** properties.

Here is a sample:

```
<des:ContextMenu id="ContextMenu" runat="server" [various properties]>
  <des:CommandMenuItem CommandID="100" CommandLabel="Show Cards"
    OnClickScript="ShowCards();" />
  <des:CommandMenuItem CommandID="101" CommandLabel="Bid $10"
    CommandKey="B" OnClickScript="Bid(10.00);" />
</des:ContextMenu>
```

Programmatically Add a CommandMenuItem Object

To add a Command Row programmatically, calls the method `AddCommand()` on the ContextMenu control. There are 4 overloaded versions. Use the last two if you want to postback or navigate to a URL. Each returns a `PeterBlum.DES.Web.WebControls.CommandMenuItem` object that has already been added as the last element of the **Items** property. You can further edit its properties as needed.

[C#]

```
public PeterBlum.DES.Web.WebControls.CommandMenuItem AddCommand(short pCommandID,
    string pCommandLabel);

public PeterBlum.DES.Web.WebControls.CommandMenuItem AddCommand(short pCommandID,
    string pCommandLabel,
    string pCommandKey, string pOnClickScript,
    bool pCausesValidation, string pValidationGroup,
    string pConfirmMessage);

public PeterBlum.DES.Web.WebControls.CommandMenuItem AddCommand(short pCommandID,
    string pCommandLabel,
    string pCommandKey, string pOnClickScript,
    bool pCausesValidation, string pValidationGroup,
    string pConfirmMessage, bool pPostBack);
```

```
public PeterBlum.DES.Web.WebControls.CommandMenuItem AddCommand(short pCommandID,
    string pCommandLabel,
    string pCommandKey, string pOnClickScript,
    bool pCausesValidation, string pValidationGroup,
    string pConfirmMessage, string pNavigateUrl);
```

[VB]

```
Public Function AddCommand(ByVal pCommandID As System.Int16, _
    ByVal pCommandLabel As String) As PeterBlum.DES.Web.WebControls.CommandMenuItem

Public Function AddCommand(ByVal pCommandID As System.Int16, _
    ByVal pCommandLabel As String, ByVal pCommandKey As String, _
    ByVal pOnClickScript As String, _
    ByVal pCausesValidation As Boolean, ByVal pValidationGroup As String, _
    ByVal pConfirmMessage As String) As PeterBlum.DES.Web.WebControls.CommandMenuItem

Public Function AddCommand(ByVal CommandID As System.Int16, _
    ByVal pCommandLabel As String, ByVal pCommandKey As String, _
    ByVal pOnClickScript As String, _
    ByVal pCausesValidation As Boolean, ByVal pValidationGroup As String, _
    ByVal pConfirmMessage As String, _
    ByVal pPostBack As Boolean) As PeterBlum.DES.Web.WebControls.CommandMenuItem

Public Function AddCommand(ByVal CommandID As System.Int16, _
    ByVal pCommandLabel As String, ByVal pCommandKey As String, _
    ByVal pOnClickScript As String, _
    ByVal pCausesValidation As Boolean, ByVal pValidationGroup As String, _
    ByVal pConfirmMessage As String, _
    ByVal pNavigateUrl As String) As PeterBlum.DES.Web.WebControls.CommandMenuItem
```

Parameters

pCommandID

Assigned to CommandID. Always assign it a unique value.

pCommandLabel

Assigned to the CommandLabel property.

pCommandKey

Assigned to the CommandKey property. If not used, pass "".

pOnClickScript

Assigned to the OnClickScript property. If not used, pass "".

pCausesValidation

Assigned to the CausesValidation property. If not used, pass false.

pValidationGroup

Assigned to the ValidationGroup property. If not used, pass "". To validate all groups, pass "*".

pConfirmMessage

Assigned to the ConfirmMessage property. If not used, pass "".

pPostBack

Assigned to the PostBack property. If not used, pass false.

pNavigateUrl

Assigned to the NavigateUrl property. If not used, pass "".

Return Value

Returns a `PeterBlum.DES.Web.WebControls.CommandMenuItem` object with its properties assigned to the values passed in. It has already been added to the **Items** property at as the last element. You can further edit the properties of the `PeterBlum.DES.Web.WebControls.CommandMenuItem` object as needed.

Example

In this example, the `Page_Load()` method will add the same two commands shown above in the ASP.NET example to the Context Menu whose ID is CM1.

[C#]

```
protected void Page_Load(object sender, System.EventArgs e)
{
    ContextMenu1.AddCommand(100, "Show Cards");
    ContextMenu1.AddCommand(101, "Bid $10",
        "B", "Bid(10.00);", // command key and OnClickScript
        false, "", // validation
        "Do you want to place a $10 bid?"); // confirm message
}
```

[VB]

```
Protected Sub Page_Load(ByVal sender As Object, _
    ByVal e As System.EventArgs)
    ContextMenu1.AddCommand(100, "Show Cards")
    ContextMenu1.AddCommand(101, "Bid $10", _
        "B", "Bid(10.00);", _ ' command key and OnClickScript
        False, "", _ ' validation
        "Do you want to place a $10 bid?" ) ' confirm message
End Sub
```

Properties for PeterBlum.DES.Web.WebControls.CommandMenuItem

- **CommandID** (string) – This is an integer which allows you to uniquely identify a row. You must define a unique value for it. It has several usages:
 - To find and modify it programmatically.
 - When using the context menu's **ProcessCommandFunctionName** property, it tells your script which command was invoked.
 - When using the context menu's **EnableItemFunctionName** property, it tells your script which menu item needs to be enabled or disabled.

- **CommandLabel** (string) – The text to appear in the column for the command label. It is required – it cannot be blank. Lengthy commands may overlap the keystroke forcing you to adjust the value of the context menu's **Width** property.

- **CommandLabelLookupID** (string) – Gets the value for **CommandLabel** through the String Lookup System. (See “String Lookup System” in the **General Features Guide**.) The LookupID and its value should be defined within the String Group of **ContextMenus**. If no match is found OR this is blank, **CommandLabel** will be used.

The String Lookup System lets you define a common set of terms so the programmer doesn't uniquely define them each time. It also provides localization based on the current culture.

To use it, define a LookupID and associated textual value in your data source (resource, database, etc). Assign the same LookupID to this property.

It defaults to "".

- **CommandKey** (string) – The text to appear in the column for the keystroke. When this is a command, it will only show one keystroke and make only the first character uppercase. So it will truncate at the first space. (If it gets “ENTER T D PAGEUP”, it will strip off everything after “Enter”.) When this is a command, uses the exact text that you supply. The string can include the Control key modifier by using the text “CTRL+” before the key. For example, “CTRL+T”.
- *Note: This field displays text representing a key. It does not actually care what the text is and certainly doesn't cause the keys that it contains to do anything when those keys are typed. The Context Menu doesn't actually have any keyboard features of its own.*
- **OnClickScript** (string) – JavaScript code to execute when the command is selected. Only applies when this is a command, not on a hint. It must be a complete JavaScript statement, ending in a semicolon. It should not start with “JavaScript:” as the control adds this text for you.

Your script can signal DES to stop processing the remaining actions of the command with this line of code:

```
vStop = true;
```

- **CausesValidation** (Boolean) – When `true`, the command first attempts to validate all controls whose validation group matches the **ValidationGroup** property. If it succeeds, the remaining actions will proceed.

Use the **ValidationGroup** property to specify the validation group name. It supports "*" to identify all groups. DES and native Validators are supported.

It defaults to `false`.

- **ValidationGroup** (string) – When **CausesValidation** is `true`, this is the validation group name that identifies the validators to fire. It can be "*" to validate all on the page.

It defaults to "".

- **PostBack** (Boolean) – When `true`, the last action taken will be to postback.

The postback will occur after attempting validation (when `CausesValidation=true`), running **OnClickScript**, and running the Menu's **ProcessCommandFunction**.

Use the **Menu.MenuSelected** event to intercept the postback. Its **CommandID** is passed to **MenuSelected**. It helps you know what action occurred.

Overrides **NavigateUrl** if that is also assigned.

It defaults to `false`.

- **PostBackTracksFocus** (Boolean) – When **PostBack=true** and this is **true**, it attempts to keep the focus on the last control with focus prior to posting back.
It defaults to **false**.
- **NavigateUrl** (string) – When assigned, direct the browser to this URL.
If **PostBack=true**, this property is not used. The redirect will occur after attempting validation (when **CausesValidation=true**), running **OnClickScript**, and running the Menu's **ProcessCommandFunction**.
It defaults to "".
- **ConfirmMessage** (string) – When assigned, prompt the user to confirm with this message.
If the user clicks **Cancel**, the process is stopped. If **CausesValidation = true**, validation runs first. The confirm message runs before running any scripts.
It defaults to "".
- **ConfirmMessageLookupID** (string) – Gets the value for **ConfirmMessage** through the String Lookup System. (See “String Lookup System” in the **General Features Guide**.) The **LookupID** and its value should be defined within the String Group of **ConfirmMessages**. If no match is found OR this is blank, **ConfirmMessage** will be used.
The String Lookup System lets you define a common set of terms so the programmer doesn't uniquely define them each time. It also provides localization based on the current culture.
To use it, define a **LookupID** and associated textual value in your data source (resource, database, etc). Assign the same **LookupID** to this property.
It defaults to "".

Menu Separators: PeterBlum.DES.Web.WebControls.SeparatorMenuItem class

A menu separator is a horizontal line between menu commands. To create a menu separator, add a `PeterBlum.DES.Web.WebControls.SeparatorMenuItem` object to the **Items** property on the `ContextMenu`.

Click on any of these topics to jump to them:

- ◆ [Appearance of Menu Separator Rows](#)
- ◆ [Adding a PeterBlum.DES.Web.WebControls.SeparatorMenuItem to the ContextMenu](#)
- ◆ [Properties for PeterBlum.DES.Web.WebControls.SeparatorMenuItem](#)

Appearance of Menu Separator Rows

Style sheet classes determine the appearance of the menu separator. The default classes are defined in `IDES\Appearance\Interactive Pages\Menu.css`.

By default, the `SeparatorMenuItem` uses the class “`DESMenuSeparator`” to define the overall height of the separator line, but you can change it in the `SeparatorCssClass` property on the `ContextMenu` control.

Here is the default style sheet:

```
.DESMenuSeparator
{
    height: 6px;
    font-size: 2pt;
}
```

The horizontal line itself is a second style sheet class that uses the `border` attribute to create the line. It always must declare a second class name of “`Line`” after the same name used in the `SeparatorCssClass` property.

```
.DESMenuSeparator .Line
{
    border-top: #a9a9a9 1px solid;
    width: 100%;
    height: 1px;
    margin-top: 2px;
    margin-bottom: 2px;
}
```

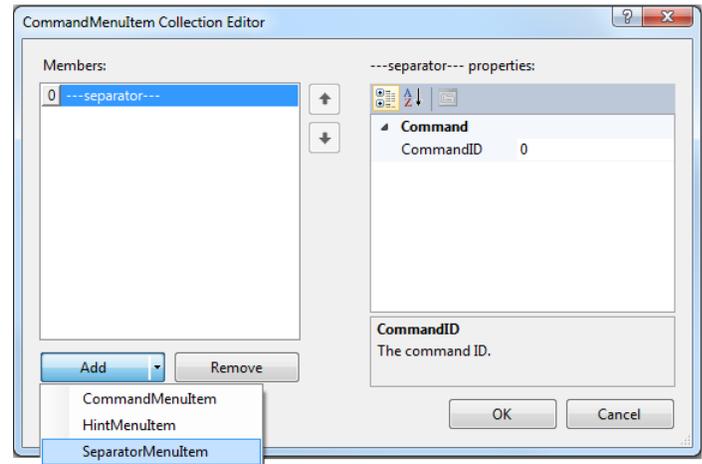
Adding a PeterBlum.DES.Web.WebControls.SeparatorMenuItem to the ContextMenu

Visual Studio and Visual Web Developer Design Mode

In the Properties Editor, click on the button to the right of the **Items** property or in the SmartTag , select **Define Menu Items**.

Click the **Add** button and select **SeparatorMenuItem**.

Only fill in the **CommandID** property if you need to programmatically access this element.



ASP.NET Markup

Define the SeparatorMenuItem as a child of the ContextMenu with no properties.

```
<des:ContextMenu id="ContextMenu1" runat="server" [various properties]>
  <des:SeparatorMenuItem />
</des:ContextMenu>
```

Programmatically Add a SeparatorMenuItem Object

To add a Menu Separator programmatically, calls the method `AddSeparator()` on the ContextMenu control. It will be added to the end of the **Items** property.

[C#]

```
public PeterBlum.DES.Web.WebControls.SeparatorMenuItem AddSeparator();
```

[VB]

```
Public Function AddSeparator() As PeterBlum.DES.Web.WebControls.SeparatorMenuItem
```

Example

[C#]

```
protected void Page_Load(object sender, System.EventArgs e)
{
    ContextMenu1.AddSeparator();
}
```

[VB]

```
Private Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs)
    ContextMenu1.AddSeparator()
End Sub
```

Properties for PeterBlum.DES.Web.WebControls.SeparatorMenuItem

- **CommandID** (string) – This is an integer which allows you to uniquely identify a row. It has several usages:
 - To find and modify it programmatically.
 - When using the context menu's **EnableItemFunctionName** property, it tells your script which menu item needs to be enabled or disabled.

You can leave it unassigned (default of 0) if you do not need it for either of these cases.

Hint Rows: PeterBlum.DES.Web.WebControls.HintMenuItem class

A hint is a row whose text fills the row and is not selectable. It's used for descriptive information.

To create a hint, add a `PeterBlum.DES.Web.WebControls.HintMenuItem` object to the **Items** property on the `ContextMenu`.

Click on any of these topics to jump to them:

- ◆ [Appearance of Menu Hint Rows](#)
- ◆ [Adding a PeterBlum.DES.Web.WebControls.HintMenuItem to the ContextMenu](#)
- ◆ [Properties for PeterBlum.DES.Web.WebControls.HintMenuItem](#)

Appearance of Menu Hint Rows

Style sheet classes determine the appearance of the Hint. The default classes are defined in `\DES\AppearanceInteractive Pages\Menu.css`.

By default, the hint uses the class "DESMenuHint", but you can change it in the `CommandHintCssClass` property on the `ContextMenu` control.

Here is the default style sheet:

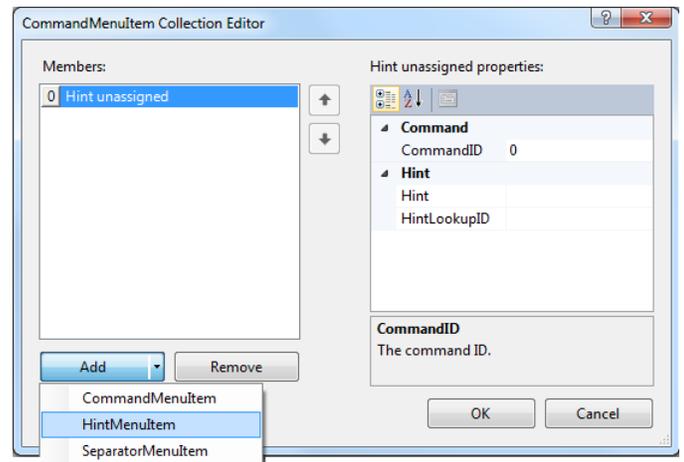
```
.DESMenuHint
{
    color: #a52a2a; /* brown */
    background-color: #ffff99;
    border-right: #d3d3d3 thin inset;
    border-top: #d3d3d3 thin inset;
    border-left: #d3d3d3 thin inset;
    border-bottom: #d3d3d3 thin inset;
    padding-top: 2px;
    padding-left: 2px;
    padding-right: 2px;
    padding-bottom: 2px;
    margin-left: 2px;
    margin-right: 2px;
}
```

Adding a PeterBlum.DES.Web.WebControls.HintMenuItem to the ContextMenu

Visual Studio and Visual Web Developer Design Mode

In the Properties Editor, click on the button to the right of the **Items** property or in the SmartTag , select **Define Menu Items**.

Click the **Add** button and select **HintMenuItem**. Assign its properties including **Hint** for the text of the hint.



ASP.NET Markup

Define the HintMenuItem as a child of the ContextMenu. Assign the **Hint** property to the hint text. Only assign the **CommandID** property if you need to retrieve the HintMenuItem object programmatically.

```
<des:ContextMenu id="ContextMenu1" runat="server" [various properties]>
  <des:HintMenuItem Hint="Maximum bet is $50" />
</des:ContextMenu>
```

Programmatically Add a HintMenuItem Object

To add a hint programmatically, call the method `AddHint()` on the ContextMenu control. It will be added to the end of the **Items** property.

```
[C#]
public PeterBlum.DES.Web.WebControls.HintMenuItem AddHint(string pHint);

[VB]
Public Function AddHint(ByVal pHint As String) _
    As PeterBlum.DES.Web.WebControls.HintMenuItem
```

Parameters

pHint

The text of the hint.

Return value

The `PeterBlum.DES.Web.WebControls.HintMenuItem` object, already assigned to the **Items** property and with the **Hint** property assigned.

Example

```
[C#]
protected void Page_Load(object sender, EventArgs e)
{
    ContextMenu1.AddHint("Maximum bet is $50");
}
```

[VB]

```
Private Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs)
    ContextMenu1.AddHint("Maximum bet is $50")
End Sub
```

Properties for PeterBlum.DES.Web.WebControls.HintMenuItem

- **CommandID** (string) – This is an integer which allows you to uniquely identify a row. It has several usages:
 - To find and modify it programmatically.
 - When using the context menu's **EnableItemFunctionName** property, it tells your script which menu item needs to be enabled or disabled.

You can leave it unassigned (default of 0) if you do not need it for either of these cases.

- **Hint** (string) – The text of the hint. It is required – it cannot be blank.
- **HintLookupID** (string) – Gets the value for **Hint** through the String Lookup System. (See “String Lookup System” in the **General Features Guide**.) The LookupID and its value should be defined within the String Group of **ContextMenus**. If no match is found OR this is blank, **Hint** will be used.

The String Lookup System lets you define a common set of terms so the programmer doesn't uniquely define them each time. It also provides localization based on the current culture.

To use it, define a LookupID and associated textual value in your data source (resource, database, etc). Assign the same LookupID to this property.

It defaults to "".

Popup Controls: PeterBlum.DES.Web.WebControls.MenuActivator class

You must tell the Context Menu which controls popup a context menu by adding `PeterBlum.DES.Web.WebControls.MenuActivator` object to the **MenuActivators** property. You can have as many as you'd like. You also define if it pops up on a left or right mouse button. Left buttons always drop down from the element pressed. Right buttons popup at the current mouse position.

There are two types of elements which Context Menu supports:

- Any document object model (DOM) element with an ID. When you create a page, all WebControls are assigned an ID automatically and these IDs are what the Context Menu needs. If you use an `HtmlControl`, make sure it has an "ID=" parameter.
- The document `.body` DOM element. When the user clicks on this, it always pops up to the current mouse point, even if the left mouse button is clicked.

Click on any of these topics to jump to them:

- ◆ [Inserting Variables Into Your Scripts](#)
- ◆ [Adding a PeterBlum.DES.Web.WebControls.MenuActivator to the ContextMenu](#)
- ◆ [Properties for PeterBlum.DES.Web.WebControls.MenuActivator](#)

Inserting Variables Into Your Scripts

When you write scripts using the **ProcessCommandFunctionName** property on the ContextMenu, you can let each MenuActivator supply up to two variables that your scripts can use. The most common usage is to provide the DHTML element id of the control itself so your scripts can modify that element on the page. For example, DES's DateTextBox provides a context menu with commands to modify the current date shown in the textbox. It supplies the ID of the TextBox element so the context menu can pass it or its DHTML element to functions that support the DateTextBox.

These variables can take several types of information: reference to a control which is converted to the ClientID of that control, string, integer, double, and boolean. If you have other types, consider passing data through a string and writing client-side code that interprets your data.

Variables are set with the **Variable1InScript** and **Variable2InScript** properties. Inside your ProcessCommand function, the Args parameter is an object that contains two properties: Token1 and Token2. They will be in the native type of string (including for Control's ClientID), integer, floating point, or boolean.

You assign these VariableInScript properties programmatically because their data is usually dynamic. With this in mind, you probably will create `PeterBlum.DES.Web.WebControls.MenuActivator` objects programmatically when using variables. For example:

```
ContextMenu1.AddMenuActivator(ImageOnTextBox, PeterBlum.DES.MouseButtonType.Right,
    TextBox1, "Start Date")
```

See "[Adding a PeterBlum.DES.Web.WebControls.MenuActivator to the ContextMenu](#)".

Adding a *PeterBlum.DES.Web.WebControls.MenuActivator* to the *ContextMenu*

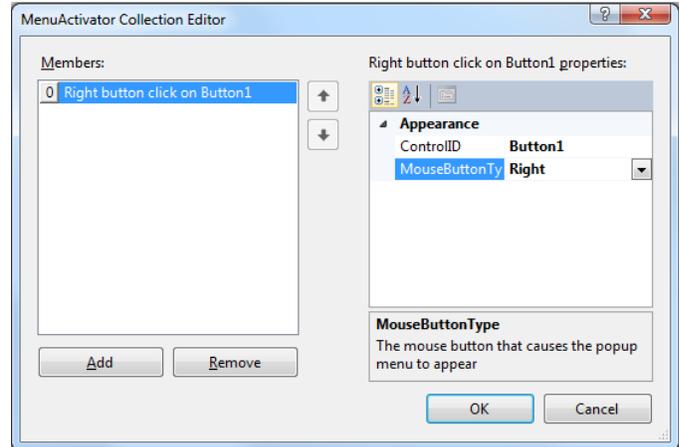
Add the *PeterBlum.DES.Web.WebControls.MenuActivator* objects to the **MenuActivators** property on the *ContextMenu*. If you intend to support the **Variable1InScript** and **Variable2InScript** properties, add *MenuActivator* objects programmatically.

Visual Studio and Visual Web Developer Design Mode

In the Properties Editor, click on the button to the right of the **MenuActivators** property or in the SmartTag , select **Controls that Popup the Menu**.

Click the **Add** button and assign its properties.

If you need to assign **Variable1InScript** and **Variable2InScript** properties, you must do so programmatically. It's probably easiest to create the *MenuActivator* programmatically in that case.



ASP.NET Markup

Add a child tag called “<MenuActivators>” and define the *MenuActivator* as a child of that tag:

```
<des:ContextMenu id="ContextMenu1" runat="server" [various properties]>
  <MenuActivators>
    <des:MenuActivator ControlId="TextBox1" MouseButton="Left" />
    <des:MenuActivator ControlId="" MouseButton="Right" />
  </MenuActivators>
</des:ContextMenu>
```

If you need to assign **Variable1InScript** and **Variable2InScript** properties, you must do so programmatically. It's probably easiest to create the *MenuActivator* programmatically in that case.

Programmatically Add a *HintMenuItem* Object

To add a *PeterBlum.DES.Web.WebControls.MenuActivator* programmatically, call the *AddMenuActivator()* method on the *ContextMenu* control.

[C#]

```
public PeterBlum.DES.Web.WebControls.MenuActivator AddMenuActivator(
    string pControlID, PeterBlum.DES.MouseButtonType pButtonType);
public PeterBlum.DES.Web.WebControls.MenuActivator AddMenuActivator(
    Control pControl, PeterBlum.DES.MouseButtonType pButtonType);
public PeterBlum.DES.Web.WebControls.MenuActivator AddMenuActivator(
    string pControlID, PeterBlum.DES.MouseButtonType pButtonType,
    object pVariable1InScript, object pVariable2InScript);
public PeterBlum.DES.Web.WebControls.MenuActivator AddMenuActivator(
    Control pControl, PeterBlum.DES.MouseButtonType pButtonType,
    object pVariable1InScript, object pVariable2InScript);
```

[VB]

```
Public Function AddMenuActivator(ByVal pControlID As String, _
    ByVal pButtonType As PeterBlum.DES.MouseButtonType) As
PeterBlum.DES.Web.WebControls.MenuActivator

Public Function AddMenuActivator(ByVal pControl As Control, _
    ByVal pButtonType As PeterBlum.DES.MouseButtonType) As
PeterBlum.DES.Web.WebControls.MenuActivator

Public Function AddMenuActivator(ByVal pControlID As String, _
    ByVal pButtonType As PeterBlum.DES.MouseButtonType, _
    ByVal pVariable1InScript As Object, ByVal pVariable2InScript As Object) _
    As PeterBlum.DES.Web.WebControls.MenuActivator

Public Function AddMenuActivator(ByVal pControl As Control, _
    ByVal pButtonType As PeterBlum.DES.MouseButtonType, _
    ByVal pVariable1InScript As Object, ByVal pVariable2InScript As Object) _
    As PeterBlum.DES.Web.WebControls.MenuActivator
```

Parameters*pControlID*Assigned to the **ControlID** property. When using document .body, pass "".*pControl*Assigned to the **ControlInstance** property. When using document .body, pass null.*pButtonType*Assigned to **MouseButtonType** property.*pVariable1InScript*Assigned to the **Variable1InScript** property. Pass null if not used.*pVariable2InScript*Assigned to the **Variable2InScript** property. Pass null if not used.**Return Value**Returns the PeterBlum.DES.Web.WebControls.MenuActivator object that was created. It has already been added to the **MenuActivators** property of the ContextMenu and its properties are assigned to the values passed in.**Example**

In this example, the Page_Load () method will add the same elements shown above in the ASP.NET Markup example.

[C#]

```
protected void Page_Load(object sender, System.EventArgs e)
{
    ContextMenu1.AddMenuActivator("TextBox1", MouseButton.Left);
    ContextMenu1.AddMenuActivator(null, MouseButton.Right);
}
```

[VB]

```
Protected Sub Page_Load(ByVal sender As Object, _
    ByVal e As System.EventArgs)
    ContextMenu1.AddMenuActivator("TextBox1", MouseButton.Left)
    ContextMenu1.AddMenuActivator(Nothing, MouseButton.Right)
End Sub
```

Properties for PeterBlum.DES.Web.WebControls.MenuActivator

- **ControlID** (string) – When a control pops it up, set the ID of the control. The control must be in the same or a parent Naming Container of the Context Menu. When the control is not in the same naming container, assign the control reference programmatically to the **ControlInstance** property (below).

If you want to select `document.body`, leave this unassigned.

- **ControlInstance** (Control) – A reference to the control that pops up the ContextMenu. It is an alternative to **ControlID** that allows the control to be anywhere on the page instead of the same naming container as the ContextMenu control. You must assign it programmatically.

When assigned, it overrides the value of **ControlID**.

- **MouseButtonType** (enum PeterBlum.DES.MouseButtonType) – Which mouse button pops it up: `Left`, `Right`, or `Both`.
- **Variable1InScript** (object) – Provides a value for use by the ProcessCommand function that is unique for this MenuActivator. It is available to your function in the `Args.Token1` property. See “[Inserting Variables Into Your Scripts](#)”.

This is an object type to allow any type passed. However, it’s limited to these types:

Control (will use the `clientID`), string, integer, double, and boolean.

When `null`, nothing is set up.

It defaults to `null`.

- **Variable2InScript** (object) – Provides a value for use by the ProcessCommand function that is unique for this MenuActivator. It is available to your function in the `Args.Token2` property. See “[Inserting Variables Into Your Scripts](#)”.

This is an object type to allow any type passed. However, it’s limited to these types:

Control (will use the `clientID`), string, integer, double, and boolean.

When `null`, nothing is set up.

It defaults to `null`.

Using the DropDownMenu control

The DropDownMenu is a button that has an attached ContextMenu. Most of the setup involves modifying the **Menu** property. It is a full ContextMenu control (aside from not having an ID property and no setup for the **MenuActivators** property is needed.) See “[Using the Context Menu](#)”.

This topic focuses on customizing the button.

Click on any of these topics to jump to them:

- ◆ [Customizing the Toggle button](#)
- ◆ [ContextMenu Overall Appearance](#)
- ◆ [Menu Commands: PeterBlum.DES.Web.WebControls.CommandMenuItem class](#)
- ◆ [Menu Separators: PeterBlum.DES.Web.WebControls.SeparatorMenuItem class](#)
- ◆ [Hint Rows: PeterBlum.DES.Web.WebControls.HintMenuItem class](#)
- ◆ [Online examples](#)

Customizing the Toggle button

When using the DropDownMenu, the user initially sees a toggle button. When clicked, it pops up the ContextMenu. The popup’s toggle can be an image, button, or text. Set it using these properties: **ToggleType**, **ToggleImageUrl**, and **ToggleText**. See “!!!”.

These properties default to use the image ▼ at **\DES\Appearance\Shared\SmallDownArrow.gif** with the style sheet class **DESMenuPopup** from the style sheet file **\DES\Appearance\Interactive Pages\Menu.css**. If you prefer to show a textual label, set **ToggleType** to **ToggleType.Text** and assign your label to **ToggleText**.

You can have it automatically popup when the mouse passes over the button by setting **PopupOnMouseOver** to **true**.

Adding a ContextMenu



These steps ask you to jump around the document using clicks on links. Adobe Reader offers a **Previous View** command to return to the link. Look for this in the Adobe Reader (shown v6.0)

1. Prepare the page for DES controls. See “Preparing a page for DES controls” in the **General Features Guide**. It covers issues like style sheets, AJAX, and localization.
2. Add a ContextMenu control to the page.

Visual Studio and Visual Web Developer Users

Drag the ContextMenu control from the Toolbox onto your web form.

Text Entry Users

Add the control (inside the <form> area):

```
<des:ContextMenu id="[YourControlID]" runat="server" />
```

Programmatically creating the ContextMenu control

- Identify the control which you will add the ContextMenu control to its **Controls** collection. Like all ASP.NET controls, the ContextMenu can be added to any control that supports child controls, like Panel, User Control, or TableCell. If you want to add it directly to the Page, first add a Placeholder at the desired location and use the Placeholder.
- Create an instance of the ContextMenu control class. The constructor takes no parameters.
- Assign the **ID** property.
- Add the ContextMenu control to the **Controls** collection.

In this example, the ContextMenu is created with an **ID** of “ContextMenu1”. It is added to Placeholder1.

[C#]

```
PeterBlum.DES.Web.WebControls.ContextMenu vContextMenu =
    new PeterBlum.DES.Web.WebControls.ContextMenu();
vContextMenu.ID = "ContextMenu1";
Placeholder1.Controls.Add(vContextMenu);
```

[VB]

```
Dim vContextMenu As PeterBlum.DES.Web.WebControls.ContextMenu = _
    New PeterBlum.DES.Web.WebControls.ContextMenu()
vContextMenu.ID = "ContextMenu1"
Placeholder1.Controls.Add(vContextMenu)
```

Guidelines for setting properties

- Design mode users can use the Properties Editor or the Expanded Properties Editor. (See “Expanded Properties Editor” in the **General Features Guide**.) The SmartTag  also offers some of the most important properties.
- Text entry users should add the properties into the <des:ControlClass> tag in this format:
propertyname="value"
- When setting a property programmatically, have a reference to the control’s object and set the property according to your language’s rules.

3. Add menu items to the **Items** property.

- ◆ [Adding a PeterBlum.DES.Web.WebControls.CommandMenuItem to the ContextMenu](#)
- ◆ [Adding a PeterBlum.DES.Web.WebControls.SeparatorMenuItem to the ContextMenu](#)
- ◆ [Adding a PeterBlum.DES.Web.WebControls.HintMenuItem to the ContextMenu](#)

4. Connect the context menu to controls where it will popup when the mouse is clicked. You can specify which mouse button (left or right) and either a web control or the entire document area of the page. See “[Adding a PeterBlum.DES.Web.WebControls.MenuActivator to the ContextMenu](#)”.
5. If desired, set up your ProcessCommand function script on the page and assign its name to the **ProcessCommandFunctionName** property. See “[ProcessCommandFunctionName property](#)”.
6. Customize the overall appearance. See “[Overall Appearance](#)”.
7. Here are some other considerations:
 - If you are using an AJAX system to update this control, set the **InAJAXUpdate** property to `true`. Also make sure the PageManager control or AJAXManager object has been setup for AJAX. See “Using these Controls With AJAX” in the **General Features Guide**. *Failure to follow these directions can result in incorrect behavior and javascript errors.*
 - This control does not preserve most of its properties in the ViewState, to limit its impact on the page. If you need to use the ViewState to retain the value of a property, see “The ViewState and Preserving Properties forPostBack” in the **General Features Guide**.
 - If you encounter errors, see the “[Troubleshooting](#)” section for extensive topics based on several years of tech support’s experience with customers.
 - See also “[Additional Topics for Using These Controls](#)”.

Complete Example

ASP.NET Markup

This example consolidates the previous examples for ASP.NET.

```
<des:ContextMenu id="ContextMenu1" runat="server" [various properties]>
  <des:CommandMenuItem CommandID="100" CommandLabel="Show Cards"
    OnClickScript="ShowCards();" />
  <des:SeparatorMenuItem />
  <des:CommandMenuItem CommandID="101" CommandLabel="Bid $10"
    CommandKey="B" OnClickScript="Bid(10.00);" />
  <des:HintMenuItem Hint="Maximum bet is $50" />
  <MenuActivators>
    <des:MenuActivator ControlID="TextBox1" MouseButton="Left" />
    <des:MenuActivator ControlID="" MouseButton="Right" />
  </MenuActivators>
</des:ContextMenu>
```

Programmatic Example

This example consolidates the previous examples for programming.

[C#]

```
protected void Page_Load(object sender, System.EventArgs e)
{
    ContextMenu1.AddCommand(100, "Show Cards", "", "ShowCards();");
    ContextMenu1.AddCommand(101, "Bid $10", "B", "Bid(10.00);");
    ContextMenu1.AddSeparator();
    ContextMenu1.AddHint("Maximum bet is $50");

    ContextMenu1.AddMenuActivator("MyTextBoxID", MouseButton.Left);
    ContextMenu1.AddMenuActivator("", PeterBlum.DES.MouseButton.Right);
}
```

[VB]

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs)
    ContextMenu1.AddCommand(100, "Show Cards", "", "ShowCards();")
    ContextMenu1.AddCommand(101, "Bid $10", "B", "Bid(10.00);")
    ContextMenu1.AddSeparator()
    ContextMenu1.AddHint("Maximum bet is $50")

    ContextMenu1.AddMenuActivator("MyTextBoxID", PeterBlum.DES.MouseButton.Left)
    ContextMenu1.AddMenuActivator("", PeterBlum.DES.MouseButton.Right)
End Sub
```

Adding a DropDownMenu



These steps ask you to jump around the document using clicks on links. Adobe Reader offers a **Previous View** command to return to the link. Look for this in the Adobe Reader (shown v6.0)

1. Prepare the page for DES controls. See “Preparing a page for DES controls” in the **General Features Guide**. It covers issues like style sheets, AJAX, and localization.
2. Add a DropDownMenu control to the page.

Visual Studio and Visual Web Developer Users

Drag the DropDownMenu control from the Toolbox onto your web form.

Text Entry Users

Add the control (inside the <form> area):

```
<des:DropDownMenu id="[YourControlID]" runat="server" />
```

Programmatically creating the DropDownMenu control

- Identify the control which you will add the DropDownMenu control to its **Controls** collection. Like all ASP.NET controls, the DropDownMenu can be added to any control that supports child controls, like Panel, User Control, or TableCell. If you want to add it directly to the Page, first add a Placeholder at the desired location and use the Placeholder.
- Create an instance of the DropDownMenu control class. The constructor takes no parameters.
- Assign the **ID** property.
- Add the DropDownMenu control to the **Controls** collection.

In this example, the DropDownMenu is created with an **ID** of “DropDownMenu1”. It is added to Placeholder1.

[C#]

```
PeterBlum.DES.Web.WebControls.DropDownMenu vDropDownMenu =
    new PeterBlum.DES.Web.WebControls.DropDownMenu();
vDropDownMenu.ID = "DropDownMenu1";
Placeholder1.Controls.Add(vDropDownMenu);
```

[VB]

```
Dim vDropDownMenu As PeterBlum.DES.Web.WebControls.DropDownMenu = _
    New PeterBlum.DES.Web.WebControls.DropDownMenu()
vDropDownMenu.ID = "DropDownMenu1"
Placeholder1.Controls.Add(vDropDownMenu)
```

Guidelines for setting properties

- Design mode users can use the Properties Editor or the Expanded Properties Editor. (See “Expanded Properties Editor” in the **General Features Guide**.) The SmartTag  also offers some of the most important properties.
- Text entry users should add the properties into the <des:ControlClass> tag in this format:
propertyname="value"
- When setting a property programmatically, have a reference to the control’s object and set the property according to your language’s rules.

3. Set the properties associated with the DropDownMenu such as **ToggleType** and **ToggleImageUrl**. See “!!!”.
4. The **Menu** property contains ContextMenu control. See “[Properties of the ContextMenu](#)”. Add MenuItem objects to **Items** property.

- ◆ [Adding a PeterBlum.DES.Web.WebControls.CommandMenuItem to the ContextMenu](#)
- ◆ [Adding a PeterBlum.DES.Web.WebControls.SeparatorMenuItem to the ContextMenu](#)
- ◆ [Adding a PeterBlum.DES.Web.WebControls.HintMenuItem to the ContextMenu](#)

The ASP.NET markup for these nested controls is shown here (the **Items** property can be omitted).

```
<des:DropDownMenu id="ContextMenu1" runat="server" [various properties]>
  <Menu [various properties]>
    <des:CommandMenuItem [various properties] />
    <des:CommandMenuItem [various properties] />
    <des:CommandMenuItem [various properties] />
  </Menu>
</des:DropDownMenu>
```

5. If desired, set up your ProcessCommand function script on the page and assign its name to the **ProcessCommandFunctionName** property. See “[ProcessCommandFunctionName property](#)”.
6. Customize the overall appearance of the menu. See “[Overall Appearance](#)”.
7. Here are some other considerations:
 - If you are using an AJAX system to update this control, set the **InAJAXUpdate** property to true. Also make sure the PageManager control or AJAXManager object has been setup for AJAX. See “[Using these Controls With AJAX](#)” in the **General Features Guide**. *Failure to follow these directions can result in incorrect behavior and javascript errors.*
 - This control does not preserve most of its properties in the ViewState, to limit its impact on the page. If you need to use the ViewState to retain the value of a property, see “[The ViewState and Preserving Properties forPostBack](#)” in the **General Features Guide**.
 - If you encounter errors, see the “[Troubleshooting](#)” section for extensive topics based on several years of tech support’s experience with customers.
 - See also “[Additional Topics for Using These Controls](#)”.

Complete Example

ASP.NET Markup

This example consolidates the previous examples for ASP.NET.

```
<des:DropDownMenu id="DropDownMenu1" runat="server" [various properties]>
  <Menu>
    <des:CommandMenuItem CommandID="100" CommandLabel="Show Cards"
      OnClickScript="ShowCards();" />
    <des:SeparatorMenuItem />
    <des:CommandMenuItem CommandID="101" CommandLabel="Bid $10"
      CommandKey="B" OnClickScript="Bid(10.00);" />
    <des:HintMenuItem Hint="Maximum bet is $50" />
  </Menu>
</des:DropDownMenu>
```

Programmatic Example

This example consolidates the previous examples for programming.

[C#]

```
protected void Page_Load(object sender, System.EventArgs e)
{
    DropDownMenu1.Menu.AddCommand(100, "Show Cards", "", "ShowCards();");
    DropDownMenu1.Menu.AddCommand(101, "Bid $10", "B", "Bid(10.00);");
    DropDownMenu1.Menu.AddSeparator();
    DropDownMenu1.Menu.AddHint("Maximum bet is $50");

    DropDownMenu1.Menu.AddMenuActivator("MyTextBoxID",
        PeterBlum.DES.MouseButtonType.Left);
    DropDownMenu1.Menu.AddMenuActivator("", PeterBlum.DES.MouseButtonType.Right);
}
```

[VB]

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs)
    DropDownMenu1.Menu.AddCommand(100, "Show Cards", "", "ShowCards();")
    DropDownMenu1.Menu.AddCommand(101, "Bid $10", "B", "Bid(10.00);")
    DropDownMenu1.Menu.AddSeparator()
    DropDownMenu1.Menu.AddHint("Maximum bet is $50")

    DropDownMenu1.Menu.AddMenuActivator("MyTextBoxID", _
        PeterBlum.DES.MouseButtonType.Left)
    DropDownMenu1.Menu.AddMenuActivator("", PeterBlum.DES.MouseButtonType.Right)
End Sub
```

Properties of the ContextMenu

The ContextMenu is subclassed from the Microsoft supplied control Panel (System.Web.UI.WebControls.Panel). See [System.Web.UI.WebControls.Panel Members](#) for any properties inherited from Panel.

Click on any of these topics to jump to them:

- ◆ [Menu Structure Properties](#)
- ◆ [Menu Item Appearance Properties](#)
- ◆ [Overall Appearance Properties](#)
- ◆ [Popup Behavior Properties](#)
- ◆ [Behavior Properties](#)
- ◆ [Popup Location Properties](#)

Properties on other classes. Click on any of these topics to jump to them:

- ◆ [Properties for PeterBlum.DES.Web.WebControls.CommandMenuItem](#)
- ◆ [Properties for PeterBlum.DES.Web.WebControls.SeparatorMenuItem](#)
- ◆ [Properties for PeterBlum.DES.Web.WebControls.HintMenuItem](#)
- ◆ [Properties for PeterBlum.DES.Web.WebControls.MenuActivator](#)

Menu Structure Properties

The Properties Editor lists these properties in the “Menu Structure” category.

- **Items** (PeterBlum.DES.Web.HtmlMenuItems) – A collection of the items that are displayed in the context menu, including menu commands, menu separators, and hints. Add the following objects to it:
 - ◆ [Menu Commands: PeterBlum.DES.Web.WebControls.CommandMenuItem class](#)
 - ◆ [Menu Separators: PeterBlum.DES.Web.WebControls.SeparatorMenuItem class](#)
 - ◆ [Hint Rows: PeterBlum.DES.Web.WebControls.HintMenuItem class](#)
- **MenuActivators** (PeterBlum.DES.MenuActivators) – A list of controls that will popup the context menu. Add PeterBlum.DES.Web.WebControls.MenuActivator objects to it.
See “[Popup Controls: PeterBlum.DES.Web.WebControls.MenuActivator class](#)”.

Menu Item Appearance Properties

The Properties Editor lists these properties in the “Menu Item Appearance” category.

- **CommandRowCssClass** (string) – A style sheet class name applied to rows showing a menu command (PeterBlum.DES.Web.WebControls.CommandMenuItem class).

A row is defined in HTML by a <div> tag assigned to this style sheet class and contains two <div> tags. The left <div> tag displays the command label (**CommandMenuItem.CommandLabel**). The right <div> tag displays the keystroke (**CommandMenuItem.CommandKey**).

Use the style sheet classes DESMenuLabel for the command label and DESMenuKey for the keystroke. To achieve the correct appearance, DESMenuLabel and DESMenuKey use special positioning attributes of float and position:relative. At times, the label and key will overlap due to these attributes. When that happens, increase the width of the control in the **Width** property.

```
<div class="DESMenuCommand">
  <div class="DESMenuLabel">Command Label</div>
  <div class="DESMenuKey">Command Key</div>
</div>
```

It defaults to "DESMenuCommand".

This style is declared in **DES\Appearance\Interactive Pages\Menu.css**.

```
.DESMenuCommand
{
  height:20px;
  padding-top:4px;
}
```

- **CommandLabelCssClass** (string) – The style sheet class name applied to the CommandLabel of the CommandMenuItem. See above for the structure of a Menu Command Row.

It defaults to "DESMenuLabel".

To achieve the correct appearance, DESMenuLabel uses special positioning attributes of float and position:relative. At times, the label and key will overlap due to these attributes. When that happens, increase the width of the control in the **Width** property.

This style is declared in **DES\Appearance\Interactive Pages\Menu.css**.

```
.DESMenuLabel
{
  text-align:left;
  /* float and position here allow varying width labels to be complimented by
  varying width keystrokes. If these two elements overlap, consider changing
  the overall width of the ContextMenu control in its Width property. */
  float:left;
  position:relative;
  left:16px;
}
```

- **CommandKeyCssClass** (string) – The style sheet class name applied to the CommandKey of the CommandMenuItem. See above for the structure of a Menu Command Row.

It defaults to "DESMenuKey".

To achieve the correct appearance, DESMenuKey uses special positioning attributes of float and position:relative. At times, the label and key will overlap due to these attributes. When that happens, increase the width of the control in the **Width** property.

This style is declared in **DES\Appearance\Interactive Pages\Menu.css**.

```
.DESMenuKey
{
    text-align:left;
    /* float and position here allow varying width labels to be complimented by
    varying width keystrokes. If these two elements overlap, consider changing
    the overall width of the ContextMenu control in its Width property. */
    float:right;
    position:relative;
    left:-16px;
}
```

- **CommandHintCssClass** (string) – The style sheet class name applied to hint rows, as defined by PeterBlum.DES.Web.WebControls.HintMenuItem objects. Hints should look different from menu commands. So the default style sheet provides a different font color, background color, and establishes borders.

It defaults to "DESMenuHint".

This style is declared in **DES\Appearance\Interactive Pages\Menu.css**.

```
.DESMenuHint
{
    color: #a52a2a; /* brown */
    background-color: #ffff99;
    border-right: #d3d3d3 thin inset;
    border-top: #d3d3d3 thin inset;
    border-left: #d3d3d3 thin inset;
    border-bottom: #d3d3d3 thin inset;
    padding-top: 2px;
    padding-left: 2px;
    padding-right: 2px;
    padding-bottom: 2px;
    margin-left: 2px;
    margin-right: 2px;
}
```

- **SeparatorCssClass** (string) – The style sheet class name applied to menu separator rows, as defined by PeterBlum.DES.Web.WebControls.SeparatorMenuItem objects.

It defaults to "DESMenuSeparator". This class defines the overall height of the separator line. The horizontal line itself is a second style sheet class that uses the border attribute to create the line. It always must declare a second class name of "Line" after the same name used in the **SeparatorCssClass** property.

For example, when **SeparatorCssClass** = "SeparatorClass":

```
.SeparatorClass .Line
```

These styles are declared in **DES\Appearance\Interactive Pages\Menu.css**.

```
.DESMenuSeparator
{
    height:6px;
    font-size:2pt;
}

.DEsmenuSeparator .Line
{
    border-top: #a9a9a9 1px solid;
    width: 100%;
    height: 1px;
    margin-top: 2px;
    margin-bottom: 2px;
}
```

- **MouseOverCssClass** (string) – The style sheet class name that creates the mouseover effect on Menu Command Rows.

It can be merged with the current style sheet name by putting a + character before the style sheet name in the **MouseOverCssClass** property. When + is before the name, the actual class will be [normalclass] [thisclass].

When there is no leading + character, only this class is applied to the row.

It defaults to "+DESMenuMouseOver". The default style is designed to merge, by only changing the background color of Menu Comand Rows.

This style is declared in **DES\Appearance\Interactive Pages\Menu.css**.

```
.DESMenuMouseOver
{
    background-color: #3366cc;
}
```

Overall Appearance Properties

The Properties Editor lists these properties in the “Overall Appearance” category.

- **CssClass** – The style sheet class for the overall control. Use it to establish font, background, and borders. Its attributes can be further customized on individual rows, by using the styles declared by the properties in “[Menu Item Appearance Properties](#)”.

It defaults to "DESMenu". If you want to establish a shadow effect, set the **UseShadowEffect** property to true.

This style is declared in **DES\Appearance\Interactive Pages\Menu.css**.

```
.DESMenu
{
    background-color: white;
    color: black;
    font-size: 8pt;
    font-family: Arial;
    border-right: #a9a9a9 1px solid; /* dark grey */
    border-top: #a9a9a9 1px solid;
    border-left: #a9a9a9 1px solid;
    border-bottom: #a9a9a9 1px solid;
    padding-top: 2px;
    padding-left: 2px;
    padding-right: 2px;
    padding-bottom: 2px;
}
```

- **Font, BackColor, BackImageUrl, BorderStyle, BorderWidth, BorderColor** and **ForeColor** – These properties are alternatives to using the **CssClass** property. If you have both assigned, these properties override their counterparts in **CssClass**. Recommendation: Use the style sheet class defined in **CssClass**.
- **Width** ([System.Web.UI.WebControls.Unit](#)) – The overall width of the context menu. It should be adjusted based on the largest command label and keystroke, because they may overlap each other if the width is too small. It defaults to 200px.
- **Height** ([System.Web.UI.WebControls.Unit](#)) – When unassigned, the menu establishes its size to fit all menu items that are shown. If you have many menu items, the height may need to be limited to fit on the page. When set, the menu will add a vertical scrollbar so the user can move through all commands. It is unassigned by default.

Popup Behavior Properties

These properties affect how the popup panel pops up and down. The Properties Editor lists these properties in the “Popup Behavior” category.

- **UsePopupEffect** (Boolean) – When `true`, Internet Explorer users will see the context menu fade in as it pops up and fade out as it pops down. This effect can be customized. See “Customizing the Popup Effect” in the **Date and Time User’s Guide**. It defaults to `true`.

Note: On pages that are very large, either in bytes or screen real-estate, this feature can cause popups and popdowns to have a delay. Set this property to false when that is the case.

- **IsPopup** (Boolean) – When menu can popup or be displayed full time as a control on the page. When `true`, it pops up. When `false`, it does not. It defaults to `true`.
- **OnPopup** (string) – Specify a client side JavaScript function that is called when the control is popping up. It is called just prior to making the control visible. Use it to transfer data into the popup. It should not include the heading "javascript:". It should always conclude with a semicolon or end brace as multiple users can append or prefix any code you add with their own code. It defaults to "".
- **OnPopDown** (string) - Specify a client side JavaScript function that is called when the control is popping down. It is called just prior to making the control invisible. Use **OnPopDown** for any cleanup that always happens on pop down. It should not include the heading "javascript:". It should always conclude with a semicolon or end brace as multiple users can append or prefix any code you add with their own code. It defaults to "".
- **IEFixPopupOverList** (Boolean) – Internet Explorer for Windows versions 5.0 through 6. have a problem allowing absolutely positioned objects appearing over ListBox and DropDownLists. There is a special hack that uses an IFrame and filter style sheet to make it appear like it’s over these controls. This property enables that hack on IE versions 5.5-6. (IE 5 doesn’t support the hack; IE 7 doesn’t require the hack.)

The hack is imperfect. It breaks when another IFrame is in the same area of the page. By "breaks", this means the popup usually looks incorrect including being transparent.

Turn off the hack to work around this problem. Set this property to `false`. But you should only do this when the popup does not overlap any listboxes or dropdownlists. If there is overlap, you have to make a design decision to change your positioning or avoid using the IFrame.

When `true`, the hack is used when the browser is Internet Explorer for Windows versions 5.5 through 6..

When `false`, the hack not used. Choose this when the hack causes visual problems such as a transparent popup.

It defaults to `true`.

Behavior Properties

The Properties Editor lists these properties in the “Behavior” category.

- **Visible** (Boolean) – When `false`, the control does not output any HTML. The control is effectively turned off. It defaults to `true`.

When `false`, no HTML is written to the page. If you want to be able to show and hide the control on the client-side, leave this property set to `true` so that all of the HTML is generated. Then use the `FieldStateController` to change the visibility. See the **Interactive Pages User’s Guide** for details on the `FieldStateController`.

- **InAJAXUpdate** (Boolean) – When using AJAX on this page, set this to `true` if the control is involved in an AJAX update. See “Using These Controls with AJAX” in the **General Features Guide**. It defaults to `false`.
- **ProcessCommandFunctionName** (string) – Optional client-side function that is called when a menu item is selected. This function allows standardized code to be in one place that handles multiple commands. For example, the `DateTextBox` predefines a mapping to each command and has a function to handle its commands in its script file. It also allows multiple controls to use the same context menu by supplying two variables from each `PeterBlum.DES.Web.WebControls.MenuActivator` object's **Variable1InScript** and **Variable2InScript** properties. See “[Inserting Variables Into Your Scripts](#)”.

The `ProcessCommand` function takes these parameters:

- **CommandID** - Integer. The command ID that was invoked. Its value comes from the `CommandMenuItem.CommandID`. If the `CommandMenuItem.CommandID` is 0, it never calls your `ProcessCommand` function.
- **Args** – JavaScript object containing the following properties:
 - **MenuID** - string. **ClientID** of the `ContextMenu` control that invoked this function.
 - **TglID** - string. “ToggleID”. **ClientID** of the control that activated the menu. If `null`, `document.body` activated this menu. Use the `DES_GetById()` function if you need to convert `TglID` into its DHTML element.
 - **Token1** - The value from `MenuActivator.Variable1InScript`. It may be `null`.
 - **Token2** - The value from `MenuActivator.Variable2InScript`. It may be `null`.
 - **Src** - The DHTML element associated with the click that opens the menu. It may not be the same as the control associated with the `MenuActivator`, especially because the control contains child HTML elements and the actual HTML element under the mouse pointer is what is returned. You may have to search through the parent elements to find the desired element.

This property will be `null` if the menu was invoked without a mouse click, such as through a keystroke command.

Your function should return `true` to continue running scripts and `false` if no further processing should occur.

Note: Many users make the mistake of assigning JavaScript code to the `ProcessCommandFunctionName` property. This will cause JavaScript errors. GOOD: “`MyFunction`”. BAD: “`MyFunction();`” and “`alert('stop it')`”.

*Note: JavaScript is case sensitive. Be sure the value of **ProcessCommandFunctionName** exactly matches the function definition.*

See also “[Adding Your JavaScript to the Page](#)”.

`PeterBlum.DES.Web.WebControls.CommandMenuItems` will run validation, confirm message, and the **OnClickScript** before calling this function.

EXAMPLES START ON THE NEXT PAGE

Example: Using multiple CommandIDs

This example is a snippet from the DES DateTextBox's context menu function. It assumes **MenuActivator.Variable1InScript** is the **id** of the DateTextBox so it knows how to update that control's value. The **ProcessCommandFunctionName** property is assigned "DES_DTBMenuCmd". The script functions are documented in the **Date and Time User's Guide**.

```
function DES_DTBMenuCmd(pCmdID, pArgs)
{
    var vDTB = DES_GetById(pArgs.Token1); // contains id to textbox
    switch (pCmdID)
    {
        case 10: // next day
            DES_DTBAddDays(vDTB, -1);
            break;
        case 11: // previous day
            DES_DTBAddDays(vDTB, 1);
            break;
        case 2: // today
            DES_DTBTodayCmd(vDTB);
            break;
    }
    return true;
}
```

ANOTHER EXAMPLE IS ON THE NEXT PAGE

Example: Using the Args.Src property

A DES Calendar control needs an “Add Appointment” command for each date in the menu. It will use this instead of the predefined context menu offered by the calendar. It defines the command ID 1000 whose task is to determine the date of the selected cell, write that date as a string to a hidden field and postback. On the server side, the ContextMenu **MenuSelected** event is used to retrieve that string, convert it to a DateTime, and use it.

```
<script type="text/javascript">
function AddAppointment(pCmdID, pArgs)
{
    if (pCmdID != 1000) return false; // error check against invalid commands

    if (pArgs.Src == null) return false;

    // Use pArgs.Src to get the DHTML element that invoked the context menu
    // Convert that to the Date of the date cell. If not found, stop
    // The Calendar assigns a property called "Date" to each Date cell.
    // That is used to detect the DateCell and get the javascript date object
    var vDate = null;
    var vCellRole = null;
    for (var vSE = pArgs.Src; (vSE != document) && (vSE != null);
        vSE = vSE.parentNode)
    {
        if (vSE.Date) // found a Date cell
        {
            vDate = vSE.Date;
            vCellRole = vSE.CellRole;
            break;
        }
    }

    if (vDate == null)
    {
        alert("Please click on a date element of the calendar.");
        return false;
    }

    // vCellRole determines the context of the cell.
    // Values 0-9 indicate a selectable date in the current month.
    // 10 and 11 indicate a selectable date in the previous and next months
    // respectively. All values above 11 are unselectable.
    // 12 = unselectable due to SpecialDates.
    // 14 = out of range MinDate-MaxDate
    if (vCellRole > 11)
    {
        if (vCellRole == 12)
            alert("That date cannot be selected for an appointment.");
        return false;
    }

    var vStorage = DES_GetById("<% =DateSelected.ClientID %>");
    var vFormattedDate = DES_FmtDate2(vDate, "yyyy-MM-dd", 0, null);
    vStorage.value = vFormattedDate; // save for postback to use
    return true;
}
</script>
```

CONTINUED ON THE NEXT PAGE

```
<des:Calendar ID="Calendar1" runat="Server" EnableContextMenu="false" />
<des:ContextMenu ID="ContextMenu" runat="server"
    ProcessCommandFunctionName="AddAppointment"
    OnMenuSelected="ContextMenu_MenuSelected">
    <des:CommandMenuItem CommandID="1000" CommandLabel="Add Appointment"
        PostBack="True" />
    <MenuActivators>
        <des:MenuActivator ControlID="Calendar1" MouseButton="Right" />
    </MenuActivators>
</des:ContextMenu>
<asp:HiddenField ID="DateSelected" runat="server" />
```

[C#]

```
protected void ContextMenu_MenuSelected(object pSender,
    PeterBlum.DES.MenuCommandIDEventArgs pArgs)
{
    if (pArgs.CommandID != 1000) return;

    if (!String.IsNullOrEmpty(DateSelected.Value))
    {
        DateTime vDateSelected;

        if (DateTime.TryParse(DateSelected.Value, out vDateSelected))
        {
            // use the Date
            // Example: Select the date
            Calendar1.SelectedDate = vDateSelected;
        }
    }
}
```

[VB]

```
Protected Sub ContextMenu_MenuSelected(ByVal pSender As Object, _
    ByVal pArgs As PeterBlum.DES.MenuCommandIDEventArgs)

    If pArgs.CommandID != 1000 Return

    If Not String.IsNullOrEmpty(DateSelected.Value) Then
        Dim vDateSelected As DateTime

        If DateTime.TryParse(DateSelected.Value, vDateSelected) Then
            ' use the Date
            ' Example: Select the date
            Calendar1.SelectedDate = vDateSelected
        End If
    End If
End Sub
```

- **EnableItemsFunctionName** (string) - You can add a JavaScript function that makes commands visible or invisible prior to popping up. Use a function when commands vary based on conditions of the page. In this property, define the name of a client-side function that will indicate if a menu command is enabled (actually visible) based on its **CommandID**.

Your function will be called as the context menu is popped up. Every commandID will be passed to your function, one at a time. Your function should evaluate the commandID and return either true to show it or false to hide it.

The function takes two parameters, the **ClientID** of the ContextMenu and the CommandID. It must return true or false.

The value from **MenuActivator.Variable1InScript** is found in the global variable `gDES_Menu.Token1`. The value of **MenuActivator.Variable2InScript** is found in the global variable `gDES_Menu.Token2`. So both are available to your code. See ["Inserting Variables Into Your Scripts"](#).

It defaults to "".

Note: Many users make the mistake of assigning JavaScript code to the EnableItemsFunctionName property. This will cause JavaScript errors. GOOD: "MyFunction". BAD: "MyFunction();" and "alert('stop it')".

Note: JavaScript is case sensitive. Be sure the value of EnableItemsFunctionName exactly matches the function definition.

See also ["Adding Your JavaScript to the Page"](#).

Example

```
function EnableMenu(pMenuID, pCmdID)
{
    // if the context menuID is "ContextMenu1", hide CommandID 20
    if ((pMenuID == "ContextMenu1") && (pCmdID == 20))
        return false;
    return true;
}
```

- **MenuSelected** event – Called on a postback invoked by a [CommandMenuItem](#) that has its **PostBack** property set to true. Its arguments provide the CommandID that was selected. Here is the event handler definition:

[C#]

```
public void MenuCommandID(object pSender,
    PeterBlum.DES.MenuCommandIDEventArgs pArgs);
```

[VB]

```
Public Sub MenuCommandID(object pSender,
    PeterBlum.DES.MenuCommandIDEventArgs pArgs);
```

Parameters

sender

An internal representation of the ContextMenu.

args

The `PeterBlum.DES.MenuCommandIDEventArgs` class provides additional inputs that are useful to your event handler. The **CommandID** property is from the [CommandMenuItem](#) object's **CommandID** property value.

[C#]

```
public class MenuCommandIDEventArgs : System.EventArgs
{ public short CommandID { get; } }
```

[VB]

```
Public Class MenuCommandIDEventArgs Inherits System.EventArgs
    Public ReadOnly Property CommandID As Short
End Class
```

See also ["Example: Using the Args.Src property"](#).

- **ClientSideCreatesHTML** (enum PeterBlum.DES.Web.ClientSideCreatesHTML) – Determines if some of the HTML is created on the client-side. This reduces the size of the HTML output, but may slow down the initialization of the page or the time to first open the popup.

The enumerated type PeterBlum.DES.Web.ClientSideCreatesHTML has these values:

- **Default** – Uses the default from **ClientSideCreatesHTMLPopup** which is set in the **Global Settings Editor**.
- **None** - Fully created by the server and transferred in the page's HTML.
- **EventScripts** - While the HTML will be created on the server side, don't create the embedded DHTML events. Instead, let the client-side set them up. That will reduce the HTML size and put more work on the client-side during initialization, but not as much as BrowserLoads and FirstPopup.
- **BrowserLoads** - As the page is loading into the browser. May cause a slightly longer page initialization.
- **FirstPopup** - When the calendar is first popped up. May cause a delay before the control is popped up.

It defaults to ClientSideCreatesHTML.Default.

*Note: When **PeterBlum.DES.Globals.WebFormDirector.Browser.SupportsClientSideCreatesHTML** is false, it always prepares the HTML on the server side.*

- **ViewStateMgr** (PeterBlum.DES.Web.WebControls.ViewStateMgr) – Enhances the ViewState on this control to provide more optimal storage and other benefits. Normally, the properties of this control and its segments are not preserved in the ViewState. When working in ASP.NET markup, define a pipe delimited string of properties in the **PropertiesToTrack** property. When working in code, call `ViewStateMgr.TrackProperty("propertyname")` to save the property. Individual segments have a similar method: `TrackPropertyInViewState("propertyname")`.

For more details, see “The ViewState and Preserving Properties forPostBack” in the **General Features User’s Guide**.

- **PropertiesToTrack** (string) – A pipe delimited list of properties to track. Designed for use in markup and the properties editor. The ViewState is not automatically used by most of these properties. To include a property, add it to this pipe delimited list.

For example, "Group|MayMoveOnClick".

When working programmatically, use `ViewStateMgr.TrackProperty("PropertyName")`.

Popup Location Properties

Customize how the popup is positioned relative to a control specified in the **MenuActivator.ControlID** property. Only applies on a left mouse click as right mouse clicks are relative to the mouse position.

The Properties Editor lists these properties in the “Popup Location” category.

- **HorizPosition** (enum PeterBlum.DES.HorizPosition) - Positions the popup panel relative to the toggle control. It has these values:
 - `LeftSidesAlign` – Left sides of both controls are flush.
 - `Center` – Objects are centered to each other.
 - `RightSidesAlign` – Right sides of both controls are flush. This is the default.
 - `PopupToRight` – Left side of the popup is flush with the right side of the toggle.
 - `PopupToLeft` – Right side of the popup is flush with the left side of the toggle.
- **HorizPositionOffset** (Int16) – Adjusts the horizontal position of the popup by a number of pixels to allow more precise positioning for **HorizPosition**. If negative, the popup panel moves left. Positive moves right. Zero does nothing. It defaults to 0.
- **VertPosition** (enum PeterBlum.DES.VertPosition) – Positions the popup panel relative to the toggle control. It has these values:
 - `PopupBelow` – Top of the popup is below the toggle. This is the default.
 - `Center` – Objects are centered to each other.
 - `PopupAbove` – Bottom of the popup is above the toggle.
 - `TopSidesAlign` - Tops of both are flush.
- **VertPositionOffset** (Int16) – Adjusts the vertical position of the popup by a number of pixels to allow more precise positioning for **VertPosition**. If negative, the popup panel moves up. Positive moves down. Zero does nothing. It defaults to 0.

Properties of the DropDownMenu

The next sections include properties found directly on the DropDownMenu control.

Click on any of these topics to jump to them:

- ◆ [Toggle Control Properties](#)
- ◆ [Popup Panel Properties](#)
- ◆ [Popup Behavior Properties](#)
- ◆ [Behavior Properties](#)
- ◆ [Properties of the ContextMenu](#)

Toggle Control Properties

The properties of the toggle control are set in the Properties Editor under the “Appearance” and “Toggle Control” categories.

- **ToggleType** (enum PeterBlum.DES.ToggleType) – Determines the appearance of the toggle button.

The enumerated type has these values:

- Text – Uses a label (which is a `` tag). The label’s text is supplied by **ToggleText**. The style sheet class is supplied by **CssClass**. If **ToggleText** contains the “{IMAGE}” token, it is replaced by an `` tag to the **ToggleImageUrl**.
 - Button – Uses an HTML button (which is an `<input type='button'>` tag). Its text is supplied by **ToggleText**.
 - Image – Uses an image (which is an `` tag). The URL is supplied by **ToggleImageUrl**. This is the default.
 - HyperLink – Uses a hyperlink (which is an `<a>` tag). Uses either or both **ToggleText** and **ToggleImageUrl**. Define the text in **ToggleText** and use the “{IMAGE}” token if you want an `` tag with the URL in **ToggleImageUrl**. Examples: “{IMAGE}”, “{IMAGE} ContextMenu”.
- **ToggleText** (string) – The label for the control when **ToggleType** is `ToggleType.Text` or `ToggleType.Button`. It defaults to “Menu”.
 - **ToggleImageUrl** (string) – The URL to the image file shown when **ToggleType** is `ToggleType.Image`. It is also used when **ToggleType**=`Text` or `HyperLink` if **ToggleText** contains the “{IMAGE}” token. That token is replaced by an `` tag to this URL.

It defaults to “{APPEARANCE}/Shared/SmallDownArrow.gif” which is this graphic: ▼.

It uses the style sheet class from **CssClass**. The `` tag gets its value for the `alt=` attribute from **ToggleText**.

Special Symbols for URLs

The “{APPEARANCE}” token will be replaced by the default path to the **Appearance** folder, which you defined as you set up the web site.

Supports the use of the tilde (~) as the first character to be replaced by the virtual path to the web application.

Images for Pressed and MouseOver Effects

You can have images for pressed and mouseover effects as well as the normal image. The names of the image files determine their purpose. Define the name of the normal image. For example, “myimage.gif”. Create the pressed version and give it the same name, with “Pressed” added before the extension. For example, “myimagepressed.gif”. Create the mouseover version and give it the same name, with “MouseOver” added before the extension. For example, myimagemouseover.gif.

The **ToggleImageUrl** property should refer to the normal image. DES will detect the presence of the other two files. If any are missing, DES continues to use the normal image for that case. *Note: Auto detection only works when the URL is a virtual path to a file. You can manage this capability with the [PeterBlum.DES.Globals.WebFormDirector.EnableButtonImageEffects](#).*

If you need more control over paths for pressed and mouseover images, you can embed up to 3 URLs into this property using a pipe (|) delimited list. The order is important: normal |pressed |mouseover. If you want to omit the pressed image, use: normal | |mouseover. If you want to omit the mouseover image, use: normal |pressed.

- **CssClass** (string) – The style name applied to the toggle control.

You can define pressed and mouseover styles by using the same style sheet class name plus the text “Pressed” or “MouseOver”. These styles will merge with the style sheet class defined here. So any properties in the pressed and mouseover classes will override properties in this, but not the entire style.

If blank, it is not used.

To use browser sensitive style sheet class names, start with an ! character. (See “Browser Sensitive Style Sheet Class Names” in the **General Features Guide**.)

It defaults to "DES_MenuPopup" which is declared in **DES\Appearance\Interactive Pages\Menu.css**:

```
.DESMenuPopup
{
    white-space:nowrap;
    text-align:center;
}
.DESMenuPopupPressed
{
    color: #00008b; /* darkblue */
}
.DESMenuPopupMouseOver
{
    color: blue;
}
.DESMenuPopup a
{
    text-decoration: none;
    color : black;
}
.DESMenuPopupPressed a
{
    text-decoration: none;
    color: #00008b; /* darkblue */
}
.DESMenuPopupMouseOver a
{
    text-decoration: none;
    color: blue;
}
.DESMenuPopup img
{
    background-color:transparent;
    margin-left: 0px;
    margin-top: 0px;
    margin-bottom:0px;
    margin-right:0px;
    border: 0pt none;
}
.DESMenuPopupPressed img
{
}
.DESMenuPopupMouseOver img
{
}
```

- **ToggleImageAlign** (enum `ImageAlign`) – The vertical position of the image when **ToggleType** is `Toggle.Image`. While it contains values `ImageAlign.Left` and `ImageAlign.Right`, do not use these. It defaults to `ImageAlign.Top`.
- **MarginLeftAdjustment** (Boolean) – When `true`, add `style="margin-left:##px"` so the toggle is flush with the textbox to its left. When `false`, handle this in your style sheet class.

The actual value of margin-left is defined by

PeterBlum.DES.Globals.WebFormDirector.Browser.ToggleButtonMarginLeftAdjustment.

It defaults to `true`.

Popup Panel Properties

The properties for the Popup Panel which contains the ContextMenu are in the Popup Panel category of the Properties Editor.

- **Menu** (ContextMenu) – This is the ContextMenu control. It contains numerous properties to configure the ContextMenu. See “[Properties of the ContextMenu](#)” for details. To edit its properties in the Properties Editor, click on the button on the **Menu** field.

ASP.NET Representation of Nested Properties

The ASP.NET representation for the nested properties. Here is how to format ASP.NET on a DropDownList control for all of these properties:

```
<des:DropDownMenu id="DropDownMenu1" runat="server" [various properties]>
  <Menu [various properties]>
    </Menu>
</des:DropDownMenu>
```

- **UseShadowEffect** (Boolean) – When true, Internet Explorer for Windows browsers will use a filter effect to give the ContextMenu a shadow. It defaults to true.

If you want to remove the feature globally, set this property in Page_Load () on all pages that use DES controls:

[C#]

```
PeterBlum.DES.Globals.WebFormDirector.Browser.SupportsFilterStyles = false;
```

[VB]

```
PeterBlum.DES.Globals.WebFormDirector.Browser.SupportsFilterStyles = False
```

WARNING: This is the same property that controls the popup and ContextMenu animation effects. By turning it off, you remove those other effects too.

- **HorizPosition** (enum PeterBlum.DES.HorizPosition) - Positions the popup panel relative to the toggle control.

The enumerated type has these values:

- LeftSidesAlign - Left sides of both controls are flush.
 - Center - Objects are centered to each other.
 - RightSidesAlign - Right sides of both controls are flush. This is the default.
 - PopupToRight - Left side of the popup is flush with the right side of the toggle.
 - PopupToLeft – Right side of the popup is flush with the left side of the toggle.
 - **HorizPositionOffset** (Int16) – Adjusts the horizontal position of the popup by a number of pixels to allow more precise positioning for **HorizPosition**. If negative, the popup panel moves left. Positive moves right. Zero does nothing. It defaults to 0.
 - **VertPosition** (enum PeterBlum.DES.VertPosition) – Positions the popup panel relative to the toggle control.
- The enumerated type has these values:
- PopupBelow - Top of the popup is below the toggle. This is the default.
 - Center - Objects are centered to each other.
 - PopupAbove - Bottom of the popup is above the toggle.
 - TopSidesAlign - Tops of both are flush.
 - **VertPositionOffset** (Int16) – Adjusts the vertical position of the popup by a number of pixels to allow more precise positioning for **VertPosition**. If negative, the popup panel moves up. Positive moves down. Zero does nothing. It defaults to 0.

Popup Behavior Properties

These properties affect how the popup panel pops up and down. They are found in the “Behavior” category of the Properties Editor.

- **UsePopupEffect** (Boolean) – When `true`, Internet Explorer users will see the ContextMenu fade in as it pops up and fade out as it pops down. This effect can be customized. See “Customizing the Popup Effect” in the **Date and Time User’s Guide**. It defaults to `true`.

Note: On pages that are very large, either in bytes or screen real-estate, this feature can cause popups and popdowns to have a delay. Set this property to `false` when that is the case.

- **PopupOnMouseOver** (Boolean) – When `true`, the user can point to the DropDownMenu toggle button. After a short delay, the ContextMenu will automatically popup. It defaults to `false`.

The delay is defined in **PopupOnMouseOverDelay**.

- **PopupOnMouseOverDelay** (integer) - When **PopupOnMouseOver** is `true`, this is the time delay between when the mouse moves over the toggle until it pops up. The value is in milliseconds.

If 0, it pops up immediately.

If -1, it uses a global default from **DefaultPopupOnMouseOverDelay**, which defaults to 500 (.5 seconds). Change this default in the **Visual Effects** section of the **Global Settings Editor**.

It defaults to -1.

- **OnPopup** (string) – Specify JavaScript code that is executed when the control is popping up. It is called just prior to making the control visible. Use it to transfer data into the popup. It should not include the heading "javascript:". It should always conclude with a semicolon or end brace as multiple users can append or prefix any code you add with their own code. It defaults to "".
- **OnPopDown** (string) - Specify JavaScript code that is executed when the control is popping down. It is called just prior to making the control invisible.
- **IEFixPopupOverList** (Boolean) – Internet Explorer for Windows versions 5.0 through 6. have a problem allowing absolutely positioned objects appearing over ListBox and DropDownLists. There is a special hack that uses an IFrame and filter style sheet to make it appear like its over these controls. This property enables that hack on IE versions 5.5-6. (IE 5 doesn't support the hack; IE 7 doesn't require the hack.)

The hack is imperfect. It breaks when another IFrame is in the same area of the page. By "breaks", this means the popup usually looks incorrect including being transparent.

If the problem is affecting the ContextMenu, set the **UseShadowEffect** property to `false` on the ContextMenu control.

Turn off the hack to work around this problem. Set this property to `false`. But you should only do this when the popup does not overlap any listboxes or dropdownlists. If there is overlap, you have to make a design decision to change your positioning or avoid using the IFrame.

When `true`, the hack is used when the browser is Internet Explorer for Windows versions 5.5 through 6.

When `false`, the hack not used. Choose this when the hack causes visual problems such as a transparent popup.

It defaults to `true`.

Behavior Properties

- **Visible** (Boolean) – Determines if the control is added to the page at runtime or not. When `false`, it is not added to the page. It defaults to `true`.

When `false`, no HTML is written to the page. If you want to be able to show and hide the control on the client-side, leave this property set to `true` so that all of the HTML is generated. Then use the `FieldStateController` to change the visibility. See the **Interactive Pages User's Guide** for details on the `FieldStateController`.

- **InAJAXUpdate** (Boolean) – When using AJAX on this page, set this to `true` if the control is involved in an AJAX update. See “Using These Controls with AJAX” in the **General Features Guide**. It defaults to `false`.

Enhanced Buttons

The three buttons in ASP.NET – Button, LinkButton, and ImageButton – have been subclassed in DES to extend them in many ways. Some of the features are needed by the DES Validation Framework (and are documented in the Validation User’s Guide). The rest are part of **Peter’s Interactive Pages** and described here.

Click on any of these topics to jump to them:

- ◆ [Features](#)
- ◆ [Using the Enhanced Buttons](#)
- ◆ [Adding an Enhanced Button](#)
- ◆ [Properties on Enhanced Buttons](#)
- ◆ [Programmatically Adding These Features to Non-DES Buttons](#)
 - [The PeterBlum.DES.SubmitBehavior Class](#)

Features

- Use the **ConfirmMessage** property to display a confirmation message. If the user answers No to the prompt, it will prevent the postback. Combined with the [ChangeMonitor](#) and the **ChangeMonitorUsesConfirm** property, you can have the message shown only after an edit occurred.
- Use the **ChangeMonitorEnables** property to determine when the button is enabled as the [ChangeMonitor](#) determines the page has been edited. When setup, the button is disabled as the page is loaded.
- Use the **DisableOnSubmit** property to disable the button after the user clicks, to limit the chance of a double submission.
- Use the **MayMoveOnClick** property when validation is causing the user to click the button twice before it will submit. The button is actually moving after the first click because validation is removing its error messages causing the page to reposition its contents. *This property does not require any license.*
- Built in support for “[Interactive Hints](#)” and “[Enhanced ToolTips](#)”.
- When using the DES Validation Framework, validation groups support special tokens to match to all groups (“*”) and assign group names based on their naming container (“+”). With the **SkipPostBackEventsWhenInvalid** property, they can skip calling your **Click** and **Command** event handler methods if validation errors are detected. See the **Validation User’s Guide** for details.
- ImageButtons can use separate graphic files to provide mouse pressed and mouseover effects. Use the **MultipleImages** property or specify a pipe delimited list of URLs in the **ImageUrl** property.
- ImageButtons will actually dim (using style sheet opacity) when disabled by the [ChangeMonitor](#), **DisableOnSubmit** property, or the [FieldStateController](#).
- LinkButtons normally show the contents of their href= attribute, which is javascript code, in the browser’s status bar. Unless prevented by the browser, DES’s LinkButtons will hide the script from the status bar. If you have a tooltip assigned, its text is used as a replacement.

The DES buttons are direct subclasses of the native buttons, making it very easy to switch to them.

Using the Enhanced Buttons

Start by selecting the Button controls from DES instead of the native controls. When in design mode, the toolbox has them in the Peter's Data Entry Suite tab. When in ASP.NET Markup, type `<des:buttontype>` instead of `<asp:buttontype>`. When writing code, use the namespace `PeterBlum.DES` instead of `System.Web.UI.WebControls`.

If you already have a page setup with the native controls, run the **Web Application Updater** with the option **Convert native controls to their DES equivalents** as described in the **Installation Guide**.

From there, set the desired properties as shown in “[Features](#)” above. See “[Properties on Enhanced Buttons](#)”.

Adding an Enhanced Button



These steps ask you to jump around the document using clicks on links. Adobe Reader offers a **Previous View** command to return to the link. Look for this in the Adobe Reader (shown v6.0)

1. Prepare the page for DES controls. See “Preparing a page for DES controls” in the **General Features Guide**. It covers issues like style sheets, AJAX, and localization.
2. If you have button with existing native button controls, convert them to DES controls by run the **Web Application Updater** with the option **Convert native controls to their DES equivalents** as described in the **Installation Guide**.
3. Add a DES Button, LinkButton, or ImageButton control to the page.

Visual Studio and Visual Web Developer Design Mode Users

Drag the Button, LinkButton, or ImageButton control from the Toolbox onto your web form. *Be sure to select DES’s control, not the native control. Look in the “Peter’s Data Entry Suite” tab.*

Text Entry Users

Add the control (inside the <form> area):

```
<des:Button id="[YourControlID]" runat="server" />
<des:LinkButton id="[YourControlID]" runat="server" />
<des:ImageButton id="[YourControlID]" runat="server" />
```

Programmatically creating the Button control

- Identify the control which you will add the Button, LinkButton, or ImageButton control to its **Controls** collection. Like all ASP.NET controls, the Button can be added to any control that supports child controls, like Panel, UserControl, or TableCell. If you want to add it directly to the Page, first add a Placeholder at the desired location and use the Placeholder.
- Create an instance of the Button, LinkButton, or ImageButton control class. The constructor takes no parameters.
- Assign the **ID** property.
- Add the Button control to the **Controls** collection.

In this example, the Button is created with an **ID** of “Button1”. It is added to Placeholder1.

[C#]

```
PeterBlum.DES.Web.WebControls.Button vButton =
    new PeterBlum.DES.Web.WebControls.Button();
vButton.ID = "Button1";
Placeholder1.Controls.Add(vButton);
```

*Note: The namespace for these controls is PeterBlum.DES. If you prefer, add a **using** clause to that namespace on your form.*

[VB]

```
Dim vButton As PeterBlum.DES.Web.WebControls.Button = _
    New PeterBlum.DES.Web.WebControls.Button()
vButton.ID = "Button1"
Placeholder1.Controls.Add(vButton)
```

*Note: The namespace for these controls is PeterBlum.DES. If you prefer, add an **Imports** clause to that namespace on your form.*

Guidelines for setting properties

- Design mode users can use the Properties Editor or the Expanded Properties Editor. (See “Expanded Properties Editor” in the **General Features Guide**.) The SmartTag  also offers some of the most important properties.
- Text entry users should add the properties into the `<des:ControlClass>` tag in this format:
propertyname="value"
- When setting a property programmatically, have a reference to the control’s object and set the property according to your language’s rules.

4. Set the properties associated with the Button, LinkButton, or ImageButton. See “[Properties on Enhanced Buttons](#)”.
5. If you want a confirmation message, set it in the **ConfirmMessage** property.
6. If you want to disable this button on submit, set **DisableOnSubmit** to `true`.
7. If you are using the ChangeMonitor, review the setting of **ChangeMonitorEnables**. By default, it enables the button after a change only when **CausesValidation** is `true`.
8. If you are using validation, set the desired validation group in the **Group** property. If this button should not validate, set **CausesValidation** to `false`. If the button may move due to a validation error showing or hiding, set **MayMoveOnClick** to `true`.
9. Here are some other considerations:
 - If you are using an AJAX system to update this control, set the **InAJAXUpdate** property to `true`. Also make sure the PageManager control or AJAXManager object has been setup for AJAX. See “Using these Controls With AJAX” in the **General Features Guide**. *Failure to follow these directions can result in incorrect behavior and javascript errors.*
 - If you encounter errors, see the “[Troubleshooting](#)” section for extensive topics based on several years of tech support’s experience with customers.
 - See also “[Additional Topics for Using These Controls](#)”.

Properties on Enhanced Buttons

These controls are subclassed from the native ASP.NET buttons. This section describes properties introduced or important to DES. For the rest, see: [System.Web.UI.WebControls.Button](#), [System.Web.UI.WebControls.LinkButton](#), and [System.Web.UI.WebControls.ImageButton](#).

Click on any of these topics to jump to them:

- ◆ [Behavior Properties](#)
- ◆ [ChangeMonitor Properties](#)
- ◆ [Validation Properties](#)
- ◆ [Hint and ToolTip Properties](#)
- ◆ [Appearance Properties](#)

Behavior Properties

- **ConfirmMessage** (string) – Displays a confirmation message when the button is clicked. It uses the JavaScript function `confirm()` which shows the text of this property and offers OK and Cancel buttons. (You cannot customize the title or buttons.) When the user clicks OK, the page will submit. If they click Cancel, it will not.

When using the DES Validation Framework on this page, it has its own confirmation message in [PeterBlum.DES.Globals.WebFormDirector.ConfirmMessage](#). When assigned, this button's property overrides the other property. You also need to determine if the confirmation message shows prior to validation or after validation reports no errors. Use the [PeterBlum.DES.Globals.WebFormDirector.SubmitOrder](#) property, which by default shows the confirmation message before it attempts to validate.

When using the [ChangeMonitor](#), the confirmation message can appear based on the changed state of the page. Use the [ChangeMonitorUsesConfirm](#) property.

- **ConfirmMessageLookupID** (string) – Gets the value for **ConfirmMessage** through the String Lookup System. (See “String Lookup System” in the **General Features Guide**.) The LookupID and its value should be defined within the String Group of [ConfirmMessages](#). If no match is found OR this is blank, **ConfirmMessage** will be used.

The String Lookup System lets you define a common set of terms so the programmer doesn't uniquely define them each time. It also provides localization based on the current culture.

To use it, define a LookupID and associated textual value in your data source (resource, database, etc). Assign the same LookupID to this property.

It defaults to "".

- **DisableOnSubmit** (Boolean) – When `true`, the control will be disabled after the page submits. If an AJAX callback is used, it disables and re-enables when the callback is completed.

It defaults to `false`.

- **MayMoveOnClick** (Boolean) – If the button requires an extra click to submit the page, its because it jumped as the user clicks on it. Set this to `true` to avoid that extra click.

This solves the following problem:

When the user edits a control and immediately clicks on the button, the `onchange` event of the control fires, running validation. If validation removes error message and the `ValidationSummary`, the button may jump. This happens before the button's `onclick` event, preventing that `onclick` event to run because the mouse button is no longer on the button.

When `true`, the feature is enabled.

It defaults to `false`.

- **InAJAXUpdate** (Boolean) – When using AJAX on this page, set this to `true` if the control is involved in an AJAX update. See “Using These Controls with AJAX” in the **General Features Guide**. It defaults to `false`.
- **Visible** (Boolean) – When `false`, no HTML is output. This control is entirely unused. It defaults to `true`.

ChangeMonitor Properties

- **ChangeMonitorEnables** (enum `PeterBlum.DES.Web.WebControls.ChangeMonitorEnablesSubmitControl`) – Determines if the button switches its state between disabled and enabled. When enabled, the button is disabled as the page is loaded. After the first edit, it becomes enabled.

The enumerated type `PeterBlum.DES.Web.WebControls.ChangeMonitorEnablesSubmitControl` has these values:

- No - The button will not change its enable state.
- Yes - The button will change its enabled state.
- `CausesValidationIsTrue` - When the button's **CausesValidation** property is `true`, it will change its enabled state.
- `CausesValidationIsFalse` - When the button's **CausesValidation** property is `false`, it will change its enabled state.

It defaults to `ChangeMonitorEnablesSubmitControl.CausesValidationIsTrue`.

Note: ImageButtons normally do not have a visual appearance for disabled. DES's ImageButtons change their appearance by changing the opacity of the button when the state is changed by the ChangeMonitor.

- **ChangeMonitorGroups** (string) – When using the ChangeMonitor, the group name(s) defined here is marked changed when the button is edited.

The button's **Group** property is used by the ChangeMonitor unless **ChangeMonitor.UseValidationGroups** is `false`. (**ChangeMonitor** is accessed programmatically through **PeterBlum.DES.Globals.WebFormDirector** and in the **PageManager** control.)

Unless the Group property does not specify the desired group, you can leave this blank.

The ChangeMonitor is enabled when **ChangeMonitor.Enabled** to `True` or the global setting **DefaultChangeMonitorEnabled** is `True` in the **Global Settings Editor**.

The value of "" is a valid group name.

For a list of group names, use the pipe character as a delimiter. For example: "GroupName1|GroupName2". If one of the groups has the name "", start this string with the pipe character: "|GroupName2".

Use "*" to indicate all groups apply.

It defaults to "".

- **ChangeMonitorUsesConfirm** (enum `PeterBlum.DES.Web.WebControls.ChangeMonitorUsesConfirm`) – When the button uses a confirmation message from its **ConfirmMessage** property, it normally displays the message on any click. When using the ChangeMonitor, you can make it display based on the changed state of the page. Use the **ChangeMonitorUsesConfirm** property on the button.

The enumerated type `PeterBlum.DES.Web.WebControls.ChangeMonitorUsesConfirm` has these values:

- No - ChangeMonitor does not affect the confirmation message.
- Changed - Only show the confirmation message if changes were made.
- NotChanged - Only show the confirm message if NO changes were made.

It defaults to `ChangeMonitorUsesConfirm.No`.

Validation Properties

- **Group** (string) – *Only used by the DES Validation Framework.* Group determines which validators are invoked when this button is clicked. Those that match the value in this will be run.

The **ValidationGroup** property, inherited from the base class, also works the same way. If you assign both, **Group** overrides **ValidationGroup**.

Group names are blank by default. When left blank, this runs all validators whose Group property is also blank.

You can also use the string "*" to run every validator on the page.

When the button is shown on multiple rows (naming containers) of a GridView, DataGrid or Repeater, you can make each row have a unique group name by adding a plus (+) character as the first character of the group name.

Note: The pipe character (|) feature is not supported to allow a delimited list of groups. The delimited list feature is only supported on Validators and ValidationSummary controls.

Just be sure to use an identical name in the validators associated with this button.

It defaults to "".

- **CausesValidation** (string) – Determines if the button fires validators. When `true` it does.

It defaults to `true`.

It fires validators first on the client-side. Then again on the server side immediately before calling your **Click** or **Command** event handler method. You should always set up server side validation as follows:

DES Validation Framework

You can have the button never call your **Click** or **Command** event handler by setting

SkipPostBackEventsWhenInvalid property to `TrueFalseDefault.True` or use the global setting **ButtonsSkipPostBackEventsWhenInvalid** in the "Other Validation Properties" topic of the **Global Settings Editor**.

Otherwise, test **PeterBlum.DES.Globals.WebFormDirector.IsValid** is `true` before saving or otherwise using the data from the page.

Native Validation Framework

Test **Page.IsValid** is `true` before saving or otherwise using the data from the page.

- **SkipPostBackEventsWhenInvalid** (enum `PeterBlum.DES.TrueFalseDefault`) – Determines if the button fires its server-side **Click** and **Command** events when there are validation errors detected.

Only applies to buttons that have **CausesValidation** = `true`.

The enumerated type `PeterBlum.DES.TrueFalseDefault` has these values:

- `True` - When **CausesValidation** is `true`, the **Click** and **Command** events are fired only when **IsValid** is `true`. When **CausesValidation**=`false`, the events always fire.
- `False` - The **Click** and **Command** events are always fired on postback.
- `Default` - Determine the value from the global setting **ButtonsSkipPostBackEventsWhenInvalid**, which defaults to `true`. It is set in the "Other Validation Properties" topic of the **Global Settings Editor**.

It defaults to `TrueFalseDefault.Default`.

Hint and ToolTip Properties

License Note: This feature requires a license for the Peter's Interactive Pages.

For an overview, see "[Interactive Hints](#)".

The Properties Editor shows these properties in the "Hint" category for hints and "Appearance" for ToolTips.

Note: The terms "Hint" and "ToolTip" both describe ways to provide documentation to the user. A Hint displays the message when focus enters the field and is best for data entry controls. A ToolTip displays the message when the mouse points to the control. It can be used on almost any type of control.

- **Hint** (string) – When using the Interactive Hints system, this is the text of the hint.

When blank, if the TextBox is using its **ToolTip** property, the **ToolTip** is used as the text of the hint unless you set the **HintManager.ToolTipAsHints** property to `False`.

HTML tags are permitted. ENTER and LINEFEED characters are not. Use the token "{NEWLINE}" where you need a linefeed.

When the hint is shown in the browser's status bar, HTML tags will automatically be stripped.

It defaults to "".

- **HintLookupID** (string) – Gets the value for **Hint** through the String Lookup System. (See "The String Lookup System" in the **General Features Guide**.) The LookupID and its value should be defined within the String Group of [Hint](#). If no match is found OR this is blank, **Hint** will be used.

The String Lookup System lets you define a common set of terms so the programmer doesn't uniquely define them each time. It also provides localization based on the current culture.

To use it, define a LookupID and associated textual value in your data source (resource, database, etc). Assign the same LookupID to this property.

It defaults to "".

- **HintHelp** (string) – When the Hint uses a PopupView, this provides data for use by the Help Button and other features on the PopupView. Its use depends on the **PopupView.HelpBehavior** property. (The PopupView is determined by the HintFormatter with its **PopupViewName** property.)

The PopupView has an optional Help button. When setup, the user can click it to bring up additional information, such as a new page of help text.

Here is how to use the **HintHelp** based on **PopupView.HelpBehavior**:

- **None** - Do not show a Help Button. The **HintHelp** property is not used.
- **ButtonAppends** - Add the text from **HintHelp** after the existing message. Use **PopupView.AppendHelpSeparator** to separate the two parts. When clicked, the Help button disappears and the message box is redrawn.
- **ButtonReplaces** - Replace the text in the message with the **HintHelp**. When clicked, the Help button disappears and the message box is redrawn.
- **Title** - The text appears in the header as the title. It replaces the **PopupView.HeaderText**. There is no Help Button. If **HintHelp** is blank, **PopupView.HeaderText** is used.
- **Hyperlink** - Provide a Hyperlink. The Help Info text will appear in the "{0}" token of **PopupView.HyperlinkUrlForHelpButton**.

For example, the **HyperlinkUrlForHelpButton** property may be "{0}" and this property is the complete URL `"/helpfiles/helptopic1000.aspx"`.

Another example uses the token for just a querystring parameter, like this: **HyperlinkUrlForHelpButton** = `"/gethelp.aspx?topicid={0}"` and this property contains the number of the ID.

- **HyperlinkNewWindow** - Provide a Hyperlink that opens a new window. The **HintHelp** text will appear in the "{0}" token of **PopupView.HyperlinkUrlForHelpButton**.

- `ButtonRunsScript` - Runs the script supplied in `PopupView.ScriptForHelpButton`. The `HintHelp` text will replace the token "{0}" in that script.

This defaults to "".

- **HintHelpLookupID** (string) – Gets the value for `HintHelp` through the String Lookup System. (See “The String Lookup System” in the **General Features Guide**.) The LookupID and its value should be defined within the String Group of `Hint`. If no match is found OR this is blank, `HintHelp` will be used.

The String Lookup System lets you define a common set of terms so the programmer doesn't uniquely define them each time. It also provides localization based on the current culture.

To use it, define a LookupID and associated textual value in your data source (resource, database, etc). Assign the same LookupID to this property.

It defaults to "".

- **SharedHintFormatterName** (string) – Specify the name of the desired `HintFormatter` object found in `HintManager.SharedHintFormatters`. (`HintManager` is accessed programmatically through `PeterBlum.DES.Globals.WebFormDirector` and in the `PageManager` control.) Alternatively, specify the name of a `PopupView` defined in the “`PopupView` definitions used by `HintFormatters`” of the **Global Settings Editor**.

The `PeterBlum.DES.Web.WebControls.HintFormatter` class describes how the hint text will be displayed. It provides its name, display mode - on the page or in a `PopupView`, if it's also in the tooltip and/or status bar, and more.

The `HintManager.SharedHintFormatters` property defines various ways to display a hint with `PeterBlum.DES.Web.WebControls.HintFormatter` objects. It lets you share a `HintFormatter` definition amongst controls on this page. It not only makes changes to the `HintFormatter` quick, but it also reduces the JavaScript output. If you want to create a `HintFormatter` specific to this control, set `SharedHintFormatterName` to "" and edit the properties of `LocalHintFormatter` (see below).

If you specify the name of a `PopupView` and there is a definition with that name, a `HintFormatter` is automatically added to `HintManager.SharedHintFormatters` with its name matching the name of the `PopupView`. This is an easy way to work with `PopupViews` without the extra step of setting up `HintFormatters`. The `HintFormatter` defined will also show the hint as a tooltip but it will not show the hint in the status bar. If you need more control over the `HintFormatter`'s properties, you must create the `HintFormatter` yourself.

See “[Interactive Hints](#)” for details.

Use the token "{DEFAULT}" to get the name from `HintManager.DefaultSharedHintFormatterName`.

It defaults to "{DEFAULT}".

- **LocalHintFormatter** (`PeterBlum.DES.Web.WebControls.HintFormatter`) – When none of the `HintFormatter` objects defined in `HintManager.SharedHintFormatters` is appropriate, use this property. (`HintManager` is accessed programmatically through `PeterBlum.DES.Globals.WebFormDirector` and in the `PageManager` control.)

The `PeterBlum.DES.Web.WebControls.HintFormatter` class describes how the hint text will be displayed. It provides its display mode - on the page or in a `PopupView`, if it's also in the tooltip and/or status bar, and more. See the “[Interactive Hints](#)” section of the **Interactive Pages User's Guide** for directions on using the `PeterBlum.DES.Web.WebControls.HintFormatter` class.

You must set `SharedHintFormatterName` to "" for this to be used.

- **ToolTip** (string) – When assigned, a tooltip with this text is shown when the user points to the textbox. If you are using the `Hint` feature, it can be used as the hint when the `Hint` property is "". When using the “[Enhanced ToolTips](#)” feature, the browser's tooltip will be replaced by a `PopupView`.
- **ToolTipUsesPopupViewName** (string) – When using the “[Enhanced ToolTips](#)” feature, this determines which `PopupView` definition is used. Specify the name from the `PopupView` definition or use the token "{DEFAULT}" to select the name from the global setting `DefaultToolTipPopupViewName`, which is set with the **Global Settings Editor**.

A `PopupView` definition describes the name, style sheets, images, behaviors, and size of a `PopupView`. Use the **Global Settings Editor** to create and edit these `PopupView` definitions in the “`PopupView` definitions used by the `HintManager`” section.

Tooltips are only converted to PopupViews when **HintManager.EnableToolTipsUsePopupViews** is `True`. (**HintManager** is accessed programmatically through **PeterBlum.DES.Globals.WebFormDirector** and in the **PageManager** control.)

Here are the predefined values: `LtYellow-Small`, `LtYellow-Medium`, `LtYellow-Large`, `ToolTip-Small`, `ToolTip-Medium`, and `ToolTip-Large`. All of these are light yellow. Their widths vary from 200px to 600px. Those named “ToolTip” have the callout feature disabled. Those named “LtYellow” have the callout feature enabled.

It defaults to “{DEFAULT}”.

Note: When the name is unknown, it also uses the factory default. This allows the software to operate even if a `PopupView` definition is deleted or renamed.

Note: When the `HintManager.ToolTipsAsHints` feature is enabled, anything other than “” or “{DEFAULT}” assigned to `ToolTipUsesPopupViewName` will prevent the `ToolTip` text from being assigned as a `Hint`. You must explicitly assign the `Hint` text if you want the tooltip and hint to share the same text.

Appearance Properties

Most of the properties that provide the appearance of the buttons is documented here:

[System.Web.UI.WebControls.Button](#), [System.Web.UI.WebControls.LinkButton](#), and [System.Web.UI.WebControls.ImageButton](#).

The Properties Editor shows these properties in the “Appearance” category.

- **ImageUrl** (string) – *ImageButton only*. The Url to the image file. No initial image is supplied, so always assign it.

Special Symbols for URLs

The “{APPEARANCE}” token will be replaced by the default path to the **Appearance** folder, which you defined as you set up the web site.

Supports the use of the tilde (~) as the first character to be replaced by the virtual path to the web application.

Images for Pressed and MouseOver Effects

License Note: Requires a license for Peter's Interactive Pages.

You can have images for pressed and mouseover effects as well as the normal image. The names of the image files determine their purpose. Define the name of the normal image. For example, “myimage.gif”. Create the pressed version and give it the same name, with “Pressed” added before the extension. For example, “myimagepressed.gif”. Create the mouseover version and give it the same name, with “MouseOver” added before the extension. For example, myimagemouseover.gif.

The **ImageUrl** property should refer to the normal image. Set **MultipleImages** to `true` and DES will detect the presence of the other two files. If any are missing, DES continues to use the normal image for that case. *Note: Auto detection only works when the URL is a virtual path to a file. You can manage this capability with the [PeterBlum.DES.Globals.WebFormDirector.ButtonEffectsManager.EnableButtonImageEffects](#).*

If you need more control over paths for pressed and mouseover images, you can embed up to 3 URLs into this property using a pipe (|) delimited list. The order is important: `normal|pressed|mouseover`. If you want to omit the pressed image, use: `normal||mouseover`. If you want to omit the mouseover image, use: `normal|pressed`.

- **WhenDisabledImageUrl** (string) – *ImageButton only*. The Url to the image file shown when the button is disabled. No initial image is supplied, so always assign it.

License Note: This property requires a license for Peter's Interactive Pages.

- **MultipleImages** (boolean) – *ImageButton only*. When `true`, you have images for pressed and mouseover effects. The **ImageUrl** points to the normal graphic. Pressed and MouseOver are files in the same folder with the "Pressed" and "MouseOver" inserted in name, just before the file extension.

The code will detect the presence of these files. If neither are found, it is not used. If one is found, it is used and the other is not. So it is safe to set this to `true` all of the time, except for the extra overhead of time used.

The [PeterBlum.DES.Globals.WebFormDirector.ButtonEffectsManager.EnableButtonImageEffects](#) property can override this behavior.

When `false`, it does not look for multiple images. However, the **ImageUrl** can override this by having a pipe delimited list of URLs in this format: `Normal|Pressed|MouseOver`.

When `true`, it looks for multiple images.

It defaults to `false`.

License Note: This property requires a license for Peter's Interactive Pages.

- **WhenDisabledCssClass** (string) – The style sheet class used when the button is disabled. When unassigned, it is not used. It defaults to "".

Note: FireFox does not support this for LinkButtons. It does not have a disabled mode for <a> tags.

License Note: This property requires a license for Peter's Interactive Pages.

Programmatically Adding These Features to Non-DES Buttons

You can add some of these features to non-DES buttons in two ways, through the NativeControlExtender or through writing code. This section shows how to write code to update non-DES buttons.

The features supported here are the ConfirmMessage, DisableOnSubmit, MayMoveOnClick, ChangeMonitor, and DES validation. See [“Adding a Hint to any Control Programmatically: PeterBlum.DES.Globals.WebFormDirector.AddHintToControl Method”](#) for using Interactive Hints on your controls.

There are three steps to programmatically connecting these features to your submit control.

1. Create a `PeterBlum.DES.SubmitBehavior` object. The constructor takes a reference to your control. See [“Constructors”](#).
2. Assign the desired properties. See [“Properties”](#).
3. Pass the object to `PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager.RegisterSubmitControl()`.

Example

Adds **DisableOnSubmit** and validation group “group1” to Button1.

[C#]

```
PeterBlum.DES.SubmitBehavior vSubmitBehavior =  
    new PeterBlum.DES.SubmitBehavior(Button1);  
vSubmitBehavior.DisableOnSubmit = true;  
vSubmitBehavior.Group = "group1";  
PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager.RegisterSubmitControl(vSubmitBehavior);
```

[VB]

```
Dim vSubmitBehavior As PeterBlum.DES.SubmitBehavior = _  
    New PeterBlum.DES.SubmitBehavior(Button1)  
vSubmitBehavior.DisableOnSubmit = True  
vSubmitBehavior.Group = "group1"  
PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager.RegisterSubmitControl(vSubmitBehavior)
```

The PeterBlum.DES.SubmitBehavior Class

Properties

- **SubmitControl** (System.Web.UI.WebControls.Control) – The control that is getting the additional functionality. It is always assigned the constructor of this class.
- **Group** (string) – *Only used by the DES Validation Framework.* Group determines which validators are invoked when this button is clicked. Those that match the value in this will be run.

Group names are blank by default. When left blank, this runs all validators whose Group property is also blank.

You can also use the string "*" to run every validator on the page.

When the button is shown on multiple rows (naming containers) of a GridView, DataGrid or Repeater, you can make each row have a unique group name by adding a plus (+) character as the first character of the group name. (This is supported for multiple group names with "+groupname|+groupname2".)

Just be sure to use an identical name in the validators associated with this button.

It defaults to "".

- **CausesValidation** (string) – Determines if the button fires DES Framework Validators on the client side. When true it does.

It defaults to true.

It only validates on the client-side. You still must set up server side validation in your control's post back event handler method like this:

[C#]

```
PeterBlum.DES.Globals.WebFormDirector.Validate("validation group name");
if (PeterBlum.DES.Globals.WebFormDirector.IsValid)
{
    // save or use the page data here
}
```

[VB]

```
PeterBlum.DES.Globals.WebFormDirector.Validate("validation group name")
If PeterBlum.DES.Globals.WebFormDirector.IsValid Then
    ' save or use the page data here
End If
```

If you have no validation group, you can pass "" or call Validate() without any parameter.

- **ConfirmMessage** (string) – *Requires a license covering the Interactive Pages module.* Displays a confirmation message when the button is clicked. It uses the JavaScript function confirm() which shows the text of this property and offers OK and Cancel buttons. (You cannot customize the title or buttons.) When the user clicks OK, the page will submit. If they click Cancel, it will not.

When using the DES Validation Framework on this page, it has its own confirmation message in [PeterBlum.DES.Globals.WebFormDirector.ConfirmMessage](#). When assigned, this button's property overrides the other property. You also need to determine if the confirmation message shows prior to validation or after validation reports no errors. Use the [PeterBlum.DES.Globals.WebFormDirector.SubmitOrder](#) property, which by default shows the confirmation message before it attempts to validate.

When using the [ChangeMonitor](#), the confirmation message can appear based on the changed state of the page. Use the [ChangeMonitorUsesConfirm](#) property.

- **DisableOnSubmit** (Boolean) – *Requires a license covering the Interactive Pages module.* When true, the control will be disabled after the page submits. If an AJAX callback is used, it disables and re-enables when the callback is completed. To disable, DES sets the disabled property to true in the HTML element for the Submit control. If that element is an <input type="image"> or , it changes the opacity of the control to dim it.

It defaults to false.

- **MayMoveOnClick** (Boolean) – If the button requires an extra click to submit the page, it's because it jumped as the user clicks on it. Set this to `true` to avoid that extra click.

This solves the following problem:

When the user edits a control and immediately clicks on the button, the `onchange` event of the control fires, running validation. If validation removes error message and the `ValidationSummary`, the button may jump. This happens before the button's `onclick` event, preventing that `onclick` event to run because the mouse button is no longer on the button.

When `true`, the feature is enabled.

It defaults to `false`.

- **ChangeMonitorEnables** (enum `PeterBlum.DES.Web.WebControls.ChangeMonitorEnablesSubmitControl`) – *Requires a license covering the Interactive Pages module.* Determines if the button switches its state between disabled and enabled. When enabled, the button is disabled as the page is loaded. After the first edit, it becomes enabled.

The enumerated type `PeterBlum.DES.Web.WebControls.ChangeMonitorEnablesSubmitControl` has these values:

- `No` - The button will not change its enable state.
- `Yes` - The button will change its enabled state.
- `CausesValidationIsTrue` - When the button's **CausesValidation** property is `true`, it will change its enabled state.
- `CausesValidationIsFalse` - When the button's **CausesValidation** property is `false`, it will change its enabled state.

It defaults to `ChangeMonitorEnablesSubmitControl.CausesValidationIsTrue`.

Note: ImageButtons normally do not have a visual appearance for disabled. DES's ImageButtons change their appearance by changing the opacity of the button when the state is changed by the ChangeMonitor.

- **ChangeMonitorUsesConfirm** (enum `PeterBlum.DES.Web.WebControls.ChangeMonitorUsesConfirm`) – *Requires a license covering the Interactive Pages module.* When the button uses a confirmation message from its **ConfirmMessage** property, it normally displays the message on any click. When using the `ChangeMonitor`, you can make it display based on the changed state of the page. Use the **ChangeMonitorUsesConfirm** property on the button.

The enumerated type `PeterBlum.DES.Web.WebControls.ChangeMonitorUsesConfirm` has these values:

- `No` - `ChangeMonitor` does not affect the confirmation message.
- `Changed` - Only show the confirmation message if changes were made.
- `NotChanged` - Only show the confirm message if NO changes were made.

It defaults to `ChangeMonitorUsesConfirm.No`.

- **InAJAXUpdate** (Boolean) – When using AJAX on this page, set this to `true` if the control is involved in an AJAX update. See “Using These Controls with AJAX” in the **General Features Guide**. It defaults to `false`.

Constructors

The following constructors have parameters that match various properties shown above.

[C#]

```
SubmitBehavior(Control pSubmitControl)
```

```
SubmitBehavior(Control pSubmitControl, string pValidationGroup)
```

```
SubmitBehavior(Control pSubmitControl, string pValidationGroup,  
string pConfirmMessage)
```

[VB]

```
SubmitBehavior(ByVal pSubmitControl As Control)
```

```
SubmitBehavior(ByVal pSubmitControl As Control, _  
ByVal pValidationGroup As String)
```

```
SubmitBehavior(ByVal pSubmitControl As Control, _  
ByVal pValidationGroup As String, _  
ByVal pConfirmMessage As String)
```

ChangeMonitor

The ChangeMonitor watches for edits in the form and changes the appearance of buttons and other fields upon the first detected edit.

The classic case is to have a disabled OK button that gets enabled as you start typing. Another case is to show a message like “This form has changed” in a label. Both of these cases are handled.

DES’s enhanced buttons are already capable of showing a confirmation message. With the ChangeMonitor in use, that message can be shown based on whether or not the user has edited the form.

Click on any of these topics to jump to them:

- ◆ [Features](#)
- ◆ [Using the ChangeMonitor](#)
 - [The ChangeMonitor Property](#)
 - [Changing the State of Buttons](#)
 - [Making Data Entry Controls Notify Changes](#)
 - [Using the FieldStateController](#)
 - [Using your own JavaScript Code](#)
 - [Validation Group and ChangeMonitor Groups](#)
- ◆ [Properties of the PeterBlum.DES.Globals.WebFormDirector.ChangeMonitor](#)
- ◆ [ChangeMonitor Properties on Buttons](#)
- ◆ [ChangeMonitor Server Side Methods](#)
- ◆ [ChangeMonitor JavaScript Functions](#)
- ◆ [Online examples](#)

Features

- Watches for the first edit made by the user. Prior to that, it establishes an initial state of selected buttons and other controls. Buttons are usually disabled. After the edit, that state is changed to communicate to the user that an edit has occurred.

Note: It does not know the original value of controls. So if the user undoes their edit, the state does not revert back.

- The state of the ChangeMonitor is preserved through postbacks and callbacks. So once the page is edited and the user posts back, as the page is redrawn it still knows that it has been edited and sets buttons and other fields accordingly. Yet, if the page is posted back and is found to be entirely valid, you can have it draw the page again as if there were no edits. This helps when the user is entering multiple records with one page.
- DES's Enhanced Buttons automatically respond to the ChangeMonitor.
- DES's data entry controls (textboxes, MultiSegmentDataEntry, Calendar, etc) automatically notify the ChangeMonitor of changes. While most controls signal that they are changed with their DHTML onchange or onclick events, you can have them signal a change as the user makes an edit using the keyboard.
- Any data entry control with an attached validator (from the DES Validation Framework) automatically notifies the ChangeMonitor of changes.
- All other data entry controls can notify the ChangeMonitor simply by assigning the NativeControlExtender to them.
- A reset button (`<input type='reset' />`) reverts buttons controls to their unedited appearance.
- When using different Validation Groups, the page is usually separated into segments with its own buttons. ChangeMonitor follows the Validation Groups. It will only enable the button associated with the validation group that was changed. Controls that do not have a way to define a Validation Group have been given a ChangeMonitorGroup property where a group name can be defined.
- You can define an alternative grouping to Validation Groups. For example, while you have several Validation Groups on the page, you want all buttons to be enabled.
- You can change the state of any other control by using the FieldStateController with the ChangeMonitorCondition class in its **Condition** property. It will update the other control as the ChangeMonitor updates the state of buttons.
- The ChangeMonitor can call your own JavaScript code so you can take other actions as changes are made.
- DES Buttons setup to show a confirmation message can elect to show that confirmation message based on the state of the page: edited or not.

Using the ChangeMonitor

There are several parts to the ChangeMonitor:

- The **ChangeMonitor** property enables the feature and defines its general operation.
- DES Buttons are aware of the ChangeMonitor. Their **ChangeMonitorEnables** property determines if they respond to the ChangeMonitor.
- Data entry controls must notify the ChangeMonitor that they have changed.
- Other controls on the page can respond to the ChangeMonitor through the FieldStateController or a call to your own JavaScript code.

Click on any of these topics to jump to them:

- ◆ [The ChangeMonitor Property](#)
- ◆ [Changing the State of Buttons](#)
- ◆ [Making Data Entry Controls Notify Changes](#)
 - [Using the NativeControlExtender](#)
 - [Using the ChangeMonitor.RegisterForChanges\(\) Method](#)
- ◆ [Using the FieldStateController](#)
 - [The PeterBlum.DES.Web.WebControls.ChangeMonitorCondition Class](#)
- ◆ [Using your own JavaScript Code](#)
- ◆ [Validation Group and ChangeMonitor Groups](#)
- ◆ [Online examples](#)

The ChangeMonitor Property

The **ChangeMonitor** property, on the PageManager control and [PeterBlum.DES.Globals.WebFormDirector](#), enables the ChangeMonitor and defines its general operation.

When **ChangeMonitor.Enabled** is `TrueFalseDefault.True`, it is enabled. Immediately, DES Buttons will disable themselves when their **CausesValidation** property is `true`, until the first change is made that matches the Validation Group in their **Group** property. If all you want is to change the state of buttons and your Validation Groups connect to the desired buttons, there is nothing else you need to do!

ChangeMonitor.Enabled can be set globally using the **DefaultChangeMonitorEnabled** property in the “ChangeMonitor Defaults” topic of the **Global Settings Editor**.

By default, the ChangeMonitor will not detect an edit until the data entry control would notify a validator. For textboxes, lists, and dropdownlists, that happens on the DHTML onchange event. For radiobuttons and checkboxes, that happens on the onclick event. When using a textbox, it may feel more natural for the first character typed to notify the ChangeMonitor of changes. To support this, set **MonitorKeystrokes** to `TrueFalseDefault.True` or the global **DefaultChangeMonitor_MonitorKeystrokes** to `true` in the **Global Settings Editor**.

The ChangeMonitor preserves its state on a postback. This works well if the user submits the page and your code redraws the same page showing validation errors or requesting additional information. The idea is that a series of postback is still part of the same edit process so buttons should reflect that. If the page is redrawn but you need the ChangeMonitor to act as if there have been no edits, there are two ways to handle this:

- Let validation tell the ChangeMonitor that the page is valid. Set **ClearIfAllValid** to `TrueFalseDefault.True` or the global setting **DefaultChangeMonitorClearIfAllValid** in the **Global Settings Editor**.
- Call the method `ClearChanged()` or `ClearChangedOnAllGroups()` in your server side code. See [“ChangeMonitor Server Side Methods”](#).

Note: Several additional properties are described in later sections.

Changing the State of Buttons

The ChangeMonitor notifies DES Buttons when changes occurs. Use the **ChangeMonitorEnables** property on any DES Button to switch their state between disabled and enabled. Here are its values:

- No - The button will not change its enable state.
- Yes - The button will change its enabled state.
- CausesValidationIsTrue - When the button's **CausesValidation** property is true, it will change its enabled state. This is the default.
- CausesValidationIsFalse - When the button's **CausesValidation** property is false, it will change its enabled state.

Note: ImageButtons normally do not have a visual appearance for disabled. DES's ImageButtons change their appearance by changing the opacity of the button when the state is changed by the ChangeMonitor.

When the DES button uses a confirmation message from its **ConfirmMessage** property, it normally displays the message on any click. When using the ChangeMonitor, you can make it display based on the changed state of the page. Use the **ChangeMonitorUsesConfirm** property on the button. Here are its values:

- No - ChangeMonitor does not affect the confirmation message.
- Changed - Only show the confirmation message if changes were made.
- NotChanged - Only show the confirm message if NO changes were made.

Using server side code

The `PeterBlum.DES.Globals.WebFormDirector.ChangeMonitor` object has several methods that let you modify the state of the buttons. They are generally used to clear the state after a postback loads fresh data onto the form. See "[ChangeMonitor Server Side Methods](#)".

Making Data Entry Controls Notify Changes

Each data entry control must notify the ChangeMonitor that it has been changed. DES's data entry controls automatically notify the ChangeMonitor. This includes the textboxes, MultiSegmentDataEntry, Calendar, MonthYearPicker, and TimePicker controls.

For any other data entry controls, you have these options:

- When a DES validator is attached, it will detect changes and notify the ChangeMonitor. This even supports many third party controls. However, it will not set up the control for keyboard changes when **ChangeMonitor.MonitorKeystrokes** is in use. For that, you need the next option.
- Assign the NativeControlExtender to them or call the method `RegisterForChanges()` on the **PeterBlum.DES.Globals.WebFormDirector.ChangeMonitor** property. When these are used, the client-side controls fire their DHTML onchange or onclick events to notify the ChangeMonitor. If the **ChangeMonitor.MonitorKeystrokes** property is in use, it will also use the onkeypress event to monitor changes. See below.
- If your data entry control is a third party control that does not otherwise trigger the ChangeMonitor, you can write some JavaScript code that notifies ChangeMonitor by calling `DES_CMonSet()`.

Using the NativeControlExtender

Use the NativeControlExtender when working with design mode or ASP.NET Markup. When writing code, see "[Using the ChangeMonitor.RegisterForChanges\(\) Method](#)". To set up the NativeControlExtender, see the **General Features Guide**.

Here is an example with a native TextBox control:

```
<asp:TextBox id="TextBox1" runat="server" />
<des:NativeControlExtender id="NativeControlExtender1" runat="server"
  ControlIDToExtend="TextBox1" />
```

Using the ChangeMonitor.RegisterForChanges() Method

Indicates the control is to be monitored for changes. Its client-side onchange or onclick event will be hooked up so the ChangeMonitor system is notified of a change. If **ChangeMonitor.MonitorKeystrokes** is in use, it also sets up the onkeypress event.

[C#]

```
void RegisterForChanges(Control pControlToRegister,
  string pChangeMonitorGroups)
```

[VB]

```
Sub RegisterForChanges(ByVal pControlToRegister As Control,
  ByVal pChangeMonitorGroups As String)
```

Parameters

pControlToRegister

Control to monitor changes.

pChangeMonitorGroups

One or more change monitor group names. See "[Validation Group and ChangeMonitor Groups](#)".

The empty string ("") is a valid group name. If the page is not using groups, use an empty string.

For match to all groups, use "*".

If a group needs to be different based on its naming container, use "+" as the first character of the group name.

For multiple groups, use a pipe delimited list. For example: "Group1|Group2".

Using the FieldStateController

You can change the appearance of almost any control on the page using the FieldStateController (see “[FieldStateController and MultiFieldStateController Controls](#)”). With the **FieldStateController.Condition** property set to the `PeterBlum.DES.Web.WebControls.ChangeMonitorCondition` object, it will change the state of your control when the ChangeMonitor detects that the page has been edited.

For example, you want a Label saying “Changes were made. Click Submit to save them” to appear after a change is made. Here is the ASP.NET Markup for that:

```
<asp:Label id="ChangesWereMadeLabel" runat="server">
  Changes were made. Click Submit to save them</asp:Label>
<des:FieldStateController id="FieldStateController1" runat="server"
  ControlIDToChange="ChangesWereMadeLabel" ConditionFalse-Visible="false" >
  <ConditionContainer>
    <des:ChangeMonitorCondition ChangeMonitorGroups=" " />
  </ConditionContainer>
</des:FieldStateController>
```

The PeterBlum.DES.Web.WebControls.ChangeMonitorCondition Class

The following list are properties specific to this Condition:

- **ChangeMonitorGroups** (string) – One or more change monitor group names. See “[Validation Group and ChangeMonitor Groups](#)”.

The empty string ("") is a valid group name. If the page is not using groups, use an empty string.

For match to all groups, use "*".

If a group needs to be different based on its naming container, use "+" as the first character of the group name.

For multiple groups, use a pipe delimited list. For example: “Group1|Group2”.

It defaults to "".

Using your own JavaScript Code

If you want your own JavaScript code to run when the ChangeMonitor detects changes, use the **ChangeMonitor.OnChangeFunctionName** property to specify the name of a function that is called by the ChangeMonitor.

Your function must provide these properties in the order shown:

- **GroupName** (string) - The group that is being set or cleared. A value of "" is a valid group. A value of "*" indicates all groups.
- **Change** (boolean) - When `true`, the group has just been changed. When `false`, the group has been cleared of its change status.

It does not return a result.

Please position this function above the opening form tag to avoid a javascript error. If you have a Reset button on the page, support `pGroup=""` as reset will clear "*".

***ALERT:** Many users make the mistake of assigning JavaScript code to this property. This will cause JavaScript errors. **GOOD:** "MyFunction". **BAD:** "MyFunction();" and "alert('stop it')".*

***Note:** JavaScript is case sensitive. Be sure the value of this property exactly matches the function definition.*

Example

Changes the visibility of a control whose ClientID is "TextBox1". **ChangeMonitor.OnChangeFunctionName** is set to "MyChangeFunction".

```
function MyChangeFunction(pGroup, pChange)
{
    if (pChange)
        DES_GetById("TextBox1").style.visibility = "inherit";
    else
        DES_GetById("TextBox1").style.visibility = "hidden";
}
```

Validation Group and ChangeMonitor Groups

When you are using validators, you may be using their Validation Groups feature to associate specific buttons with specific validators. The ChangeMonitor respects this grouping. The ChangeMonitor will change the enabled state of the button whose validation group property (**Group** or **ValidationGroup**) matches the validation group name of the validator.

When using Validation Groups, you need to be aware of this.

- When a control is not managed by a validator, it needs the ChangeMonitor to know the correct validation group it is in. Assign that group to its **ChangeMonitorGroups** property. When using the NativeControlExtender, **ChangeMonitorGroups** is on that control. When using the [ChangeMonitor.RegisterForChanges\(\)](#) method, **ChangeMonitorGroups** is a parameter.
- When you want a button to respond to more Validation Group names than it has in its Group or ValidationGroup properties, set those additional groups in its **ChangeMonitorGroups** property.
 - To match to all groups, set **ChangeMonitorGroups** to "*".
 - To match to two or more additional groups, use a pipe delimited list. For example, "Group1|Group2".
- When you want to entirely ignore Validation Groups, set the **ChangeMonitor.UseValidationGroups** property to `False`.
- When you want to use a different grouping model, set the **ChangeMonitor.UseValidationGroups** property to `False`. Then assign the **ChangeMonitorGroups** property on each data entry control and button to new group names.

If you are not using Validation Groups, you can still use the **ChangeMonitorGroups** property to group certain data entry controls with buttons.

Example

Suppose you have a grid where the user can edit a row, or add a new record in the footer.

Start by entirely focusing on validation. You have two validation groups: row being edited (EditItemTemplate) and row being inserted (FooterTemplate).

Go through each of these templates and assign the **Group** property on each Validator and button to one of the two group names. (Proposed names: "Edit", "Insert")

If you have a ValidationSummary control, give its **Group** property the value of "*" (for all groups) or "Edit | Insert" (specifies a list of groups).

Make sure that validation works correctly.

Now let's look at the ChangeMonitor. By default, it groups buttons and edit controls by the names supplied using Validation Groups. Do you want the ChangeMonitor to work separately for Edit and Insert rows? If so, it is correctly setup. An edit in each row will only update its own buttons.

If not, ChangeMonitor should have its own grouping, using the **ChangeMonitorGroup** property on edit controls, NativeControlExtender (for non-DES edit controls) and buttons.

Set **ChangeMonitor.UseValidationGroups** to `TrueFalseDefault.False` on either **PeterBlum.DES.Globals.WebFormDirector** or the PageManager control.

In both cases, consider setting **ChangeMonitor.ClearIfAllValid** to `TrueFalseDefault.True` on either **PeterBlum.DES.Globals.WebFormDirector** or the PageManager control.

Properties of the PeterBlum.DES.Globals.WebFormDirector.ChangeMonitor

The ChangeMonitor property, found on the PageManager control and **PeterBlum.DES.Globals.WebFormDirector** property, provides properties that enable and configure the ChangeMonitor feature.

See “[Using the ChangeMonitor](#)”.

- **Enabled** (enum PeterBlum.DES.TrueFalseDefault) – Determines if the ChangeMonitor is enabled. When it’s enabled, controls will automatically start monitoring changes.

The enumerated type PeterBlum.DES.TrueFalseDefault has these values:

- True - Indicates the ChangeMonitor is enabled.
- False - Indicates the ChangeMonitor is disabled.
- Default - Use the value from the global setting **DefaultChangeMonitorEnabled** to determine if it’s enabled or not. **DefaultChangeMonitorEnabled** is set in the **Global Settings Editor** and defaults to false (disabled).

It defaults to TrueFalseDefault.Default.

- **UseValidationGroups** (enum PeterBlum.DES.TrueFalseDefault) – Determines if the Validation Group system provide group names in addition to group names from the **ChangeMonitorGroups** property on various controls.

When Controls grouped by validation group do not define the right group for monitoring, set this property to TrueFalseDefault.False.

Changes are monitored by group names which can come from two sources: the **ChangeMonitorGroups** and **Group** properties on various controls. Validation groups already provide an effective group naming system and are applied when this is TrueFalseDefault.True. However, controls marked by validation group do not always define the right group for change monitoring. So use this to turn off validation group names as the source of changes and only use the **ChangeMonitorGroups** property on the data entry controls and buttons.

The enumerated type PeterBlum.DES.TrueFalseDefault has these values:

- True - Indicates validation group names are used in addition to the **ChangeMonitorGroups** properties. If the control defines both a value in **Group** and **ChangeMonitorGroups**, they are both used.
- False - Indicates validation group names are not used. Only the **ChangeMonitorGroups** properties are used.
- Default - Use the value from the global setting **DefaultChangeMonitorUseValidationGroups** to determine if it’s enabled or not. **DefaultChangeMonitorUseValidationGroups** is set in the **Global Settings Editor** and defaults to true (enabled).

It defaults to TrueFalseDefault.Default.

- **MonitorKeystrokes** (enum PeterBlum.DES.TrueFalseDefault) – Controls that are edited through the keyboard can mark their change monitor group as changed as soon as the user changes the control.

Most controls tell the change monitor they are changed after an edit is completed, such as using the onclick or onchange event. Typing can enhance the user experience and enable a button so the user can type ENTER within the field and hit the button. If the button was disabled while focus is in the textbox, it would not press the button.

The enumerated type PeterBlum.DES.TrueFalseDefault has these values:

- True - Enables this feature.
- False - Disables this feature.
- Default - Use the value from the global setting **DefaultChangeMonitor_MonitorKeystrokes** to determine if it’s enabled or not. **DefaultChangeMonitor_MonitorKeystrokes** is set in the **Global Settings Editor** and defaults to true (enabled).

It defaults to TrueFalseDefault.Default.

- **ClearIfAllValid** (enum PeterBlum.DES.TrueFalseDefault) – Useful if the same page is used for multiple records. When there are no validation errors after a postback, that is the indication that the change monitor should clear its changed state.

On a post back, the state of the change monitor is preserved so that groups known as changed retain that state. This helps with autopostback and normal postbacks where validation errors are detected on the client-side.

While you can manually clear groups with the `ClearChanges()` and `ClearChangesOnAllGroups()` methods, this can automatically clear the group just validated if no validation error is found.

When true, if the page was validated using page-level validation, the validation group that was used is used by the change monitor to clear its own groups. If page-level validation did not occur, the state of the change monitor remains the same. Page-level validation requires either the **CausesValidation** property to be true on the submit control or a call to `PeterBlum.DES.Globals.WebFormDirector.Validate()`.

The enumerated type `PeterBlum.DES.TrueFalseDefault` has these values:

- `True` - Enables this feature.
- `False` - Disables this feature.
- `Default` - Use the value from the global setting **DefaultChangeMonitorClearIfAllValid** to determine if it's enabled or not. **DefaultChangeMonitorClearIfAllValid** is set in the **Global Settings Editor** and defaults to `false` (disabled).

It defaults to `TrueFalseDefault.Default`.

- **OnChangeFunctionName** (string) – Assign to the name of a JavaScript function that will be called as the change monitor first detects a change or gets cleared. See [“Using your own JavaScript Code”](#).

When "", no function is set up. It defaults to "".

ALERT: Many users make the mistake of assigning JavaScript code to this property. This will cause JavaScript errors.

GOOD: `“MyFunction”`. **BAD:** `“MyFunction();”` and `“alert(‘stop it’)”`.

Note: JavaScript is case sensitive. Be sure the value of this property exactly matches the function definition.

ChangeMonitor Server Side Methods

These methods are found on the `PeterBlum.DES.Globals.WebFormDirector.ChangeMonitor` object. (If you are using the `PageManager`, it also has a `ChangeMonitor` object with these methods but they should *not* be used.)

***ChangeMonitor.SetChanged()* method**

Indicates that a group is already changed.

[C#]

```
void SetChanged(string pGroupName)
```

[VB]

```
Sub SetChanged(ByVal pGroupName As String)
```

Parameters

pGroupName

Specify the `ChangeMonitor` group name that should be marked as changed.

Use "" when you are not using any groups.

It does not support a pipe delimited group list or "*" for all groups.

***ChangeMonitor.SetChangedOnAllGroups()* method**

Indicates that all groups are already changed.

[C#]

```
void SetChangedOnAllGroups()
```

[VB]

```
Sub SetChangedOnAllGroups()
```

***ChangeMonitor.ClearChanged()* method**

Indicates that a group is not changed.

[C#]

```
void ClearChanged(string pGroupName)
```

[VB]

```
Sub ClearChanged(ByVal pGroupName As String)
```

Parameters

pGroupName

Specify the `ChangeMonitor` group name that should be marked as unchanged.

Use "" when you are not using any groups.

It does not support a pipe delimited group list or "*" for all groups.

ChangeMonitor.ClearChangedOnAllGroups() method

Indicates that all groups are already changed.

[C#]

```
void ClearChangedOnAllGroups()
```

[VB]

```
Sub ClearChangedOnAllGroups()
```

ChangeMonitor.HasChanged() method

Indicates if the specified group is marked as changed. Evaluates a single group name, pipe delimited list of group names, or "*" to determine if a group is changed. So long as one group name matches to the known groups changed, it is considered changed.

[C#]

```
bool HasChanged(string pGroupName)
```

[VB]

```
Function HasChanged(ByVal pGroupName As String) As Boolean
```

Parameters

pGroupName

Specify the ChangeMonitor group name to be evaluated.

Use "" when you are not using any groups.

Supports a pipe delimited group list and "*" for all groups.

Return value

true when any group requested is changed. false when no changes are detected.

ChangeMonitor JavaScript Functions

When a control does not use the DHTML onchange event to notify of changes, you can write JavaScript code that is called by the control's alternative "on change" functionality. (Most third party controls provide an API that notifies you of a change.) Your JavaScript will call the `DES_CMonSet ()` function with the ChangeMonitor group name.

function `DES_CMonSet(pGroup)`

Indicates the specified ChangeMonitor group name has been changed.

Parameters

pGroupName

MUST BE uppercase.

Specify the ChangeMonitor group name to be evaluated.

Use "" when you are not using any groups.

Supports a pipe delimited group list and "*" for all groups.

function `DES_CMonClear(pGroup)`

Indicates the specified ChangeMonitor group name has not been changed.

Parameters

pGroupName

MUST BE uppercase.

Specify the ChangeMonitor group name to be evaluated.

Use "" when you are not using any groups.

Supports a pipe delimited group list and "*" for all groups.

function `DES_CMonIsChanged(pGroup)`

Indicates the state of the specified ChangeMonitor group name.

Parameters

pGroupName

MUST BE uppercase.

Specify the ChangeMonitor group name to be evaluated.

Use "" when you are not using any groups.

Supports a pipe delimited group list and "*" for all groups.

Return value

`true` when any group requested is changed. `false` when no changes are detected.

Direct Keystrokes to Click Buttons

DES's TextBoxes and the MultiSegmentDataEntry control offer the **EnterSubmitsControlID** property, which lets you direct the ENTER key to click a specific button or control. It's useful when you have several Submit buttons on the page, each with their own task.

Additionally, the NativeControlExtender control and `PeterBlum.DES.Globals.WebFormDirector.RegisterKeyClicksControl()` let you attach this capability to any control. This feature has several advantages:

- It allows you to define the keystroke that clicks the button. For example, ESC can hit a "Cancel" button. *Only applies to the RegisterKeyClicksControl()*
- Instead of setting it up for individual controls, you can set it up for a group of controls by attaching this to a container control, like a Panel or Table. The browsers are designed to let the onkeypress event, used here, to "bubble up" until consumed (which is what the container will do).

The client-side code calls the `click()` method on the control. This will run the control's `onclick` event. For a `<input type='submit'>` control, request that it submits the page. Other controls that have a call to `__doPostBack()` in their `onclick` event will attempt to submit the page too. (They will all validate if the submit control is set up to validate.)

You have the option of moving the focus from the data entry field to the button as a way to provide visual feedback to which button was clicked.

Click on any of these topics to jump to them:

- ◆ [Using the NativeControlExtender](#)
- ◆ [Using the RegisterKeyClicksControl\(\) Method](#)

Using the NativeControlExtender

1. Set the **ControlIDToExtend** to the control that intercepts the keystroke. If several controls need to click the same button and they are all contained in a tag like a `<div>`, `<p>`, or `<table>`, you can just use the container control's tag. It will capture the keystroke for all of its child controls.

The selected controls must have `runat=server` and an ID attribute.

2. Determine the control whose `click()` method will be invoked. Set **EnterSubmitsControlID** to that control.

There are a lot of controls that support `click()`, although they vary by browser. In addition to Buttons and ImageButtons, typical cases are hyperlinks, LinkButtons, checkboxes and radiobuttons. However, browsers don't all support the `click()` method on the same control. Here are the differences:

- Internet Explorer and Opera 7 support it on hyperlinks (and LinkButton) while Mozilla, FireFox, Netscape 7, and Safari do not.
- All support checkboxes and radiobuttons. However, Mozilla, FireFox, and Netscape 7 always remove the focus from the current field even if you don't set this feature up to move the focus (the focus is gone, not moved)
- All support Buttons the same way. This is the best choice for a control to click.

Using the RegisterKeyClicksControl() Method

1. Determine which data entry controls should use this feature when they have focus. If they are all contained in a tag like a `<div>`, `<p>`, or `<table>`, you can just use the container control's tag. It will capture the keystroke for all of its child controls.

The selected controls must have `runat=server` and an ID attribute.

2. Determine the keystroke. JavaScript uses numeric values called "keycodes" that often match to the [ASCII code table](#) but not always. ENTER (13), ESC (27), and most characters from SPACE (32) through TILDE (~) (126) are the same.

You can research the keycodes by adding a TextBox with this code:

```
TextBox1.Attributes.Add("onkeypress", "alert(event.keyCode);")
```

Then type into the textbox to see the keycodes.

3. Determine the control whose `click()` method will be invoked.

There are a lot of controls that support `click()`, although they vary by browser. In addition to Buttons and ImageButtons, typical cases are hyperlinks, LinkButtons, checkboxes and radiobuttons. However, browsers don't all support the `click()` method on the same control. Here are the differences:

- Internet Explorer and Opera 7 support it on hyperlinks (and LinkButton) while Mozilla, FireFox, Netscape 7, and Safari do not.
- All support checkboxes and radiobuttons. However, Mozilla, FireFox, and Netscape 7 always remove the focus from the current field even if you don't set this feature up to move the focus (the focus is gone, not moved)
- All support Buttons the same way. This is the best choice for a control to click.

4. In `Page_Load()` or a post back event handler, call the `PeterBlum.DES.Globals.WebFormDirector.RegisterKeyClicksControl()` method. See below.

PeterBlum.DES.Globals.WebFormDirector.RegisterKeyClicksControl Method

Sets up JavaScript that intercepts a keystroke and fires the `click()` method of a control. For example, monitor ENTER keys and fire a Submit button to validate and submit the page.

Call it within `Page_Load()` or post back event handler methods.

[C#]

```
public void RegisterKeyClicksControl(Control pControlToMonitor,
    Control pControlToClick, int pKeyCode,
    bool pSetFocus)
```

[VB]

```
Public Sub RegisterKeyClicksControl(ByVal pControlToMonitor As Control,
    ByVal pControlToClick As Control, ByVal pKeyCode As Integer,
    ByVal pSetFocus As Boolean)
```

Parameters

pControlToMonitor

The control that will monitor keystrokes. Usually a data entry control or a container control.

Note: The control must have `runat=server` and an ID attribute. Usually you can use `Page.FindControl("ID")` to retrieve the control's object.

pControlToClick

The control whose client-side `click()` method will be fired. Buttons, ImageButtons, LinkButtons, and HyperLinks are some of the most common controls for this because they fire commands.

Note: The control must have `runat=server` and an ID attribute. Usually you can use `Page.FindControl("ID")` to retrieve the control's object.

pKeyCode

Keycodes are client-side values returned by the event object's `keyCode` attribute. 13 is ENTER; 27 is ESC. The user can research other keystrokes by adding a TextBox with this code:

```
TextBox1.Attributes.Add("onkeypress", "alert(event.keyCode);")
```

Then type into the textbox to see the keycodes.

pSetFocus

When `true`, it sets focus to the control (if possible) prior to clicking it. This shows the user what they clicked better but moves the focus from the current field. When `false`, focus does not move.

Example

A group of textboxes are contained in a Panel called `Panel1`. The panel contains two Buttons, `SubmitBtn` and `CancelBtn`. When ENTER is typed, click `SubmitBtn`. When ESC is typed, click `CancelBtn`. Set focus to the button as it clicks.

This code is in `Page_Load()`:

[C#]

```
PeterBlum.DES.Globals.WebFormDirector.RegisterKeyClicksControl(Panel1, SubmitBtn,
13, true);
PeterBlum.DES.Globals.WebFormDirector.RegisterKeyClicksControl(Panel1, CancelBtn,
27, true);
```

[VB]

```
PeterBlum.DES.Globals.WebFormDirector.RegisterKeyClicksControl(Panel1, SubmitBtn,
13, True)
PeterBlum.DES.Globals.WebFormDirector.RegisterKeyClicksControl(Panel1, CancelBtn,
27, True)
```

Custom Submit Function

ASP.NET provides the [Page.RegisterOnSubmitStatement\(\)](#) method to add code that will run when a submit control is fired. It has limitations that DES addresses with the

[PeterBlum.DES.Globals.WebFormDirector.CustomSubmitFunctionName](#) property addresses.

- It is connected to the validation process. You determine if your code executes before or after Validation.
- It can stop the page from submitting by returning false. If it occurs before Validation, it will prevent validation too.
- It works with any control that submits the page and is attached to DES's Validators.

For example, after validating the page, display an absolutely positioned <div> that tells the user to wait. You would have already added the <div> to the page and made its style="visibility:hidden;display:none". You add JavaScript to display the <div> after validation has occurred.

Any submit control that has its **CausesValidation** property set to true will invoke your function if supplied.

Click on any of these topics to jump to them:

- ◆ [Using The Custom Submit Function](#)
- ◆ [Page-Level Properties](#)

Using The Custom Submit Function

Your JavaScript code should be contained in a function that has a specific parameter list and returns a Boolean value.

Your function should take one parameter, the group name (an uppercase string), which you can use if your code depends on a group. Your function should return true to continue submitting or false to stop submitting the page.

```
function MySubmitFnc(pGroup)
{
    // do your work
    if (continue)
        return true;
    else
        return false;
}
```

In [Page_Load\(\)](#), set the function name in the

[PeterBlum.DES.Globals.WebFormDirector.ValidationManager.CustomSubmitFunctionName](#) property. Determine whether the function is run before or after validation with the

[PeterBlum.DES.Globals.WebFormDirector.ValidationManager.SubmitOrder](#) property.

Example

This code appears in [Page_Load\(\)](#):

[C#]

```
PeterBlum.DES.Globals.WebFormDirector.ValidationManager.CustomSubmitFunctionName =
    "MySubmitFnc";
PeterBlum.DES.Globals.WebFormDirector.ValidationManager.SubmitOrder =
    PeterBlum.DES.Web.SubmitOrderType.ConfirmValidateCustom;
```

[VB]

```
PeterBlum.DES.Globals.WebFormDirector.ValidationManager.CustomSubmitFunctionName =
    "MySubmitFnc"
PeterBlum.DES.Globals.WebFormDirector.ValidationManager.SubmitOrder = _
    PeterBlum.DES.Web.SubmitOrderType.ConfirmValidateCustom
```

Page-Level Properties

The following properties are on the **PeterBlum.DES.Globals.WebFormDirector** object. You set them in the `Page_Load()` method.

- **CustomSubmitFunctionName** (string) – Use this to add your own JavaScript code into the page submission process. Any submit control that has its **CausesValidation** property set to `true` will invoke your function if supplied.

Your function should take one parameter, the group name (an uppercase string), which you can use if your code depends on a group. Your function should return `true` to continue submitting or `false` to stop submitting the page. This property should only contain the name of the function. See “[Using The Custom Submit Function](#)”.

Note: The group name property will always be uppercase, even when the user entered it with lowercase. Be sure that you use an uppercase group name when you compare to the parameter.

The custom submit function is part of a group of actions that occur during submission: validation, confirm message and custom submit function. Use the **PeterBlum.DES.Globals.WebFormDirector.SubmitOrder** property to determine the order of these actions.

When this property is "", no custom submit function is defined.

It defaults to "".

ALERT: Many users make the mistake of assigning JavaScript code to this property. This will cause JavaScript errors.

GOOD: “MyFunction”. **BAD:** “MyFunction(;)” and “alert(‘stop it’)”.

Note: JavaScript is case sensitive. Be sure the value of this property exactly matches the function definition.

Example

```
function MySubmitFnc(pGroup)
{
    // do your work
    if (continue)
        return true;
    else
        return false;
}
```

Additional Topics for Using These Controls

This section covers a variety of special cases when using these controls.

Click on any of these topics to jump to them:

- ◆ [Page Level Properties and Methods](#)
- ◆ [JavaScript Support Functions](#)
- ◆ [Adding Your JavaScript to the Page](#)

These topics are found in the **General Settings Guide**:

- ◆ Using these Controls with AJAX
- ◆ The ViewState and Preserving Properties for PostBack
- ◆ Establishing Default Localization for the Web Form
- ◆ Using Style Sheets
- ◆ The String Lookup System
- ◆ The Global Settings Editor
- ◆ Using Server Transfer and Using Alternative HttpHandlers
- ◆ Using a Redistribution License
- ◆ Browser Support and The TrueBrowser Class

Page Level Properties and Methods

The **PeterBlum.DES.Globals.WebFormDirector** object contains several properties that affect all controls on the page. They include setting the **CultureInfo** object, getting the Browser details, and enabling JavaScript.

The **Page** property on **PeterBlum.DES.Globals** uses the class `PeterBlum.DES.Web.WebControls.WebFormDirector`. When accessed through **PeterBlum.DES.Globals.WebFormDirector**, you will have an object that is unique to the current thread. It is really a companion to the **Page** object of a web form, hosting details related to DES. Properties set on it will not affect any other request for a page.

Properties on PeterBlum.DES.Globals.WebFormDirector

You generally assign properties to **PeterBlum.DES.Globals.WebFormDirector** in your `Page_Load()` method. Your post back event handler methods can also assign properties.

- **CultureInfo** (`System.Globalization.CultureInfo`) – Cultures define date, time, number and text formatting for a program to follow. DES uses this value within its data types (`PeterBlum.DES.DESTypeConverter` classes) as it translates between strings and values.

The **CultureInfo** property uses `CultureInfo.CurrentCulture` by default. This value is determined by the web server's .Net settings, the web.config's `<globalization>` tag, or the `<% @Page %>` tag with the **Culture** property.

Web.Config setting – Affects the entire site

```
<globalization Culture="en-US" [other properties] />
```

Page Setting – Affects a page

```
<%@Page Culture="en-US" [other page properties] %>
```

You can set it programmatically in your `Page_Init()` method or in the `Application_BeginRequest()` method of **Global.aspx**. Use the .Net Framework method `CultureInfo.CreateSpecificCulture()`. For example, assigning the US culture looks like this:

```
PeterBlum.DES.Globals.WebFormDirector.CultureInfo =
    CultureInfo.CreateSpecificCulture("en-US")
```

Changing the properties of CultureInfo programmatically

Assign values to **PeterBlum.DES.Globals.WebFormDirector.CultureInfo**. Here are some examples:

[C#]

```
System.Globalization.DateTimeFormatInfo vDTFI =
    PeterBlum.DES.Globals.WebFormDirector.CultureInfo.DateTimeFormat;
vDTFI.ShortDatePattern = "MM-dd-yyyy";
vDTFI.DateSeparator = "-";
System.Globalization.NumberFormatInfo vNFI =
    PeterBlum.DES.Globals.CultureInfo.NumberFormat;
vNFI.DecimalSeparator = ".";
vNFI.CurrencySymbol = "€";
```

[VB]

```
Dim vDTFI As System.Globalization.DateTimeFormatInfo = _
    PeterBlum.DES.Globals.WebFormDirector.CultureInfo.DateTimeFormat
vDTFI.ShortDatePattern = "MM-dd-yyyy"
vDTFI.DateSeparator = "-"
Dim vNFI As System.Globalization.NumberFormatInfo = _
    PeterBlum.DES.Globals.CultureInfo.NumberFormat
vNFI.DecimalSeparator = "."
vNFI.CurrencySymbol = "€"
```

- **HintManager** (PeterBlum.DES.Web.WebControls.HintManager) – Properties used by [Interactive Hints](#). See “[Properties on the PeterBlum.DES.Globals.WebFormDirector.HintManager Property](#)”.
- **ChangeMonitor** (PeterBlum.DES.Web.WebControls.ChangeMonitor) – Properties used by the [ChangeMonitor](#). See “[Properties of the PeterBlum.DES.Globals.WebFormDirector.ChangeMonitor](#)”.
- **FSCManager** – Hosts a list of all FieldStateControllers on the page in its **Items** property.
- **TextCounterManager** – Hosts a list of TextCounter controls on the page in its **Items** property.
- **MenuManager** – Hosts a list of Menu controls on the page in its **Items** property.
- **SubmitPageManager** (PeterBlum.DES.Web.WebControls.SubmitPageManager) – Properties used by [Enhanced Buttons](#). See “[Programmatically Adding These Features to Non-DES Buttons](#)”.
- **Browser** (PeterBlum.DES.Web.TrueBrowser) – Detects the actual browser that is requesting the page and configures the HTML and JavaScript code returned to work with that browser. If the browser doesn’t support the client-side scripting code for FieldStateControllers, the **TrueBrowser.SupportsFieldStateControllers** property is `false` and they are disabled. See “[Browser Support](#)” in the **General Features Guide**.
- **JavaScriptEnabled** (Boolean) – Determines if the browser really has JavaScript enabled. It automatically detects if JavaScript is enabled after the first post back for a session. Prior to that first post back, it is `true`. After that, it is `true` when JavaScript is enabled and `false` when it is not.

When `false`, the page will be generated as if the browser does not support JavaScript. No controls will output JavaScript and may draw themselves differently, knowing that a client-side only feature that doesn’t work is inappropriate to output.

This feature stores its state in the Session collection. If the Session is not working or has been cleared, it will reset to `true` and attempt to resolve the JavaScript state on the next post back.

If you do not want this detection feature enabled, set **DetectJavaScript** to `false`.

You can set this value directly in `Page_Load()`. It lets you turn off all of DES’s JavaScript features on demand. For example, your customers can identify if they use JavaScript on their browser in a configuration screen. It only affects the current page so set it on each page where needed.

- **DetectJavaScript** (Boolean) – When `true`, the **JavaScriptEnabled** property will monitor for JavaScript support. When `false`, it will not.

It defaults to the global **DefaultDetectJavaScript** property, which defaults to `true`. You set **DefaultDetectJavaScript** with the **Global Settings Editor**. (For details on the **Global Settings Editor**, see “[Global Settings: The Editor and custom.DES.config File](#)” in the **General Features Guide**.)

- **ButtonEffectsManager.EnableButtonImageEffects** (enum PeterBlum.DES.Web.EnableButtonImageEffects) – Many buttons can show up to 3 images: normal, pressed, and mouseover. By default, these effects are set up based on the presence of the actual files. However, DES cannot always see the files are present. For example, the URL uses `http://`. **EnableButtonImageEffects** lets you to specify that the images are present or not.

The enumerated type `PeterBlum.DES.Web.EnableButtonImageEffects` has these values:

- `None` - Never use image effects.
- `Always` - Always use image effects. Assume that all image files are available
- `Auto` - Detect the files, if possible before using them
- `Pressed` - Always set up for pressed. Never set up for mouse over
- `MouseOver` - Always set up for mouseover. Never set up for pressed

It defaults to `EnableButtonImageEffects.Auto`.

- **PageIsLoadingMsg** (string) – The error message to display on the client-side if the user interacts with this control before it is initialized. It defaults to “`Page is loading. Please wait.`”.

Validation Properties

These properties are only used with the DES Validation Framework.

- **ConfirmMessage** (string) – Allows you to show an OK/Cancel message box when the user submits the page, regardless of if there are any errors found. If they answer OK, submit is continued. With Cancel, it is cancelled. When this is blank, no alert is shown.

DES submit controls can override this with their own **ConfirmMessage** property . See “[Enhanced Buttons](#)”.

The alert appears based on the group being submitted. It must match **ConfirmMessageGroup**.

Submit controls whose **CausesValidation** property is `false` will not show this messagebox.

The confirm message is part of a group of actions that occur during submission: validation, confirm message and custom submit function. Use the [SubmitOrder](#) property to determine the order of these actions.

It defaults to the **DefaultConfirmMessage** property in the **Global Settings Editor**, which defaults to "".

- **ConfirmMessageLookupID** (string) – Gets the value for **ConfirmMessage** through the String Lookup System. (See “String Lookup System” in the **General Features Guide**.) The LookupID and its value should be defined within the String Group of [ConfirmMessages](#). If no match is found OR this is blank, **ConfirmMessage** will be used.

The String Lookup System lets you define a common set of terms so the programmer doesn't uniquely define them each time. It also provides localization based on the current culture.

To use it, define a LookupID and associated textual value in your data source (resource, database, etc). Assign the same LookupID to this property.

It defaults to the **DefaultConfirmMessageLookupID** property in the **Global Settings Editor**, which defaults to "".

- **ConfirmMessageGroup** (string) – When using the **ConfirmMessage** property, use this property to determine which group shows this message. When "", it will match group names that are blank. If this is "*", it will match all group names.

It defaults to the **DefaultConfirmMessageGroup** property in the **Global Settings Editor**, which defaults to "".

- **ValidationManager.SubmitOrder** (enum PeterBlum.DES.Web.SubmitOrderType) – Determines the order of these three client-side actions when the page is submitted:
 - Validation of fields associated with submit button's group
 - Confirm message when the **ConfirmMessage** property is set up.
 - Custom submit function when the **CustomSubmitFunctionName** property is set up.

The enumerated type `PeterBlum.DES.Web.SubmitOrderType` has these values:

- `ConfirmCustomValidate` – Confirm message, Custom submit function, Validate.
- `ConfirmValidateCustom`
- `CustomConfirmValidate`
- `CustomValidateConfirm`
- `ValidateConfirmCustom`
- `ValidateCustomConfirm`

It defaults to the **DefaultSubmitOrder** property in the **Global Settings Editor**, which defaults to `SubmitOrderType.ConfirmCustomValidate`.

Note: When the page posts back to the server, it will once again run validation. Server-side validation is not affected by this property. It always occurs after all client-side actions.

JavaScript Support Functions

This section shows how to communicate with these controls from your own JavaScript.

DES supplies the following client-side functions to any page that includes these controls.

Click on any of these topics to jump to them:

- ◆ [General Utilities](#)
- ◆ [JavaScript: Running FieldStateControllers on demand](#)
- ◆ [JavaScript: Running CalculationControllers On Demand](#)
- ◆ [Javascript functions: Show and Hide the Hint On Demand](#)
- ◆ [ChangeMonitor JavaScript Functions](#)
- ◆ [Custom Submit Function](#)

General Utilities

function *DES_GetById(pID)*

Returns the DHTML element associated with the ID supplied. This is a wrapper around the functions `document.all[]` and `document.getElementById()` so that you can get the field using browser independent code.

Parameters

pID

The **ClientID** property value from the server side control. It is the value written into the `id=` attribute of the HTML element. See “[Embedding the ClientID into your Script](#)”.

Return value

Returns the field object or null.

Example

```
var vOtherField = DES_GetById('DateTextBox1');
```

function *DES_ParseInt(pText)*

It converts it into an integer number and returns the number. While the JavaScript `parseInt()` function is supposed to do this, when there is a lead zero, `parseInt()` believes the number is octal (base 8). Thus, `08` is returned as `10`. Dates often have lead zeros. So call this instead of `parseInt()`. *Internally, it calls `parseInt()` after stripping off the lead zeroes.*

Parameters

pText

The string to convert to an integer.

Return value

An integer. If the text represented a decimal value, it will return the integer portion. If it cannot be converted, it returns `NaN` which you can detect with the JavaScript function `isNaN(value)`.

Example

```
var vNumber = DES_ParseInt("03"); // returns 3
if (!isNaN(vNumber))
    // do something with vNumber
```

function *DES_SetFocus(pID)*

Sets focus to the HTML element whose ID is passed in. It will not set focus if the element is not present or it's illegal to set focus (such as its invisible). It will also select the contents of a textbox, if the ID is to a textbox.

It calls your custom focus function defined in **PeterBlum.DES.Globals.WebFormDirector.SetFocusFunctionName** to assist it to setting focus. (See "Properties on PeterBlum.DES.Globals.WebFormDirector" in the **General Features Guide**.)

Parameters

pID

The **ClientID** property value from the server side control. It is the value written into the `id=` attribute of the HTML element. See "Embedding the ClientID into your Script".

Return value

Returns the field object or null.

Example

```
DES_SetFocus( 'DateTextBox1' );
```

function *DES_Round(pValue, pMode, pDecimalPlaces)*

Rounds a decimal value in several ways.

Parameters

pValue

The initial decimal value.

pMode

An integer representing one of the rounding modes:

0 = Truncate – Drop the decimals after `pDecimalPlaces`

1 = Currency – Round to the nearest even number

2 = Point5 – Round to the next number if .5 or higher; round down otherwise

3 = Ceiling – Returns the smallest integer greater than or equal to a number. When it's a negative number, it will return the number closest to zero.

4 = NextWhole - Returns the smallest integer greater than or equal to a number. When it's a negative number, it will return the number farthest from zero.

pDecimalPlaces

The number of decimal places to preserve. For example, when 2, it rounds based on the digits after the 2nd decimal place.

Return value

Returns the rounded decimal value.

Example

```
var PI = 3.14159;  
var vResult = DES_Round(PI, 0, 0); // Truncate: returns 3  
vResult = DES_Round(PI, 1, 2); // Currency: returns 3.14  
vResult = DES_Round(PI, 3, 0); // Ceiling: returns 4
```

function *DES_Trunc(pValue)*

Returns the integer part of the a decimal value. Converts the type from float to integer.

Parameters

pValue

The initial decimal value.

Return value

Returns the integer part of the a decimal value. Converts the type from float to integer.

Example

```
var PI = 3.14159;  
var vResult = DES_Trunc(PI); // returns 3
```

function *DES_SetInnerHTML(pField, pHTML)*

A browser independent way to update the inner HTML of a tag. Usually you will define a `` tag with an ID. The inner HTML of that tag will be updated. A `System.Web.UI.WebControls.Label` creates such a `` tag and its **ClientID** is the ID to find the tag on the page.

Parameters

pField

The DHTML element for the HTML table. Use [DES_GetById\(\)](#) to convert a **ClientID** into an DHTML element. See [“Embedding the ClientID into your Script”](#).

pHTML

The inner HTML.

Example

```
DES_SetInnerHTML(DES_GetById('Label1'), 'New Text');
```

function *DES_RERpl(pText, pFind, pReplace)*

Replaces text in a string. Internally, it uses a regular expression to do a case insensitive match for the text *pFind* and replaces it with *pReplace*.

Parameters

pText

The text to be modified.

pFind

The text to find within pText.

pReplacet

The text to replace.

Return value

The updated value of *pText*.

Example

```
var vText = "This is {0}.";   
vText = DES_RERpl(vText, '{0}', 'replaced text');
```

Adding Your JavaScript to the Page

Some of DES's features allow you to write your own JavaScript. When writing JavaScript, you can put it in three places:

- Directly on the page. It is typically placed before the <form> tag. Be sure to enclose it in <script> tags like this:

```
<script type='text/javascript' language='javascript' >
<!--
add your function code here
// -->
</script>
```

- In your Page_Load() code using the [Page.RegisterClientScriptBlock\(\)](#) method. You must still include the <script> tags in your code:

[C#]

```
uses System.Text;
...
protected void Page_Load(object sender, System.EventArgs e)
{
    StringBuilder vScript = new StringBuilder(2000);
    vScript.Append("<script type='text/javascript' language='javascript' >\n");
    vScript.Append("<!-- \n");
    vScript.Append( add your function code here );
    vScript.Append("// -->\n</script>\n");
    RegisterClientScriptBlock("KeyName", vScript.ToString());
}
```

[VB]

```
Imports System.Text
...
Protected Sub Page_Load(ByVal sender As object, _
    ByVal e As System.EventArgs)

    Dim vScript As StringBuilder = New StringBuilder(2000)
    vScript.Append("<script type='text/javascript' language='javascript' >")
    vScript.Append("<!-- ")
    vScript.Append( add your function code here )
    vScript.Append("// --></script>")
    RegisterClientScriptBlock("KeyName", vScript.ToString())
End Sub
```

- In a separate file, dedicated to JavaScript. This file doesn't need <script> tags. Instead, the page needs <script src= > tags to load it. The script tags should appear before the <form> tag.

```
<script type='text/javascript' language='javascript' src='url to the file' />
```

Embedding the ClientID into your Script

If your scripts are embedded into your web form, you can use this syntax to get the **ClientID**:

```
'<% =ControlName.ClientID %>'
```

For example:

```
DES_GetById('<% =ControlName.ClientID %>');
```

If you create the script programmatically, simply embed the **ClientID** property value. For example:

```
vScript = "DES_GetById('" + ControlName.ClientID + "')";
```

Debugging Your JavaScript

Using Internet Explorer

You can debug JavaScript in Internet Explorer by using Visual Studio as your debugger. Open the **Tools; Internet Options** menu command and select the **Advanced** tab. Then unmark **Disable Script Debugging**.

After launching your web page from Visual Studio, switch back to Visual Studio. Then select **Debug; Windows; Script Explorer** (or **Running Documents** in VS2002/3) from the menubar. Double-click on the filename containing the JavaScript function and set a breakpoint inside the function. Now resume using your browser.

Using FireFox

Use the FireBug debugger for FireFox. Get it here: <https://addons.mozilla.org/en-US/firefox/addon/1843>

The Condition classes

Many features of DES use the Condition classes. These are objects that evaluate something about your page and return “success”, “failed”, or “cannot evaluate”. They are used by:

- **Enabler** property on Validators, FieldStateControllers, and CalculationController. This is used to enable or disable the control as the page’s state changes.
- **Condition** property on FieldStateController and MultiFieldStateController. This is used to select between the ConditionTrue and ConditionFalse properties.
- ConditionCalcItem object for CalculationControllers. This is used to build IF ELSE logic into your calculation based on the state of the page.

Condition classes are also the underlying engine for the Validator controls. As a result, they are fully documented in the **Validation User’s Guide**. This section helps you quickly track down the right Condition class.

Licensing note: Most conditions require a license for either the Peter’s Professional Validation or Peter’s More Validators modules based on the associated Validator’s license. Only the Non-Data Entry conditions are available without a license.

Click on any of these topics to jump to them:

- ◆ [Evaluating Textboxes and other controls with textual values](#)
- ◆ [Evaluating Listboxes and other controls with index values](#)
- ◆ [Evaluating checkboxes, radiobuttons, and other controls](#)
- ◆ [Evaluating non-data value states of controls](#)

Evaluating Textboxes and other controls with textual values

Class name (in the name space PeterBlum.DES.Web.WebControls)	Description and usage
RequiredTextCondition	Evaluates as “success” when the textbox has text and “failed” when it is empty (or all spaces). <i>Usage:</i> Assign the ControlIDToEvaluate property to the control.
CompareToValueCondition	Evaluates as “success” when the value matches a comparison to another value and “failed” when it cannot. Evaluates as “cannot evaluate” when the text cannot be converted based on the DataType property or the text value is blank. <i>Usage:</i> Assign the ControlIDToEvaluate property to the control. Set DataType to the type the text represents. Set ValueToCompare to a string representation of the value to compare to. Set the Operator to the comparison operator.
CompareTwoFieldsCondition	Evaluates as “success” when the values of two textboxes match the comparison operator and “failed” when it cannot. Evaluates as “cannot evaluate” when the text cannot be converted based on the DataType property or the text value is blank. <i>Usage:</i> Assign the ControlIDToEvaluate and SecondControlIDToEvaluate properties to the controls to compare. Set DataType to the type the text represents. Set the Operator to the comparison operator.
RegexCondition	Evaluates as “success” when the text matches the regular expression and “failed” when it does not. <i>Usage:</i> Assign the ControlIDToEvaluate property to the control. Set Expression to the regular expression. Use properties of NotCondition , CaseInsensitive and Multiline to enhance the logic
DataTypeCheckCondition	Evaluates as “success” when the text can be converted to the type and “failed” when it cannot. Evaluates as “cannot evaluate” when the text value is blank. <i>Usage:</i> Assign the ControlIDToEvaluate property to the control and DataType to the data type.
TextLengthCondition	Evaluates as “success” when the number of characters is within a range and “failed” when it is not. <i>Usage:</i> Assign the ControlIDToEvaluate property to the textbox. Set Minimum and Maximum to the valid character range.
CharacterCondition	Evaluates as “success” when all characters are within a defined character set and “failed” when they are not. <i>Usage:</i> Assign the ControlIDToEvaluate property to the textbox. Define the character set using these boolean properties: LettersUppercase , LettersLowercase , Digits , Space , Enter , DiacriticLetters , Punctuation , VariousSymbols , EnclosureSymbols , MathSymbols , and CurrencySymbols . Define any specific characters in the OtherCharacters property. Exclude the character set defined with the Exclude property.
RangeCondition	Evaluates as “success” when the value is within a range and “failed” when it is outside the range. Evaluates as “cannot evaluate” when the text cannot be converted based on the DataType property or the text value is blank. <i>Usage:</i> Assign the ControlIDToEvaluate property to the textbox. Set DataType to the type the text represents. Set Minimum and Maximum to a string representation of the range limits or programmatically set MinimumAsNative and MaximumAsNative to the native type (integer, double, DateTime, etc).

CompareToStringsCondition	<p>Evaluates as “success” when the text matches to one or more strings and “failed” when it does not.</p> <p><i>Usage:</i> Assign the ControlIDToEvaluate property to the textbox. For each string to compare, add a CompareToStringItem object to its Items collection. Use the TextMatchRule to define how the text is matched.</p>
MultipleRequiredControlsCondition	<p>Evaluates as “success” when multiple textboxes have been assigned according to a rule and “failed” otherwise.</p> <p><i>Usage:</i> Assign the ControlIDToEvaluate and SecondControlIDToEvaluate properties to the first two textboxes. For additional textboxes, add RequiredTextControl objects to the ControlsToEvaluate collection. Use the Mode property to select the rule as All, All or None, Only one, At least one, or Range (which uses Minimum and Maximum properties).</p>
MultiCondition	<p>Builds a boolean expression from <i>any</i> Condition objects, such as “RequiredTextCondition on TextBox1 AND CheckStateCondition on CheckBox1”.</p> <p>Evaluates as “success” when its child Conditions evaluate as success based on its Operator property and “failed” when it does not.</p> <p><i>Usage:</i> Add each Condition as a child object of the Conditions collection. Set the Operator property to either AND or OR.</p>

Note: There are additional conditions that are lesser used: CustomCondition, DifferenceCondition, WordCountCondition, ABARoutingNumberCondition, CreditCardNumberCondition, and EmailAddressCondition. See the Validation User’s Guide for details.

Evaluating Listboxes and other controls with index values

Class name (in the name space PeterBlum.DES.Web.WebControls)	Description and usage
RequiredListCondition	<p>Evaluates as “success” when the list has a selected item and “failed” when it does not</p> <p><i>Usage:</i> Assign the ControlIDToEvaluate property to the control. If the first item of the list indicates “no selection”, set UnassignedIndex to 0.</p>
SelectedIndexCondition	<p>Evaluates as “success” when the list has a specific index selected or unselected and “failed” otherwise.</p> <p><i>Usage:</i> Assign the ControlIDToEvaluate property to the control. Set the Index property to the index of the list (where 0 is the first item in the list) and Selected to true if the item must be selected and false if it must be unselected.</p>
SelectedIndexRangesCondition	<p>Evaluates as “success” when the list has a specific index selected or unselected amongst a list of valid indices and “failed” otherwise.</p> <p><i>Usage:</i> Assign the ControlIDToEvaluate property to the control. To define an index or range of indices, add a SelectedIndexRange object to the Ranges collection. Set Selected to true if the item must be selected and false if it must be unselected.</p>
CountSelectionsCondition	<p>Evaluates as “success” when the number of selected items is within a range and “failed” otherwise</p> <p><i>Usage:</i> Assign the ControlIDToEvaluate property to the control. Set the range with the Minimum and Maximum properties.</p>
MultiCondition	<p>Builds a boolean expression from <i>any</i> Condition objects, such as “RequiredTextCondition on TextBox1 AND SelectedIndexCondition where Index=2 on ListBox1”.</p> <p>Evaluates as “success” when its child Conditions evaluate as success based on its Operator property and “failed” when it does not.</p> <p><i>Usage:</i> Add each Condition as a child object of the Conditions collection. Set the Operator property to either AND or OR.</p>

Note: Many List style controls also have a textual value as defined in `<asp:ListItem value="here">label</asp:ListItem>`. You can use the Conditions described in [“Evaluating Textboxes and other controls with textual values”](#) if you want to evaluate them.

Evaluating checkboxes, radiobuttons, and other controls

Class name (in the name space PeterBlum.DES.Web.WebControls)	Description and usage
CheckStateCondition (CheckBox and RadioButton)	Evaluates as “success” when a checkbox’s value matches the Checked property and “failed” when it does not. Also supports individual radiobuttons. <i>Usage:</i> Assign the ControlIDToEvaluate property to the control. Set Checked to true to require the checkbox to be marked.
SelectedIndexCondition (CheckBoxList and RadioButtonList)	Evaluates as “success” when the RadioButtonList or CheckBoxList has a specific button marked or unmarked and “failed” otherwise. <i>Usage:</i> Assign the ControlIDToEvaluate property to the control. Set the Index property to the index of the button (where 0 is the first item in the list) and Selected to true if the item must be marked and false if it must be unmarked.
SelectedIndexRangesCondition (CheckBoxList and RadioButtonList)	Evaluates as “success” when the RadioButtonList or CheckBoxList has a specific button marked or unmarked amongst a list of valid button positions and “failed” otherwise. <i>Usage:</i> Assign the ControlIDToEvaluate property to the control. To define a button position or range of positions, add a SelectedIndexRange object to the Ranges collection. Set Selected to true if the item must be marked and false if it must be unmarked.
CountSelectionsCondition (CheckBoxList)	Evaluates as “success” when the number of marked buttons in a CheckBoxList is within a range and “failed” otherwise <i>Usage:</i> Assign the ControlIDToEvaluate property to the control. Set the range with the Minimum and Maximum properties.
RequiredSelectionCondition (Calendar, MultiSelectionCalendar, MonthYearPicker, and TimePicker)	Evaluates as “success” when the control has a selected cell and “failed” if no cells are selected. <i>Usage:</i> Assign the ControlIDToEvaluate property to the control.
MultiCondition	Builds a boolean expression from <i>any</i> Condition objects, such as “RequiredTextCondition on TextBox1 AND CheckStateCondition on CheckBox1”. Evaluates as “success” when its child Conditions evaluate as success based on its Operator property and “failed” when it does not. <i>Usage:</i> Add each Condition as a child object of the Conditions collection. Set the Operator property to either AND or OR.

Evaluating non-data value states of controls

Class name (in the name space PeterBlum.DES.Web.WebControls)	Description and usage
VisibleCondition	<p>Evaluates as “success” when a control is visible and “failed” when it is not. This does NOT evaluate the Control’s own Visible property because that prevents generating HTML. It evaluates the HTML generated to see if its styled to be visible.</p> <p><i>Usage:</i> Assign the ControlIDToEvaluate property to the control.</p>
EnabledCondition	<p>Evaluates as “success” when a control’s enabled state matches the IsEnabled property and “failed” when it is not.</p> <p><i>Usage:</i> Assign the ControlIDToEvaluate property to the control. Set IsEnabled to the desired enabled state.</p>
ClassNameCondition	<p>Evaluates as “success” when a control’s style sheet name matches the ClassName property and “failed” when it is not.</p> <p><i>Usage:</i> Assign the ControlIDToEvaluate property to the control. Set the style sheet class name in the ClassName property.</p>
ReadOnlyCondition	<p>Evaluates as “success” when a textbox’s readOnly state matches the IsReadOnly property and “failed” when it is not.</p> <p><i>Usage:</i> Assign the ControlIDToEvaluate property to the control. Set IsReadOnly to the desired readonly state.</p>
CompareToValueAttributeCondition	<p>Evaluates as “success” when an HTML attribute or style matches the properties shown below and “failed” when it is not.</p> <p><i>Usage:</i> Assign the ControlIDToEvaluate property to the control. Set AttributeName to the name of the HTML attribute or style (case sensitive). Set Value to the value to match. Set AttributeType to either Attribute or Style to determine where to find the AttributeName. Set Data Type to String, Integer, or Boolean to define the type of data to compare. Set Operator to the comparison for the Value property against the actual value in the attribute or style.</p>

Troubleshooting

Here are some issues that you may run into. Remember that technical support is available from support@PeterBlum.com. We encourage you to use this knowledge base first.

This guide contains problems specific to the **Peter's Interactive Pages** module. Please see the "Troubleshooting" section of the **General Features Guide** for an extensive list of other topics including "Handling JavaScript Errors" and "Common Error Messages".

None specific to this module. Please use the General Features Guide.

Technical Support and Other Assistance

PeterBlum.com offers free technical support. This is just one of the ways to solve problems. This section provides all of your options and explains how technical support is set up.

Troubleshooting Section of this Guide

This guide includes an extensive set of problems and their solutions. See "[Troubleshooting](#)". This information will often save you time.

Developer's Kit

The Developer's Kit is a free download that provides documentation and sample code for building your own classes with this framework. It includes:

- Developer's Guide - Overviews of each class with examples, step-by-step guides, and other tools to develop new classes.
- MSDN-style help file - Browse through this help file to learn about all classes and their members.
- Sample code in C# and VB.

You can download it from <http://www.peterblum.com/DES/DevelopersKit.aspx>.

PeterBlum.Com Forums

Use the forums at <http://www.peterblum.com/forums.aspx> to discuss issues and ideas with other users.

Getting Product Updates

As minor versions are released (5.0.1 to 5.0.2 is a minor version release), you can get them for free. Go to <http://www.PeterBlum.com/DES/Home.aspx>. It will identify the current version at the top of the page. You can read about all changes in the release by clicking "Release History". Click "Get This Update" to get the update. You will need the serial number and email address used to register for the license.

As upgrades are offered (v5.0 to v5.1 or v6), PeterBlum.com will determine if there is an upgrade fee at the time. You will be notified of upgrades and how to retrieve them through email.

PeterBlum.com often adds new functionality into minor version releases.

Technical Support

You can contact Technical Support at this email address: Support@PeterBlum.com. I (Peter Blum) make every effort to respond quickly with useful information and in a pleasant manner. As the only person at PeterBlum.com, it is easy to imagine that customer support questions will take up all of my time and prevent me from delivering to you updates and cool new features. As a result, I request the following of you:

- Please review the Troubleshooting section first. See "[Troubleshooting](#)".
- Please try to include as much information about your web form or the problem as possible. I need to fully understand what you are seeing and how you have set things up.
- If you have written code that interacts with my controls or classes, please be sure you have run it through a debugger to determine that it is working in your code or the exact point of failure and error it reports.
- If you are subclassing from my controls, I provide the DES [Developer's Kit](#) that includes the Developers Guide.pdf, Classes And Types help file, and sample files. *I can only offer limited assistance as you subclass because this kind of support can be very time consuming.* I am interested in any feedback about my documentation's shortcomings so I can continue to improve it.
- I cannot offer general ASP.NET, HTML, style sheet, JavaScript, DHTML, DOM, or Regular Expression mentoring. If your problem is due to your lack of knowledge in any of these technologies, I will give you some initial help and then ask you to find assistance from the many tools available to the .Net community. They include:

- www.asp.net forums and tutorials
- Google searches. (I virtually live in Google as I try to figure things out with ASP.NET.) <http://www.Google.com>. Don't forget to search the "Groups" section of Google!
- For DHTML, Microsoft provides an excellent guide at <http://msdn2.microsoft.com/en-us/library/ms533050.aspx>.
- For DOM, start with the DHTML guide. Topics that are also in DOM are noted under the heading "Standards Information"
- For JavaScript, I recommend <https://developer.mozilla.org/en/JavaScript/Reference>.
- <http://aspnet.4guysfromrolla.com/>, <http://www.aspalliance.com/>
- Books

As customers identify issues and shortcomings with the software and its documentation, I will consider updating these areas.

Table of Contents

PETER’S INTERACTIVE PAGES OVERVIEW 2

FieldStateControllers Overview3

CalculationController Overview3

TextCounter Overview3

ContextMenu and DropDownMenu Overview4

ChangeMonitor Overview.....4

Interactive Hints Overview4

Enhanced ToolTips Overview.....4

Enhanced Buttons Overview.....4

Direct Keystrokes to Click Buttons Overview.....5

FIELDSTATECONTROLLER AND MULTIFIELDSTATECONTROLLER CONTROLS..... 6

Features7

Using the FieldStateControllers.....8

 The Condition8

 Controls that run the FieldStateController9

 Controls To Change10

 Attribute Values To Change11

 Extending the Attributes with Your Own Code12

 Client-Side Function: The Change State Function.....12

 Server Side Event Handler.....13

 Updating Validators14

 Changing Visibility on a Complex Control15

 Solution.....15

 Toggling States16

 JavaScript: Running FieldStateControllers on demand17

 Controls That Have Child Controls18

 GetChild Method18

 Installing the GetChild Method19

Example: FieldStateController.....20

Example: MultiFieldStateController21

Adding the FieldStateController Control.....22

Adding the MultiFieldStateController Control26

Properties of FieldStateController And MultiFieldStateController31

 Invoke the Change Properties32

 Controls To Change Properties.....34

 Attributes To Change Properties.....35

 Properties of ConditionTrue and ConditionFalse36

Update Validators Properties	39
When to Use the Control Properties	40
Behavior Properties	42
FSCONCOMMAND AND MULTIFSCONCOMMAND CONTROLS.....	44
Features	45
Using the FSConCommand Controls.....	46
Controls that run the FSConCommand control.....	46
Controls To Change	47
Attribute Values To Change	48
Updating Validators	49
Changing Visibility on a Complex Control	50
Example: The DateTextBox control	50
Solution.....	50
Selectively Running the Control	51
Example: FSConCommand	52
Example: MultiFSConCommand.....	53
Adding the FSConCommand Control	54
Adding the MultiFSConCommand Control.....	56
Properties of FSConCommand And MultiFSConCommand	59
Invoke the Change Properties	60
Controls To Change Properties.....	62
Attributes To Change Properties.....	63
Update Validators Properties	66
When To Use The Control Properties.....	67
Behavior Properties	69
CALCULATIONCONTROLLER.....	70
Features	71
Using the CalculationController.....	72
Creating the Expression: The CalcItem classes	73
PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem.....	74
PeterBlum.DES.Web.WebControls.ConstantCalcItem	75
PeterBlum.DES.Web.WebControls.ListConstantsCalcItem.....	76
PeterBlum.DES.Web.WebControls.CheckStateCalcItem	77
PeterBlum.DES.Web.WebControls.ParenthesisCalcItem	79
PeterBlum.DES.Web.WebControls.ConditionCalcItem.....	80
PeterBlum.DES.Web.WebControls.CalcControllerCalcItem	82
PeterBlum.DES.Web.WebControls.TotalingCalcItem	83
PeterBlum.DES.BLD.DataFieldTotalingCalcItem	84
General Guidelines for CalcItem objects	85
Displaying The Result	86
Using the Result in Validators and Conditions	87
Using the Result in Your Server-Side Code	88
JavaScript: Running CalculationControllers On Demand	89
Adding the CalculationController Control.....	90

Properties on CalculationController	92
Calculating The Value Properties	93
Showing The Value Properties	98
When to Use the Control Properties	100
Behavior Properties	103
Properties on CalcItem Classes	105
Properties Common To All CalcItem Classes	106
Properties for the PeterBlum.DES.Web.WebControls.NumericTextBoxCalcItem Class.....	107
Properties for the PeterBlum.DES.Web.WebControls.ListConstantsCalcItem Class.....	108
Properties for the PeterBlum.DES.Web.WebControls.CheckStateCalcItem Class	111
Properties for the PeterBlum.DES.Web.WebControls.ConstantCalcItem Class	112
Properties for the PeterBlum.DES.Web.WebControls.ParenthesisCalcItem Class	113
Properties for the PeterBlum.DES.Web.WebControls.ConditionCalcItem Class	114
Properties for the PeterBlum.DES.Web.WebControls.CalcControllerCalcItem Class.....	118
Properties for the PeterBlum.DES.Web.WebControls.TotalingCalcItem Class	119
The GetColumnControl event handler	120
Properties for the PeterBlum.DES.BLD.DataFieldTotalingCalcItem Class	122
Adding Custom Code to a CalcItem	123
The Client-Side Function and the CustomCalcFunctionName Property	124
The Server Side Event Handler and CustomCalculation Property.....	125
INTERACTIVE HINTS	127
Features	128
When using Labels	128
When using PopupViews.....	128
Other ways to display Hints.....	129
Interactively Customizing the Hint Text.....	129
Using Interactive Hints	130
Displaying Hints: The PeterBlum.DES.Web.WebControls.HintFormatter Class	131
Page-Level Hint Settings: The PeterBlum.DES.Globals.WebFormDirector.HintManager Property	132
Showing Validation Errors In The Hints	132
Adding HintFormatters to the SharedHintFormatters Property	133
When using a PopupView: AddSharedHintPopupView()	134
When using a Label on the Page: AddSharedHintOnPage().....	136
Using Your Own HintFormatter definition: AddSharedHintFormatter()	138
Defining PopupViews.....	139
View an existing definition.....	140
Edit a definition	142
Add a definition	142
Rename a definition.....	142
Delete a definition.....	142
Creating your own Callouts	143
Adding your own Callouts to the PopupView Definition	144
Using PopupViews	145
Defining Hints shown on the Page.....	147
Using a Label.....	147
Using a Panel containing a Label.....	148
Customize How Hints Appear: The Formatter Function	149
Using Hints shown on the Page	150
Customize the Text of the Hint: The Text Function	151
Javascript functions: Show and Hide the Hint On Demand.....	152
Providing an Initialization Function	153
Adding a Hint to any Control Programmatically: PeterBlum.DES.Globals.WebFormDirector.AddHintToControl Method	154

Properties for the PeterBlum.DES.Web.WebControls.HintFormatter Class 156

Properties on the PeterBlum.DES.Globals.WebFormDirector.HintManager Property 160

Properties for the PeterBlum.DES.Web.WebControls.HintPopupView Class 163

- Overall Appearance Properties 164
- Header Properties..... 165
- Body Properties 168
- Footer Properties..... 170
- Callout Properties 172
- Positioning Properties 174
- Other Properties 175

ENHANCED TOOLTIPS 178

Features 179

Using Enhanced ToolTips 180

- HintManager.AddToolTipPopupViewToControl() method 181

TEXTCOUNTER CONTROL 182

Features 182

Using the TextCounter Control 183

- Connecting To a TextBox..... 183
- Establishing the Limits 184
- Setting the Text and Style Sheets..... 185
- Tokens in Messages..... 186

Adding a TextCounter Control 187

Properties of the TextCounter Control..... 189

- TextBox Properties 189
- Message Properties 191
- Appearance Properties 194
- Behavior Properties 197

CONTEXT MENU AND DROPDOWNMENU CONTROLS 198

Features 199

Using the Context Menu..... 200

- Overall Appearance 201
- Menu Commands: PeterBlum.DES.Web.WebControls.CommandMenuItem class..... 202

 - Providing a Script for your Command..... 203
 - Appearance of Menu Command Rows 205
 - Adding a PeterBlum.DES.Web.WebControls.CommandMenuItem to the ContextMenu..... 207
 - Properties for PeterBlum.DES.Web.WebControls.CommandMenuItem 210

- Menu Separators: PeterBlum.DES.Web.WebControls.SeparatorMenuItem class 212

 - Appearance of Menu Separator Rows 212
 - Adding a PeterBlum.DES.Web.WebControls.SeparatorMenuItem to the ContextMenu 213
 - Properties for PeterBlum.DES.Web.WebControls.SeparatorMenuItem..... 214

- Hint Rows: PeterBlum.DES.Web.WebControls.HintMenuItem class..... 215

 - Appearance of Menu Hint Rows..... 215
 - Adding a PeterBlum.DES.Web.WebControls.HintMenuItem to the ContextMenu 216

Properties for PeterBlum.DES.Web.WebControls.HintMenuItem.....	218
Popup Controls: PeterBlum.DES.Web.WebControls.MenuActivator class	219
Inserting Variables Into Your Scripts	220
Adding a PeterBlum.DES.Web.WebControls.MenuActivator to the ContextMenu	221
Properties for PeterBlum.DES.Web.WebControls.MenuActivator	223
Using the DropDownMenu control	224
Customizing the Toggle button.....	224
Adding a ContextMenu	225
Complete Example.....	227
Adding a DropDownMenu	228
Complete Example.....	230
Properties of the ContextMenu.....	231
Menu Structure Properties	231
Menu Item Appearance Properties.....	232
Overall Appearance Properties	235
Popup Behavior Properties	236
Behavior Properties	237
Popup Location Properties.....	243
Properties of the DropDownMenu	244
Toggle Control Properties.....	245
Popup Panel Properties	247
ASP.NET Representation of Nested Properties	247
Popup Behavior Properties	248
Behavior Properties	249
ENHANCED BUTTONS	250
Features	250
Using the Enhanced Buttons	251
Adding an Enhanced Button.....	252
Properties on Enhanced Buttons	254
Behavior Properties	254
ChangeMonitor Properties	255
Validation Properties	256
Hint and ToolTip Properties	257
Appearance Properties	260
Programmatically Adding These Features to Non-DES Buttons	261
The PeterBlum.DES.SubmitBehavior Class	262
Properties	262
Constructors	264
CHANGEMONITOR	265
Features	266
Using the ChangeMonitor	267

The ChangeMonitor Property 268

Changing the State of Buttons 269

 Using server side code 269

Making Data Entry Controls Notify Changes 270

 Using the NativeControlExtender 270

 Using the ChangeMonitor.RegisterForChanges() Method 270

Using the FieldStateController 271

 The PeterBlum.DES.Web.WebControls.ChangeMonitorCondition Class 271

Using your own JavaScript Code 272

Validation Group and ChangeMonitor Groups 273

Properties of the PeterBlum.DES.Globals.WebFormDirector.ChangeMonitor 274

ChangeMonitor Server Side Methods 276

ChangeMonitor JavaScript Functions 278

DIRECT KEYSTROKES TO CLICK BUTTONS 279

Using the NativeControlExtender 280

Using the RegisterKeyClicksControl() Method 281

 PeterBlum.DES.Globals.WebFormDirector.RegisterKeyClicksControl Method 282

CUSTOM SUBMIT FUNCTION 283

Using The Custom Submit Function 283

Page-Level Properties 284

ADDITIONAL TOPICS FOR USING THESE CONTROLS 285

PAGE LEVEL PROPERTIES AND METHODS 286

Properties on PeterBlum.DES.Globals.WebFormDirector 286

 Validation Properties 288

JAVASCRIPT SUPPORT FUNCTIONS 289

General Utilities 289

ADDING YOUR JAVASCRIPT TO THE PAGE 292

Embedding the ClientID into your Script 292

Debugging Your JavaScript 293

THE CONDITION CLASSES 294

Evaluating Textboxes and other controls with textual values 295

Evaluating Listboxes and other controls with index values 297

Evaluating checkboxes, radiobuttons, and other controls 298

Evaluating non-data value states of controls 299

TROUBLESHOOTING 300

TECHNICAL SUPPORT AND OTHER ASSISTANCE..... 301

- Troubleshooting Section of this Guide 301
- Developer’s Kit..... 301
- PeterBlum.Com Forums 301
- Getting Product Updates 301
- Technical Support..... 301