PETER'S PROFESSIONAL VALIDATION USER'S GUIDE

peterblum.com
Start with better controls. Finish with better sites.

Click on any of these topics to jump to them:

- Getting Started
- Learn online
- Validator Controls ..... Adding ..... Properties
  - ABARoutingNumberValidator, CharacterValidator, CheckStateValidator
  - CompareToStringsValidator, CompareToValueValidator, CompareTwoFieldsValidator
  - CountSelectionsValidator, CountTrueConditionsValidator, CreditCardNumberValidator
  - CustomValidator, DataTypeCheckValidator, DifferenceValidator
  - DuplicateEntryValidator, EmailAddressValidator, IgnoreConditionValidator
  - ListSizeValidator, MultipleRequiredControlsValidator, MultiConditionValidator
  - RangeValidator, RegexValidator
  - RequiredListValidator, RequiredSelectionValidator, RequiredTextValidator
  - SelectedIndexValidator, SelectedIndexRangesValidator, TextLengthValidator
  - UnwantedWordsValidator, WordCountValidator
  - Changing When the Validator is Evaluated ..... Enabler ..... Validation Groups
  - Defining the Error Message and Associated Labels ..... Properties
  - ErrorFormatters: Customizing the Appearance of the Error Message ..... Using ..... Properties
  - Other Validator Properties that Customize the Appearance
- Drawing The User's Attention To The Error ..... Change style on error ..... Hilite nearby fields
- Submitting the Page ..... Adding ..... Buttons ..... Client Side
- Additional Validation Topics ..... Best Practices ..... Working with ASP.NET Controls

- ValidationSummary Control ..... Adding ..... Using ..... Properties
- CombinedErrorMessages Control ..... Adding ..... Using ..... Properties
- RequiredFieldMarker Control ..... Adding ..... Properties
- RequiredFieldsDescription Control ..... Adding ..... Properties

- Page Level Properties and Methods
- JavaScript Support Functions
- Troubleshooting
- Table of Contents

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

License Information

This document includes information for all aspects of validation found in Peter's Data Entry Suite. If you licensed the complete Suite, you have all features found in this User's Guide. If you licensed the Peter's Professional Validation module, some features are not available to you unless you also licensed the Peter's More Validators module.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Getting Started

Validation is far more than just controls which evaluate the values of your textboxes. It involves your buttons and offers other tools to communicate with the user (like the ValidationSummary Control.) It also needs a little code, specifically testing that the page is valid before you save the data submitted by the user. As a result, Validation is called the "DES Validation Framework" here. (The native ASP.NET validation system is called the "Native Validation Framework" here.)

If you are unfamiliar with the Native Validation Framework, here are the concepts used by both Native and DES.

- A **Validator control** looks at one or more data entry controls on the page. As those controls are changed, the Validator control evaluates the values of the fields against a rule, called a "condition". If the data doesn't match the condition, the Validator shows a message describing the error. If it matches the condition, the error message is hidden. The error message is defined as a property of the Validator control and positioned where you place the control on the page.
- The **ValidationSummary control** shows a list of all errors on the page in one place and appears when the user submits a page.
- The **buttons on the page** cause the Validators to show or hide themselves unless you turn the feature off. The page will not submit while there are errors.
- Validation happens on the client-side to avoid unnecessary post backs. However, every time a page is posted back, the **server side will also validate** because users are running browsers with JavaScript unavailable or the Validators themselves have JavaScript unavailable.

Now let's look through some of the major concepts.

Click on any of these topics to jump to them:

- Evaluating data: The Validator controls and their Condition objects
- Determining When Validators Fire
- Communicating Errors
- Other Controls in the DES Validation Framework
- Step-by-step instructions for setting up a page
- Validation Best Practices
- Additional Validation Topics
- Learn online

Here are some additional resources:

- **Getting Started** document in the Start menu
- **Tutorials** document in the Start menu

Peter's Business Logic Driven UI ("BLD") Users

The **BLD** framework already integrates Validator controls within its Field Templates engine. By using a ValidationAttribute within your Entity classes, you will get the appropriate DES Validator. Use the BLDDataFieldValidators control to customize the ErrorFormatter and Error Messages.

You will still want to add a ValidationSummary Control into your pages and use DES's Buttons.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Evaluating data: The Validator controls and their Condition objects

DES includes 27 Validator web controls to evaluate all kinds of data entry controls and situations. From requiring text (RequiredTextValidator) to demanding a valid credit card number (CreditCardNumberValidator) to building complex expressions out of other validators (MultiConditionValidator), you have plenty to choose from.

Each Validator control has the same features for communicating with the user, such as the **ErrorMessage** and **ErrorFormatter** properties. These will be described in later topics. Where each Validator differs is in its *evaluation rule*, and as a result, they have unique properties to support their evaluation rule.

Here's a key concept: evaluation rules are actually offloaded into objects called Conditions. These objects only check the evaluation rule. They cannot format or communicate the error. Condition objects are used throughout DES. The MultiConditionValidator uses them to create complex boolean logic. The **Enabler** property on Validators and FieldStateControllers uses them to enable and disable their control based on the state of other controls. Because they are so important, the following charts identify both the Validator and its Condition object.

Click on any of these topics to jump to them:

- Evaluating Textboxes and other controls with textual values
- Evaluating the Listbox, DropDropList, RadioButtonList, and CheckBoxList
- Evaluating CheckBoxes, RadioButtons, and other controls
- Evaluating non-data value states of controls
- Evaluating all other cases

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Evaluating Textboxes and other controls with textual values

Class name (in the name space PeterBlum.DES.Web.WebControls)	Description and usage
RequiredTextValidator and RequiredTextCondition	Tests if a field's textual value is "blank" or not. You can define what text represents "blank". *Usage:* Assign the **ControlIDToEvaluate** property to the control. *More*
CompareToValueValidator and CompareToValueCondition	Tests a field's textual value against another value that you supply. You determine the data type of the values and the operator to compare the two. *Usage:* 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. *More*
CompareTwoFieldsValidator and CompareTwoFieldsCondition	Tests the values of two fields to each other. You select the operator that is used in the comparison. *Usage:* 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. *More*
RegexValidator and RegexCondition	Tests a field's textual value against a regular expression. Use it to confirm the text fits a pattern such as a phone number, email address, or postal code. *Usage:* Assign the **ControlIDToEvaluate** property to the control. Set **Expression** to the regular expression. Use properties of **NotCondition**, **CaseInsensitive** and **Multiline** to enhance the logic. Design mode for the **Expressions** property includes predefined expressions. *More*
DataTypeCheckValidator and DataTypeCheckCondition	Tests a field's textual value to confirm it represents a particular data type. *Usage:* Assign the **ControlIDToEvaluate** property to the control and **DataType** to the data type. *More*
TextLengthValidator and TextLengthCondition	Evaluates the number of characters in a textbox against a minimum and/or maximum. *Usage:* Assign the **ControlIDToEvaluate** property to the textbox. Set **Minimum** and **Maximum** to the valid character range. *More*
CharacterValidator and CharacterCondition	Tests a field's textual value to confirm that each character is in a character set that you specify. For example, you require a password to be letters, digits, and underscores only. This Condition doesn't care about the order or pattern of characters. *Usage:* Assign the **ControlIDToEvaluate** property to the textbox. Define the characterset 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. *More*
RangeValidator and RangeCondition	Tests a field's textual value is between a minimum and maximum value. You determine the data type of the values. *Usage:* 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). *More*

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Class name (in the name space PeterBlum.DES.Web.WebControls)	Description and usage
CompareToStringsValidator and CompareToStringsCondition	Tests a field's textual value against a list of strings. If the text matches one of the strings, it is valid. *Usage:* 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. If you want to use a datasource to populate the **Items** collection, use the **DataSource**, **DataMember**, and **DataTextField** properties. *More*
MultipleRequiredControlsValidator and MultipleRequiredControlsCondition	Tests several fields to determine how many have been filled in (not "blank"). It supports these rules: All, All or none, Only one, At least one, and a range. *Usage:* 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). *More*
DifferenceValidator and DifferenceCondition	Evaluates two textbox fields to determine if the difference between their two values is a certain number apart from each other. You select the data type (including dates), a value to compare and the operator. For example, test that two dates are less than 60 days apart. *Usage:* Assign the **ControlIDToEvaluate** and **SecondControlIDToEvaluate** properties to the controls to compare. Set **DataType** to the type the text represents. Set **Operator** to the comparison operator and **DifferenceValue** to the number of units they are apart. *More*
WordCountValidator and WordCountCondition	Evaluates the number of words in a textbox against a minimum and/or maximum. *Usage:* Assign the **ControlIDToEvaluate** property to the textbox. Set **Minimum** and **Maximum** to the valid character range. *More*
UnwantedWordsValidator and UnwantedWordsCondition	Compares a list of words to the text. It reports an error when any of the words is found within the text. *Usage:* Assign the **ControlIDToEvaluate** property to the textbox. For each string to compare, add an UnwantedWordsItem object to its **Items** collection. The evaluation rules are in the properties **WholeWord**, **CaseInsensitive**, and **BrokenWords**. If you want to use a datasource to populate the **Items** collection, use the **DataSource**, **DataMember**, and **DataTextField** properties. *More*
DuplicateEntryValidator and DuplicateEntryCondition	Determines if three or more textboxes, dropdownlists, or lists have a matching value. It reports an error when a duplicate is found. *Usage:* Specify the data entry controls within the **ControlsToMatch** property with LabeledControl objects, where you identify the control and optionally its label. Use the **MatchMode** property to specify a rule for matching: exact match, case insensitive match, or use the **SelectedIndex** property (only for ListBox and DropDropList controls). Use the **IgnoreUnassigned** property to ignore blank controls. *More*
EmailAddressValidator and EmailAddressCondition	Used on Email address fields. Confirms that the pattern is valid on both client and server side. *Usage:* Assign the **ControlIDToEvaluate** property to the control. If you want to allow multiple email addresses in the same textbox, set **MultipleAddressesAllowed** to `true`. You can define the delimiter character in **DelimiterCharacter**, which defaults to a semicolon. You can allow spaces after the delimiter when **DelimiterAllowsSpaces** is `true`. *More*

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Class name (in the name space PeterBlum.DES.Web.WebControls)	Description and usage
CreditCardNumberValidator and CreditCardNumberCondition	Confirms that the pattern is valid for credit card numbers. Its evaluation rules confirm that the text is all digits, follows Luhn's formula, and if desired matches specific brands of credit cards. It does not provide credit card number approval. *Usage:* Assign the **ControlIDToEvaluate** property to the control. *More*
ABARoutingNumberValidator and ABARoutingNumberCondition	Confirms that the pattern is valid for ABA routing numbers such as check routing numbers. It confirms that the text is always 9 digits that are confirmed with a checksum rule defined by the American Bankers Association. *Usage:* Assign the **ControlIDToEvaluate** property to the control. *More*

If none of these work for you, see "Evaluating all other cases".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Evaluating the Listbox, DropDropList, RadioButtonList, and CheckBoxList

These data entry controls have a **SelectedIndex** property which is an integer that identifies the position of the selected item in the list, where 0 is the first item. The Validators shown here evaluate the SelectedIndex property.

Note: Many List style controls also have a textual value as defined in <asp:ListItem value="here">label</asp:ListItem>. You can use the Condiitons described in "Evaluating Textboxes and other controls with textual values" if you want to evaluate them.

Class name (in the name space PeterBlum.DES.Web.WebControls)	Description and usage
RequiredListValidator and RequiredListCondition	Tests if a list-style control is "unselected". *Usage:* Assign the **ControlIDToEvaluate** property to the control. If the first item of the list indicates "no selection", set **UnassignedIndex** to 0. *More*
SelectedIndexValidator and SelectedIndexCondition	Evaluates a specific index in the list style control to see if its selected or unselected. *Usage:* 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. *More*
SelectedIndexRangesValidator and SelectedIndexRangesCondition	Evaluates the selected index against a list of valid indices. *Usage:* 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. *More*
CountSelectionsValidator and CountSelectionsCondition	Evaluates CheckBoxLists and multi-selection ListBoxes to determine how many items are selected and compares the result of a range. *Usage:* Assign the **ControlIDToEvaluate** property to the control. Set the range with the **Minimum** and **Maximum** properties. *More*
DuplicateEntryValidator and DuplicateEntryCondition	Determines if three or more textboxes, dropdownlists, or lists have a matching value. It reports an error when a duplicate is found. *Usage:* Specify the data entry controls within the **ControlsToMatch** property with LabeledControl objects, where you identify the control and optionally its label. Use the **MatchMode** property to specify a rule for matching: exact match, case insensitive match, or use the **SelectedIndex** property (only for ListBox and DropDropList controls). Use the **IgnoreUnassigned** property to ignore blank controls. *More*
ListSizeValidator and ListSizeCondition	Evaluates the number of items shown in a listbox or dropdownlist. Typically used when the user interface allows adding and removing items in a list, where you have a minimum or maximum for the number of items shown. *Usage:* Assign the **ControlIDToEvaluate** property to the control. Set the range with the **Minimum** and **Maximum** properties. *More*

If none of these work for you, see "Evaluating all other cases".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Evaluating CheckBoxes, RadioButtons, and other controls

Class name (in the name space PeterBlum.DES.Web.WebControls)	Description and usage
CheckStateValidator and CheckStateCondition (CheckBox and RadioButton)	Evaluates the mark within a checkbox or radiobutton. *Usage:* Assign the **ControlIDToEvaluate** property to the control. Set Checked to true to require the checkbox to be marked. *More*
SelectedIndexValidator and SelectedIndexCondition (CheckBoxList and RadioButtonList)	Evaluates a specific index in the CheckBoxList or RadioButtonList control to see if its selected or unselected. *Usage:* 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. *More*
SelectedIndexRangesValidator and SelectedIndexRangesCondition (CheckBoxList and RadioButtonList)	Evaluates the selected index against a list of valid indices. *Usage:* 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. *More*
CountSelectionsValidator and CountSelectionsCondition (CheckBoxList)	Evaluates a CheckBoxList to determine how many items are selected and compares the result of a range. *Usage:* Assign the **ControlIDToEvaluate** property to the control. Set the range with the **Minimum** and **Maximum** properties. *More*
RequiredSelectionValidator and RequiredSelectionCondition (Calendar, MultiSelectionCalendar, MonthYearPicker, TimePicker, DataList, and ListView)	For controls whose state isn't from a textbox, list, or button. Evaluates when the control has a selected element. *Usage:* Assign the **ControlIDToEvaluate** property to the control. *More*

If none of these work for you, see "Evaluating all other cases".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Evaluating non-data value states of controls

There are no Validator controls here, only Condition objects. These are used by the MultiConditionValidator, CountTrueConditionsValidator, and the **Enabler** property to handle cases based on the state of non-data oriented properties like visibility and enabled.

Class name (in the name space PeterBlum.DES.Web.WebControls)	Description and usage
VisibleCondition	Evaluates a control's visible state. 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. *Usage:* Assign the **ControlIDToEvaluate** property to the control. *More*
EnabledCondition	Evaluates a control's enabled state. *Usage:* Assign the **ControlIDToEvaluate** property to the control. Set **IsEnabled** to the desired enabled state. *More*
ClassNameCondition	Evaluates the a control's style sheet name. *Usage:* Assign the **ControlIDToEvaluate** property to the control. Set the style sheet class name in the **ClassName** property. *More*
ReadOnlyCondition	Evaluates a textbox's readonly state. *Usage:* Assign the **ControlIDToEvaluate** property to the control. Set **IsReadOnly** to the desired readonly state. *More*
CompareToValueAttributeCondition	Evaluates an HTML attribute or style's value. *Usage:* 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 **DataType** 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. *More*

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Evaluating all other cases

Class name (in the name space PeterBlum.DES.Web.WebControls)	Description and usage
MultiConditionValidator and MultiCondition	Builds a boolean expression from *any* Condition objects, such as "RequiredTextCondition on TextBox1 AND CheckStateCondition on CheckBox1". This Validator often replaces the need for custom Validators. *Usage:* Add each Condition object as a child object of the **Conditions** collection. Set the **Operator** property to either AND or OR. *More*
CountTrueConditionsValidator and CountTrueConditions	Lets you define a list of Condition objects to evaluate a number of fields. Evaluates how many of those conditions are true and compares them to a minimum and maximum. A common usage is to count the number of marked CheckBoxes in a column of a DataList, ListView, Repeater, or GridView. *Usage:* Add each Condition object as a child object of the **Conditions** collection. Set the **Minimum** and **Maximum** properties to the range. *More*
CustomValidator and CustomCondition	Build your own evaluation rules. It supports client- and server-side evaluations. *Usage:* On the server side, create a method that evaluates data and sets the args.IsMatch property to true or false. Assign that method to the **ServerCondition** event. On the client-side, write a function that evaluates data and returns 1 ("success"), 0 ("failed"), and -1 ("cannot evaluate"). Assign the function name to the **CustomEvalFunctionName** property. *More*
IgnoreConditionValidator (there is no Condition class)	This Validator has no condition of its own. It is used to hold an error message that you manually show by setting its **IsValid** property `false` in some program code. While the CustomValidator forces you to move some of your validation logic into its own event handler, this Validator lets you keep all of your logic in one area. *Usage:* In your code that needs to report an error, create and add the IgnoreConditionValidator to the page with its **ErrorMessage** property assigned and **IsValid** = false. *More*

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Determining When Validators Fire

Frequently you must limit when a Validator fires. DES provides these rules:

- Validation Groups – Associates a button with a group of validators it fires by a unique "group name". This allows several submit buttons on the page to validate their own data entry controls. For example, a login area has its own Submit button. Assign each Validator and the Submit button to the group "Login" and it will only validate the login name and password fields, leaving the other Validators on the page untouched.

  Define a unique name for each group and assign it to the **Group** property on the button and each Validator control it invokes.

  Be sure to use DES's own button controls instead of the native ones. See "DES's Button, LinkButton, and ImageButton Controls". If you need to use the native ones, see "Using Native Button Controls to Submit the Page".

- The Enabler Property – The **Enabler** property on each Validators allows it to enable and disable itself based on any Condition you want. Conditions are defined using Condition objects. See "Evaluating data: The Validator controls and their Condition objects".

  For example, suppose you have a TextBox that is only used when a checkbox is marked. So set **Enabler** to the CheckStateCondition (from the CheckStateValidator).

There are several more rules available. See "Other Properties That Disable Validation" for details.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Communicating Errors

A Validator's task is to show an error message to the user when the condition indicates that the wrong data was found. Whenever you add text to the web page, you have your own ideas of its formatting. For example, you apply a style sheet that changes the text color to red or you add the ⚠ image to the left of the message.

With the Native Validation Framework, you have to embed HTML tags into the error message to perform some of the formatting. DES gives you properties that introduce the formatting gracefully without affecting the text of the error message. It also offers many new ways to attract the user's attention to the error.

- **ErrorMessage** and **SummaryErrorMessage** properties - Error Messages are the text that is shown to the user when a condition indicates the wrong data was found. There are two error messages on each Validator: one shown at the location of the Validator; the other shown in the ValidationSummary Control. See "Defining the Error Message".
    - **Tokens** – Each Validator control defines tokens that you can enter in the error messages. For example, "You have entered {COUNT} characters which exceeds {MAXIMUM} by {EXCEEDS}." Tokens really become powerful tools when your site must have consistent messages. Your programmers must use the exact strings you define for each case. They simply setup the properties appropriate for the condition and the label and let tokens embedded in the error messages customize the text. See "Tokens in Error Messages" for details.
    - **Standardized Error Messages** - You can define standard error messages in .resx files, a database, or other data sources using the String Lookup System. When the programmer selects an error message, they use a "Lookup ID" to the error message and never need to fill in the actual error message. Once done this way, you can change the error messages in one place and update the entire site. See "String Lookup System" in the **General Features Guide**.
    - **Localized Error Messages** - Error messages can be localized using the String Lookup System. Use the .resx files, a database, or other data source with the "Lookup ID" to define your messages in multiple cultures. See "String Lookup System" in the **General Features Guide**.
- **ErrorFormatters** – There are far more ways to show an error than with a simple label. DES supplies five "Error Formatters" and is designed to allow you to create your own. See "ErrorFormatters: Customizing the Appearance of the Error Message".

  The ErrorFormatter is assigned to the Validator's **ErrorFormatter** property. Its most important property is **Display**, with these values:
    - `Static` – When hidden, the validator uses up space of the page enough to fit the error message. This is the default.
    - `Dynamic` – When hidden, the Validator does not use up space to fit its error message. Use this when you have several validators side-by-side.
    - `None` – Always hides the error message. Use this only if you have a ValidationSummary Control to show the errors.
- **Drawing The User's Attention to the Error**– When there is an error as the field is changed or the page is submitted, there are several ways to draw the user's attention to the error.
    - Put the focus on the field with the error.
    - Change the style of the field with the error. For example, change the background to red.
    - Put up an alert box with the error message.
    - Change the style of the label or an enclosing field. For example, change the font color of the label and change the background color and border of an enclosing to `<div>`.
    - The ErrorFormatter can blink to get the user's attention. This is especially effective when the user hits Submit and you want to draw their eye to the field with the error.

  See "Drawing The User's Attention To The Error" for details.
- **NoErrorFormatter property** - When no error is shown, you can still have the Validator display text, HTML or an image. It can be shown at different times, such as any time there is no error, or only after the user corrects a validation error.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

- **ValidationSummary Control** – Displays a list of all errors on the page in one place, at the location of this control. It has extensive formatting capabilities including the ability *not* to show the errors, but to just show a message like "Fix the errors". This control is often used in place of having Validators show their individual error messages by setting the **Validator.ErrorFormatter.Display** property to `None`.

  See "ValidationSummary Control".

- **CombinedErrorMessages Control** –The CombinedErrorMessages control lets you combine error messages from several Validators under one ErrorFormatter. It has several uses:

  - When you are using an error formatter with an image, like ⚠, and the field may show several of these at the same time, you will normally see multiple images. That looks strange. The CombinedErrorMessages control will show just one image and display all of the relevant error messages.
  - When you want to show all error messages in one place, add the CombinedErrorMessages control where you want to show the error. The idea is similar to a ValidationSummary control except it appears as fields are changed and its formatting matches other Validators.

  See "CombinedErrorMessages Control".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Other Controls in the DES Validation Framework

Click on any of these topics to jump to them:

- Submitting The Page
- Showing the "Required Field" Marker
- Supported data entry controls
- Working within Grid Controls

Submitting The Page

The action of submitting the page is a critical part to validation. On the client-side it reviews all fields, including those that did not get edited, to report errors and block page submission. On the server-side, it does the same but provides additional defenses because JavaScript can be turned off, the browser may not support the client-side scripts, and hackers will try to attack your site by working around the client-side validation.

The buttons, hyperlinks, menus and other controls that submit the page run validation. They can evaluate validators based on a Validation Group name or all groups (group name= "*"). When they do this on the client-side, DES prevents submitting the page until all errors are addressed. On the server-side, DES's buttons can validate for you or you can call `PeterBlum.DES.Globals.WebFormDirector.Validate()` to validate. You should always test the **PeterBlum.DES.Globals.WebFormDirector.IsValid** property is `true` inside your postback event handler method before saving data. See "Submitting the Page: Server-Side Validation".

ALERT: New users often overlook server side validation, leaving their site vulnerable to bad data and hacking. Always set up server side validation on every submit control that needs validation.

DES must enhance the existing page submission controls to do handle client-side validation:

- **Buttons** - DES provides replacements for the Button, LinkButton, and ImageButton controls. These direct subclasses of Microsoft's controls introduce capabilities like a property for validation groups and the ability to automatically validate on both client and server sides. See "DES's Button, LinkButton, and ImageButton Controls".

  You can use the Microsoft buttons too, if you connect them to DES using the NativeControlExtender control. See "Using Native Button Controls to Submit the Page".

- **Buttons in Grids** - DES provides replacements for the ButtonColumn and EditCommandColumn controls in the DataGrid and CommandField and ButtonField in the GridView. These direct subclasses of Microsoft's controls introduce capabilities like a property for validation groups and the ability to automatically validate on both client and server sides. See "DES's Submit Controls For The DataGrid" and "DES's Submit Controls For The GridView and DetailsView".

- **Menu control** – The native ASP.NET Menu control can run client-side validation on selected menu items with the NativeControlExtender control.

- **Third party menus and toolbar controls** – See the **Using Third Party Controls** guide.

- **AutoPostBack** – When you use the AutoPostBack feature on a data entry control, it makes sense to validate it first to avoid an unnecessary post back. DES's TextBoxes do this automatically. If you use any other control, use the NativeControlExtender control. See "Using Validation with AutoPostBack".

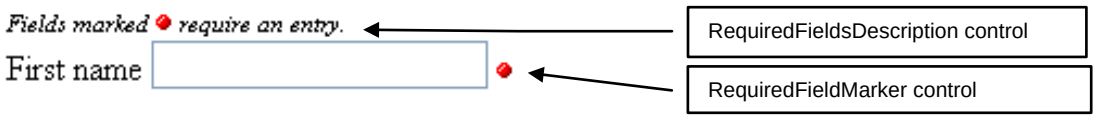
Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Showing the "Required Field" Marker

The "required field marker" is a standard identifier that indicates the associated control is required. Typical forms place a symbol, like "*" or ●, next to each field that requires an entry. They also place a description of the required field marker on the page. DES makes setting up the required field marker easy:

- **RequiredFieldsDescription Control** – Use the RequiredFieldsDescription control to place the description of the marker on the page. While it's easy to define a text label, this control retrieves the text from global values. This assures you that the description is standardized throughout your site. It supports localization. See "RequiredFieldsDescription Control".
- **RequiredFieldMarker Control** – Use the RequiredFieldMarker control to place the actual marker next to the field. This standardizes the visual image, so that you define it once. The global definition can be textual or an image. It supports localization. See "RequiredFieldMarker Control".
- **ShowRequiredFieldMarker Property** – Each Validator includes the **ShowRequiredFieldMarker** property. When set to `true`, it automatically inserts a RequiredFieldMarker control to its left.



Supported data entry controls

- DES's Validators support all data entry controls in the System.Web.UI.WebControls and System.Web.UI.HtmlControls namespaces. It also supports any subclasses of those controls that you or a third party develops. For a detailed look through the ASP.NET controls, see "Using the DES Validation Framework with each ASP.NET web control".
- DES's Validators support controls with the ValidationPropertyAttribute. This is the same attribute used by Microsoft's Validation controls and allows your new custom controls to be supported automatically. In addition, specialized DES Validators can use any control they desire, not just those with the ValidationPropertyAttribute. For example, the CheckStateValidator uses the CheckBox control, which does not support the ValidationPropertyAttribute.
- Any third party control described in the **Using Third Party Controls Guide** including those from Telerik and Infragistics.
- Validators and their conditions can easily support third party controls that are textboxes, radiobuttons, checkboxes, listboxes and dropdownlists that do not use the ValidationPropertyAttribute. See the section "Supporting Other Data Entry Controls" in the **Using Third Party Controls Guide**.

Working within Grid Controls

DES works well in most Grid controls that provide an editable row mode where you are able to add any webcontrols to the row. Microsoft's GridView is one example.

Telerik RadGrid and Infragistics **UltraWebGrid** require some additional setup. See the **Using Third Party Controls.pdf** document.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Adding Validation to a WebForm

Here is the general approach to add validation to your web form. Many users miss steps 1, 2, 11, 12, and 16 which are important.

Big Picture

The step-by-step instructions below provide extensive details, giving you a wide variety of options. Here is the big picture of what you should do:

- Add Validator controls, setting their condition rules, formatting, and error messages. See "Validator Controls".
- Use DES's Button controls to submit the page. For any button that should not validate, set its **CausesValidation** property to `false`. See "Submitting the Page: Server-Side Validation".
- In the post back event handler where you save data, first confirm **PeterBlum.DES.Globals.WebFormDirector.IsValid** is `true`.
- If desired, add the ValidationSummary Control and Required Field Marker Controls.
- If desired, add other ways to draw the user's attention to the error. See "Drawing The User's Attention To The Error".
- When using AJAX, be sure DES is properly configured. See "Using these controls with AJAX" in the **General Features Guide**.

Learn online

Step-by-Step

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 your web form already has controls from the Native Validation Framework or native buttons, you can quickly convert it to use DES equivalent controls with the **Web Application Updater** program. Use the **Convert native controls to their DES equivalents** option.
3. Determine the type of Validator that you need. See "Evaluating data: The Validator controls and their Condition objects".
4. Select the location on the page where the Validator will appear. It can be moved later if needed. If you have limited space, the Validator has formatting options to greatly reduce its size. So position it at your first choice of location.
5. Add a Validator control to the page.

   Visual Studio and Visual Web Developers Design Mode Users

   Drag the Validator control from the Toolbox onto your web form.

   Text Entry Users

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

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

   For example, to add the RequiredTextValidator, use:

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

   Programmatically creating the Validator control

   - Identify the control which you will add the Validator control to its **Controls** collection. Like all ASP.NET controls, the Validator 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 control class. The constructor takes no parameters.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- Assign the **ID** property.
- Add the Validator control to the **Controls** collection.

In this example, the RequiredTextValidator is created with an **ID** of "TextBox1RequiredValidator". It is added to PlaceHolder1.

[C#]

```
PeterBlum.DES.Web.WebControls.RequiredTextValidator vValidator =
   new PeterBlum.DES.Web.WebControls.RequiredTextValidator();
vValidator.ID = "TextBox1RequiredValidator";
PlaceHolder1.Controls.Add(vValidator);
```

*Note: The namespace for these controls is PeterBlum.DES. If you prefer, add a **using** clause to that namespace on your form.*

[VB]

```
Dim vValidator As PeterBlum.DES.Web.WebControls.RequiredTextValidator = _
   New PeterBlum.DES.Web.WebControls.RequiredTextValidator()
vValidator.ID = "TextBox1RequiredValidator"
PlaceHolder1.Controls.Add(vValidator)
```

*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.

6. Set the properties associated with the Condition (evaluation rule) for the Validator.

Click on the Validator in this chart to see its Condition properties:		
ABARoutingNumberValidator	CharacterValidator	CheckStateValidator
CompareToStringsValidator	CompareToValueValidator	CompareTwoFieldsValidator
CountSelectionsValidator	CountTrueConditionsValidator	CreditCardNumberValidator
CustomValidator	DataTypeCheckValidator	DifferenceValidator
DuplicateEntryValidator	EmailAddressValidator	IgnoreConditionValidator
ListSizeValidator	MultipleRequiredControlsValidator	MultiConditionValidator
RangeValidator	RegexValidator	
RequiredListValidator	RequiredSelectionValidator	RequiredTextValidator
SelectedIndexValidator	SelectedIndexRangesValidator	TextLengthValidator
UnwantedWordsValidator	WordCountValidator	

7. Set the error message using the **ErrorMessage** property. If you are using the ValidationSummary control and a different error message is needed, also use the **SummaryErrorMessage** property. There are options for tokens, localization and standardized message libraries. See "Defining the Error Message".

CONTINUED ON THE NEXT PAGE

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

8. Set the appearance of the Validator in the location of the Validation control using the **ErrorFormatter** property. See "ErrorFormatters: Customizing the Appearance of the Error Message".
   - Select an overall format from one of styles provided: Text, Hyperlink with Alert, Image with Tooltip, Image with Alert, and PopupView. If you have space limits, avoid the Text ErrorFormatter.

     Note: Within the Properties Editor, you can customize the default properties for the ErrorFormatters. You can even create new names for types with various configurations. That way, you can just select the ErrorFormatter type from the list to set it up quickly.
   - If you do not want to show the Validator except in the ValidationSummary, set the **ErrorFormatter.Display** property to `None`.
   - If there are several Validators whose messages you'd like to combine under one ErrorFormatter, also add a CombinedErrorMessages Control.
   - If you want to indicate that an error has been fixed, use the **NoErrorFormatter** property.
9. Set cases where your Validator should not be evaluated.
   - When you have two or more submit controls that are associated with their own Validators, use **validation groups** by setting the **Group** property to the name of the group. See "Validation Groups".
   - When the Validator should not validate unless there is a specific setting on the page, use the **Enabler** property to specify the rule that enables it. See "The Enabler Property".
   - The **Enabled** property can fully disable the Validator. The **EventsThatValidate** property can determine what client-side events cause validation. See "Other Properties That Disable Validation".
10. You have finished the setup of the Validator control. Return to step 3 to add another. Otherwise, continue to establish other behaviors on the page with the remaining steps.
11. Make your submit controls run DES validation on the client and server side.
   - For buttons, use DES's Button, LinkButton, and ImageButton controls. See "DES's Button, LinkButton, and ImageButton Controls".
   - For the DataGrid, use DES's EditCommandColumn and ButtonColumn controls. See "DES's Submit Controls For The DataGrid".
   - For the GridView and DetailView control, use DES's ButtonField and CommandField controls. See "DES's Submit Controls For The GridView and DetailsView".
   - If you don't want to use DES's submit controls, you need to hookup code to your existing buttons to force client-side validation. See "Using Native Button Controls to Submit the Page" and "Using the Menu Control to Validate the Page". On the server side, your **Click** post back event handler should call `PeterBlum.DES.Globals.WebFormDirector.Validate()`.
   - If a submit control should not validate, set its **CausesValidation** property to `false`. For example, a Cancel or Back button.
   - If you are using the **Group** property on Validators, assign the group name to the **Group** property on the submit control.

CONTINUED ON THE NEXT PAGE

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

12. Your post back event handler method should check that **PeterBlum.DES.Globals.WebFormDirector.IsValid** is `true` before saving or otherwise using the data. If you are not using DES's submit controls, this should appear after the call to `Validate()`.

    [C#]

    ```
    protected void Button1_Click(object sender, EventArgs e)
    {
       // if Button1 is not a DES button,
       // call PeterBlum.DES.Globals.WebFormDirector.Validate("validation group
    name") here
       if (PeterBlum.DES.Globals.WebFormDirector.IsValid)
       {
          // code that saves or uses the data of the web form
       }
    }
    ```

    [VB]

    ```
    Protected Sub Button1_Click(ByVal sender As Object, ByVal e As EventArgs) _
       Handles Button1.Click

       ' if Button1 is not a DES button
       ' call PeterBlum.DES.Globals.WebFormDirector.Validate("validation group
    name") here
       If PeterBlum.DES.Globals.WebFormDirector.IsValid Then
          ' code that saves or uses the data of the web form
       End If
    End Sub
    ```

13. Set up properties that draw the user's attention to the error. DES provides several techniques including setting focus to the field with the error, showing an alert, changing a style of the field with the error or its label, and blinking the errorformatter. See "Drawing The User's Attention To The Error".

14. If desired, add a ValidationSummary control. It provides a consolidated list of all errors on the page. See "ValidationSummary Control".

15. If desired, add required field markers to fields that require an entry.

    - Use the RequiredFieldDescriptor control as a descriptive heading.
    - Either use the Validator's **ShowRequiredFieldMarker** property or add the RequiredFieldMarker control next to the data entry control.

    See "Required Field Marker Controls".

16. If you need to run scripts before or after validation, see "Running a Script when validation occurs".

17. If your license covers **Peter's Interactive Pages** module, see the **Interactive Pages User's Guide** for adding any of these features:

    - Prompt to confirm saving the page on submit. There is a default confirmation message on **PeterBlum.DES.Globals.WebFormDirector.ValidationManager.ConfirmMessage**. DES's submit controls can provide individualized messages with their own **ConfirmMessage** property.
    - Disable the submit buttons to limit double submissions using the **DisableOnSubmit** property on DES's submit controls.

CONTINUED ON THE NEXT PAGE

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

18. 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. In addition, you can display a report that gives the page's validation configuration. See "Analyzing the Page's Validation configuration".
- See also "Additional Validation Topics".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Validator Controls

Validation is a framework of controls and classes that notify users when fields contain invalid data on the page. The **Validator control** class defines the rules to detect an error – called a **condition**, the error message, and its formatting. DES includes 25 Validator controls, each with its own condition; plus two more where you define the rules to detect an error. You can also create your own Validator controls and conditions with the Developer's Kit.

Click on any of these topics to jump to them:

- Evaluation rules for Validators: Condition
  - Available Validator controls
  - Finding the Validator by the Data Entry Control
  - Properties Common to most Conditions
  - Non-Data Entry Conditions
- Changing When the Validator is Evaluated
- Defining the Error Message
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Other Properties
- Drawing The User's Attention To The Error
- Submitting the Page: Server-Side Validation
- Submitting the Page: Client-Side Validation
- Additional Validation Topics

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Evaluation rules for Validators: Condition objects

Validator controls use **Conditions** to determine whether the fields are valid or not. A Condition is a very powerful tool. In fact, the Condition concept has been placed into a separate class of objects so it can be used in a variety of ways. Each Validator control internally contains a Condition object and exposes its properties. The **Enabler** property on all Validator controls lets you use a Condition object to determine if the Validator control should validate at all. For example, your Validator on a textbox shouldn't run if the textbox is invisible.

Each Condition defines a rule that is evaluated when the Validator is notified to evaluate. There are 30 Condition classes defined in DES. Some are **Non-Data Entry Condition**s that evaluate attributes of a field on the page such as visibility or the enabled state. The Non-Data Entry Conditions do not have a Validator control equivalent. They are used within the **Enabler** property and several Validator controls. You can add your own Conditions through delegation (attaching an event handler) or by subclassing.

One of the most powerful Conditions within DES is the **MultiCondition** class and its associated MultiConditionValidator Control. Use it to build Boolean expressions from the results of any number of Conditions. For example, (condition1 AND condition2 OR (condition3 AND condition4)). It is often used, as the rules of your pages can get complex.

When a Condition is evaluated, it may return one of three values: "success", "failed", or "cannot evaluate". "Cannot evaluate" indicates that the fields do not contain the right kind of data that can be evaluated. For example, when you have set up a Validator to compare the contents of a TextBox to an integer, the TextBox must contain an integer before the comparison can occur. If the user entered something else, the Condition would return "cannot evaluate". In this case, you would set up another Validator to confirm the data is an integer. The Validator control notifies the user when the condition failed. It hides the error when the condition succeeded. When the Condition returns "cannot evaluate", any existing error is hidden.

Available Validator controls

Click on the Validator in this chart to see its Condition properties:

ABARoutingNumberValidator	CharacterValidator	CheckStateValidator
CompareToStringsValidator	CompareToValueValidator	CompareTwoFieldsValidator
CountSelectionsValidator	CountTrueConditionsValidator	CreditCardNumberValidator
CustomValidator	DataTypeCheckValidator	DifferenceValidator
DuplicateEntryValidator	EmailAddressValidator	IgnoreConditionValidator
ListSizeValidator	MultipleRequiredControlsValidator	MultiConditionValidator
RangeValidator	RegexValidator	
RequiredListValidator	RequiredSelectionValidator	RequiredTextValidator
SelectedIndexValidator	SelectedIndexRangesValidator	TextLengthValidator
UnwantedWordsValidator	WordCountValidator	

Finding the Validator by the Data Entry Control

Click on any of these topics to jump to them:

- Evaluating Textboxes and other controls with textual values
- Evaluating the Listbox, DropDropList, RadioButtonList, and CheckBoxList
- Evaluating CheckBoxes, RadioButtons, and other controls
- Evaluating all other cases
- Non-Data Entry Condition Classes

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Properties Common to most Conditions

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties common to most Conditions. Individual Validator controls describe additional properties in the following sections.

- **ControlIDToEvaluate** (string) – Identifies the data entry control that will be evaluated. This property takes the ID of the control. When a Validator control or Condition class includes this property, it must be assigned unless you are using the **ControlToEvaluate** property or the CustomValidator.

  If the Condition supports multiple controls, this refers to the first control.

  An exception is thrown at runtime when this is blank, unknown, not in the same or ancestor naming container, is Visible=false, or a control class that is not supported.

  Each condition class specifies the types of controls it supports. See the descriptions of each condition. Start at "Quick jump to the Condition properties of individual Validator controls".

  > *Note: DES supports third party controls that are based on or emulate TextBox, CheckBox, RadioButton, ListBox, and DropDownList. See the topic "Supporting Third Party Data Entry Controls" in the Installation Guide*

  #### 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]
  ```

  This is commonly used to connect a validator to a data entry control inside of a textbox. See "Attaching Validators to controls inside of a User Control".

- **ControlToEvaluate** (System.Web.UI.Control) – An alternative to **ControlIDToEvaluate**. Use it when the data entry control is not in the same or ancestor naming container. It must be assigned programmatically. For example, if you have a Validator instance in the variable "Val1" and a textbox instance in the variable "TextBox1", write code like this: `Val1.ControlToEvaluate = TextBox1`.

  When programmatically assigning properties to a Validator control or Condition class, if you have access to the control object that will be evaluated, it is better to assign it here than assign its ID to the **ControlIDToEvaluate** property because DES operates faster using **ControlToEvaluate**.

- **ValueToEvaluate** (*depends on the class*) – Allows the Validator or Condition to be used without a control attached. Instead, data comes from the value assigned to this property. When used, it replaces **ControlIDToEvaluate** and **ControlToEvaluate**.

  The value assigned depends on its type.

  - object – Supports either a string or a native type based on the DataType property. For example, if DataType=Date, you can pass in a DateTime object or a string formatted as a date in the current culture.
  - String – Used by many validators.
  - Integer – An index into a list (RequiredListValidator, SelectedIndexValidator, SelectedIndexRangesValidator)
  - Boolean – Used by CheckStateValidator.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

- IList<Integer> - A list of indexes into a list (CountSelectionsValidator).

See "Using Validator controls to evaluate data not from Controls".

- **NotCondition** (Boolean) – When `true`, it reverses the result of the Condition's evaluation. If the evaluation was `success`, it will be `failed`. If `failed`, it will be `success`. This is especially useful when building Boolean expressions within the MultiConditionValidator. It defaults to `false`.
- **Trim** (Boolean) – When `true`, Conditions will trim the leading and trailing spaces from the strings they retrieve from data entry controls prior to evaluating them. Set to `false` when your server side retrieval code for the field doesn't trim, thus would be tripped up by the leading and trailing spaces. It defaults to `true`.
- **ReportErrorsAfter** (enum PeterBlum.DES.ReportErrorsAfter) – Any validator that can evaluate two or more controls can use this property to delay the onchange validation until the user moves out of those controls. It allows them to edit all controls first before seeing an error.

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

  - `EachEdit` – Validate after an individual control is edited. This is the normal way validation works.
  - `AllEdits` – Validate after focus leaves all controls assigned to the validator, if any were changed.
  - `AllEditsWhenError` – When there is a validation error, wait until focus leaves all controls assigned to the validator. If the error is fixed, remove the error message immediately after any of the fields are edited.
  - `AllEditsFirstTime` – The first time editing any of the controls will wait until focus leaves all fields assigned to the validator. After the validator has been evaluated once, validate after each of the fields are edited.

  Its default depends on the validator.

- **ExtraControlsToRunThisAction** (PeterBlum.DES.Connections) – Identifies additional controls and elements on the page that validate this Validator when clicked or edited.

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

  - 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 CustomValidator uses controls that are not specified by **ControlIDToEvaluate** and **SecondControlIDToEvaluate**, add those controls to this property.
  - The IgnoreConditionValidator does not offer the **ControlIDToEvaluate** and **SecondControlIDToEvaluate** properties. If you are using the **OverrideClientSideEvaluation** property to provide a client-side experience, you can add controls involved in your condition here.

  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 the control assigned to this collection has the `runat=server` property.

ASP.NET Markup for Adding ControlConnections

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

```
<des:ValidatorClass id="Validator1" runat="server">

   <ExtraControlsToRunThisAction>
      <des:ControlConnection ControlID="TextBox1" />
      <des:ControlConnection ControlID="Label1" />
   </ExtraControlsToRunThisAction>

</des:ValidatorClass>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Programmatically Adding ControlConnections

The **ExtraControlsToRunThisAction** property will create `PeterBlum.DES.Web.WebControls.ControlConnections` for you by calling its `Add()` method. `ExtraControlsToRunThisAction.Add()` takes a reference to the control object or its ID. (The reference is recommended for performance.)

[C#]

```
Validator1.ExtraControlsToRunThisAction.Add(RadioButton1);  // reference
Validator1.ExtraControlsToRunThisAction.Add("RadioButton2"); // ID
```

[VB]

```
Validator1.ExtraControlsToRunThisAction.Add(RadioButton1)   ' reference
Validator1.ExtraControlsToRunThisAction.Add("RadioButton2") ' ID
```

Example

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)
      Validator1.ExtraControlsToRunThisAction[1];
vConnection.ControlInstance = Label1;
// add TextBox2. It can be either a control reference or its ID
Validator1.ExtraControlsToRunThisAction.Add(TextBox2);
```

[VB]

```
Imports PeterBlum.DES
...
Dim vConnection As ControlConnection = _
   CType(Validator1.ExtraControlsToRunThisAction(1), ControlConnection)
vConnection.ControlInstance = Label1
' add TextBox2. It can be either a control reference or its ID
Validator1.ExtraControlsToRunThisAction.Add(TextBox2)
```

Quick jump to the Condition properties of individual Validator controls

Click on the Validator in this chart to see its Condition properties:		
ABARoutingNumberValidator	CharacterValidator	CheckStateValidator
CompareToStringsValidator	CompareToValueValidator	CompareTwoFieldsValidator
CountSelectionsValidator	CountTrueConditionsValidator	CreditCardNumberValidator
CustomValidator	DataTypeCheckValidator	DifferenceValidator
DuplicateEntryValidator	EmailAddressValidator	IgnoreConditionValidator
ListSizeValidator	MultipleRequiredControlsValidator	MultiConditionValidator
RangeValidator	RegexValidator	
RequiredListValidator	RequiredSelectionValidator	RequiredTextValidator
SelectedIndexValidator	SelectedIndexRangesValidator	TextLengthValidator
UnwantedWordsValidator	WordCountValidator	

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

RequiredTextValidator

Condition Class	RequiredTextCondition
License Required	Peter's Professional Validation
Supported controls	All that declare the ValidationPropertyAttribute including TextBox, ListBox, DropDownList
Can evaluate blank fields	Yes

The RequiredTextValidator tests if a field's textual value is "blank" or not. You can define what text represents "blank" such as the empty string (the default) or "Enter Text Here".

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for RequiredTextValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Using This Condition

Specify the control to evaluate with the **ControlIDToEvaluate** property.

When used on a TextBox, it evaluates the **Text** property. When used on a ListBox and DropDownList, it evaluates to the **Value** property on selected ListItem.

The **UnassignedValues** property determines what text is considered unassigned. It defaults to an empty string. It allows you to define multiple strings considered unassigned, simply by separating each with a pipe (|). For example, to consider the values of an empty string and "Item1" as unassigned, enter "`|Item1`". If you want those to be the only valid values, set the **NotCondition** property to `true`.

Most of the remaining Validator controls do nothing when the field's textual value is the empty string, generally requiring you to add this Validator. If you want a Validator that also uses this rule, use the MultiConditionValidator with the RequiredTextCondition and the desired Validator's condition. Use a logical AND operator on the MultiConditionValidator.

Normally validators are invoked if a field is changed. If you want this validator to display an error just by tabbing out of a blank textbox, set the **RequiredFieldsReportErrorsOnBlur** property to true on the PageManager control.

Example

Determine if TextBox1 is blank.

This field is required

```
<des:RequiredTextValidator id="RequiredTextValidator1" runat="server"
   ControlIDToEvaluate="TextBox1" ErrorMessage="This field is required">
</des:RequiredTextValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Condition Properties for RequiredTextValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, **ValueToEvaluate**, **Trim**, and **NotCondition** – See "Properties Common to most Conditions".
- **UnassignedValues** (string) – The textual value of the control that means it is unassigned. When the control's value matches this value, the condition indicates failure. Matching may be case sensitive or insensitive based on **UnassignedCaseInsensitive**. It defaults to "".

  Typically, a blank field (empty string) represents unassigned. Some sites prefer to have a term like "Please fill this in" that when replaced is no longer unassigned. If you use DES's TextBoxes with their **ValueWhenBlank** property set, this validator automatically treats the **ValueWhenBlank** text as a blank field.

  Since the RequiredTextCondition can get its value from the selected item in a ListBox or DropDownList, assign this to text of the **Value** property on the ListItem that indicates the list is unassigned.

  This property supports multiple items that reflect the unassigned state. Separate each item with a pipe character (|). For example, a textbox is unassigned when either blank or "Enter a value" is present. Use "|Enter a value". (The part prior to the pipe is an empty string.) Lead and trailing spaces are removed on each item within the **UnassignedValues**.
- **UnassignedCaseInsenstive** (Boolean) - Determines if the **UnassignedValues** property is matched case insensitive or not. When `true`, it is case insensitive. It defaults to `true`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

RequiredListValidator

Condition Class	RequiredListCondition
License Required	Peter's Professional Validation
Supported controls	ListBox, DropDownList, RadioButtonList, CheckBoxList, and HtmlSelect
Can evaluate blank fields	Yes

The RequiredListValidator tests if a list-style control has no selection. By default, "no selection" means no item is highlighted or marked. You can change "no selection" to be a particular item, such as the first item in the list.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for RequiredListValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Using This Condition

Specify the control to evaluate with the **ControlIDToEvaluate** property. Use the **UnassignedIndex** property to change the "no selection" item.

*ALERT: Do not use this Validator with the ASP.NET AJAX Control Toolkit's CascadingDropDown control. The CascadingDropDown does not support the **SelectedIndex** property on the server side. It only supports the textual value. Instead use the RequiredTextValidator on the List control. Set up its UnassignedValues property to be "|your prompt text". (That is a pipe character as the first letter.)*

Normally validators are invoked if a field is changed. If you want this validator to display an error just by tabbing out of a blank textbox, set the **RequiredFieldsReportErrorsOnBlur** property to true on the PageManager control.

Example

Determine if ListBox1 has no selection.

```
<des:RequiredListValidator id="RequiredListValidator1" runat="server"
   ControlIDToEvaluate="ListBox1" ErrorMessage="Please make a selection">
</des:RequiredListValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Condition Properties for RequiredListValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, and **NotCondition** – See "Properties Common to most Conditions".
- **UnassignedIndex** (Integer) – The item number of the item that is considered unselected. When -1, there are no selected items. 0 is the first item in the list. It defaults to -1.

  Change it when an item, such as the first element of a RadioButtonList, ListBox, or DropDownList indicates an unassigned state. The CheckBoxList control does not use this property because a CheckBoxList is unassigned when all checkboxes are unmarked.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

DataTypeCheckValidator

Condition Class	DataTypeCheckCondition
License Required	Peter's Professional Validation
Supported controls	All that declare the ValidationPropertyAttribute including TextBox, ListBox, DropDownList
Can evaluate blank fields	No (Use RequiredTextValidator)

The DataTypeCheckValidator tests a field's textual value conforms to the format a particular data type. DES includes an extensive and expandable list of data types.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for DataTypeCheckValidator
  - Range Properties on the DataTypeCheckCondition Class
  - ControlToNative() method
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Using This Condition

Specify the control to evaluate with the **ControlIDToEvaluate** property. The **DataType** property specifies the data type. If you assign this Validator to any of the Numeric TextBoxes, it will automatically adopt all of the data type rules from that textbox. It will ignore the setting in the **DataType** property.

Data types like date and currency will follow culture specific formatting set on the current web page or overridden on **PeterBlum.DES.Globals.WebFormDirector.CultureInfo**.

If you are get a value from a textbox by retrieving a string from the **Text** property, let this validator assist you in converting that string to its native type, such as Integer or DateTime, with the ControlToNative() method.

When used with any of DES's numeric, date or time textboxes, it can respect their minimum and maximum values (**MinValue/MaxValue** for numeric, **MinDate/MaxDate** for date, and **MinTime/MaxTime** for time) when the textbox control's **DataTypeCheckReportsRangeErrors** property is `true`.

Example

Determine if TextBox1 contains an integer.

abc The value must be an integer

```
<des:DataTypeCheckValidator id="DataTypeCheckValidator1" runat="server"
   ControlIDToEvaluate="TextBox1" DataType="Integer"
   ErrorMessage="The value must be an integer">
</des:DataTypeCheckValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Condition Properties for DataTypeCheckValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, **ValueToEvaluate**, **Trim**, and **NotCondition** – See "Properties Common to most Conditions".
- **DataType** (string) – The name of the data type that this condition is looking for. It defaults to "String". You can change it to any of these values: "String-Case Insensitive", "Integer", "Double", "Date", "Time", "Duration", "Currency", "Currency with Symbol", "Positive Integer", "Positive Double", "Positive Currency", and "Positive Currency with Symbol".

  Numeric, currency, and date types will follow culture specific formatting set on the current web page or overridden on **PeterBlum.DES.Globals.WebFormDirector.CultureInfo**. Integers are represented by signed 32 bit integer values. Any larger integer value than 2,147,483,648 will be considered invalid.

  The "String" Data Type is really meaningless as every textual entry is a string. *Suggestion: if you are looking for a way to limit text to letters, digits, or any particular characters, use the CharacterValidator.*

  When using any of the numeric, date or time textboxes from the **Peter's TextBoxes** or **Peter's Date and Time** modules, this property is almost always ignored. The textbox will tell the validator exactly how it needs to be configured. **DataType** is only used when evaluating a TimeOfDayTextBox with the **DateTextBoxControlID** assigned. In that case, set **DataType** to `DateTime` or `Time` to select whether to validate both DateTextBox+TimeOfDayTextBox together or just the TimeOfDayTextBox alone.

  You can programmatically define new data types. See the Developer's Guide for details.

Range Properties on the DataTypeCheckCondition Class

The DataTypeCheckCondition (but not the validator) offers **Minimum** and **Maximum** properties. When set, they are the range used by this condition to report an error.

Sometimes an out of range value is considered a data type check error. More often, it is better to evaluate it separately in a RangeValidator to let the user know explicitly by an error message "The value is out of range."

Only use this when there is a range situation which the validator's current error message is helpful enough. For example, since a percentage is usually a positive number, establish a minimum of 0 here. Reporting "-10" as an illegal value will make sense to the user without saying "The value -10 is out of range.".

The DataTypeCheckValidator respects the range defined on a DES's numeric, date, or time textboxes with their **DataTypeCheckReportsRangeErrors** property is `true`.

- **Minimum** (string) – The minimum value of a range in a string formatted to match the selected **DataType**. For example, if the **DataType**=`Integer`, this value can only contain digits. Alternatively, assign a native type, like Integer or DateTime, to **MinimumAsNative**, and **Minimum** will be correctly formatted for you.

  If it is "", there is no minimum.

  It defaults to "".
- **Maximum** (string) – The maximum value of a range in a string formatted to match the selected **DataType**. For example, if the **DataType**=`Integer`, this value can only contain digits. Alternatively, assign a native type, like Integer or DateTime, to **MaximumAsNative**, and **Maximum** will be correctly formatted for you.

  If it is "", there is no maximum.

  It defaults to "".
- **MinimumAsNative** (object) - Alternative to the **Minimum** property. It takes a data type other than a string and converts it to the string stored in the **Minimum** property. This property is not shown in the Properties Editor.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

When setting this property, use the type given for the **DataType** property values listed here:

DataType	Supported types
Integer, Positive Integer	32 bit integer (System.Int32)
Double, Positive Double, Currency (all types)	Double (System.Double)
Date, DateTime	System.DateTime
Time, Duration	System.DateTime, System.TimeSpan, Integer (number of seconds), Double (number of hours)

If you set an illegal value, it will throw an exception.

When getting this property, if the **Minimum** property is the empty string, it will return `null` or the **MinValue** property of the expected datatype in some cases. If the **Minimum** property cannot be converted from a string, it will return `null`.

- **MaximumAsNative** (object) - Alternative to the **Maximum** property. It takes a data type other than a string and converts it to the string stored in the **Maximum** property. This property is not shown in the Properties Editor.

  When setting this property, use the type given for the **DataType** property values listed here:

DataType	Supported types
Integer, Positive Integer	32 bit integer (System.Int32)
Double, Positive Double, Currency (all types)	Double (System.Double)
Date, DateTime	System.DateTime
Time, Duration	System.DateTime, System.TimeSpan, Integer (number of seconds), Double (number of hours)

  If you set an illegal value, it will throw an exception.

  When getting this property, if the **Maximum** property is the empty string, it will return `null` or the **MinValue** property of the expected datatype in some cases. If the **Maximum** property cannot be converted from a string, it will return `null`.

ControlToNative() method

The DataTypeCheckValidator and DataTypeCheckCondition provide the `ControlToNative()` method to convert the textual value of the textbox to its native type. This is especially important because of the extensive formatting rules permitted by the validator. For example, the Currency types may allow the currency symbol, thousands separator, and negative format, all following the current CultureInfo object.

[C#]

```
object ControlToNative()
```

[VB]

```
Function ControlToNative() As Object
```

Return value

Returns an object representing the data type defined in the **DataType** property. You should typecast it to the expected type. (See the example below.)

If the text cannot be converted, it returns `null`.

DataType	Native Type Returned
Integer, Positive Integer	32 bit integer (System.Int32)
Double, Positive Double, Currency (all types)	Double (System.Double)
Date, DateTime, Time, Duration	System.DateTime

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Example

[C#]

```
int vResult = (int) DataTypeCheckValidator1.ControlToNative();
```

[VB]

```
Dim vResult As Integer = CType(DataTypeCheckValidator1.ControlToNative(), Integer)
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

CompareToValueValidator

Condition	CompareToValueCondition
License Required	Peter's Professional Validation
Supported controls	All that declare the ValidationPropertyAttribute including TextBox, ListBox, DropDownList. Also supports DES's Calendar.
Can evaluate blank fields	No (Use RequiredTextValidator)

The CompareToValueValidator compares a field's value to another value that you supply. You specify the data type of the values and the operator used to compare them. Operators include Less Than, Equal, Greater, Not Equal, Less Than and Equal, Greater Than and Equal.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for CompareToValueCondition
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Using This Condition

Specify the control to evaluate with the **ControlIDToEvaluate** property. When pointing to a ListBox or DropDownList, it evaluates the **ListItem.Value** property of the selected item, not the **ListItem.Text** property which is displayed to the user.

The **DataType** property specifies the data type. Data types like date and currency will follow culture specific formatting set on the current web page or overridden on **PeterBlum.DES.Globals.WebFormDirector.CultureInfo**. DES's data type textboxes (DateTextBox, IntegerTextBox, etc) automatically set **DataType** for you.

Use the **Operator** property for the operator and **ValueToCompare** for the value. The **ValueToCompare** is a string but should reflect the same data type you specified, following the culture format set in **PeterBlum.DES.Globals.WebFormDirector.CultureInfo**. Alternatively, you can programmatically assign a native type, like an integer or DateTime, to the **ValueToCompareAsNative** property and it will convert the value into a correctly formatted string.

This Validator cannot evaluate when the control's text is blank or it cannot be converted to the data type. Use the RequiredTextValidator and DataTypeCheckValidator to handle each of those cases or combine their conditions within the MultiConditionValidator to provide one error message.

The **ErrorMessage** property supports the token "{VALUETOCOMPARE}". It displays the value from the **ValueToCompare** property. See "Tokens in Error Messages" for details on tokens.

AN EXAMPLE IS ON THE NEXT PAGE

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Example

Confirm that TextBox1's value is greater than 10. TextBox1 must contain an integer.

3 The value must be greater than 10

```
<des:CompareToValueValidator id="CompareToValueValidator1" runat="server"
   ControlIDToEvaluate="TextBox1" DataType="Integer"
   ValueToCompare="10" Operator="GreaterThan"
   ErrorMessage="The value must greater than {VALUETOCOMPARE}">
</des:CompareToValueValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Condition Properties for CompareToValueCondition

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, **ValueToEvaluate**, **Trim**, and **NotCondition** – See "Properties Common to most Conditions".
- **DataType** (string) – The name of the data type that this condition evaluates. If the user does not enter a string that is compatible with this data type, the condition will not evaluate. (So use a DataTypeCheckValidator to detect errors in the format.) It defaults to "String". You can change it to any of these values: "String-Case Insensitive", "Integer", "Double", "Date", "Time", "Duration", "Currency", "Currency with Symbol", "Positive Integer", "Positive Double", "Positive Currency", and "Positive Currency with Symbol".

  Numeric, currency, and date types will follow culture specific formatting set on the current web page or overridden on **PeterBlum.DES.Globals.WebFormDirector.CultureInfo**. Integers are represented by signed 32 bit integer values. Any larger integer value than 2,147,483,648 will not be evaluated.

  When using any of the numeric, date or time textboxes from the **Peter's TextBoxes** or **Peter's Date and Time** modules, this property is almost always ignored. The textbox will tell the validator exactly how it needs to be configured. **DataType** is only used when evaluating a TimeOfDayTextBox with the **DateTextBoxControlID** assigned. In that case, set **DataType** to `DateTime` or `Time` to select whether to validate both DateTextBox+TimeOfDayTextBox together or just the TimeOfDayTextBox alone.

  You can programmatically define new data types. See the Developer's Guide for details.

- **ValueToCompare** (string) – A string representing the value to be compared. The value you enter here must be compatible with the **DataType** selected. Otherwise, it will throw an exception at runtime. For example, if you set **DataType** to `Integer`, you must enter only digits here. Alternatively, assign a native type to the **ValueToCompareAsNative** property and it will assign **ValueToCompare** with the correct formatting.

  If it is an empty string, it will still be used for comparison of String data types and report an exception for other data types.

  This value is the right side of the expression. The value of **ControlIDToEvaluate** is on the left side.

- **Operator** (enum PeterBlum.DES.ConditionOperator) – Determines how the data entry control is compared to **ValueToCompare**. The enumerated type `PeterBlum.DES.ConditionOperator` has the following values:
  - `Equal` (This is the default.) ControlIDToEvaluate = ValueToCompare
  - `NotEqual` ControlIDToEvaluate <> ValueToCompare
  - `GreaterThan` ControlIDToEvaluate > ValueToCompare
  - `GreaterThanEqual` ControlIDToEvaluate >= ValueToCompare
  - `LessThan` ControlIDToEvaluate < ValueToCompare
  - `LessThanEqual` ControlIDToEvaluate <= ValueToCompare

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- **ValueToCompareAsNative** (object) - Alternative to the **ValueToCompare** property. It takes a data type other than a string and converts it to the string stored in the **ValueToCompare** property. This property is not shown in the Properties Editor.

  When setting this property, use the type given for the **DataType** property values listed here:

DataType	Supported types
Integer, Positive Integer	32 bit integer (System.Int32)
Double, Positive Double, Currency (all types)	Double (System.Double)
Date, DateTime	System.DateTime
Time, Duration	System.DateTime, System.TimeSpan, Integer (number of seconds), Double (number of hours)

  If you set an illegal value, it will throw an exception.

  When getting this property, if the **ValueToCompare** property is the empty string, it will return `null` or the **MinValue** property of the expected datatype in some cases. If the **ValueToCompare** property cannot be converted from a string, it will return `null`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

CompareTwoFieldsValidator

Condition	CompareTwoFieldsCondition
License Required	Peter's Professional Validation
Supported controls	All that declare the ValidationPropertyAttribute including TextBox, ListBox, DropDownList
Can evaluate blank fields	No (Use RequiredTextValidator or MultipleRequiredControlsValidator on both fields)

The CompareTwoFieldsValidator compares the textual values of two fields to each other. You specify the data type of the values and the operator used to compare them. Operators include Less Than, Equal, Greater, Not Equal, Less Than and Equal, Greater Than and Equal.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for CompareTwoFieldsCondition
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Using This Condition

Specify the two controls to evaluate with the **ControlIDToEvaluate** and **SecondControlIDToEvaluate** properties. The **DataType** property specifies the data type. Data types like date and currency will follow culture specific formatting set on the current web page or overridden on **PeterBlum.DES.Globals.WebFormDirector.CultureInfo**.

Use the **Operator** property for the operator.

This Validator cannot evaluate when either control's text is blank or it cannot be converted to the data type. Use the RequiredTextValidator and DataTypeCheckValidator to handle each of those cases or combine their conditions within the MultiConditionValidator to provide one error message.

Example

Confirm that TextBox1 has an integer that is less than or equal to TextBox2.

3 thru 2 The Thru textbox must be a larger number

```
<des:CompareTwoFieldsValidator id="CompareTwoFieldsValidator1" runat="server"
   ControlIDToEvaluate="TextBox1" SecondControlIDToEvaluate="TextBox2"
   DataType="Integer" Operator="LessThanEqual"
   ErrorMessage="The Thru textbox must be a larger number">
</des:CompareTwoFieldsValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Condition Properties for CompareTwoFieldsCondition

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, **ValueToEvaluate**, **ReportErrorsAfter**, **Trim**, and **NotCondition** – See "Properties Common to most Conditions".
- **SecondControlIDToEvaluate** (string) – Identifies the second data entry control that will be evaluated. This property takes the ID of the control. When a Validator control or Condition class includes this property, it must be assigned unless you are using the **SecondControlToEvaluate** property.

  An exception is thrown at runtime when this is blank, unknown, not in the same or ancestor naming container, is **Visible**=`false`, or a control class that is not supported.

  If you want to connect to a control that is inside of a usercontrol, see "Attaching Validators to controls inside of a User Control".
- **SecondControlToEvaluate** (System.Web.UI.Control) – An alternative to **SecondControlIDToEvaluate**. Use it when the data entry control is not in the same or ancestor naming container. It must be assigned programmatically. For example, if you have a Validator instance in the variable "Val2" and a textbox instance in the variable "TextBox2", write code like this: `Val2.SecondControlToEvaluate = TextBox2`.

  When programmatically assigning properties to a Validator control or Condition class, if you have access to the control object that will be evaluated, it is better to assign it here than assign its ID to the **SecondControlIDToEvaluate** property because DES operates faster using **SecondControlToEvaluate**.
- **SecondValueToEvaluate** (object) – Allows the Validator or Condition to be used without a control attached. Instead, data comes from the value assigned to this property. The value must be the native type for the specified **DataType**, or a string that can convert to the native type. When used, it replaces **SecondControlIDToEvaluate** and **SecondControlToEvaluate**.

  See "Using Validator controls to evaluate data not from Controls".
- **DataType** (string) – The name of the data type that this condition evaluates. If the user does not enter a string that is compatible with this data type, the condition will not evaluate. (So use a DataTypeCheckValidator to detect errors in the format.) It defaults to "String". You can change it to any of these values: "String-Case Insensitive", "Integer", "Double", "Date", "Time", "Duration", "Currency", "Currency with Symbol", "Positive Integer", "Positive Double", "Positive Currency", and "Positive Currency with Symbol".

  Numeric, currency, and date types will follow culture specific formatting set on the current web page or overridden on **PeterBlum.DES.Globals.WebFormDirector.CultureInfo**. Integers are represented by signed 32 bit integer values. Any larger integer value than 2,147,483,648 will not be evaluated.

  When using any of the numeric, date or time textboxes from the **Peter's TextBoxes** or **Peter's Date and Time** modules, this property is almost always ignored. The textbox will tell the validator exactly how it needs to be configured. **DataType** is only used when evaluating a TimeOfDayTextBox with the **DateTextBoxControlID** assigned. In that case, set **DataType** to `DateTime` or `Time` to select whether to validate both DateTextBox+TimeOfDayTextBox together or just the TimeOfDayTextBox alone.

  You can programmatically define new data types. See the Developer's Guide for details.
- **Operator** (enum PeterBlum.DES.ConditionOperator) – Determines how **ControlIDToEvaluate** is compared to **SecondControlIDToEvaluate**. The enumerated type `PeterBlum.DES.ConditionOperator` has the following values:
  - `Equal` (This is the default.) ControlIDToEvaluate = SecondControlIDToEvaluate
  - `NotEqual` ControlIDToEvaluate <> SecondControlIDToEvaluate
  - `GreaterThan` ControlIDToEvaluate > SecondControlIDToEvaluate
  - `GreaterThanEqual` ControlIDToEvaluate >= SecondControlIDToEvaluate
  - `LessThan` ControlIDToEvaluate < SecondControlIDToEvaluate

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- `LessThanEqual` ControlIDToEvaluate <= SecondControlIDToEvaluate

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

RangeValidator

Condition	RangeCondition
License Required	Peter's Professional Validation
Supported controls	All that declare the ValidationPropertyAttribute including TextBox, ListBox, DropDownList
Can evaluate blank fields	No (Use RequiredTextValidator)

The RangeValidator determines if a field's textual value is between a minimum and maximum value. You determine the data type of the values. The condition indicates success when the value is greater or equal to the minimum and less than or equal to the maximum.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for RangeValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Using This Condition

Specify the control to evaluate with the **ControlIDToEvaluate** property. The **DataType** property specifies the data type. Data types like date and currency will follow culture specific formatting set on the current web page or overridden on **PeterBlum.DES.Globals.WebFormDirector.CultureInfo**.

Use the **Minimum** and **Maximum** properties for the range. The **Minimum** and **Maximum** are strings but should reflect the same data type you specified, following the culture format set in **PeterBlum.DES.Globals.WebFormDirector.CultureInfo**. Alternatively, you can programmatically assign a native type, like an integer or DateTime, to the **MinimumAsNative** and **MaximumAsNative** properties and it will convert the value into a correctly formatted string.

This Validator cannot evaluate when the control's text is blank or it cannot be converted to the data type. Use the RequiredTextValidator and DataTypeCheckValidator to handle each of those cases or combine their conditions within the MultiConditionValidator to provide one error message.

The Validator's **ErrorMessage** property supports the tokens "{MINIMUM}" and "{MAXIMUM}" which are replaced by the values from the **Minimum** and **Maximum** properties. See "Tokens in Error Messages" for details on tokens.

Example

TextBox1 must have a currency value that is between 0.01 and 10000.00.

0 Only values between 0.01 and 10000.00 are accepted.

```
<des:RangeValidator id=RangeValidator1 runat="server"
   ControlIDToEvaluate="TextBox1" DataType="Currency"
   Minimum="0.01" Maximum="10000"
   ErrorMessage="Only values between {MINIMUM} and {MAXIMUM} are accepted.">
</des:RangeValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Condition Properties for RangeValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, **ValueToEvaluate**, **Trim**, and **NotCondition** – See "Properties Common to most Conditions".
- **DataType** (string) – The name of the data type that this condition evaluates. If the user does not enter a string that is compatible with this data type, the condition will not evaluate. (So use a DataTypeCheckValidator to detect errors in the format.) It defaults to "String". You can change it to any of these values: "String-Case Insensitive", "Integer", "Double", "Date", "Time", "Duration", "Currency", "Currency with Symbol", "Positive Integer", "Positive Double", "Positive Currency", and "Positive Currency with Symbol".

  Numeric, currency, and date types will follow culture specific formatting set on the current web page or overridden on **PeterBlum.DES.Globals.WebFormDirector.CultureInfo**. Integers are represented by signed 32 bit integer values. Any larger integer value than 2,147,483,648 will not be evaluated.

  When using any of the numeric, date or time textboxes from the **Peter's TextBoxes** or **Peter's Date and Time** modules, this property is almost always ignored. The textbox will tell the validator exactly how it needs to be configured. **DataType** is only used when evaluating a TimeOfDayTextBox with the **DateTextBoxControlID** assigned. In that case, set **DataType** to `DateTime` or `Time` to select whether to validate both DateTextBox+TimeOfDayTextBox together or just the TimeOfDayTextBox alone.

  You can programmatically define new data types. See the Developer's Guide for details.
- **Minimum** (string) – The minimum value of a range in a string formatted to match the selected **DataType**. For example, if the **DataType**=`Integer`, this value can only contain digits. Alternatively, assign a native type, like Integer or DateTime, to **MinimumAsNative**, and **Minimum** will be correctly formatted for you.

  If it is "", there is no minimum.

  It defaults to "".
- **Maximum** (string) – The maximum value of a range in a string formatted to match the selected **DataType**. For example, if the **DataType**=`Integer`, this value can only contain digits. Alternatively, assign a native type, like Integer or DateTime, to **MaximumAsNative**, and **Maximum** will be correctly formatted for you.

  If it is "", there is no maximum.

  It defaults to "".
- **MinimumAsNative** (object) - Alternative to the **Minimum** property. It takes a data type other than a string and converts it to the string stored in the **Minimum** property. This property is not shown in the Properties Editor.

  When setting this property, use the type given for the **DataType** property values listed here:

DataType	Supported types
Integer, Positive Integer	32 bit integer (System.Int32)
Double, Positive Double, Currency (all types)	Double (System.Double)
Date, DateTime	System.DateTime
Time, Duration	System.DateTime, System.TimeSpan, Integer (number of seconds), Double (number of hours)

  If you set an illegal value, it will throw an exception.

  When getting this property, if the **Minimum** property is the empty string, it will return `null` or the **MinValue** property of the expected datatype in some cases. If the **Minimum** property cannot be converted from a string, it will return `null`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- **MaximumAsNative** (object) - Alternative to the **Maximum** property. It takes a data type other than a string and converts it to the string stored in the **Maximum** property. This property is not shown in the Properties Editor.

  When setting this property, use the type given for the **DataType** property values listed here:

DataType	Supported types
Integer, Positive Integer	32 bit integer (System.Int32)
Double, Positive Double, Currency (all types)	Double (System.Double)
Date, DateTime	System.DateTime
Time, Duration	System.DateTime, System.TimeSpan, Integer (number of seconds), Double (number of hours)

  If you set an illegal value, it will throw an exception.

  When getting this property, if the **Maximum** property is the empty string, it will return `null` or the **MinValue** property of the expected datatype in some cases. If the **Maximum** property cannot be converted from a string, it will return `null`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

RegexValidator

Condition	RegexCondition
License Required	Peter's Professional Validation
Supported controls	All that declare the ValidationPropertyAttribute including TextBox, ListBox, DropDownList
Can evaluate blank fields	Yes when the IgnoreBlankText property is false

The RegexValidator tests a field's textual value against a regular expression. It is one of the most flexible Validators as it can look at the patterns of text to determine if it matches. Regular expressions can be used to verify phone numbers, IP Addresses, ID numbers, URLs, and so much more.

DES predefines a number of regular expressions that you can access in the Properties Editor for the **Expression** property (see below) or by opening the **des.config** file to the `<RegExPatterns>` section. Here are the descriptive names of the expressions already defined:

- U.S. Phone
- French Phone
- P.R.C Phone
- MailTo URL
- U.S. Zip Code (5 and 9 digits)
- French Postal Code
- P.R.C Postal Code
- U.S. Social Security Number
- International Phone
- Japanese Phone
- Email Address
- FTP URL
- US and Canadian Postal Codes
- German Postal Code
- Credit Card with spaces
- P.R.C Social Security (ID)
- Any Phone
- German Phone
- Web URL
- Any URL
- Canadian Postal Code
- Japanese Postal Code
- Credit Card without spaces

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Learning About Regular Expressions
- Using This Condition
- Condition Properties for RegexValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Javascript expressions don't consistently work
- Additional Validation Topics
- Online example

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Learning About Regular Expressions

Regular expressions are a complex language that requires an education in the topic. Here are several links that may assist you:

MSDN Overview of Regular Expressions in .Net: http://msdn2.microsoft.com/en-us/library/hs600312.aspx

MSDN Regular Expression Language Elements: http://msdn2.microsoft.com/en-us/library/az24scfc.aspx

MSDN definition of the .Net Regex class: http://msdn2.microsoft.com/en-us/library/system.text.regularexpressions.regex.aspx

The JavaScript language implementation of regular expressions: http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference:Global_Objects:RegExp

RegExLib.com, A Regular Expression Library: http://www.regexlib.com/

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Using This Condition

Specify the control to evaluate with the **ControlIDToEvaluate** property.

Assign the regular expression in the **Expression** property. When you edit this property in the Properties Editor, it offers an extensive and customizable list of pre-defined expressions. It also has a feature to test expressions.

Note: While regular expressions can be used to validate a particular data type, first consider using the DataTypeCheckValidator for that type of validation.

The RegexValidator must be able to run your regular expression in JavaScript unless you elect to use server-side only validation by setting the Validator's **EnableClientScript** property to `false`. JavaScript's definition of regular expressions is a subset of the Regex class in .Net.

Example

Confirm that TextBox1 matches the pattern of a US Phone number.

ABC The phone number must be in this format (###) ###-####

```
<des:RegexValidator id="RegexValidator1" runat="server"
   ControlIDToEvaluate="TextBox1"
   Expression="^([1])?\s*(\(\d{3}\))?\s*\d{3}[\s\-]?\d{4}"
   ErrorMessage="The phone number must be in this format (###) ###-####">
</des:RegexValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

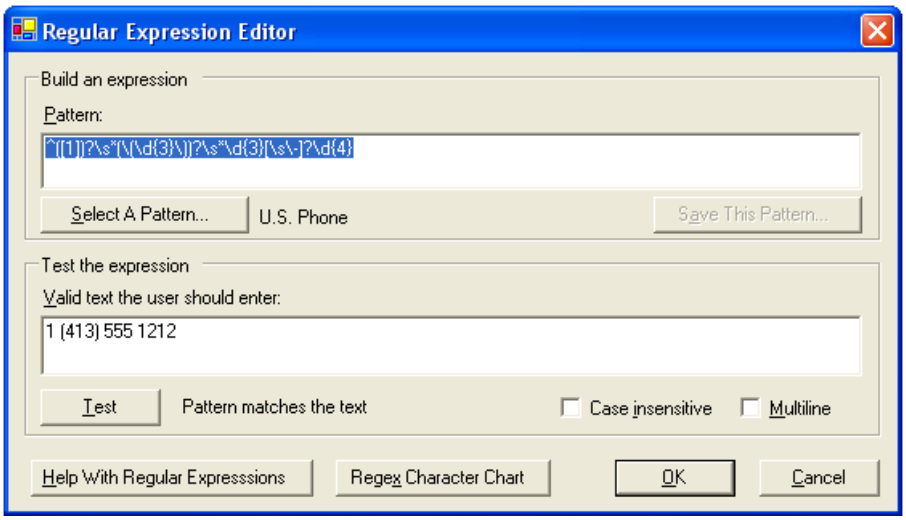
Condition Properties for RegexValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, **ValueToEvaluate**, **Trim**, and **NotCondition** – See "Properties Common to most Conditions".
- **Expression** (string) – A regular expression that will be evaluated against the value of the field specified in **ControlIDToEvaluate**. It must be compatible with both .Net and JavaScripts regular expression parser. When the value of the control is blank, the condition cannot evaluate.

  The Properties Editor offers this window to define, test, load and save regular expressions:



  Enter the regular expression within the **Pattern** textbox. Click **Select a Pattern** to pick from a list of existing patterns. You can add your own expressions or replace existing expressions with the **Save This Pattern** button.

  Note: Regular expression patterns are written into the custom.DES.config file.

  If you cannot find an existing pattern, the **Help With Regular Expressions** and **Regex Character Chart** buttons open web pages with details on using regular expressions.

  Test your expression before applying it to your RegexValidator control with the **Test the expression** box. This is especially important if you select from the list of existing patterns because those defined by PeterBlum.com may not exactly match your needs. *You can edit the existing pattern and replace it under the same name. Later if you want to return to the original pattern, the Select A Pattern dialog box offers a Restore To Default button.*

  See also "Javascript expressions don't consistently work".

- **CaseInsensitive** (Boolean) – When `true`, the regular expression parser will perform a case insensitive match. When `false`, it will perform a case sensitive match. It defaults to `true`.
- **Multiline** (Boolean) – When `true`, it changes the meaning of the regular expression symbols "^" and "$" so they match at the beginning and end, respectively, of any line, and not just the beginning and end of the entire string. It defaults to `false`.

  WARNING: Some browsers do not support the Multiline flag on client-side validation. This includes IE Mac 5.x and Netscape 4.x. These browsers will abort processing the client-side JavaScript condition code when this option is detected. Use this option with care.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- **IgnoreBlankText** (Boolean) – Determines how blank text is evaluated. When `true`, the condition cannot be evaluated. When `false`, the condition evaluates as failed (reports an error). It defaults to `true`.

  There are two common cases for setting this to `false`:

  - To merge the RequiredTextCondition into this condition.
  - When blank text is something you consider an illegal format no different than the regular expression determining an illegal format.

- **GlobalStorageName** (string) – Use this property to improve speed. When your **Expression** property never changes, assign a name here and the **Expression** will be compiled into a .net Regex object and cached so the compilation process doesn't happen on each page request.

  You define a unique name for this expression and assign it to **GlobalStorageName**. For example, "PhoneNumber" or "123abc".

  If you have several RegexValidators that use the same Expression value, use the same value of **GlobalStorageName** so that they all share the same Regex object. This further improves your application's speed.

  When used, the .net Regex object is stored in the Application collection. `Application[GlobalStorageName]` is assigned the Expression string. `Application[GlobalStorageName + "_RE"]` is assigned the Regex object. So give this a name that will not conflict with anything else that you place into the Application collection.

  WARNING: Several other Validators offer the GlobalStorageName property. Never use the same value amongst different types of Validators.

- **PrepareProperties** (delegate PeterBlum.DES.RegexPrepareProperties) – When using the **GlobalStorageName** and you run time consuming code to create the expression (like a database query), put your time consuming code in a method assigned to **PrepareProperties**. Your code will only be called if the expression is not yet cached. This increases the application speed by avoiding that time consuming code when it's not needed.

  For example, your code accesses the database to retrieve a string that is searched within the Expression. Database queries are time consuming.

  Your method will update the properties of the RegexCondition, the Condition object of the RegexValidator. The RegexCondition and RegexValidator share the same properties described throughout this section. Normally you assign the **Expression** property. However, any of the properties can be changed.

  Here is the format of a method assigned to **PrepareProperties**.

  [C#]

  ```
  public void MethodName(PeterBlum.DES.IRegexCondition pCondition)
  ```

  [VB]

  ```
  Public Sub MethodName(ByVal pCondition As PeterBlum.DES.IRegexCondition)
  ```

  Parameters

  pCondition

  The Condition class for the RegexValidator. It has the same properties as shown in this section.

  You assign your method programmatically like this:

  [C#]

  ```
  MyValidator.PrepareProperties =
     new PeterBlum.DES.RegexPrepareProperties(methodname);
  ```

  [VB]

  ```
  MyValidator.PrepareProperties = _
     New PeterBlum.DES.RegexPrepareProperties(AddressOf methodname)
  ```

  Example

  Assigns the results from the `GetPersonsName()` method to the **Expression** property.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

[C#]

```
public void ApplyExpression(PeterBlum.DES.IRegexCondition pCondition)
{
   pCondition.Expression = GetPersonsName();
   pCondition.CaseInsensitive = true;
}
```

[VB]

```
Public Sub ApplyExpression(ByVal pCondition As _
   PeterBlum.DES.IRegexCondition)
   pCondition.Expression = GetPersonsName()
   pCondition.CaseInsensitive = True
End Sub
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

CheckStateValidator

Condition	CheckStateCondition
License Required	Peter's Professional Validation
Supported controls	CheckBox, RadioButton, HtmlInputCheckBox, HtmlInputRadioButton
Can evaluate blank fields	n/a

The CheckStateValidator evaluates the mark within a checkbox or radio button. You can select whether the condition allows a marked or unmarked button.

The Condition of this Validator, CheckStateCondition, is more frequently used than the Validator itself. The **Enabler** property often turns on a Validator based on the state of a checkbox or radio button. MultiConditionValidators often evaluate checkboxes and radio buttons as part of their logic.

If you want to evaluate a RadioButtonList or CheckBoxList control, use the RequiredListValidator or SelectedIndexValidator.

If you want to evaluate that at least one CheckBox or RadioButton control in different rows of a grid or Repeater, use the CountTrueConditionsValidator with CheckStateConditions in its **Conditions** property.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for CheckStateValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Using This Condition

Specify the control to evaluate with the **ControlIDToEvaluate** property. The **Checked** property determines what state the CheckBox or RadioButton's **Checked** property must match to evaluate successfully.

RadioButtons only run their client-side 'onclick' event when clicked, not when they become unmarked as another radio button is marked. If you want this Condition to be evaluated when you unmark a radiobutton not specified with **ControlIDToEvaluate**, specify the other radiobuttons in **ExtraControlsToRunThisAction**.

Example

CheckBox1 must be marked checked.

☐ I have read the license agreement You must mark this field

```
<des:CheckStateValidator id="CheckStateValidator1" runat="server"
   ControlIDToEvaluate="CheckBox1" Checked="true"
   ErrorMessage="You must mark this field">
</des:CheckStateValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Condition Properties for CheckStateValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, **NotCondition**, and **ExtraControlsToRunThisAction** – See "Properties Common to most Conditions".
- **Checked** (Boolean) – When `true`, the Condition indicates success when the CheckBox or RadioButton is marked. When `false`, the Condition indicates success when the checkbox or radiobutton is not marked. It defaults to `true`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

SelectedIndexValidator

Condition	SelectedIndexCondition
License Required	Peter's Professional Validation
Supported controls	ListBox, DropDownList, RadioButtonList, and CheckBoxList
Can evaluate blank fields	No (Use RequiredListValidator)

The SelectedIndexValidator evaluates the **SelectedIndex** property of a ListBox, DropDownList, RadioButtonList, or CheckBoxList. You provide an index into the list (starting at 0 for the first item) and whether you want the item at that index to be highlighted/marked or not. The condition indicates success when the item at the given index matches the desired highlight or mark state.

If you want to determine if one of these controls is unselected, use the RequiredListValidator.

If you want to determine if any of several SelectedIndexes are selected, use the "SelectedIndexRangesValidator".

*ALERT: Do not use this with the ASP.NET AJAX Control Toolkit's CascadingDropDown control. The CascadingDropDown does not support the **SelectedIndex** property on the server side. It only supports the textual value. So use a validator that evaluates the text value, such as CompareToValueValidator.*

Click on any of these topics to jump to them:

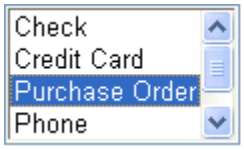
- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for SelectedIndexValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Using This Condition

Specify the control to evaluate with the **ControlIDToEvaluate** property. The **Index** property determines the position whose selected state is evaluated, and the **Selected** property determines the state.

Example

Select only the second item in ListBox1.



You must select Credit Card when downloading the product.

```
<des:SelectedIndexValidator id="SelectedIndexValidator1" runat="server"
   ControlIDToEvaluate="ListBox1" Index="1" Selected="true"
   ErrorMessage="You must select Credit Card when downloading the product.">
</des:SelectedIndexValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Condition Properties for SelectedIndexValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, and **NotCondition** – See "Properties Common to most Conditions".
- **Index** (Integer) – The position in the list to have its selection state evaluated. The first item in the list is position 0. If you supply an index beyond the end of the list, the Condition cannot evaluate. It defaults to 0.
- **Selected** (Boolean) – Determines the state of the list's selection at the position in **Index**. If that position's state matches this value, the Condition indicates success. When `true`, you are looking for the item to be selected or marked. When `false`, you are looking for the item to be unselected or unmarked.

  When evaluating to a CheckBoxList or RadioButtonList, it compares to the **Checked** property on the control. For ListBox and DropDownList controls, it compares the **SelectedIndex** property to **Index**. It defaults to `true`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

TextLengthValidator

Condition	TextLengthCondition
License Required	Peter's Professional Validation
Supported controls	TextBox, HtmlInputText
Can evaluate blank fields	Yes

The TextLengthValidator evaluates the number of characters in a textbox against a minimum and/or maximum. It can optionally evaluate the total characters in two textboxes. *If you want to count the number of words, use the WordCountValidator*.

The browser will automatically maintain a maximum limit on `<input type='text'>` fields when you set the **maxlength** attribute. So consider using the **MaxLength** property on the TextBox control instead of this Validator when TextBox is set to **TextMode**=`SingleLine`.

DES provides the TextCounter control in the **Peter's Interactive Pages** module. It can compliment the TextLengthValidator.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for TextLengthValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Using This Condition

Specify the control to evaluate with the **ControlIDToEvaluate** property; if you want to evaluate the combined length of two controls, specify the second in **SecondControlIDToEvaluate**. The **Minimum** and **Maximum** properties determine the limits when they are not 0. When using the **Minimum** property, if you do not want errors reported for an empty textbox, set **IgnoreBlankText** to `true`.

The Validator's **ErrorMessage** property supports several tokens. "{COUNT}" is the number of characters entered. "{EXCEEDS}" is the number of characters exceeding the minimum or maximum. "{EXCEEDS:character:characters}" uses the word "character" when {EXCEEDS} is 1; otherwise it uses the word "characters". "{MINIMUM}" is replaced by the property **Minimum**. "{MAXIMUM}" is replaced by the property **Maximum**. See "Tokens in Error Messages" for details on tokens.

Examples

When, in the course of human events,
there comes a time for all programmers
to find entertainment away from the
computer.

You have entered 21 too many characters.

```
<des:TextLengthValidator id="TextLengthValidator1" runat="server"
   ControlIDToEvaluate="TextBox1" Maximum="100"
   ErrorMessage="You have entered {EXCEEDS} too many
         {EXCEEDS:character:characters}.">
</des:TextLengthValidator>
```

abc

Please enter at least 5 characters.

```
<des:TextLengthValidator id="TextLengthValidator2" runat="server"
   ControlIDToEvaluate="TextBox2" Minimum="5"
   ErrorMessage="Please enter at least {MINIMUM} characters.">
</des:TextLengthValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Condition Properties for TextLengthValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, **ValueToEvaluate**, **ReportErrorsAfter**, **Trim**, and **NotCondition** – See "Properties Common to most Conditions".
- **Minimum** (Integer) – The minimum number of characters in the textbox. When it is 0, there is no minimum. It defaults to 0.
- **Maximum** (Integer) – The maximum number of characters in the textbox. When it is 0, there is no maximum. It defaults to 0.
- **IgnoreBlankText** (Boolean) – By default, when you use the **Minimum** property, a blank textbox will be reported as an error. If you do not require text in the textbox, set **IgnoreBlankText** to `true`. It defaults to `false`.
- **SecondControlIDToEvaluate** (string) – Identifies a second TextBox control. When assigned, the total characters in the two textboxes are added together before comparing to the minimum and maximum. Leave it blank if you only need to evaluate one TextBox. This property takes the ID of the control.

  An exception is thrown at runtime when this is unknown, not in the same or ancestor naming container, is Visible=false, or a control class that is not supported.

  If you want to connect to a control that is inside of a usercontrol, see "Attaching Validators to controls inside of a User Control".
- **SecondControlToEvaluate** (System.Web.UI.Control) – An alternative to **SecondControlIDToEvaluate**. Use it when the TextBox control is not in the same or ancestor naming container. It must be assigned programmatically. For example, if you have a Validator instance in the variable "Val2" and a textbox instance in the variable "TextBox2", write code like this: `Val2.SecondControlToEvaluate = TextBox2`.

  When programmatically assigning properties to a Validator control or Condition class, if you have access to the control object that will be evaluated, it is better to assign it here than assign its ID to the **SecondControlIDToEvaluate** property because DES operates faster using **SecondControlToEvaluate**.
- **SecondValueToEvaluate** (string) – Allows the Validator or Condition to be used without a control attached. Instead, data comes from the value assigned to this property. When used, it replaces **SecondControlIDToEvaluate** and **SecondControlToEvaluate**.

  See "Using Validator controls to evaluate data not from Controls".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

EmailAddressValidator

Condition	EmailAddressCondition
License Required	Peter's Professional Validation
Supported controls	TextBox, HtmlInputText
Can evaluate blank fields	Yes when the IgnoreBlankText property is false

The EmailAddressValidator reviews the text of an email address to confirm that it matches valid email address patterns. In addition, you can hook it up to a third party email validation engine such as Advanced Intellect's aspNetMX. AspNetMX can confirm that the address is genuine in several ways. AspNetMX must be purchased separately, but it is not required to use this Validator to check the email address pattern.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for EmailAddressValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Using aspNetMX
- Changing the Default Email Address Pattern
- Online example

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Using This Condition

Specify the textbox to evaluate with the **ControlIDToEvaluate** property. If you want to allow multiple email addresses in the same textbox, set **MultipleAddressesAllowed** to `true`. You can define the delimiter character in **DelimiterCharacter**, which defaults to a semicolon. You can allow spaces after the delimiter when **DelimiterAllowsSpaces** is `true`.

If you are using a third party validation engine, set up the **OnEmailAddressValidate** event handler to call that engine and return `true` if the email address is valid.

The following example does not show the use of **OnEmailAddressValidate**. See the properties for details on using this event handler.

```
<des:EmailAddressValidator id="EmailAddressValidator1" runat="server"
   ControlIDToEvaluate="TextBox1"
   ErrorMessage="Invalid email address">
</des:EmailAddressValidator>
```

Omitting Certain Domains

Often users want to prevent email addresses containing specific domains. The EmailAddressValidator cannot handle this, but DES's UnwantedWordsValidator can. By using a separate validator, you can offer a clear error message on what's wrong with an otherwise valid email address.

Here is a definition of the UnwantedWordsValidator that omits emails from "hotmail.com", "yahoo.com", and "gmail.com".

```
<des:UnwantedWordsValidator id="UnwantedWordsValidator1" runat="server"
ErrorMessage="The domain {UNWANTED} is not supported by this site. Please try
another email address." ControlIDToEvaluate="TextBox1">
  <Items>
    <des:UnwantedWordsItem Value="hotmail.com"></des:UnwantedWordsItem>
    <des:UnwantedWordsItem Value="yahoo.com"></des:UnwantedWordsItem>
    <des:UnwantedWordsItem Value="gmail.com"></des:UnwantedWordsItem>
  </Items>
</des:UnwantedWordsValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Condition Properties for EmailAddressValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, **ValueToEvaluate**, **Trim**, and **NotCondition** – See "Properties Common to most Conditions".
- **IgnoreBlankText** (Boolean) – Determines how blank text is evaluated. When `true`, the condition cannot be evaluated. When `false`, the condition evaluates as failed (reports an error). It defaults to `true`.

  This is often set to `false` to merge RequiredTextCondition into this condition.
- **MultipleAddressesAllowed** (Boolean) – Determines if the user can enter multiple email addresses, separated by the delimiter character defined in **DelimiterCharacter** and optionally followed by a space, when **DelimiterAllowsSpaces** is `true`.

  It defaults to `false`.
- **DelimiterCharacter** (String) – A single character used as the delimiter between email addresses. Used when MultipleAddressesAllowed is `true`. Common delimiters are semicolon (`;`), comma (`,`) and space.

  It defaults to a semicolon (`;`).
- **DelimiterAllowsSpaces** (Boolean) – When multiple email addresses are allowed, this determines if spaces are allowed after the delimiter character. It allows users to have some flexibility in formatting. When `true`, one or more spaces are permitted after the delimiter. When `false`, no space is allowed (unless the delimiter itself is a space).

  It defaults to `false`.
- **AllowExtraText** (Boolean) – Determines if the email address should permit additional text after it. When `true`, it permits additional text. No validation is performed on that additional text. When `false`, only the email address is permitted. It defaults to `false`.
- **OnEmailAddressValidate** (delegate PeterBlum.DES.ExtendedEmailValidation) – Set up this event handler when you are using a third party email address validation system. Write a method that uses the third party validation engine to analyze the email text and return `true` if it's valid.

  Your method will only be called if the text was not blank and the built-in pattern evaluation approved the text.

  Here is the definition of the delegate:

  [C#]

  ```
  public delegate bool ExtendedEmailValidation(object pSender,
     string pEmailText);
  ```

  [VB]

  ```
  Public Delegate Function ExtendedEmailValidation(_
     ByVal pSender As Object, ByVal pEmailText As String) As Boolean
  ```

  #### Parameters

  pSender

  The EmailAddressCondition object that is getting evaluated.

  pEmailText

  The text from the TextBox. Evaluate this string to confirm that it is a valid email address.

  #### Return value

  When `true`, the pEmailText is a valid email address. When `false`, it is not.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Example

This example uses a fictional object SuperEmailTest to determine if pEmailText is valid or not. Its `Validate()` method returns `true` when it is valid. If you are using Advanced Intellect's aspNetMX product, see below for explicit instructions.

[C#]

```
protected void Page_Load(object sender, System.EventArgs e)
{
// set up the Event Handler on the validator
// notice that it does not use the += notation common to many event handlers
   EmailAddressValidator1.OnEmailAddressValidate =
      new PeterBlum.DES.ExtendedEmailValidation(CheckEmailAddress);
}

protected bool CheckEmailAddress(object pSender, string pEmailText)
{
   return SuperEmailTest.Validate(pEmailText);  // fictional engine
}
```

[VB]

```
Protected Sub Page_Load(ByVal sender As object, ByVal e As System.EventArgs)
' set up the Event Handler on the validator
' notice no AddHandler here, unlike most event handlers
   EmailAddressValidator1.OnEmailAddressValidate =
      New PeterBlum.DES.ExtendedEmailValidation(AddressOf CheckEmailAddress)
End Sub

Protected Function CheckEmailAddress( _
   ByVal pSender As Object, ByVal pEmailText As String) As Boolean
   Return SuperEmailTest.Validate(pEmailText)   ' fictional engine
End Function
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Using aspNetMX

Advanced Intellect's aspNetMX provides a terrific extension to this Validator. While the Validator can only analyze the text pattern to be sure it conforms to the standards of email addresses, it takes aspNetMX to determine if the email address is a real one. This tool can help you avoid filling up your database with useless email addresses. AspNetMX is a separate product available at http://www.aspNetMX.com.

Note: PeterBlum.com does not provide technical support on aspNetMX. You should install and test this product following the aspNetMX documentation and technical support.

Here is how to use it in your web application with DES.

1. Install AspNetMX according to the instructions. It will add the assembly **aspNetMX.dll** into the Global Assembly Cache. Don't forget to install it onto each web server that will use this web application.
2. If you are using Visual Studio, add a reference to **aspNetMX.dll** into your web application project.
3. You will set up the MXValidate object in the `Application_Start()` method of **Global.asax**. Set up involves these actions (examples are shown below):
   - Create the object.
   - Save a reference to it in the Application collection so it's available throughout your web app.
   - Establish any properties you'd like. Consider DnsServer, KnownProperties, and anything else described in the "Tips and Tricks" section of the aspNetMX help file.
4. On the page where you collect Email addresses, set up the `Page_Load()` method with the **OnEmailAddressValidate** event handler.
5. Code the **OnEmailAddressValidate** event handler to retrieve the MXValidate object from the Application object and call its `MXValidate.Validate()` method. AspNetMX recommends that you establish a level of `MXValidateLevel.MXRecords` as described in the "Tips and Tricks" section. Return `true` when the `MXValidate.Validate()` method returns a level that you consider acceptable.

Here is an example of the `Application_Start()`, `Page_Load()`, and CheckEmailAddress event handler methods.

[C#]

```
// Application_Start goes in your Global.asax file
protected void Application_Start(Object sender, EventArgs e)
{
   aspNetMX.MXValidate mx = new aspNetMX.MXValidate();
   Application.Add("aspNetMX", mx);
// set properties. These examples are taken from Tips and Tricks
   mx.DnsServer = "1.2.3.4";
   string domains =
      "hotmail.com;aol.com;yahoo.com;usa.net;bigfoot.com;earthlink.net";
   mx.AddKnownDomains(domains);
   mx.SMTPHello = "myserver.mydomain.com";
   string bad = "test@test.com;a@a.com;asdf@asdf.com";
   mx.AddBadEmails(bad);
   // establish any other attributes you might like
}
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

```
// Page_Load and CheckEmailAddress go into the web form
protected void Page_Load(object sender, System.EventArgs e)
{
// set up the Event Handler on the validator
// notice that it does not use the += notation common to many event handlers
   EmailAddressValidator1.OnEmailAddressValidate =
      new PeterBlum.DES.ExtendedEmailValidation(CheckEmailAddress);
}

protected bool CheckEmailAddress(object pSender, string pEmailText)
{
   aspNetMX.MXValidate mx =
      (aspNetMX.MXValidate) this.Application["aspNetMX"];
// you must determine your preferred levels below
   return mx.Validate(pEmailText, aspNetMX.MXValidateLevel.MXRecords) >=
      aspNetMX.ValidateLevel.MXRecords;
}
```

[VB]

```
' Application_Start goes in your Global.asax file
Sub Application_Start(ByVal sender As Object, ByVal e As EventArgs)
   Dim mx As aspNetMX.MXValidate = New aspNetMX.MXValidate()
   Application.Add("aspNetMX", mx)
' set properties. These examples are taken from Tips and Tricks
   mx.DnsServer = "1.2.3.4"
   Dim domains As String = _
      "hotmail.com;aol.com;yahoo.com;usa.net;bigfoot.com;earthlink.net"
   mx.AddKnownDomains(domains)
   mx.SMTPHello = "myserver.mydomain.com"
   Dim bad As String = "test@test.com;a@a.com;asdf@asdf.com"
   mx.AddBadEmails(bad)
   ' establish any other attributes you might like
End Sub

' Page_Load and CheckEmailAddress go into the web form
Protected Sub Page_Load(ByVal sender As object, ByVal e As System.EventArgs)
' set up the Event Handler on the validator
' notice no AddHandler here, unlike most event handlers
   EmailAddressValidator1.OnEmailAddressValidate =
      New PeterBlum.DES.ExtendedEmailValidation(AddressOf CheckEmailAddress)
End Sub

Protected Function CheckEmailAddress( _
   ByVal pSender As Object, ByVal pEmailText As String) As Boolean
   Dim mx As aspNetMX.MXValidate = _
      CType(Me.Application["aspNetMX"], aspNetMX.MXValidate)
' you must determine your preferred levels below
   Return mx.Validate(pEmailText, aspNetMX.MXValidateLevel.MXRecords)_
       >= aspNetMX.ValidateLevel.MXRecords
End Function
```

Changing the Default Email Address Pattern

There are times when you expect certain email addresses to be rejected that are accepted by this validator. The validator uses a regular expression to determine validity. The default is:

```
^([\w\.!#\$%\-+.'_]+@[A-Za-z0-9\-]+(\.[A-Za-z0-9\-]{2,})+)
```

If you want to edit it, open the **Global Settings Editor** and edit it in the **EmailAddressValidatorRegEx** property in the "Other Validator Properties" topic.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

MultipleRequiredControlsValidator

Condition	MultipleRequiredControlsCondition
License Required	Peter's More Validators
Supported controls	All that declare the ValidationPropertyAttribute including TextBox, ListBox, DropDownList. Also CheckBox and RadioButton
Can evaluate blank fields	Yes

The MultipleRequiredControlsValidator evaluates two or more data entry controls to evaluate if they are blank or have data. This is an extension of the idea behind the RequiredTextValidator and RequiredListValidator by looking at the state of several controls and determine if the right combination has data or not.

Based on the **Mode** property, you can demand the following:

- All
- All or None
- Only one
- At least one
- A range

It supports a mixture of data entry controls. All DES TextBoxes, native TextBox, ListBox, DropDownList, CheckBoxList, RadioButtonList, CheckBox, and RadioButton. With lists, it can evaluate either the current textual value or the selected index.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
  - Adding Textual Controls to ControlsToEvaluate
  - Adding List Controls to ControlsToEvaluate
  - Adding CheckBox/RadioButton Controls to ControlsToEvaluate
  - Determining the number of controls that are required
- Condition Properties for MultipleRequiredControlsValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Using This Condition

Specify the first two textboxes to evaluate with the **ControlIDToEvaluate** and **SecondControlIDToEvaluate** properties. *If you don't have textboxes, it is acceptable to leave them unassigned.* Use the **ControlsToEvaluate** property to define a list of additional controls (textual and list-type).

Normally validators are invoked if a field is changed. If you want this validator to display an error just by tabbing out of a blank textbox, set the **RequiredFieldsReportErrorsOnBlur** property to true on the PageManager control.

Click on any of these topics to jump to them:

- Adding Textual Controls to ControlsToEvaluate
- Adding List Controls to ControlsToEvaluate
- Adding CheckBox/RadioButton Controls to ControlsToEvaluate
- Determining the number of controls that are required

Adding Textual Controls to ControlsToEvaluate

You add `PeterBlum.DES.Web.WebControls.RequiredTextControl` objects to **ControlsToEvaluate** for evaluating text. Assign the textual control's ID to the **RequiredTextControl.ControlIDToEvaluate** property. Define an alternative to what is considered unassigned with the **UnassignedValues** and **UnassignedCaseInsensitive** properties.

Adding to ControlsToEvaluate in ASP.NET Markup

```
<des:MultipleRequiredControlsValidator [properties] >
  <ControlsToEvaluate>
    <des:RequiredTextControl ControlIDToEvaluate="TextBox1"
      UnassignedValues="Fill me in" UnassignedCaseInsensitive="false" />
  </ControlsToEvaluate>
</des:MultipleRequiredControlsValidator>
```

Adding to ControlsToEvaluate Programmatically

Call the `AddTextControl()` method on the **ControlsToEvaluate** property, passing in a reference to the textbox and optionally values for the **UnassignedValues** and **UnassignedCaseInsensitive** properties.

[C#]

```
void AddTextControl(Control pControlToEvaluate);

void AddTextControl(Control pControlToEvaluate, string pUnassignedValues);

void AddTextControl(Control pControlToEvaluate, string pUnassignedValues,
  bool pUnassignedCaseInsensitive);
```

[VB]

```
Sub AddTextControl(ByVal pControlToEvaluate As Control)

Sub AddTextControl(ByVal pControlToEvaluate As Control, _
  ByVal pUnassignedValues As String)

Sub AddTextControl(ByVal pControlToEvaluate As Control, _
  ByVal pUnassignedValues As String, _
  ByVal pUnassignedCaseInsensitive As Boolean)
```

For example:

```
MRCValidator1.ControlsToEvaluate.AddTextControl(TextBox1);
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Adding List Controls to ControlsToEvaluate

Add `PeterBlum.DES.Web.WebControls.RequiredListControl` objects to **ControlsToEvaluate** for evaluating the **SelectedIndex** property of list controls. Assign the list control's ID to the **RequiredListControl.ControlIDToEvaluate** property. Define the index considered unassigned with the **UnassignedIndex** property.

*ALERT: The ASP.NET AJAX Control Toolkit's CascadingDropDown control does not support the **SelectedIndex** property on the server side. It only supports the textual value. Treat its values like a textbox, not a DropDownList. Assign its controls using the ControlsToEvaluate property with a RequiredTextControl, not a RequiredListControl. Set the RequiredTextControl.UnassignedValues property to be "|your prompt text". (That is a pipe character as the first letter.)*

Using ASP.NET Markup

```
<des:MultipleRequiredControlsValidator [properties] >
  <ControlsToEvaluate>
    <des:RequiredListControl ControlIDToEvaluate="ListBox1"
      UnassignedIndex="0" />
  </ControlsToEvaluate>
</des:MultipleRequiredControlsValidator>
```

Programmatically

Call the `AddListControl()` method on the **ControlsToEvaluate** property, passing in a reference to the list control and optionally the value for the **UnassignedIndex** property.

[C#]

```
void AddListControl(Control pControlToEvaluate);

void AddListControl(Control pControlToEvaluate, int pUnassignedIndex);
```

[VB]

```
Sub AddListControl(ByVal pControlToEvaluate As Control)

Sub AddListControl (ByVal pControlToEvaluate As Control, _
  ByVal pUnassignedIndex As Integer)
```

For example:

```
MRCValidator1.ControlsToEvaluate.AddListControl(ListBox1, 0);
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Adding CheckBox/RadioButton Controls to ControlsToEvaluate

You add `PeterBlum.DES.Web.WebControls.RequiredCheckStateControl` objects to **ControlsToEvaluate**. Assign the control's ID to the **RequiredCheckStateControl.ControlIDToEvaluate** property. By default, it assumes an unchecked control is unassigned and checked is assigned. If you want to switch that, set the **RequiredCheckStateControl.CheckedIsAssigned** property to `false`.

Adding to ControlsToEvaluate in ASP.NET Markup

```
<des:MultipleRequiredControlsValidator [properties] >
  <ControlsToEvaluate>
    <des:RequiredCheckStateControl ControlIDToEvaluate="CheckBox1" />
    <des:RequiredCheckStateControl ControlIDToEvaluate="CheckBox2"
      CheckedIsAssigned="false" />
  </ControlsToEvaluate>
</des:MultipleRequiredControlsValidator>
```

Adding to ControlsToEvaluate Programmatically

Call the `AddCheckStateControl()` method on the **ControlsToEvaluate** property, passing in a reference to the CheckBox or RadioButton and if **CheckIsAssigned** should be `false`, pass `false` as the second parameter.

[C#]

```
void AddCheckStateControl(Control pControlToEvaluate);
```

```
void AddCheckStateControl(Control pControlToEvaluate, bool pCheckedIsAssigned);
```

[VB]

```
Sub AddCheckStateControl(ByVal pControlToEvaluate As Control)
```

```
Sub AddCheckStateControl(ByVal pControlToEvaluate As Control, _
  ByVal pCheckedIsAssigned As Boolean)
```

For example:

```
MRCValidator1.ControlsToEvaluate.AddCheckStateControl(CheckBox1);
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Determining the number of controls that are required

Use the **Mode** property to determine how many must be assigned. If you select `Range`, fill in **Minimum** and **Maximum**.

Example 1: TextBoxes

5 textboxes are evaluated. At least one must be assigned.

```
<des:MultipleRequiredControlsValidator id="MRCV1" runat="server"
   ErrorMessage="Fill in at least one."
   ControlIDToEvaluate="TextBox1" SecondControlIDToEvaluate="TextBox2"
   Mode="AtLeastOne">
   <ControlsToEvaluate>
      <des:RequiredTextControl ControlIDToEvaluate="TextBox3" />
      <des:RequiredTextControl ControlIDToEvaluate="TextBox4" />
      <des:RequiredTextControl ControlIDToEvaluate="TextBox5" />
   </ControlsToEvaluate>
</des:MultipleRequiredControlsValidator>
```

Example 2: ListBoxes

3 listboxes are evaluated. All or None must be assigned.

```
<des:MultipleRequiredControlsValidator id="MRCV2" runat="server"
   ErrorMessage="Fill all lists in or leave them unassigned."
   Mode="AllOrNone">
   <ControlsToEvaluate>
      <des:RequiredListControl ControlIDToEvaluate="ListBox1" />
      <des:RequiredListControl ControlIDToEvaluate="ListBox2" />
      <des:RequiredListControl ControlIDToEvaluate="ListBox3" />
   </ControlsToEvaluate>
</des:MultipleRequiredControlsValidator>
```

Example 3: CheckBoxes

3 CheckBoxes are evaluated. All or None must be assigned.

```
<des:MultipleRequiredControlsValidator id="MRCV3" runat="server"
   ErrorMessage="Mark all or leave them unassigned."
   Mode="AllOrNone">
   <ControlsToEvaluate>
      <des:RequiredCheckStateControl ControlIDToEvaluate="CheckBox1" />
      <des:RequiredCheckStateControl ControlIDToEvaluate="CheckBox2" />
      <des:RequiredCheckStateControl ControlIDToEvaluate="CheckBox3" />
   </ControlsToEvaluate>
</des:MultipleRequiredControlsValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Condition Properties for MultipleRequiredControlsValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate** (string) – Identifies a textual data entry control that will be evaluated. This property takes the ID of the control. It is not required. Use it when you have a textbox whose unassigned value is "".

  Add additional controls using the **SecondControlIDToEvaluate** and **ControlsToEvaluate** properties.

- **ControlToEvaluate** (System.Web.UI.Control) – An alternative to **ControlIDToEvaluate**. Use it when the textual data entry control is not in the same or ancestor naming container. It must be assigned programmatically. For example, if you have a Validator instance in the variable "Val1" and a textbox instance in the variable "TextBox1", write code like this: `Val1.ControlToEvaluate = TextBox1`.

  When programmatically assigning properties to a Validator control or Condition class, if you have access to the control object that will be evaluated, it is better to assign it here than assign its ID to the **ControlIDToEvaluate** property because DES operates faster using **ControlToEvaluate**.

- **SecondControlIDToEvaluate** (string) – Identifies a textual data entry control that will be evaluated. This property takes the ID of the control. It is not required. Use it when you have a textbox whose unassigned value is "".

  Add additional controls using the **ControlsToEvaluate** property.

  If you want to connect to a control that is inside of a usercontrol, see "Attaching Validators to controls inside of a User Control".

- **SecondControlToEvaluate** (System.Web.UI.Control) – An alternative to **SecondControlIDToEvaluate**. Use it when the textual data entry control is not in the same or ancestor naming container. It must be assigned programmatically. For example, if you have a Validator instance in the variable "Val2" and a textbox instance in the variable "TextBox2", write code like this: `Val2.SecondControlToEvaluate = TextBox2`.

  When programmatically assigning properties to a Validator control or Condition class, if you have access to the control object that will be evaluated, it is better to assign it here than assign its ID to the **SecondControlIDToEvaluate** property because DES operates faster using **SecondControlToEvaluate**.

- **SecondValueToEvaluate** (string) – Allows the Validator or Condition to be used without a control attached. Instead, data comes from the value assigned to this property. When used, it replaces **SecondControlIDToEvaluate** and **SecondControlToEvaluate**.

  See "Using Validator controls to evaluate data not from Controls".

- **ControlsToEvaluate** (PeterBlum.DES.Web.WebControls.RequiredControlsToEvaluate) – Create a list of data entry controls to evaluate in addition to those defined in **ControlIDToEvaluate** and **SecondControlIDToEvaluate**. It can contain a mixture of control types.

  When using a textual data entry controls, add a `PeterBlum.DES.Web.WebControls.RequiredTextControl` class to the list. See "Adding Textual Controls to ControlsToEvaluate".

  When evaluating the **SelectedIndex** of a list type control, add a `PeterBlum.DES.Web.WebControls.RequiredListControl` class to the list. See "Adding List Controls to ControlsToEvaluate".

  When evaluating a CheckBox or RadioButton control, add a `PeterBlum.DES.Web.WebControls.RequiredCheckStateControl` class to the list. See "Adding CheckBox/RadioButton Controls to ControlsToEvaluate".

- **Mode** (enum PeterBlum.DES.MultipleRequiredControlsMode) – Determines how many of the data entry controls must have data to avoid an error.

  It determines the data entry controls through your assignments made to the **ControlIDToEvaluate**, **SecondControlIDToEvaluate** and **ControlsToEvaluate** properties.

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

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- `All` – All must have data.
- `AllOrNone` – Either all must have data or none have data.
- `AtLeastOne` – At least one must have data.
- `OnlyOne` – Only one can have data.
- `Range` – Count the number that have data. That number of fields must be inside a range determined by the **Minimum** and **Maximum** properties.

It defaults to `MultipleRequiredControlsMode.All`.

- **Minimum** (integer) – When **Mode** is set to `Range`, it determines the minimum number of data entry controls that are required. If 0, it is ignored. It defaults to 0.
- **Maximum** (integer) – When **Mode** is set to `Range`, it determines the maximum number of data entry controls that are required. If 0, it is ignored. It defaults to 0.
- **ReportErrorsAfter**, **Trim** and **NotCondition**– See "Properties Common to most Conditions".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

RequiredSelectionValidator

Condition	RequiredSelectionCondition
License Required	Peter's More Validators
Supported controls	ListView, DataList, Calendar, MultiSelectionCalendar, MonthYearPicker, TimePicker. It is extendable.
Can evaluate blank fields	Yes

The RequiredSelectionValidator is another validator that checks for required data to be assigned. In this case, it is for non-textual and non-list controls that display data, usually in a HTML table. Examples are ListView, Calendar, and MonthYearPicker. All of these are supported automatically by default.

When your data is textual, use the RequiredTextValidator. When it's a list build upon HTML form elements (`<select>` and `<input>`, use RequiredListValidator.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Supporting Other Controls
- Condition Properties for RequiredSelectionValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Using This Condition

Specify the control to evaluate with the **ControlIDToEvaluate** property.

When you use a ListView or DataList control, it only performs server side validation, using the control's **SelectedIndex** property. When **SelectedIndex** is -1, it does not have a selection.

When you use a DES Calendar, MultiSelectionCalendar, MonthYearPicker or TimePicker, it performs client and server side validation.

Example

```
<des:RequiredSelectionValidator id="RequiredSelectionValidator1" runat="server"
   ControlIDToEvaluate="Calendar1" ErrorMessage="Required">
</des:RequiredSelectionValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Supporting Other Controls

You can extend this validator to support many controls that display data without using standard form elements of `<input>`, `<textarea>`, or `<select>`. You will define a subclass of `PeterBlum.DES.Web.BaseHtmlControlValueAdapter<System.Web.UI.Control>` object with its `IsValueNull()` method returning `true` if there is a selection and `false` if not. Then add it with the `PeterBlum.DES.ControlAdapterFactory.Register()` method within the `Application_Start()` method of **Global.asax**.

Here is a snippit from the CalendarControlValueAdapter, which handles the DES's Calendar, where its `IsEmpty()` method returns true to indicate no selection.

[C#]

```
public class CalendarControlValueAdapter :
   PeterBlum.DES.Web.WebControls.BaseSWUControlValueAdapter
{
   public CalendarControlValueAdapter(Control pControl) : base(pControl) { }
   public CalendarControlValueAdapter() : base() { }
   public override bool HasNullValues(Control pControl)
   {
      return true;
   }
   public override bool IsValueNull(Control pControl)
   {
      Calendar vCalendar = EnsureControl(pControl);
      return vCalendar.IsEmpty();
   }
}
```

[VB]

```
Public Class CalendarControlValueAdapter
   Inherits PeterBlum.DES.Web.WebControls.BaseSWUControlValueAdapter
   Public Sub New(pControl As Control)
      MyBase.New(pControl)
   End Sub
   Public Sub New()
      MyBase.New()
   End Sub
   Public Overrides Function HasNullValues(pControl As Control) As Boolean
      Return True
   End Function
   Public Overrides Function IsValueNull(pControl As Control) As Boolean
      Dim vCalendar As Calendar = EnsureControl(pControl)
      Return vCalendar.IsEmpty()
   End Function
End Class
```

Adding your class to the ControlAdapterFactory

In the `Application_Start()` method of **Global.asax**, supply the type of your class as the first parameter of the Register() method and true as the second parameter:

[C#]

```
PeterBlum.DES.ControlAdapterFactory.Register(typeof(yourclass), true);
```

[VB]

```
PeterBlum.DES.ControlAdapterFactory.Register(GetType(yourclass), True)
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Setting up Client Side Validation

Client-side validation can only be supported if you write a JavaScript function that takes the id of the control and returns `true` if the value has NO selection and `false` if it has a selection. (This function is an "Is Null Value" function.) Some third party controls may already have such a function written.

Here is the function used by DES's Calendar control:

```
function DES_CALIsValNull(pID)
{
   return DES_CALGetUTCDate(pID) == null;
}
```

Now update your ControlValueAdapter class, described in the previous section, to return that function as a string by overriding the `GetIsValueNullScript()` method. For using the Calendar example:

[C#]

```
public class CalendarControlValueAdapter :
   PeterBlum.DES.Web.WebControls.BaseSWUControlValueAdapter
{
   public CalendarControlValueAdapter(Control pControl) : base(pControl) { }
   public CalendarControlValueAdapter() : base() { }
   public override bool HasNullValues(Control pControl)
   {
      return true;
   }
   public override bool IsValueNull(Control pControl)
   {
      Calendar vCalendar = EnsureControl(pControl);
      return vCalendar.IsEmpty();
   }
   protected override string GetIsValueNullScript()
   {
      return "function DES_CALIsValNull(pID) {" +
         "return DES_CALGetUTCDate(pID) == null; }";
   }
}
```

[VB]

```
Public Class CalendarControlValueAdapter
   Inherits PeterBlum.DES.Web.WebControls.BaseSWUControlValueAdapter
   Public Sub New(pControl As Control)
      MyBase.New(pControl)
   End Sub
   Public Sub New()
      MyBase.New()
   End Sub
   Public Overrides Function HasNullValues(pControl As Control) As Boolean
      Return True
   End Function
   Public Overrides Function IsValueNull(pControl As Control) As Boolean
      Dim vCalendar As Calendar = EnsureControl(pControl)
      Return vCalendar.IsEmpty()
   End Function
   Protected Overrides Function GetIsValueNullScript() As String
      Return "function DES_CALIsValNull(pID) {" +
         "return DES_CALGetUTCDate(pID) == null; }";
   End Function
End Class
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Condition Properties for RequiredSelectionValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, and **NotCondition** – See "Properties Common to most Conditions".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

CharacterValidator

Condition	CharacterCondition
License Required	Peter's More Validators
Supported controls	All that declare the ValidationPropertyAttribute including TextBox, ListBox, DropDownList
Can evaluate blank fields	Yes when the IgnoreBlankText property is false

The CharacterValidator determines if your textbox has text where every character is in a character set that you specify. It doesn't define a pattern or order to the characters. Patterns, like phone numbers or addresses, need the power of the RegexValidator. Some examples where a limited character set applies:

- Passwords – perhaps you only allow letters, digits, space and underscore
- Credit card numbers – only allow digits
- Person's first name – only allow letters

When used with the FilteredTextBox, it automatically uses the FilteredTextBox's properties to define the character set. It ignores properties that define the character set on itself.

Click on any of these topics to jump to them:

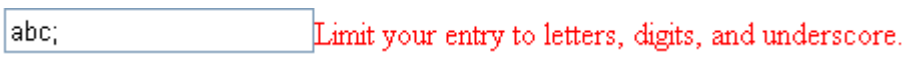
- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for CharacterValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Using This Condition

Specify the control to evaluate with the **ControlIDToEvaluate** property. Use these Boolean properties to determine the set of characters: **LettersUppercase**, **LettersLowercase**, **DiacriticLetters**, **Digits**, **Space**, **Enter**, **Punctuation**, **CurrencySymbols**, **EnclosureSymbols**, **MathSymbols**, and **VariousSymbols**. For any other characters or to define a subset of the predefined character definitions, use the **OtherCharacters** property to list the characters you want. If you want all characters except a specific character set, define the character set using the preceding properties and set the property **Exclude** to `true`.

Example

Limit TextBox1 to only letters, digits, and the underscore character.




```
<des:CharacterValidator id="CharV1" runat="server"
   ErrorMessage="Limit your entry to letters, digits, and underscore."
   ControlIDToEvaluate="TextBox1"
   LettersUppercase="true" LettersLowercase="true"
   Digits="true" OtherCharacters="_">
</des:CharacterValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Condition Properties for CharacterValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

Note: When used with the FilteredTextBox, this Condition automatically uses the FilteredTextBox's properties to define the character set. The following properties are ignored: LettersLowercase, LettersUppercase, Digits, Space, Enter, OtherCharacters, and Exclude.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, and **Trim** – See "Properties Common to most Conditions".
- **LettersLowercase** (Boolean) – When `true`, all lowercase letters are part of the characterset. When `false`, no lowercase letters are included unless they are specified in the **OtherCharacters** property. It defaults to `false`.
- **LettersUppercase** (Boolean) – When `true`, all uppercase letters are part of the characterset. When `false`, no uppercase letters are included unless they are specified in the **OtherCharacters** property. It defaults to `false`.
- **DiacriticLetters** (Boolean) – When `true`, diacritic (accented) letters are part of the characterset. There are two groups of diacritics available to users: ASCII and Unicode. They are both supported.

  When `false`, no diacritic letters are included unless they are specified in the **OtherCharacters** property. It defaults to `false`.

  ASCII Diacritic Letters Supported

  `âäàåáÄÅ çÇ éêëèÉ æÆ ƒ ïîìí ñÑ ôöòóÖ üûùúÜ ÿ`

  These letters are found in the ASCII character set between ASCII 128 and 165. Users on Windows type `ALT+1##` to insert them into the textbox. Users on MacOSX use **Edit; Special Characters** to open the Character Palette. These characters are in the Glyph View.

  Unicode Diacritic Letters Supported

  `àáâãäåÀÁÂÃÄÅ æÆ çÇ èéêëÈÉÊË ìíîïÌÍÎÏ ñÑ òóôõöøÒÓÔÕÖØ ùúûüÙÚÛÜ ýÿÝ ðÐ þÞ ß`

  These letters are found in Unicode's Latin-1 Supplement character set between 0192 (Hex 00C0) and 0255 (Hex 00FF). Users on Windows type `ALT+0###` to insert them into the textbox. Users on MacOSX use **Edit; Special Characters** to open the Character Palette. These characters are in the Unicode View.

- **Digits** (Boolean) – When `true`, all digits are part of the characterset. When `false`, no digits are included unless they are specified in the **OtherCharacters** property. It defaults to `false`.
- **Space** (Boolean) – When `true`, the Space character is part of the characterset. It defaults to `false`. The **Trim** property automatically strips leading and trailing spaces when `true`. It will not affect embedded spaces.
- **Enter** (Boolean) – When `true`, the Enter character is part of the characterset. It defaults to `false`.
- **Punctuation** (Boolean) – When `true`, punctuation characters are part of the characterset. When `false`, no punctuation characters are included unless they are specified in the **OtherCharacters** property. It defaults to `false`.

  Punctuation Characters Supported

  `. , ! ? ' " - ; :`

  (period, comma, exclamation point, question mark, single quote, double quote, dash, semicolon, colon)

  ALERT: Hackers use the single quote and dash for SQL Injection attacks. If you permit these characters, see the DES: Peter's Input Security User's Guide for defensive measures.

- **CurrencySymbols** (Boolean) – When `true`, currency symbol characters are part of the characterset. When `false`, no currency symbol characters are included unless they are specified in the **OtherCharacters** property. It defaults to `false`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Currency Symbols Supported

`$ ¢ ¥ ¥ £ ¤`

(US Dollar, cents – UNICODE 0162, Yen – ASCII 157, Yen – UNICODE 0164, Pounds – UNICODE 0163)

- **EnclosureSymbols** (Boolean) – When `true`, characters that enclose (bracket) text are part of the characterset. When `false`, no enclosure characters are included unless they are specified in the **OtherCharacters** property. It defaults to `false`.

Enclosure Symbols Supported

`( ) [ ] { }`

- **MathSymbols** (Boolean) – When `true`, math symbol characters are part of the characterset. When `false`, no math symbol characters are included unless they are specified in the **OtherCharacters** property. It defaults to `false`.

Math Symbols Supported

`+ - * / = ( ) < > . % ± × ÷ ± ÷`

(plus, minus, asterisk, equals, left paren, right paren, less than, greater than, period, percent, plus/minus – UNICODE 0177, multiply – UNICODE 0215, divide – UNICODE - 0247, divide – ASCII 246)

> *ALERT: Hackers use the dash for SQL Injection attacks. If you permit the dash, see the DES: Peter's Input Security User's Guide for defensive measures.*

- **VariousSymbols** (Boolean) – When `true`, various symbol characters shown below are part of the characterset. When `false`, none of the characters shown below are included unless they are specified in the **OtherCharacters** property. It defaults to `false`.

Various Symbols Supported

`_ @ # ^ & * ~ ¿ ¿ ¡ \ / | ¦ § © ® ` ´`

(Underscore, at, circumflex, ampersand, asterisk, tilde, inverted question mark – ASCII 168, inverted question mark – UNICODE 0191, inverted exclamation point – UNICODE 0161, left slash, right slash, pipe, broken bar, section sign, copyright, registered, grave accent, acute accent)

- **OtherCharacters** (string) – Enter each unique character that you want in the character. Don't enter any characters that are already covered by the previously stated properties. Suppose you want to support only "n", "x", and the underscore character here. You would enter "`nx_`". Suppose you want some common punctuation characters. You would enter "`.!,?;:`".

- **Exclude** (Boolean) – When `true`, the character set defined in the above properties is excluded; all other characters are permitted. When `false`, the character set defined above are the only characters permitted. It defaults to `false`.

- **IgnoreBlankText** (Boolean) – Determines how blank text is evaluated. When `true`, the condition cannot be evaluated. When `false`, the condition evaluates as failed (reports an error). It defaults to `true`.

- **GlobalStorageName** (string) – Use this property to improve speed. This Condition uses a .net Regex object to evaluate text. It takes time for a .net Regex object to convert a regular expression into its internal format. With this setting that will only happen once per application start, and it is further optimized by using the "Compiled" option on the .net Regex object.

  Do not use this when the characterset will change. It should only be used when the characterset remains identical while the application runs.

  You define a unique name for this set of rules and assign it to **GlobalStorageName**. For example, "PersonsName" or "VAL_PhoneNumber".

  If you have several CharacterValidators that use identical rules, use the same value of **GlobalStorageName** so that they all share the same .net Regex object. This further improves your application's speed.

  When used, the .net Regex object is stored in the Application collection. `Application[GlobalStorageName]` is assigned the string containing the regular expression. `Application[GlobalStorageName + "_RE"]` is

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

assigned the Regex object. So give this a name that will not conflict with anything else that you place into the Application collection.

WARNING: Several other Validators offer the GlobalStorageName property. Never use the same value amongst different types of Validators.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

CompareToStringsValidator

Condition	CompareToStringsCondition
License Required	Peter's More Validators
Supported controls	All that declare the ValidationPropertyAttribute including TextBox, ListBox, DropDownList
Can evaluate blank fields	Yes when the IgnoreBlankText property is false

The CompareToStringsValidator determines if your textbox has text that matches to any item in a list of strings that you supply. For example, you can define a list of UserNames to compare to the textbox. *Note: This control has to copy the list of the strings to the client-side if you allow the control to validate on the client-side. When you supply a list of user names or passwords, you should set **EnableClientScript** to false to force server-side evaluation or use **OverrideClientSideEvaluation** to provide a lightweight client-side interface.*

This control supports retrieving data from a data source such as a DataSet, DataTable or a `System.Collections.IEnumerable` collection, in the same way you populate a ListBox or DropList.

You can establish a rule where the text can match to a string in the list in the following ways: exact match, contained within a string, start the string, or end the string.

Note: This Validator uses a Regular Expression to evaluate the list of strings. In fact, it is subclassed from RegexValidator and RegexCondition.

Note: This Validator is not meant for long lists of strings, especially when you validate on the client side. It has to copy all of the strings to the client-side, which slows transmission times. There are likely limits to the size of a regular expression pattern. Test your page against the data to be sure this Validator will work for you.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for CompareToStringsValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Using This Condition

Specify the control to evaluate with the **ControlIDToEvaluate** property. The data comes from three properties that are familiar if you have used databinding on ListBoxes or DataGrids: **DataSource**, **DataMember**, and **DataTextField**. However, you don't need to call the `DataBind()` method unless you use databinding notation in the ASP.NET text (as shown below.) Alternatively, you can add strings to the **Items** property.

fish Enter a category

This example uses the **DataSource** property.

```
<des:CompareToStringsValidator id="CTSV1" runat="server"
   ErrorMessage="Enter a category" ControlIDToEvaluate="TextBox1"
   DataMember="Categories" DataSource="<%# dataSet1 %>"
   DataTextField="CategoryName">
</des:CompareToStringsValidator>
```

This example uses the **Items** property.

```
<des:CompareToStringsValidator id="CTSV2" runat="server"
   ErrorMessage="Enter a category" ControlIDToEvaluate="TextBox1" >
<Items>
   <des:CompareToStringsItem Value="hardware" />
   <des:CompareToStringsItem Value="software" />
</Items>
</des:CompareToStringsValidator>
```

It defaults to a case insensitive match. If you prefer case sensitive, set **CaseInsensitive** to `false`.

It defaults to matching the full string entered to each string. Use the **Mode** property to change it to match these strings to the start of the string, end of the string, or contained in the string.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Condition Properties for CompareToStringsValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, **ValueToEvaluate**, **Trim**, and **NotCondition** – See "Properties Common to most Conditions".
- **Items** (PeterBlum.DES.CompareToStringsItems) – This collection contains the strings which will be compared against the value of the textbox. Add `PeterBlum.DES.CompareToStringsItem` objects. These objects have one property, **Value**, where you assign the string.

  You can also populate this collection using a data source defined with the **DataSource**, **DataMember**, and **DataTextField** properties.

  #### Entering Items in ASP.NET Markup

  Strings are defined in `PeterBlum.DES.CompareToStringsItem` objects.

  Define strings like this:

  ```
  <Items>
     <des:CompareToStringsItem Value="string1" />
     <des:CompareToStringsItem Value="string2" />
  </Items>
  ```

  #### Entering Items Programmatically

  Use the `Add()` method on the **Items** property. Pass a string or `PeterBlum.DES.CompareToStringsItem` object.

  [C#]

  ```
  CompareToStringValidator1.Items.Add("string1");
  ```

  [VB]

  ```
  CompareToStringValidator1.Items.Add("string1")
  ```

- **CaseInsensitive** (Boolean) – When `true`, the regular expression parser will perform a case insensitive match. When `false`, it will perform a case sensitive match. It defaults to `true`.
- **MatchTextRule** (enum PeterBlum.DES.MatchTextRule) – Determines how the value of the text field is compared to each string in the list. This enumerated type has these values:
  - `Exact` – The value must exactly match a string. This is the default.
  - `Contains` – The value must be somewhere within a string.
  - `StartsWith` – The value must appear at the start of a string.
  - `EndsWith` – The value must appear at the end of a string.
- **IgnoreBlankText** (Boolean) – Determines how blank text is evaluated. When `true`, the condition cannot be evaluated. When `false`, the condition evaluates as failed (reports an error). It defaults to `true`.
- **ItemsByDelimitedString** (string) – *Only available on the Condition, not on the validator.* An alternative to defining the **Items** property through child CompareToStringsItems or using a Data Source, use this to supply a pipe delimited list of strings to compare. For example, "Blue|Red|Green".

  If used, the **Items** collection is not used, nor are the **DataSource** and related properties.

  It is unassigned by default.

  This property was created as a way to use the CompareToStringsCondition inside of a `PeterBlum.DES.DataAnnotations.ConditionDependencyAttribute` because Attributes located in metadata cannot have child elements.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- **DataSource** (object) – You can populate the **Items** property from a data source using this property. Your data source can be a DataSet, DataTable, or a collection that implements `System.Collections.IEnumerable` such as an ArrayList. If you use the design time features of the Properties Editor, you can select a DataSet object already on your design mode surface with the "(DataBindings)" property.
    - When using a DataSet, also set the **DataMember** property to the name of the table that contains the column you are using. You can leave DataMember blank only if DataSet only has one table. Always set **DataTextField** to the name of the column that you are using.
    - When using a DataTable, set the **DataTextField** property to the name of the column that you are using.
    - When using a collection that implements IEnumerable, you can populate it with strings or any object class that has a string property with the desired data. If you use an object that has a string property, give the name of that property in the **DataTextField** property.

    Note: If you set up the DataSource property in the Properties Editor, you must call the DataBind() method on the Validator in your Page_Load() method. You do not need to call DataBind() when you assign this property programmatically.

- **DataMember** (string) – When using the **DataSource** property with a DataSet, assign the name of the table within that DataSet that contains the column you are using. It defaults to "".

- **DataTextField** (string) – When using the **DataSource** property, assign this to the name of the column or property that contains the data. When it is a DataSet or DataTable, this should be the name of the column within a table. When it is an IEnumerable collection, it should be the name of a property on an object. When it is a collection that only contains strings, leave this blank. It defaults to "".

- **GlobalStorageName** (string) – Use this property to improve speed. This Condition uses a .net Regex object to evaluate text. It takes time for a .net Regex object to convert a regular expression into its internal format. With this setting that will only happen once per application start, and it is further optimized by using the "Compiled" option on the Regex object.

    Do not use this when the list of strings will change. It should only be used when the list of strings remains identical while the application runs.

    You define a unique name for this expression and assign it to **GlobalStorageName**. For example, "ReviewStatusNames" or "123abc".

    If you have several CompareToStringValidators that use identical rules, use the same value of **GlobalStorageName** so that they all share the same Regex object. This further improves your application's speed.

    When used, the .net Regex object is stored in the Application collection. `Application[GlobalStorageName]` is assigned the string containing the regular expression. `Application[GlobalStorageName + "_RE"]` is assigned the Regex object. So give this a name that will not conflict with anything else that you place into the Application collection.

    WARNING: Several other Validators offer the GlobalStorageName property. Never use the same value amongst different types of Validators.

- **PrepareProperties** (delegate PeterBlum.DES.RegexPrepareProperties) – When using the **GlobalStorageName** and you run time consuming code to create the list of strings, put your time consuming code in a method assigned to **PrepareProperties**. Your code will only be called once and its results will be cached. This increases the application speed by avoiding that time consuming code when it's not needed.

    This is recommended when you use the **DataSource** property as database queries are time consuming.

    Your method will update the properties of the CompareToStringsCondition, the Condition object of the CompareToStringsValidator. The CompareToStringsCondition and CompareToStringsValidator share the same properties described throughout this section. Normally you assign the **Items** or **DataSource** property. However, any of the properties can be changed.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Here is the format of a method assigned to **PrepareProperties**.

[C#]

```
public void MethodName(PeterBlum.DES.IRegexCondition pCondition)
```

[VB]

```
Public Sub MethodName(ByVal pCondition As PeterBlum.DES.IRegexCondition)
```

Parameters

pCondition

The Condition class for the CompareToStringsValidator. It is declared as RegexCondition, which is a base class to the CompareToStringsCondition. So typecast *pCondition* to `PeterBlum.DES.Web.WebControls.CompareToStringsCondition` to access its properties. CompareToStringsCondition has the same properties as shown in this section.

[C#]

```
MyValidator.PrepareProperties =
   new PeterBlum.DES.RegexPrepareProperties(methodname);
```

[VB]

```
MyValidator.PrepareProperties = _
   New PeterBlum.DES.RegexPrepareProperties(AddressOf methodname)
```

Example

Assigns a `DataSet` to the **DataSource** property after populating it. The sqlDataAdapter1 is a `SQLDataAdapter` already on the page and fully attached to a `SqlConnection` and `SqlCommand` object. dataSet1 is a `DataSet` already on the page. Assume that they are all objects on this page generated by Visual Studio.net. The **DataMember** and **DataTextName** properties retain their previously assigned values in this example.

[C#]

```
public void ApplyExpression(PeterBlum.DES.IRegexCondition pRegexCondition)
{
   sqlDataAdapter1.Fill(dataSet1);
   ((PeterBlum.DES.Web.WebControls.CompareToStringsCondition)pCondition).
         DataSource = dataSet1;
}
```

[VB]

```
Public Sub ApplyExpression(ByVal pRegexCondition As _
      PeterBlum.DES.IRegexCondition)
   sqlDataAdapter1.Fill(dataSet1)
   CType(pCondition, _
      PeterBlum.DES.Web.WebControls.CompareToStringsCondition).DataSource = _
      dataSet1
End Sub
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

WordCountValidator

Condition	WordCountCondition
License Required	Peter's More Validators
Supported controls	TextBox, HtmlInputText
Can evaluate blank fields	Yes

The WordCountValidator evaluates the number of words in a textbox against a minimum and/or maximum. It can optionally evaluate the total words in two textboxes.

A "word" is defined as any unbroken string of these characters: A-Z, a-z, 0-9, underscore (_) and single quote ('). All other characters are word separators.

DES provides the TextCounter control in the **Peter's Interactive Pages** module. It can compliment the TextLengthValidator.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for WordCountValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Using This Condition

Specify the control to evaluate with the **ControlIDToEvaluate** property; if you want to evaluate the total of two controls, specify the second in **SecondControlIDToEvaluate**.

The **Minimum** and **Maximum** properties determine the limits when they are not 0. When using the **Minimum** property, if you do not want errors reported for an empty textbox, set **IgnoreBlankText** to `true`.

The Validator's **ErrorMessage** property supports several tokens. "{COUNT}" is the number of words entered. "{EXCEEDS}" is the number of words exceeding the minimum or maximum. "{EXCEEDS:word:words}" uses the word "word" when {EXCEEDS} is 1; otherwise it uses the word "words". "{MINIMUM}" is replaced by the property **Minimum**. "{MAXIMUM}" is replaced by the property **Maximum**. See "Tokens in Error Messages" for details on tokens.

Example

```
When in the course of human events,
there comes a time for all programmers
to find entertainment away from the
computer.
```

Please limit your entry to 15 words. You have entered 21 words.

```
<des:WordCountValidator id="WordCountValidator1" runat="server"
   ControlIDToEvaluate="TextBox1" Maximum="15"
   ErrorMessage="Please limit your entry to {MAXIMUM} words.
      You have entered {COUNT} {COUNT:word:words}.">
</des:WordCountValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Condition Properties for WordCountValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, **ValueToEvaluate**, **ReportErrorsAfter**, **Trim**, and **NotCondition** – See "Properties Common to most Conditions".
- **Minimum** (Integer) – The minimum number of words in the textbox. When it is 0, there is no minimum. It defaults to 0.
- **Maximum** (Integer) – The maximum number of words in the textbox. When it is 0, there is no maximum. It defaults to 0.
- **IgnoreBlankText** (Boolean) – By default, when you use the **Minimum** property, a blank textbox will be reported as an error. If you do not require text in the textbox, set **IgnoreBlankText** to `true`. It defaults to `false`.
- **SecondControlIDToEvaluate** (string) – Identifies a second TextBox control. When assigned, the total words in the two textboxes are added together before comparing to the minimum and maximum. Leave it blank if you only need to evaluate one TextBox. This property takes the ID of the control.

  An exception is thrown at runtime when this is unknown, not in the same or ancestor naming container, is **Visible**=`false`, or a control class that is not supported.

  If you want to connect to a control that is inside of a usercontrol, see "Attaching Validators to controls inside of a User Control".
- **SecondControlToEvaluate** (System.Web.UI.Control) – An alternative to **SecondControlIDToEvaluate**. Use it when the TextBox control is not in the same or ancestor naming container. It must be assigned programmatically. For example, if you have a Validator instance in the variable "Val2" and a textbox instance in the variable "TextBox2", write code like this: `Val2.SecondControlToEvaluate = TextBox2`.

  When programmatically assigning properties to a Validator control or Condition class, if you have access to the control object that will be evaluated, it is better to assign it here than assign its ID to the **SecondControlIDToEvaluate** property because DES operates faster using **SecondControlToEvaluate**.
- **SecondValueToEvaluate** (string) – Allows the Validator or Condition to be used without a control attached. Instead, data comes from the value assigned to this property. When used, it replaces **SecondControlIDToEvaluate** and **SecondControlToEvaluate**.

  See "Using Validator controls to evaluate data not from Controls".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

DifferenceValidator

Condition	DifferenceCondition
License Required	Peter's More Validators
Supported controls	All that declare the ValidationPropertyAttribute including TextBox, ListBox, DropDownList
Can evaluate blank fields	No (Use RequiredTextValidator or MultipleRequiredControlsValidator on both fields)

The DifferenceValidator evaluates two textbox fields to determine if the difference between their two values is a certain number apart from each other. You specify the data type, a value to compare to the difference and the operator.

The data type can be any of these:

- Integer – For example, they must be at least a value of 5 apart
- Double – For example, they must be within 1.5 of each other.
- Currency – For example, they must be exactly $10.00 difference
- Date – For example, test that two dates are less than 60 days apart.
- Time and Duration – The number of seconds between the two fields.
- DateTime – The number of seconds between two pairs of Date and TimeOfDay TextBoxes.

> *Note: Integer, Double, and Currency offer several forms of data types such as Positive Integer and Currency with Symbol. All of these are supported.*

> *Note: Time, Duration, and DateTime only evaluate controls found in the Peter's Date and Time module.*

The DifferenceCondition first determines the numeric value of the two text items. (Dates have a numeric value too; the number of days.) Then it subtracts them and keeps the absolute value of the result. It uses the operator you select (less than, equal, etc.) and compares the difference to the value you supply.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for DifferenceValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Using This Condition

Specify the two controls to evaluate with the **ControlIDToEvaluate** and **SecondControlIDToEvaluate** properties. The **DataType** property specifies the data type. Data types like date and currency will follow culture specific formatting set on the current web page or overridden on **PeterBlum.DES.Globals.WebFormDirector.CultureInfo**.

Use the **Operator** property for the operator and the **DifferenceValue** property for the numeric difference between the two fields. The value in **DifferenceValue** can have different meanings. If you enter 1, do you mean 1 hour, 1 minute, or 1 day? To clarify that meaning, use the **OneEquals** property.

This Validator cannot evaluate when either control's text is blank or it cannot be converted to the data type. Use the RequiredTextValidator and DataTypeCheckValidator to handle each of those cases or combine their conditions within the MultiConditionValidator to provide one error message.

The Validator's **ErrorMessage** property supports several tokens. "{DIFFERENCEVALUE} is replaced by the property **DifferenceValue**. "{DIFFRESULT}" is replaced by the calculated difference. See "Tokens in Error Messages" for details on tokens.

Example

2/1/2003 thru 4/15/2003 Keep the dates within 60 days of each other. They are 73 days apart.

```
<des:DifferenceValidator id="DifferenceValidator1" runat="server"
   ControlIDToEvaluate="TextBox1" SecondControlIDToEvaluate="TextBox2"
   DataType="Date" Operator="LessThanEqual" DifferenceValue="60"
   ErrorMessage="Keep the dates within {DIFFERENCEVALUE} days of each other.
      They are {DIFFRESULT} days apart.">
</des:DifferenceValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Condition Properties for DifferenceValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, **ValueToEvaluate**, **ReportErrorsAfter**, **Trim**, and **NotCondition** – See "Properties Common to most Conditions".
- **SecondControlIDToEvaluate** (string) – Identifies the second data entry control that will be evaluated. This property takes the ID of the control. When a Validator control or Condition class includes this property, it must be assigned unless you are using the **SecondControlToEvaluate** property, the CustomValidator or CustomCondition.

  An exception is thrown at runtime when this is blank, unknown, not in the same or ancestor naming container, is **Visible**=`false`, or a control class that is not supported.

  If you want to connect to a control that is inside of a usercontrol, see "Attaching Validators to controls inside of a User Control".
- **SecondControlToEvaluate** (System.Web.UI.Control) – An alternative to **SecondControlIDToEvaluate**. Use it when the data entry control is not in the same or ancestor naming container. It must be assigned programmatically. For example, if you have a Validator instance in the variable "Val2" and a textbox instance in the variable "TextBox2", write code like this: `Val2.ControlToEvaluate = TextBox2`.

  When programmatically assigning properties to a Validator control or Condition class, if you have access to the control object that will be evaluated, it is better to assign it here than assign its ID to the **SecondControlIDToEvaluate** property because DES operates faster using **SecondControlToEvaluate**.
- **SecondValueToEvaluate** (object) – Allows the Validator or Condition to be used without a control attached. Instead, data comes from the value assigned to this property. The value must be the native type for the specified **DataType**, or a string that can convert to the native type. When used, it replaces **SecondControlIDToEvaluate** and **SecondControlToEvaluate**.

  See "Using Validator controls to evaluate data not from Controls".
- **DataType** (string) – The name of the data type that this condition evaluates. If the user does not enter a string that is compatible with this data type, the condition will not evaluate. (So use a DataTypeCheckValidator to detect errors in the format.) It defaults to "String". You can change it to any of these values: "String-Case Insensitive", "Integer", "Double", "Date", "Time", "Duration", "Currency", "Currency with Symbol", "Positive Integer", "Positive Double", "Positive Currency", and "Positive Currency with Symbol".

  Numeric, currency, and date types will follow culture specific formatting set on the current web page or overridden on **PeterBlum.DES.Globals.WebFormDirector.CultureInfo**. Integers are represented by signed 32 bit integer values. Any larger integer value than 2,147,483,648 will not be evaluated.

  Dates are compared as a number of days between the two fields. For example, the difference between January 1, 2003 and January 2, 2003 is one day. Times are compared as a number of seconds.

  Do not use String or String-Case Insensitive. They are not supported.

  When using any of the numeric, date or time textboxes from the **Peter's TextBoxes** or **Peter's Date and Time** modules, this property is almost always ignored. The textbox will tell the validator exactly how it needs to be configured. **DataType** is only used when evaluating a TimeOfDayTextBox with the **DateTextBoxControlID** assigned. In that case, set **DataType** to `DateTime` or `Time` to select whether to validate both DateTextBox+TimeOfDayTextBox together or just the TimeOfDayTextBox alone.

  You can programmatically define new data types. See the Developer's Guide for details.

CONTINUED ON THE NEXT PAGE

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

- **DifferenceValue** (Double) – The value to be compared to the difference between the two values. (The difference is always an absolute value.) It must be 0 or higher. (When set to 0, you have effectively made a CompareTwoFieldsCondition.)

  This value is the right side of the expression. The calculated difference is on the left side.

  Use the **OneEquals** property to clarify what the value means, such as 1 means 1 minute or 1 hour.

- **Operator** (enum PeterBlum.DES.ConditionOperator) – Determines how the calculated difference is compared to **DifferenceValue**. The enumerated type `PeterBlum.DES.ConditionOperator` has the following values:
  - `Equal` (This is the default.) abs(ControlIDToEvaluate – SecondControlIDToEvaluate) = DifferenceValue
  - `NotEqual` abs(ControlIDToEvaluate – SecondControlIDToEvaluate) <> DifferenceValue
  - `GreaterThan` abs(ControlIDToEvaluate – SecondControlIDToEvaluate) > DifferenceValue
  - `GreaterThanEqual` abs(ControlIDToEvaluate – SecondControlIDToEvaluate) >= DifferenceValue
  - `LessThan` abs(ControlIDToEvaluate – SecondControlIDToEvaluate) < DifferenceValue
  - `LessThanEqual` abs(ControlIDToEvaluate – SecondControlIDToEvaluate) <= DifferenceValue

- **OneEquals** (enum PeterBlum.DES.OneEquals) – Determines how to scale the value entered into **DifferenceValue**. You identify what the value of 1 means.

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

  - `Auto` – Based on the data type's rules. Number types like Integer and Decimal use 1 = 1.00. Dates use 1 = 1 day. Times use 1 = 1 second. DataType="DateTime" uses 1 = 1 day. This is the default.
  - `Second` – Use on time and duration data types. 1 = 1 second.
  - `Minute` – Use on time and duration data types. 1 = 1 minute.
  - `Hour` – Use on time and duration data types. 1 = 1 hour.
  - `Day` – Use on date oriented data types. 1 = 1 day.
  - `Month` – Use on date oriented data types. 1 = 1 month, which is always 30 days here.
  - `Year` – Use on date oriented data types. 1 = 1 year, which is always 365 days here
  - `One` – Use on numeric data types. 1 = 1
  - `Ten` – Use on numeric data types. 1 = 10
  - `Hundred` – Use on numeric data types. 1 = 100
  - `Thousand` – Use on numeric data types. 1 = 1000
  - `TenThousand` – Use on numeric data types. 1 = 10,000
  - `HundredThousand` – Use on numeric data types. 1 = 100,000
  - `Million` – Use on numeric data types. 1 = 1,000,000
  - `Billion` – Use on numeric data types. 1 = 1,000,000,00

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

CountSelectionsValidator

Condition	CountSelectionsCondition
License Required	Peter's More Validators
Supported controls	CheckBoxList, ListBox
Can evaluate blank fields	Yes

The CountSelectionsValidator evaluates CheckBoxLists and multi-selection ListBoxes to determine how many items are selected. You can establish a minimum and maximum. Optionally you can assign it to a second CheckBoxList or multi-selection ListBox to compare to the combined total selection to the range.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for CountSelectionsValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

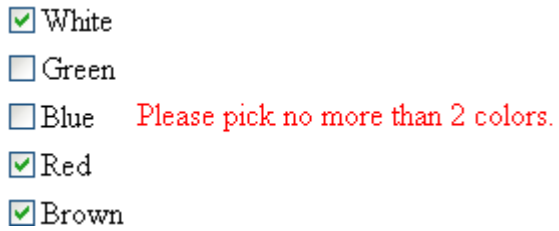
Using This Condition

Specify the control to evaluate with the **ControlIDToEvaluate** property. If you want to evaluate the total of two controls, specify the second in **SecondControlIDToEvaluate**.

The **Minimum** and **Maximum** properties determine the limits when they are not 0.

This Validator's **ErrorMessage** property supports several tokens. "{COUNT}" is the number of selected items. "{MINIMUM}" is replaced by the property **Minimum**. "{MAXIMUM}" is replaced by the property **Maximum**. See "Tokens in Error Messages" for details on tokens.

Example




```
<des:CountSelectionsValidator id="CountSelectionsValidator1" runat="server"
   ControlIDToEvaluate="CheckListBox1" Maximum="2"
   ErrorMessage="Please pick no more than {MAXIMUM} colors.">
</des:CountSelectionsValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Condition Properties for CountSelectionsValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, **ValueToEvaluate**, **ReportErrorsAfter**, and **NotCondition** – See "Properties Common to most Conditions".
- **Minimum** (Integer) – The minimum number of selected items in the CheckBoxList or multi-selection ListBox. When it is 0, there is no minimum. It defaults to 0.
- **Maximum** (Integer) – The maximum number of selected items in the CheckBoxList or multi-selection ListBox. It must be greater or equal to 1. It defaults to 99999.
- **SecondControlIDToEvaluate** (string) – Identifies a second CheckBoxList or multi-selection ListBox control. When assigned, the total selections in the two controls are added together before comparing to the minimum and maximum. Leave it blank if you only need to evaluate one control. This property takes the ID of the control.

  An exception is thrown at runtime when this is unknown, a different control class than **ControlIDToEvaluate** not in the same or ancestor naming container, is **Visible** = `false`, or a control class that is not supported.

  If you want to connect to a control that is inside of a usercontrol, see "Attaching Validators to controls inside of a User Control".
- **SecondControlToEvaluate** (System.Web.UI.Control) – An alternative to **SecondControlIDToEvaluate**. Use it when the control is not in the same or ancestor naming container. It must be assigned programmatically. For example, if you have a Validator instance in the variable "Val2" and a CheckBoxList instance in the variable "CheckBoxList2", write code like this: `Val2.SecondControlToEvaluate = CheckBoxList2`.

  When programmatically assigning properties to a Validator control or Condition class, if you have access to the control object that will be evaluated, it is better to assign it here than assign its ID to the **SecondControlIDToEvaluate** property because DES operates faster using **SecondControlToEvaluate**.
- **SecondValueToEvaluate** (System.Collections.Generics.IList<Integer>) – Allows the Validator or Condition to be used without a control attached. Instead, data comes from the value assigned to this property. The value must be a list of integers, each specifying an index into the list that is selected. When used, it replaces **SecondControlIDToEvaluate** and **SecondControlToEvaluate**.

  See "Using Validator controls to evaluate data not from Controls".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

DuplicateEntryValidator

Condition	DuplicateEntryCondition
License Required	Peter's More Validators
Supported controls	TextBox, ListBox, DropDownList
Can evaluate blank fields	Yes

The DuplicateEntryValidator evaluates three or more data entry controls to determine if there are any duplicate entries. You can compare textual values in any of the controls supported: TextBox, ListBox, and DropDownList. For ListBox and DropDownList, the textual value is the **Value** property associated with selected item. If you are only using ListBoxes or DropDownLists, you can compare the selected indexes to confirm that there are no duplicates.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for DuplicateEntryValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Using This Condition

Specify the data entry controls within the **ControlsToMatch** property. It is a collection that holds `PeterBlum.DES.Web.WebControls.LabeledControl` objects, where you identify the control and optionally its label.

Use the **MatchMode** property to specify a rule for matching: exact match, case insensitive match, or use the **SelectedIndex** property (only for ListBox and DropDropList controls).

Use the **IgnoreUnassigned** property to ignore blank controls.

The label is used in the **ErrorMessage** when you include the tokens "{LABEL1}" and "{LABEL2}". They help lead the user to a pair of fields found with duplicate values. Your **ErrorMessage** can also contain the token "{TEXTVALUE}" to show the value found to be duplicated. See "Tokens in Error Messages" for details on tokens.

Example

Field 1 tree

Field 2 stream

Field 3 tree

Field 4 sky

The value "tree" is shown in both Field 1 and Field 3. Only unique values are accepted.

```
<des:DuplicateEntryValidator id=DuplicateEntryValidator1 runat="server"
   IgnoreBlankText="True" MatchMode="CaseInsensitive"
   ErrorMessage='The value "{TEXTVALUE}" is shown in both {LABEL1} and
{LABEL2}. Only unique values are accepted.'>
<ControlsToMatch>
   <des:LabeledControl ControlID="TextBox1" Label-LabelControlID="Label1">
   </des:LabeledControl>
   <des:LabeledControl ControlID="TextBox2" Label-LabelControlID="Label2">
   </des:LabeledControl>
   <des:LabeledControl ControlID="TextBox3" Label-Text="Field 3">
   </des:LabeledControl>
   <des:LabeledControl ControlID="TextBox4" Label-Text="Field 4">
   </des:LabeledControl>
</ControlsToMatch>
</des:DuplicateEntryValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Condition Properties for DuplicateEntryValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ReportErrorsAfter**, **Trim** and **NotCondition** – See "Properties Common to most Conditions".
- **ControlsToMatch** (System.Collections.IList) – A list identifying each data entry control and optionally, its label. Add `PeterBlum.DES.Web.WebControls.LabeledControl` objects to this collection.

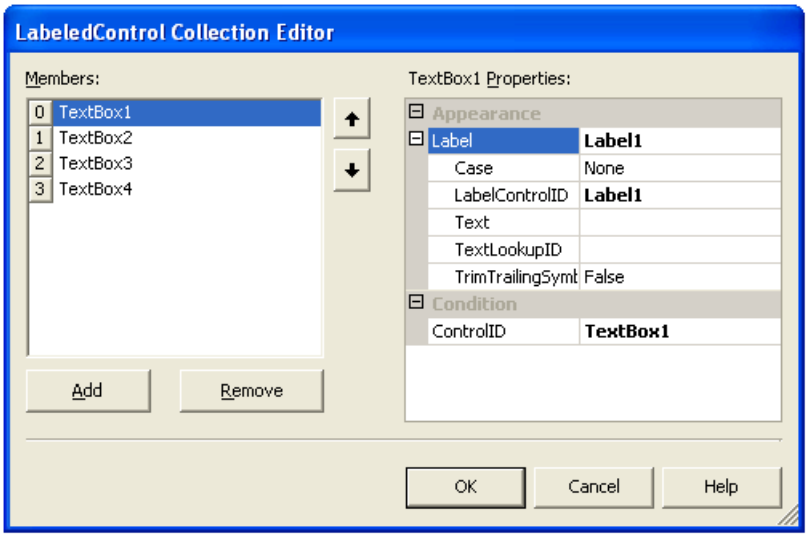
  You can only specify TextBox, ListBox, and DropDownList controls. If the **MatchMode** property is set to `SelectedIndex`, do not add TextBoxes.

  The `PeterBlum.DES.Web.WebControls.LabeledControl` object has these properties:

  - **ControlID** (string) – The ID of the data entry control. The control must be in the same or a parent naming container as the DuplicateEntryValidator. Alternatively, use the **ControlInstance** property.
  - **ControlInstance** (System.Web.UI.Control) – An object reference to the data entry control. It is an alternative to ControlID that allows a control from any naming container. If you programmatically set up LabeledControl objects, it is better to assign this property as DES processes it faster.
  - **Label** (PeterBlum.DES.IBaseLabelText) – Specifies an optional label associated with the data entry control. It is used in the ErrorMessage tokens "{LABEL1}" and "{LABEL2}". So set it when using those tokens. You can specify a Label control in its **LabelControlID** or **LabelControl** properties or enter text for the label in its **Text** property. See "Properties for Error Messages" for details on the properties of the LabelText class.

Adding to ControlsToMatch In Design Mode

The Properties Editor for the **ControlsToMatch** property provides a window where you can define LabeledControl objects.



Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Adding to ControlsToMatch with ASP.NET Markup

ControlsToMatch is a type of collection. Therefore its ASP.NET text is nested as a series of LabeledControl objects within the `<ControlsToMatch>` tag. Each LabeledControl is a tag with `<des:LabeledControl>` followed by the properties.

The following example represents the same LabeledControl objects shown in the editor window above. The first two textboxes have labels defined using Label controls. The last two do not and require the **Text** property to be used.

```
<des:DuplicateEntryValidator id="Validator1" runat="server">

<ControlsToMatch>
   <des:LabeledControl ControlID="TextBox1" Label-LabelControlID="Label1" />
   <des:LabeledControl ControlID="TextBox2" Label-LabelControlID="Label2" />
   <des:LabeledControl ControlID="TextBox3" Label-Text="Field 3" />
   <des:LabeledControl ControlID="TextBox4" Label-Text="Field 4" />
</ControlsToMatch>

</des:DuplicateEntryValidator>
```

Adding To ControlsToMatch Programmatically

Use the `Add()` method on **ControlsToMatch** to supply the ID or reference of the control to evaluate. Optionally specify its Label. The `Add()` method is overloaded with these choices:

[C#]

```
int Add(string pDataControlID)

int Add(string pDataControlID, string pLabelText)

int Add(string pDataControlID, string pLabelControlID,
   PeterBlum.DES.CaseConversion pCase)

int Add(Control pDataControl)

int Add(Control pDataControl, string pLabelText)

int Add(Control pDataControl, Control pLabelControl,
   PeterBlum.DES.CaseConversion pCase)
```

[VB]

```
Function Add(ByVal pDataControlID As String) As Integer

Function Add (ByVal pDataControlID As String, _
   ByVal pLabelText As String) As Integer

Function Add (ByVal pDataControlID As String, _
   ByVal pLabelControlID As String, _
   ByVal pCase As PeterBlum.DES.CaseConversion) As Integer

Function Add (ByVal pDataControl As Control) As Integer

Function Add (ByVal pDataControl As Control, _
   ByVal pLabelText As String) As Integer

Function Add (ByVal pDataControl As Control, _
   ByVal pLabelControl As Control, _
   ByVal pCase As PeterBlum.DES.CaseConversion) As Integer
```

pDataControlID maps to the **ControlID** property. *pDataControl* maps to the **ControlInstance** property. *pLabelText* maps to the **Label.Text** property. *pLabelControl* maps to the **Label.LabelControl** property. *pCase* maps to the **Label.Case** property.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Example

Add items to ControlsToMatch, for TextBox1 and TextBox3. TextBox1 uses a label in a Label1. TextBox3 uses text for its label.

[C#]

```
Validator1.ControlsToMatch.Add(TextBox1, Label1);
Validator1.ControlsToMatch.Add(TextBox3, "Field 3");
```

[VB]

```
Validator1.ControlsToMatch.Add(TextBox1, Label1)
Validator1.ControlsToMatch.Add(TextBox3, "Field 3")
```

- **MatchMode** (enum PeterBlum.DES.DuplicateMatchMode) – Choose the rule to interpret the value of each data entry control. The enumerated type `PeterBlum.DES.DuplicateMatchMode` has these values:
  - `Exact` - Use a case sensitive textual comparison.
  - `CaseInsensitive` - Use a case insensitive textual comparison.
  - `SelectedIndex` – Compare the **SelectedIndex** properties on ListBoxes and DropDownLists. (ListBoxes and DropDownLists also support `Exact` and `CaseInsensitive` modes.)

  It defaults to `DuplicateMatchMode.Exact`.

- **IgnoreUnassigned** (Boolean) – Determines how to handle controls that are unassigned (as indicated by the **UnassignedText** or **UnassignedIndex** properties.)

  When `true`, if the data entry control's value matches **UnassignedText** or **UnassignedIndex**, it is never matched. When `false`, the data entry control's value is always used, even when blank. It defaults to `true`.

- **UnassignedText** (string) – Used when **IgnoreUnassigned** is `true`. When this value matches the textual value of the data entry control, that value is never matched to the others. It is considered "blank" and should be ignored. It defaults to "".

- **UnassignedIndex** (integer) – Used when **IgnoreUnassigned** is `true`. When this **SelectedIndex** value matches the **SelectedIndex** of the list-type control, that value is never matched to the others. It is considered "no selection" and should be ignored. The value -1 indicates the list has no selection. 0 is the first item in the list. It defaults to `-1`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

UnwantedWordsValidator

Condition	UnwantedWordsCondition
License Required	Peter's More Validators
Supported controls	All that declare the ValidationPropertyAttribute including TextBox, ListBox, DropDownList
Can evaluate blank fields	Yes when the IgnoreBlankText property is false

The UnwantedWordsValidator compares the contents of a textbox to a list of strings. If it matches one of the strings, the condition fails.

This control can retrieve a list of strings from a data source such as a DataSet, DataTable or a `System.Collections.IEnumerable` collection, in the same way you populate a ListBox or DropList.

It has rules to match "whole words", case insensitive, and "broken words". A broken word is a word that is separated by non-letter characters in an attempt to avoid detection. For example, if the word "flower" is unwanted, an error will be reported on "f*l*o*w*e*r".

Note: This Validator uses a Regular Expression to evaluate the list of strings. In fact, it is subclassed from RegexValidator and RegexCondition.

Note: This Validator is not meant for long lists of strings, especially when you validate on the client side. It has to copy all of the strings to the client-side, which slows transmission times. There are likely limits to the size of a regular expression pattern. Test your page against the data to be sure this Validator will work for you.

Note: This Validator should not be used for detecting SQL Injection or Script Injection attacks. Hackers have so many textual patterns that they can use, they usually will get around your efforts to block them here. Instead, use the Validator controls from the Peter's Input Security module.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for UnwantedWordsValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Using This Condition

Specify the control to evaluate with the **ControlIDToEvaluate** property.

The **Items** property holds each unwanted word in a `PeterBlum.DES.Web.WebControls.UnwantedWordsItem` object. Assign its **Value** property to the unwanted word.

The **Items** property can also be populated from a DataSource. The data comes from three properties that are familiar if you have used databinding on ListBoxes or DataGrids: **DataSource**, **DataMember**, and **DataTextField**. However, you don't need to call the `DataBind()` method unless you use databinding notation in the ASP.NET text (as shown below.)

The evaluation rules are in the properties **WholeWord**, **CaseInsensitive**, and **BrokenWords**.

This Validator's **ErrorMessage** supports the token "{UNWANTED}" to state the offending word. See "Tokens in Error Messages" for details on tokens.

Examples

damnit! You cannot be serious!

Please do not enter damnit.

This example uses the **DataSource** property.

```
<des:UnwantedWordsValidator id="UWWV1" runat="server"
   ControlIDToEvaluate="TextBox1"
   ErrorMessage="Please do not enter {UNWANTED}."
   DataSource="<%# dataSet1 %>" DataMember="SwearWords"
   DataTextField="Word"
   BrokenWords="True" CaseInsensitive="True" >
</des:UnwantedWordsValidator>
```

This example uses the **Items** property.

```
<des:UnwantedWordsValidator id="UWWV2" runat="server"
   ControlIDToEvaluate="TextBox1"
   ErrorMessage="Please do not enter {UNWANTED}."
   BrokenWords="True" CaseInsensitive="True" >
   <Items>
      <des:UnwantedWordsItem Value="crap"></des:UnwantedWordsItem>
      <des:UnwantedWordsItem Value="damnit"></des:UnwantedWordsItem>
   </Items>
</des:UnwantedWordsValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Condition Properties for UnwantedWordsValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, **ValueToEvaluate**, **Trim**, and **NotCondition** – See "Properties Common to most Conditions".
- **Items** (PeterBlum.DES.UnwantedWordsItems) – This collection contains the strings which are unwanted. It contains `PeterBlum.DES.UnwantedWordsItem` objects. These objects have one property, **Value**, where you assign an unwanted word.

  #### Entering Items in ASP.NET Markup

  Strings are defined in `PeterBlum.DES.UnwantedWordsItem` objects.

  Define strings like this:

  ```
  <Items>
     <des:UnwantedWordsItem Value="unwanted1" />
     <des:UnwantedWordsItem Value="unwanted2" />
  </Items>
  ```

  #### Entering Items Programmatically

  Use the `Add()` method on the **Items** property. Pass a string or `PeterBlum.DES.UnwantedWordsItem` object containing the unwanted word.

  [C#]

  ```
  UnwantedWordsValidator1.Items.Add("unwanted");
  ```

  [VB]

  ```
  UnwantedWordsValidator1.Items.Add("unwanted")
  ```

- **DataSource** (object) – You can populate the **Items** property from a data source using this property. Your data source can be a DataSet, DataTable, or a collection that implements `System.Collections.IEnumerable` such as an ArrayList. If you use the design time features of the Properties Editor, you can select a DataSet object already on your design mode surface with the "(DataBindings)" property.
  - When using a DataSet, also set the **DataMember** property to the name of the table that contains the column you are using. You can leave DataMember blank only if DataSet only has one table. Always set **DataTextField** to the name of the column that you are using.
  - When using a DataTable, set the **DataTextField** property to the name of the column that you are using.
  - When using a collection that implements IEnumerable, you can populate it with strings or any object class that has a string property with the desired data. If you use an object that has a string property, give the name of that property in the **DataTextField** property.

  Note: If you set up the DataSource property in the Properties Editor, you must call the DataBind() method on the Validator in your Page_Load() method. You do not need to call DataBind() when you assign this property programmatically.

- **DataMember** (string) – When using the **DataSource** property with a DataSet, assign the name of the table within that DataSet that contains the column you are using. It defaults to "".
- **DataTextField** (string) – When using the DataSource property, assign this to the name of the column or property that contains the data. When it is a DataSet or DataTable, this should be the name of the column within a table. When it is an IEnumerable collection, it should be the name of a property on an object. When it is a collection that only contains strings, leave this blank. It defaults to "".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

- **WholeWord** (Boolean) – When `true`, all unwanted words must match to whole word. That means their first letter is not preceded by another letter and their last letter is not followed by another letter. When `false`, the unwanted work can be found inside another word. It defaults to `false`.

- **BrokenWords** (Boolean) – Look for a word whose characters are broken up by non-letter characters. For example, the unwanted word "darn" can be written "d&a!r-n" and it will be detected. Spaces are valid separators. So the word "darn" will be found in "radar noise" unless you set **WholeWord** to `true`. As a result, **WholeWord** mode is strongly recommended.

  When `true`, the feature is enabled. When `false`, it is disabled. It defaults to `false`.

- **CaseInsensitive** (Boolean) – When `true`, the regular expression parser will perform a case insensitive match. When `false`, it will perform a case sensitive match. It defaults to `true`.

- **IgnoreBlankText** (Boolean) – Determines how blank text is evaluated. When `true`, the condition cannot be evaluated. When `false`, the condition evaluates as failed (reports an error). It defaults to `true`.

- **ItemsByDelimitedString** (string) – *Only available on the Condition, not on the validator.* An alternative to defining the **Items** property through child UnwantedWordstems or using a Data Source, use this to supply a pipe delimited list of strings to compare. For example, "lol|lmao|rofl".

  If used, the **Items** collection is not used, nor are the **DataSource** and related properties.

  This property was created as a way to use the UnwantedWordsCondition inside of a `PeterBlum.DES.DataAnnotations.ConditionDependencyAttribute` because Attributes located in metadata cannot have child elements.

  It is unassigned by default.

- **GlobalStorageName** (string) – Use this property to improve speed. This Condition uses a .net Regex object to evaluate text. It takes time for a .net Regex object to convert a regular expression into its internal format. With this setting that will only happen once per application start, and it is further optimized by using the "Compiled" option on the .net Regex object.

  Do not use this when the list of strings will change. It should only be used when the list of strings remains identical while the application runs.

  You define a unique name for the list in **Items** and assign it to **GlobalStorageName**. For example, "CussWords" or "123abc".

  If you have several UnwantedWordsValidators that use identical rules, use the same value of **GlobalStorageName** so that they all share the same Regex object. This further improves your application's speed.

  When used, the .net Regex object is stored in the Application collection. `Application[GlobalStorageName]` is assigned the string containing the regular expression. `Application[GlobalStorageName + "_RE"]` is assigned the Regex object. So give this a name that will not conflict with anything else that you place into the Application collection.

  > *WARNING: Several other Validators offer the GlobalStorageName property. Never use the same value amongst different types of Validators.*

- **PrepareProperties** (delegate PeterBlum.DES.RegexPrepareProperties) – When using the **GlobalStorageName** and you run time consuming code to create the list of strings, put your time consuming code in a method assigned to **PrepareProperties**. Your code will only be called once and its results will be cached. This increases the application speed by avoiding that time consuming code when it's not needed.

  This is recommended when you use the **DataSource** property as database queries are time consuming.

  Your method will update the properties of the UnwantedWordsCondition, the Condition object of the UnwantedWordsValidator. The UnwantedWordsCondition and UnwantedWordsValidator share the same properties described throughout this section. Normally you assign the **Items** or **DataSource** property. However, any of the properties can be changed.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Here is the format of a method assigned to **PrepareProperties**.

[C#]

```
public void MethodName(PeterBlum.DES.IRegexCondition pCondition)
```

[VB]

```
Public Sub MethodName(ByVal pCondition As PeterBlum.DES.IRegexCondition)
```

Parameters

pCondition

The Condition class for the UnwantedWordsValidator. It is declared as RegexCondition, which is a base class to the UnwantedWordsCondition. So typecast *pCondition* to `PeterBlum.DES.Web.WebControls.UnwantedWordsCondition` to access its properties. UnwantedWordsCondition has the same properties as shown in this section.

[C#]

```
MyValidator.PrepareProperties =
   new PeterBlum.DES.RegexPrepareProperties(methodname);
```

[VB]

```
MyValidator.PrepareProperties = _
   New PeterBlum.DES.RegexPrepareProperties(AddressOf methodname)
```

Example

Assigns a `DataSet` to the **DataSource** property after populating it. The sqlDataAdapter1 is a `SQLDataAdapter` already on the page and fully attached to a `SqlConnection` and `SqlCommand` object. dataSet1 is a `DataSet` already on the page. Assume that they are all objects on this page generated by Visual Studio.net. The **DataMember** and **DataTextName** properties retain their previously assigned values in this example.

[C#]

```
public void ApplyExpression(PeterBlum.DES.IRegexCondition pRegexCondition)
{
   sqlDataAdapter1.Fill(dataSet1);
   ((PeterBlum.DES.Web.WebControls.UnwantedWordsCondition)pCondition).
         DataSource = dataSet1;
}
```

[VB]

```
Public Sub ApplyExpression(ByVal pRegexCondition As _
      PeterBlum.DES.IRegexCondition)
   sqlDataAdapter1.Fill(dataSet1)
   CType(pCondition, _
      PeterBlum.DES.Web.WebControls.UnwantedWordsCondition).DataSource = _
      dataSet1
End Sub
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

SelectedIndexRangesValidator

Condition	SelectedIndexRangesCondition
License Required	Peter's More Validators
Supported controls	ListBox, DropDownList, RadioButtonList
Can evaluate blank fields	Yes

The SelectedIndexRangesValidator evaluates the **SelectedIndex** property on ListBoxes, DropDownLists, and RadioButtonLists. It is an extension of the idea behind the SelectedIndexValidator where you indicate a **SelectedIndex** that's valid. In this Validator, you provide one or more ranges that the **SelectedIndex** can match to. For example, if you have a dropdownlist of 20 items and you want items 1, 10-12, and 15-20 to be valid, this is the right Validator.

*ALERT: Do not use this with the ASP.NET AJAX Control Toolkit's CascadingDropDown control. The CascadingDropDown does not support the **SelectedIndex** property on the server side. It only supports the textual value. So use a validator that evaluates the text value, such as CompareToStringsValidator.*

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for SelectedIndexRangesValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

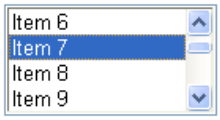
Using This Condition

Specify the control to evaluate with the **ControlIDToEvaluate** property.

Establish the indexes that the user can select in the **Ranges** property, a collection where you define ranges in `PeterBlum.DES.SelectedIndexRange` objects. Use the **StartIndex** and **EndIndex** properties of `PeterBlum.DES.SelectedIndexRange` to establish a range. When **StartIndex** = **EndIndex**, a single index is defined.

This Validator's **ErrorMessage** property supports the token "{TEXTVALUE}" to state the value that is out of range.

Example



Do not select "Item 7"

```
<des:SelectedIndexRangesValidator id="SIRValidator1" runat="server"
   ControlIDToEvaluate="ListBox1" ErrorMessage='Do not select "{TEXTVALUE}"'>
   <Ranges>
      <des:SelectedIndexRange StartIndex="10" EndIndex="13" />
      <des:SelectedIndexRange StartIndex="25" EndIndex="9999" />
      <des:SelectedIndexRange EndIndex="0" />
      <des:SelectedIndexRange StartIndex="4" EndIndex="4" />
   </Ranges>
</des:SelectedIndexRangesValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Condition Properties for SelectedIndexRangesValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, and **NotCondition** – See "Properties Common to most Conditions".
- **Ranges** (PeterBlum.DES.SelectedIndexesRanges) – This is a collection where you define index ranges and single indexes that are selectable. Any index not included in this list will evaluate as "failed". (You can use the **NotCondition** property to reverse this logic.)

  You add `PeterBlum.DES.SelectedIndexesRange` objects to this collection. Each object has two properties, **StartIndex** and **EndIndex**. Use them to specify a range of indexes. **StartIndex** must always be less than or equal to **EndIndex**. If they have the same value, the range is just one index.

  StartIndex and **EndIndex** use a value of -1 to indicate that a list with no selection is valid. **StartIndex** defaults to -1. **EndIndex** defaults to 9999.

Entering Items in ASP.NET Markup

Ranges is a type of collection. Therefore its ASP.NET text is nested as a series of `PeterBlum.DES.SelectedIndexesRange` objects within the `<Ranges>` tag. Each SelectedIndexRange object is a tag with `<des:SelectedIndexRange>` followed by the properties.

For example:

```
<Ranges>
   <des:SelectedIndexRange StartIndex="10" EndIndex="13">
   </des:SelectedIndexRange>
   <des:SelectedIndexRange StartIndex="25" EndIndex="9999">
   </des:SelectedIndexRange>
   <des:SelectedIndexRange StartIndex="4" EndIndex="4">
   </des:SelectedIndexRange>
</Ranges>
```

Entering Items Programmatically

Use the `Add()` method on the **Ranges** property. Pass an index or a range of indices. Here is the overloaded method definition.

`PeterBlum.DES.SelectedIndexesRange` has these constructors:

[C#]

```
int Add(int pIndex)     // sets StartIndex=EndIndex

int Add(int pStartIndex, int pEndIndex)
```

[VB]

```
Function Add(ByVal pIndex As Integer) As Integer ' StartIndex=EndIndex

Function Add(ByVal pStartIndex As Integer, _
   ByVal pEndIndex As Integer) As Integer
```

This example builds the same ranges as shown above.

[C#]

```
SelectedIndexRangesValidator1.Items.Add(10, 13);
SelectedIndexRangesValidator1.Items.Add(25, 9999);
SelectedIndexRangesValidator1.Items.Add(4);
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

[VB]

```
SelectedIndexRangesValidator1.Items.Add(10, 13)
SelectedIndexRangesValidator1.Items.Add(25, 9999)
SelectedIndexRangesValidator1.Items.Add(4)
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

ListSizeValidator

Condition	ListSizeCondition
License Required	Peter's More Validators
Supported controls	ListBox, DropDownList
Can evaluate blank fields	Yes

The ListSizeValidator evaluates a listbox or dropdownlist to determine if has enough elements listed. It is used when the user interface allows adding and removing elements to a list. Typically users want to report an error when the list is empty, however, this allows a range (Minimum/Maximum) to provide more flexibility.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for ListSizeValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Using This Condition

Specify the ListBox to evaluate with the **ControlIDToEvaluate** property. By default, it requires at least one in the list. Set a different minimum with the **Minimum** property and introduce a maximum with the **Maximum** property.

The **ErrorMessage** and **SummaryErrorMessage** properties support several tokens:

- {MINIMUM} – The value of the Minimum property.
- {MAXIMUM} – The value of the Maximum property.
- {COUNT} – The number of items in the list.
- {COUNT:singular:plural} – Shows either the word in the "singular" position or "plural" position based on the count. Singular is shown if the count is 1.

EXAMPLE SHOWN ON THE NEXT PAGE

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Example

Require between 0 and 4 items in ListBoxA. Require between 2 and 3 items in ListBoxB.

```
<asp:ListBox ID="ListBoxA" runat="server" >
   <asp:ListItem>1</asp:ListItem>
   <asp:ListItem>2</asp:ListItem>
   <asp:ListItem>3</asp:ListItem>
</asp:ListBox>
<des:Button ID="MoveAtoB" runat="server" CausesValidation="False"
   OnClick="MoveAtoB_Click" Text="Move >>" />
<des:Button ID="MoveBtoA" runat="server" CausesValidation="False"
   OnClick="MoveBtoA_Click" Text="<< Move" />
<asp:ListBox ID="ListBoxB" runat="server">
   <asp:ListItem>4</asp:ListItem>
   <asp:ListItem>5</asp:ListItem>
   <asp:ListItem>6</asp:ListItem>
</asp:ListBox>
<des:ListSizeValidator id="ListSizeValidator1" runat="server"
   ControlIDToEvaluate="ListBoxA" Maximum="4" Minimum="0"
   ErrorMessage="Max {MAXIMUM}. You have {COUNT} {COUNT:item:items}." >
</des:ListSizeValidator>
<des:ListSizeValidator id="ListSizeValidator2" runat="server"
   ControlIDToEvaluate="ListBoxB" Maximum="3" Minimum="2"
   ErrorMessage="Between {MINIMUM} and {MAXIMUM}.
      You have {COUNT} {COUNT:item:items}." >
</des:ListSizeValidator>
```

[C#]

```
protected void MoveAtoB_Click(object sender, EventArgs e)
{
   Transfer(ListBoxA, ListBoxB);
}
protected void MoveBtoA_Click(object sender, EventArgs e)
{
   Transfer(ListBoxB, ListBoxA);
}
protected void Transfer(ListBox pSourceList, ListBox pDestList)
{
   if (pSourceList.SelectedItem != null)
   {
      ListItem vItem = pSourceList.SelectedItem;
      vItem.Selected = false;
      pDestList.Items.Add(vItem);
      pSourceList.Items.Remove(vItem);
   }
}
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Condition Properties for ListSizeValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, and **NotCondition** – See "Properties Common to most Conditions".
- **UnassignedIndex** (Integer) – The item number of the item that is considered unselected. When -1, there are no selected items. 0 is the first item in the list. It defaults to -1.

  Change it when an item, such as the first element of a RadioButtonList, ListBox, or DropDownList indicates an unassigned state. The CheckBoxList control does not use this property because a CheckBoxList is unassigned when all checkboxes are unmarked.
- **Minimum** (Integer) – The minimum value of the range. When 0, it allows an empty list.

  It defaults to 1.
- **Maximum** (Integer) – The maximum value of the range. When 0, there is no maximum.

  It defaults to 0.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

CreditCardNumberValidator

Condition	CreditCardNumberCondition
License Required	Peter's More Validators
Supported controls	TextBox, HtmlInputText
Can evaluate blank fields	Yes when the IgnoreBlankText property is false

The CreditCardNumberValidator reviews the text pattern to confirm it matches the rules for credit card numbers. Four rules are applied:

- The text is all digits. *You have the option of allowing the user to type spaces with the AllowSpaces property. They are ignored by the validator.*
- Luhn's formula. This is an algorithm based on the values of digits across the number. All credit card numbers are designed to match this formula.
- Match to a particular brand by lead digits. For example, all Visa cards start with '4' while MasterCard starts with '51', '52', '53', '54', or '55'.
- Match to a particular brand by length of the text. For example, Visa is either 13 or 16 characters while MasterCard is only 16 characters.

This validator does not submit the credit card number against an online service. You must get other software for that and can use the IgnoreConditionValidator or CustomValidator to report errors from that service.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for CreditCardNumberValidator
- Editing Credit Card Brands
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Using This Condition

Specify the control to evaluate with the **ControlIDToEvaluate** property.

The first two rules are always applied. The last two depend on these factors:

- The **CheckByBrand** property is `true`. (It defaults to true.)
- The credit card brands that you support are defined in the **des.config** and **custom.des.config** files. Open each of these files to the `<CreditCards>` section. **des.config** should not be edited. It is the factory list of supported cards. Use the **custom.des.config** file to remove factory defaults that you do not want and add new rules. See "Editing Credit Card Brands". Alternatively, you can programmatically assign credit card brand information to the CreditCardNumberValidator or CreditCardNumberCondition using the `AddBrand()` method.

Example

4102551272103 Invalid credit card number

```
<des:CreditCardNumberValidator id="CreditCardNumberValidator1" runat="server"
   ControlIDToEvaluate="TextBox1"
   ErrorMessage="Invalid credit card number">
</des:CreditCardNumberValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Condition Properties for CreditCardNumberValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, **Trim**, **ValueToEvaluate**, and **NotCondition** – See "Properties Common to most Conditions".
- **CheckByBrand** (Boolean) – Determines if the credit card brands are checked. When `true`, they are.

  Credit card brands are defined in the **des.config** and **custom.des.config** files, in the `<CreditCards>` section. You can modify the **custom.des.config** file to add new cards or remove those defined in the **des.config** file (the factory defaults.) See "Editing Credit Card Brands".

  It defaults to `true`.
- **IgnoreBlankText** (Boolean) – Determines how blank text is evaluated. When `true`, the Condition cannot be evaluated. When `false`, the Condition evaluates as failed (reports an error). It defaults to `true`.
- **AllowSpaces** (Boolean) – You may prefer to let users enter spaces in their credit card number. Your server side code will strip out those spaces for its own use. If this is the case, set **AllowSpaces** to `true`. The validator does not care where the spaces are located or if there are several in a row.

  It defaults to `false`.

  The **CleanedUpNumber** property gives you the number entered without the spaces.
- **CleanedUpNumber** (Boolean) – Read Only. Gets the text that is evaluated by this validator without the spaces that were permitted by the **AllowSpaces** property.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Editing Credit Card Brands

The CreditCardNumberValidator can restrict credit card numbers to particular brands when you set the **CheckByBrand** property to `true`. It uses a list of brands defined in both the **des.config** and **custom.des.config** files. You can override the values from the config files using the `AddBrand()` method. The **des.config** file provides the factory default list. You should not edit it because each release of DES replaces this file. Instead, you edit the **custom.des.config** file, adding new credit cards, changing the prefix property on existing ones, and identifying those defined in **des.config** that you want to remove.

Use your favorite text editor to open both **des.config** and **custom.des.config** files within your web application's **\DES** folder. Locate the `<CreditCards>` section in both files.

Reminder: XML is case sensitive. Type the tag and attribute names in exactly as shown.

- To add a new brand, add a new row to the `<CreditCards>` section of **custom.des.config** in this format:

  ```
  <CreditCard name="[brand name]" length="[length]" prefixes="[prefixes]" />
  ```

  - name – The credit card brand name. It must be unique against all `<CreditCard>` entries in both files.
  - length – The number of digits defined for this brand. If the credit card has multiple lengths, create separate `<CreditCard>` rows, each with a different length and a different name like "Card-14chars" and "Card-15chars".
  - prefixes – Each brand defines one or more prefixes to the number. For example, Visa uses "4" and MasterCard uses "51"-"55". When defining multiple prefixes, specify each separated by the pipe character (`|`).

    ```
    <CreditCard name="ABCCard" length="14" prefixes="591|595|596" />
    ```

- To edit the prefixes of a card defined in **des.config**, simply add it again in **custom.des.config** with the complete list of prefixes for that card. It will override the entry in **des.config**.
- To remove a brand already listed in **des.config**, add a new row to the `<CreditCards>` section of **custom.des.config** in this format:

  ```
  <CreditCard name="[brand name]" enabled="false" />
  ```

  - name – The credit card brand name. It must be identical to the name in **des.config**.

    ```
    <CreditCard name="MasterCard" enabled="false" />
    ```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

AddBrand Method

You can override the global credit card brands defined in **des.config** and **custom.des.config** on each CreditCardNumberValidator and CreditCardNumberCondition using the `AddBrand()` method.

[C#]

```
void AddBrand(string pPrefixes, int pLength)
```

[VB]

```
Sub AddBrand(ByVal pPrefixes As String, ByVal pLength As Integer)
```

Parameters

pPrefixes

One or more prefixes that identify the brand of credit card. Each separate prefix must be separated by a pipe character. For example, Master Card uses '51'-'55'. Format this like '51|52|53|54|55'.

pLength

The number of digits required by this brand.

The first call to `AddBrand()` will replace all global credit card brands with the one you specify.

Generally you will call this method in the `Page_Load()` method. Here is an example where MasterCard and Visa are added to CreditCardNumberValidator1.

[C#]

```
CreditCardNumberValidator1.AddBrand("51|52|53|54|55", 16);
CreditCardNumberValidator1.AddBrand("4", 13);
CreditCardNumberValidator1.AddBrand("4", 16);
```

[VB]

```
CreditCardNumberValidator1.AddBrand("51|52|53|54|55", 16)
CreditCardNumberValidator1.AddBrand("4", 13)
CreditCardNumberValidator1.AddBrand("4", 16)
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

ABARoutingNumberValidator

Condition	ABARoutingNumberCondition
License Required	Peter's More Validators
Supported controls	TextBox, HtmlInputText
Can evaluate blank fields	Yes when the IgnoreBlankText property is false

The ABARoutingNumberValidator reviews the text pattern to confirm it matches the rules for check routing numbers. Three rules are applied:

- The text is all digits.
- The text must be exactly 9 digits.
- The pattern much match a checksum validation rule specified by the American Bankers Association.

The ABARoutingNumberValidator does not validate the number against actual institution routing numbers. You must supply another piece of software to do that. Many sites prefer to separate the data entry form from the validation phase of a transaction through a confirmation page. Thus, they will want validation against actual institutions to be separated. If you want to keep them on the same page, use the IgnoreConditionValidator or CustomValidator to call functions on your ABA routing number software package.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for ABARoutingNumberValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Using This Condition

Specify the control to evaluate with the **ControlIDToEvaluate** property.

Example

123456789 Invalid routing number

```
<des:ABARoutingNumberValidator id="ABARoutingNumberValidator1" runat="server"
   ControlIDToEvaluate="TextBox1"
   ErrorMessage="Invalid routing number">
</des:ABARoutingNumberValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Condition Properties for ABARoutingNumberValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, **ValueToEvaluate**, **Trim**, and **NotCondition** – See "Properties Common to most Conditions".
- **IgnoreBlankText** (Boolean) – Determines how blank text is evaluated. When `true`, the Condition cannot be evaluated. When `false`, the Condition evaluates as failed (reports an error). It defaults to `true`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

CountTrueConditionsValidator

Condition	CountTrueConditions
License Required	Peter's More Validators
Supported controls	Uses the fields of other Conditions that you select
Can evaluate blank fields	Yes

The CountTrueConditionsValidator lets you define a list of conditions for a number of fields. Evaluates how many of those Conditions are true (evaluate as "success") and compares them to a minimum and maximum. The condition indicates success when the count of true conditions is greater or equal to the minimum and less than or equal to the maximum.

Here are some common uses for this Validator:

- Limit the number of marks amongst a list of checkboxes (not using a CheckBoxList). Set the maximum to the limit. Use one CheckStateCondition for each checkbox.
- Require that at least one checkbox is marked in a column of checkboxes in a DataGrid. Simply set the minimum to 1 and add CheckStateCondition objects programmatically in the DataGrid's ItemCreated event handler.
- Require that only one of several textboxes have text. Set the minimum and maximum to 1. Use one RequiredTextCondition for each textbox.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for CountTrueConditionsValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Using This Condition

Specify a list of conditions in the **Conditions** property. You can add any Condition class available within DES. See "Evaluation rules for Validators: Condition objects".

The **Minimum** and **Maximum** properties determine the limits when they are not 0.

You can use MultiCondition classes to build Boolean expressions that represent a single Condition in the collection. When the Boolean expression is evaluated as `true`, the count is incremented. The child condition objects of the expression do not contribute to the count.

If your Conditions are associated with controls inside of a ListView, GridView, DataList or Repeater, this Condition can self populate its Conditions property. Just specify the ID of the list or grid in the **ListControlID** property and add a single Condition that would be used in each row to the **Conditions** property.

This Validator's **ErrorMessage** property supports several tokens. "{COUNT}" is the number of selected items. "{MINIMUM}" is replaced by the property **Minimum**. "{MAXIMUM}" is replaced by the property **Maximum**. See "Tokens in Error Messages" for details on tokens.

Example 1

```
<des:CountTrueConditionsValidator id="CountTrueCondValidator1" runat="server"
      Maximum="2"
      ErrorMessage="Please pick no more than {MAXIMUM} colors.">
   <Conditions>
      <des:CheckStateCondition ControlIDToEvaluate="Checkbox1" />
      <des:CheckStateCondition ControlIDToEvaluate="Checkbox2" />
      <des:CheckStateCondition ControlIDToEvaluate="Checkbox3" />
      <des:CheckStateCondition ControlIDToEvaluate="Checkbox4" />
      <des:CheckStateCondition ControlIDToEvaluate="Checkbox5" />
   </Conditions>
</des:CountTrueConditionsValidator>
```

Example 2: Using ListControlID

Goes through ListView1 and creates CheckStateConditions for each row.

```
<asp:ListView id="ListView1" runat="server">
  <ItemTemplate>
    <asp:CheckBox id="Checkbox1" runat="server" />
  </ItemTemplate>
</asp:ListView>
<des:CountTrueConditionsValidator id="CountTrueCondValidator1" runat="server"
      ListControlID="ListView1" Maximum="2"
      ErrorMessage="Please pick no more than {MAXIMUM} colors.">
   <Conditions>
      <des:CheckStateCondition ControlIDToEvaluate="Checkbox1" />
   </Conditions>
</des:CountTrueConditionsValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Condition Properties for CountTrueConditionsValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

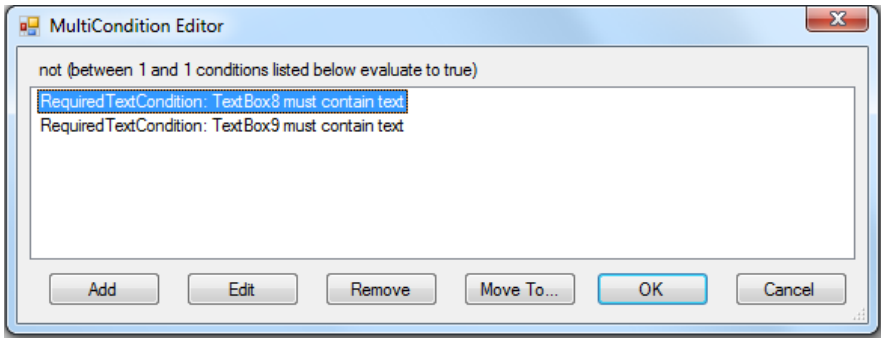
- **NotCondition** – See "Properties Common to most Conditions".
- **Conditions** (PeterBlum.DES.ConditionsCollection) – A collection of Condition classes. Each Condition object will be evaluated. If its condition indicates success, the count is incremented. If the condition indicates failure or it cannot evaluate, the count remains unchanged. The total count is compared to the range established with the **Minimum** and **Maximum** properties where the count must be greater or equal to **Minimum** and less than or equal to **Maximum**.

  If using the **ListControlID** or **ListInstance** properties, only specify one Condition object.

  You can add any Condition class available within DES. See "Evaluation rules for Validators: Condition objects" for a list.

Adding Conditions In Design Mode

The Properties Editor for the **Conditions** property provides a window where you can define conditions.



- Click the **Add** button to add a new Condition. A list of Conditions appears. Select one and click **OK**. An editor window for the condition appears. Fill in its property grid and click **OK**.
- Click the **Edit** button to open a Condition and edit its property grid.
- Click the **Remove** button to remove a Condition.

Adding Conditions with ASP.NET Markup

Conditions is a type of collection. Therefore its ASP.NET text is nested as a series of child controls within the `<Conditions>` tag. Each Condition is a tag with `<des:conditionclass>` followed by the properties.

The following example represents the same conditions shown in the editor window above. They are CheckStateConditions.

```
<des:CountTrueConditionsValidator id="Validator1" runat="server">
   <Conditions>
      <des:CheckStateCondition ControlIDToEvaluate="CheckBox2" />
      <des:CheckStateCondition ControlIDToEvaluate="CheckBox3" />
      <des:CheckStateCondition ControlIDToEvaluate="CheckBox4" />
      <des:CheckStateCondition ControlIDToEvaluate="CheckBox5" />
      <des:CheckStateCondition ControlIDToEvaluate="CheckBox6" />
   </Conditions>
</des:CountTrueConditionsValidator>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Adding Conditions Programmatically

1. Create an instance of the desired Condition. 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 CountTrueConditionsValidator object.

2. Assign property values.

3. Use the `Add()` method on the **Conditions** property to add the Condition object.

In this example, add two CheckStateConditions for CheckBox2 and CheckBox3.

[C#]

```
PeterBlum.DES.Web.WebControls.CheckStateCondition vCond1 =
   new PeterBlum.DES.Web.WebControls.CheckStateCondition();
vCond1.ControlToEvaluate = CheckBox2;     // instance example
Validator1.Conditions.Add(vCond1);
PeterBlum.DES.Web.WebControls.CheckStateCondition vCond2 =
   new PeterBlum.DES.Web.WebControls.CheckStateCondition();
vCond2.ControlIDToEvaluate = "CheckBox3"; // ID example
Validator1.Conditions.Add(vCond2);
```

[VB]

```
Dim vCond1 As PeterBlum.DES.Web.WebControls.CheckStateCondition =_
   New PeterBlum.DES.Web.WebControls.CheckStateCondition()
vCond1.ControlToEvaluate = CheckBox2      ' instance example ()
Validator1.Conditions.Add(vCond1)
Dim vCond2 As PeterBlum.DES.Web.WebControls.CheckStateCondition =_
   New PeterBlum.DES.Web.WebControls.CheckStateCondition()
vCond2.ControlIDToEvaluate = "CheckBox3"  ' ID example
Validator1.Conditions.Add(vCond2)
```

- **Minimum** (Integer) – The minimum number of Conditions that evaluate successfully in the **Conditions** property. It only counts the results of the immediate children in the **Conditions** property. When it is 0, there is no minimum. It defaults to 0.

- **Maximum** (Integer) – The maximum number of Conditions that evaluate successfully in the **Conditions** property. It only counts the results of the immediate children in the **Conditions** property. It must be greater or equal to 1. It defaults to 99999.

- **ListControlID** (string) – The ID of a list or grid control. When specified, the **Conditions** property will be populated by creating a Condition for each row in the list or grid (excluding header and footer rows). Always add a single Condition object to the Conditions property that reflects the object to use on each row.

  You can assign it to any of these types of controls: ListView, GridView, DataList, and Repeater.

  If unassigned, you must add Condition objects for each row in your list or grid.

  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.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

MultiConditionValidator

Condition	MultiCondition
License Required	Peter's Professional Validation
Supported controls	Uses the fields of other conditions that you select
Can evaluate blank fields	Depends on child conditions

The MultiConditionValidator evaluates multiple Conditions within a Boolean expression. Often, your validation rules cannot be expressed with one Condition. MultiConditionValidator will handle many of those cases. This Validator is frequently used. When you ask "how can I combine several Validators?", choose the MultiConditionValidator.

For example, the user picks a product from a listbox and enters a quantity in a text box. If each product has a different quantity limit, you need to determine if the user exceeds the limit. Suppose the first product in the drop down list is limited to 5. The Boolean expression to describe this rule could be written this way:

```
(SelectedIndexCondition on ListBox1 has SelectedIndex = 0)
AND
(CompareToValueCondition on TextBox1 has the value Less than or equal to 5)
```

The MultiConditionValidator will show its error message when the Boolean expression evaluates as `false`.

Often the MultiConditionValidator is used to show one error message in place of individual error messages of several Validator controls. For example, normally you have to set up two Validators with separate errors messages to handle the cases that the field must not be blank (RequiredTextValidator) and it contains an integer (DataTypeCheckValidator). If you want one error message, use a MultiConditionValidator with the conditions of the original Validators – RequiredTextCondition and DataTypeCheckCondition – and the logical AND. If either the field is blank or it's not an integer, one error message appears.

HINT: The Tutorial file includes several detailed examples on using the MultiConditionValidator.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Condition Properties for MultiConditionValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

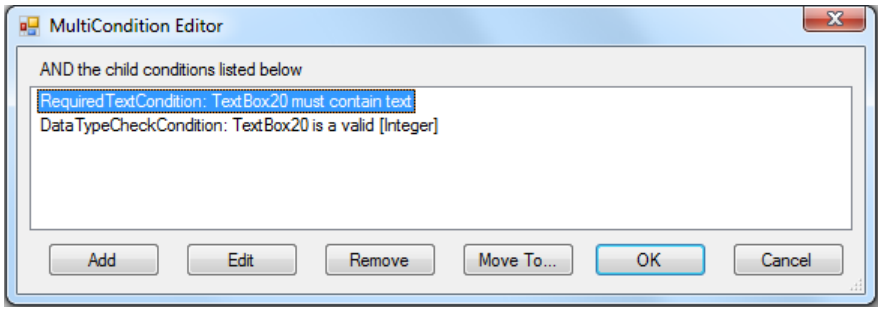
Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Using This Condition

Specify a list of conditions in the **Conditions** property. Use the **Operator** to determine if the conditions are evaluated with AND or OR logic.

The MultiConditionValidator offers a unique editor from within the Properties Editor to help you construct the expression. Here is the editor describing the same Boolean expression:




```
<des:MultiConditionValidator id="MultiConditionValidator1" runat="server"
      Operator="AND"
      ErrorMessage="The quantity is limited to 5.">
   <Conditions>
      <des:SelectedIndexCondition Name="SelectedIndexCondition"
         ControlIDToEvaluate="ListBox1" Index="0" Selected="True" />
      <des:CompareToValueCondition Name="CompareToValueCondition"
         ControlIDToEvaluate="TextBox1"
         DataType="Integer" Operator="LessThanEqual" ValueToCompare="5" />
   </Conditions>
</des:MultiConditionValidator>
```

The MultiConditionValidator can use any of the Conditions classes defined throughout DES, including those listed in this section with each Validator (see "Evaluation rules for Validators: Condition objects") and the Non-Data Entry Conditions described in the next section. The Non-Data Entry Conditions can evaluate attributes of fields that the user doesn't edit, such as visibility, enabled, and style settings.

If you want a Validator control to be enabled only when a certain Condition is met, you can use the MultiConditionValidator. However, you will find that each Validator control's **Enabler** property is a better choice. In fact, the **Enabler** property supports the MultiCondition, allowing you to build Boolean expressions that enable the control.

For example, suppose you have a textbox with a RequiredTextValidator. The RequiredTextValidator should only be enabled when two checkboxes on the form are marked. You would set up the **Enabler** property with a MultiCondition, like this:

```
<des:RequiredTextValidator id="RequiredTextValidator1" runat="server"
      ControlIDToEvaluate="TextBox1"
      ErrorMessage="Please assign a value here.">
   <EnablerContainer>
      <des:MultiCondition Name="MultiCondition" Operator="AND">
      <Conditions>
         <des:CheckStateCondition ControlIDToEvaluate="CheckBox1" />
         <des:CheckStateCondition ControlIDToEvaluate="CheckBox2" />
      </Conditions>
      </des:MultiCondition>
   </EnablerContainer>
</des:RequiredTextValidator>
```

*Note: The tag "<EnablerContainer>" is used by the **Enabler** property. Normally you would expect to see the tag with the same name as the property (<Enabler>). In this case, DES is working around a limitation of the .Net framework that only supports polymorphic properties by nesting them into a collection class. (EnablerContainer is a hidden property that is a collection class.)*

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Condition Properties for MultiConditionValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ReportErrorsAfter** and **NotCondition** – See "Properties Common to most Conditions".
- **Operator** (enum PeterBlum.DES.AND_OROperator) – Determines if the immediate child conditions are evaluated with a logical AND or logical OR operator. The enumerated type `PeterBlum.DES.AND_OROperator` has these values:
    - `OR` – This is the default
    - `AND`
- **Conditions** (PeterBlum.DES.ConditionsCollection) – A collection of Condition classes. See "Evaluation rules for Validators: Condition objects".

  Each Condition object will be part of a Boolean expression whose Boolean operators are determined by the **Operator** and **NotCondition** properties.

  As individual Conditions are evaluated, if they indicate success, they are considered "true" in the Boolean expression. If they indicate failure, they are considered "false". Any condition that cannot be evaluated will be ignored as if you didn't define it.

  You can use MultiCondition classes as items in this collection to build complex Boolean expressions. For example, to enter the expression: Checkbox1 is checked AND (TextBox1 contains text OR TextBox2 contains text), you will define a MultiConditionValidator with an AND operator and these conditions:

  CheckStateCondition on Checkbox1

  MultiCondition using the OR operator on these conditions:

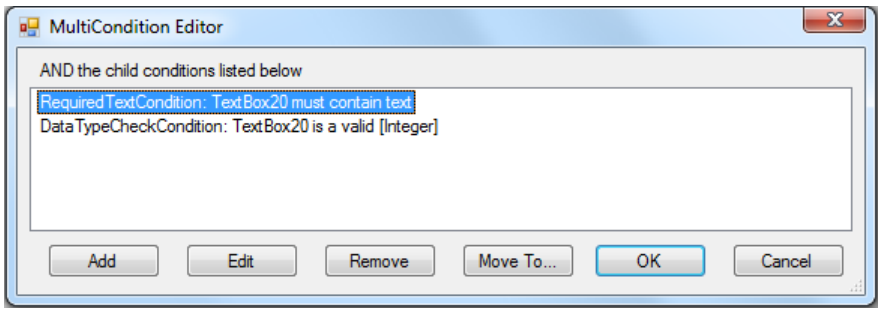
  RequiredTextCondition on TextBox1

  RequiredTextCondition on TextBox2

  HINT: The Tutorial file includes several detailed examples on using the MultiConditionValidator.

Adding Conditions In Design Mode

The Properties Editor for the **Conditions** property provides a window where you can define conditions.



- Click the **Add** button to add a new Condition. A list of Conditions appears. Select one and click **OK**. An editor window for the Condition appears. Fill in its property grid and click **OK**.

  If you want to add a child Condition to a MultCondition node, first click on it. Then click **Add**.
- Click the **Edit** button to open a Condition and edit its property grid.
- Click the **Remove** button to remove a Condition.
- Click the **Move To** button to relocate a Condition between the MultiConditionValidator and any of its child nodes.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Adding Conditions in ASP.NET Markup

Conditions is a type of collection. Therefore its ASP.NET text is nested as a series of child controls within the `<Conditions>` tag. Each Condition is a tag with `<des:conditionclass>` followed by the properties.

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

```
<des:MultiConditionValidator id="Validator1" runat="server">
   <Conditions>
      <des:SelectedIndexCondition ControlIDToEvaluate="ListBox1"
         Index="0" />
      <des:CompareToValueCondition ControlIDToEvaluate="TextBox1"
         ValueToCompare="5" DataType="Integer"
         Operator="LessThanEqual" />
   </Conditions>
</des:MultiConditionValidator>
```

Adding Conditions Programmatically

1. Create an instance of the desired Condition. 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 MultiConditionValidator object.

2. Assign property values.

3. Use the `Add()` method on the Conditions property to add the Condition object.

In this example, add the two Conditions shown in the above examples.

[C#]

```
PeterBlum.DES.Web.WebControls.SelectedIndexCondition vCond1 =
   new PeterBlum.DES.Web.WebControls.SelectedIndexCondition();
vCond1.ControlToEvaluate = ListBox1;
vCond1.Index = 0;
Validator1.Conditions.Add(vCond1);
PeterBlum.DES.Web.WebControls.CompareToValueCondition vCond2 =
   new PeterBlum.DES.Web.WebControls.CompareToValueCondition();
vCond2.ControlToEvaluate = TextBox1;
vCond2.ValueToCompareAsNative = 5;  // or ValueToCompare="5"
vCond2.DataType = "Integer";
vCond2.Operator = PeterBlum.DES.ConditionOperator.LessThanEqual;
Validator1.Conditions.Add(vCond2);
```

[VB]

```
Dim vCond1 As PeterBlum.DES.Web.WebControls.SelectedIndexCondition =_
   New PeterBlum.DES.Web.WebControls.SelectedIndexCondition()
vCond1.ControlToEvaluate = ListBox1
vCond1.Index = 0
Validator1.Conditions.Add(vCond1)
Dim vCond2 As PeterBlum.DES.Web.WebControls.CompareToValueCondition =_
   New PeterBlum.DES.Web.WebControls.CompareToValueCondition()
vCond2.ControlToEvaluate = TextBox1
vCond2.ValueToCompareAsNative = 5   ' or ValueToCompare="5"
vCond2.DataType = "Integer"
vCond2.Operator = PeterBlum.DES.ConditionOperator.LessThanEqual
Validator1.Conditions.Add(vCond2)
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

CustomValidator

Condition	CustomCondition
License Required	Peter's Professional Validation
Supported controls	All
Can evaluate blank fields	Depends upon the code that you write to evaluate data

The CustomValidator allows you to build your own Conditions without subclassing. (See also "IgnoreConditionValidator" for another approach to custom validation.) Use it when the existing Validators, including the MultiConditionValidator, do not provide the logic you need for your Condition. For example, you may need to compare a text field's value to data within a database.

The CustomValidator requires some programming, where you define an event handler to evaluate the Condition on the server side and a JavaScript function to evaluate the Condition on the client-side. See "Using This Condition" for details. DES can supply a lightweight client-side evaluation function when you use the **OverrideClientSideEvaluation** property.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Condition
- Steps to Defining Your CustomValidator
- Server-Side Condition
  - Server-Side Condition Example
  - Changing The Error Message On The Server Side
- Client-Side
  - The Client-Side Evaluation Function
  - Accessing Data On the Client-Side
  - Client-Side Condition Example
  - Changing The Error Message On The Client Side
- Passing around data into your evaluation function
- Condition Properties for CustomValidator
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

There are alternatives to using a CustomValidator:

- You can subclass a Condition class. This provides reusability and becomes part of the Conditions you can use when building a Boolean expression within a MultiConditionValidator or using the **Enabler** property. This way, you only need to write the new logic introduced by your Condition and let the MultiConditionValidator supply the JavaScript and server-side code for all other parts of the Condition needed on this page. For example, suppose a textbox must contain text and only then will it compare its value to a database. You would write your Condition to compare to the database and use the RequiredTextCondition along with your new condition in the MultiConditionValidator. See the Developer's Guide for details on subclassing Conditions.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

- Every Validator control provides the ability to replace both its client-side and server-side Condition code with your own. In fact, the CustomValidator is simply a Validator control that lacks its own evaluation functions and demands that you supply them. If another Validator control has a better set of properties, use it as the start of your custom Validator. You can even use its existing Condition, called from within your own code, if you just want to expand upon its Condition's rule. See "Extending Existing Validators and Conditions" for details.
- You can define a new data type to use in Conditions that have a **DataType** property. Suppose you want to use the logic of the CompareTwoFieldsValidator but the fields will contain a time of day. Subclass the `PeterBlum.DES.DESTypeConverter` class and install it into the **custom.des.config** file. Now DES's Validators and Conditions will support your data type. See the Developer's Guide for details on subclassing DESTypeConverters.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Using This Condition

This Condition requires you to create all of the evaluation logic. You should always define a server-side method that is attached to the **ServerCondition** event. You can create a similar client-side function in JavaScript and assign the function name to the **CustomEvalFunctionName** property.

Your function can access any control on the page through traditional coding. You can establish up to two controls in the **ControlIDToEvaluate** and **SecondControlIDToEvaluate** properties. When used, their values are passed to your evaluation functions automatically. On the client-side, your function will be fired when these fields are changed. Additionally, features like setting focus to the field with the error or changing its style are supported by your CustomValidator when using **ControlIDToEvaluate** and **SecondControlIDToEvaluate**.

Steps to Defining Your CustomValidator

These steps will give you the big picture for setting up a CustomValidator or replacing the conditions of other Validators. The sections that follow provide extensive details and examples.

1. **Always** set up server-side validation by testing **PeterBlum.DES.Globals.WebFormDirector.IsValid** is `true` in your post back event handler before using the data. See "Properties and Methods to Validate the Page".
2. Determine your logic. How will your code determine "success", "failure" and "cannot evaluate"? At this time, you make a decision if you can write client-side code. If your evaluation logic needs features only found on the server side, like database access, it will not offer a client-side evaluation function.
3. Determine what data entry controls on the page will be evaluated. Set **ControlIDToEvaluate** and **SecondControlIDToEvaluate** properties to any data entry controls that you want to fire your Validator on the client-side. These controls will have their textual values passed to your evaluation functions too. You can add other controls to **ExtraControlsToRunThisAction**.
4. Create a method that matches the delegate `PeterBlum.DES.ServerConditionEventHandler`. See "Server-Side Condition", below.

   Your method should set **args.IsMatch** to `true` or `false` depending on the results. If the conditions do not permit evaluation, set **args.CannotEvaluate** to `true`. For example, if you depend on the textbox to contain a date and it does not, it cannot be evaluated. Examples are shown in "Server-Side Condition Example".
5. If you want to modify the **ErrorMessage** property, see "Changing The Error Message On The Server Side".
6. Hookup your evaluation method to the **ServerCondition** event in your `Page_Load()` method. When using a grid, DetailsView, or FormView, use its **ItemCreated** or **RowCreated** event handler.

   [C#]

   ```
   CustomValidator1.ServerCondition +=
            new ServerConditionEventHandler(YourMethodName);
   ```

   [VB]

   ```
   AddHandler CustomValidator1.ServerCondition = AddressOf YourMethodName
   ```
7. While the server side evaluation event is required, a client-side evaluation function is optional. If you cannot write a client-side evaluation function, consider using the **OverrideClientSideEvaluation** property to give some interactivity. Stop at this step.
8. Write the client-side evaluation function in JavaScript. It takes one parameter, the client-side object that represents the condition. This object has properties to describe elements like the ID to the control specified in **ControlToEvaluate**. It must return one of three values: 1 (success), 0 (failed), -1 (cannot evaluate). See "Client-Side Condition" and "Client-Side Condition Example".
9. If you want to change the error message on the client-side, see "Changing The Error Message On The Client Side".
10. Assign the name of the function to the **CustomEvalFunctionName** property. In this example, the function named "CoordinateCondition" is used:

    ```
    <des:CustomValidator ID="CustomValidator1" runat="server"
       CustomEvalFunctionName="CoordinateCondition" [other properties]>
    </des:CustomValidator>
    ```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Server-Side Condition

You will write a method that matches the delegate `PeterBlum.DES.ServerConditionEventHandler` and assign it to the **ServerCondition** event on the CustomValidator control or CustomCondition class. Your code will be passed information about the Condition, controls selected by the user, and the textual value of those controls. It must return an indication whether the Condition indicates "success", "failure", or "cannot evaluate".

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

The `ServerConditionEventHandler` is defined here:

[C#]

```
public void ServerConditionEventHandler(
   PeterBlum.DES.IBaseCondition sourceCondition,
   PeterBlum.DES.IConditionEventArgs args);
```

[VB]

```
Public Sub ServerConditionEventHandler( _
   ByVal sourceCondition As PeterBlum.DES.IBaseCondition, _
   ByVal args As PeterBlum.DES.IConditionEventArgs)
```

Parameters

sourceCondition

The Condition class that is associated with this condition. It contains all the properties from the user, including **ControlIDToEvaluate**. However, *args* offers a better way to get to the controls assigned by the user.

When accessing its properties, be sure to typecast sourceCondition to the Condition subclass that you expect. When using this with CustomValidator or CustomCondition, typecast to `PeterBlum.DES.Web.WebControls.CustomCondition`.

args

The *args* parameter is used to communicate with the caller. The caller will pass a subclass of ConditionEventArgs, `PeterBlum.DES.Web.WebControls.ConditionTwoFieldEventArgs`, with references to each control assigned by the user in **ControlIDToEvaluate** and **SecondControlIDToEvaluate** (if available). It will also retrieve the textual value from each of these fields (if possible) and supply them. Your event handler should evaluate the condition based on these inputs and set the property **IsMatch** to `true` or `false`. If your event handler identifies that the Condition cannot be evaluated, it should set **CannotEvaluate** to `true`.

If your evaluation method is called from a Validator control (some CustomConditions are used in other controls), you can modify the error message and summary error message shown on the Validator if you like. Access the Validator control with the **Validator** property and modify its **ErrorMessage**, **ErrorMessageLookupID**, **SummaryErrorMessage**, and **SummaryErrorMessageLookupID** properties. See "Changing The Error Message On The Server Side".

REMINDER: Always typecast the args parameter to PeterBlum.DES.Web.WebControls.ConditionTwoFieldEventArgs when using it with a CustomValidator or CustomCondition.

Note: CustomValidator and CustomCondition are passed PeterBlum.DES. Web.WebControls.ConditionTwoFieldEventArgs, not PeterBlum.DES.ConditionEventArgs, because this system handles another case: customizing the evaluation method of any other Validator. They take PeterBlum.DES. Web.WebControls.ConditionEventArgs, PeterBlum.DES.Web.WebControls.ConditionOneFieldEventArgs or PeterBlum.DES.Web.WebControls.ConditionTwoFieldEventArgs depending on their support for the ControlIDToEvaluate and SecondControlIDToEvaluate properties

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Properties of PeterBlum.DES.Web.WebControls.ConditionTwoFieldEventArgs

- **IsMatch** (Boolean) – When your condition evaluates as "success", set this to `true`. When your condition evaluates as "failed", set this to `false`. When your condition evaluates as "cannot evaluate", leave this alone and set **CannotEvaluate** to `true`. **IsMatch** defaults to `false` so be sure to set it to `true` if it's valid.
- **CannotEvaluate** (Boolean) – When your condition cannot evaluate the data it's given, set this to `true`. An example of "cannot evaluate": The user enters X,Y coordinates. Your function evaluates that X and Y are within the range of 1-1000. You decide that cannot evaluate a blank field. (You leave this to the RequiredTextValidator.)
- **Validator** (PeterBlum.DES.IValidatorControl) – The Validator control that is being evaluated. Generally you use this to modify the error message. If you created a CustomCondition for use in another type of control, this will be `null`/`nothing`. See "Changing The Error Message On The Server Side".
- **Value** (string) – The textual value associated with **Validator.ControlIDToEvaluate**. If the **Trim** property is `true`, it will be trimmed already. If the control does not have a textual value, it will be "".
- **ValueAsNative** (object) – Many controls do not use a string as their native data type. This holds the native type of the value represented by the **Value** property, if available. It may be a string too.
- **ControlToEvaluate** (System.Web.UI.Control) – A reference to the control assigned to **Validator.ControlToEvaluate**. If **ControlToEvaluate** is not used, this is `null`/`nothing`.
- **SecondValue** (string) – The textual value associated with **Validator.SecondControlIDToEvaluate**. If the **Trim** property is `true`, it will be trimmed already. If the control does not have a textual value, it will be "".
- **SecondValueAsNative** (object) – Many controls do not use a string as their native data type. This holds the native type of the value represented by the **SecondValue** property, if available. It may be a string too.
- **SecondControlToEvaluate** (System.Web.UI.Control) – A reference to the control assigned to **Validator.SecondControlToEvaluate**. If **SecondControlToEvaluate** is not used, this is `null`/`nothing`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Server-Side Condition Example

In this example, the event handler will evaluate one textbox field to determine that it represents coordinates with the format "##,##" (a location with X,Y coordinates). It expects the user to assign **ControlIDToEvaluate** to the textbox. It cannot evaluate when the **ControlIDToEvaluate** is unassigned or the **Value** is blank.

[C#]

```
using PeterBlum.DES;
// other uses statements and the namespace declaration

public class MyForm : System.Web.UI.Page
{
   protected System.Web.UI.WebControls.TextBox TextBox1;
   protected CustomValidator CustomValidator1;
   protected void Page_Load(object sender, System.EventArgs e)
   {
      CustomValidator1.ServerCondition +=
         new ServerConditionEventHandler(CoordinateDataTypeCheck);
   }
   protected void CoordinateDataTypeCheck(IBaseCondition sourceCondition,
      IConditionEventArgs args)
   {
         // shows typecasting
      CustomCondition vCondition = (CustomCondition) sourceCondition;
      ConditionTwoFieldEventArgs vArgs =
         (ConditionTwoFieldEventArgs) args;

         // cannot evaluate when blank or ControlToEvaluate is unassigned
      if ((vArgs.Value == "") ||
         (vArgs.ControlToEvaluate == null))
         vArgs.CannotEvaluate = true;
      else
      {  // evaluate
         string[] vParts = vArgs.Value.Split(",");
         if (vParts.Length < 2)     // bad format
            vArgs.IsMatch = false;
         else
            try
            {
               Convert.ToInt32(vParts[0]); // throws exceptions if not ints
               Convert.ToInt32(vParts[1]);
               vArgs.IsMatch = true;
            }
            catch (Exception)
            {
               vArgs.IsMatch = false;
            }
      }
   }  // CoordinateDataTypeCheck
}  // class
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

[VB]

```
Imports PeterBlum.DES
' other Imports statements and the namespace declaration

Public Class MyForm
   Inherits System.Web.UI.Page
{
   Protected WithEvents TextBox1 As System.Web.UI.WebControls.TextBox
   Protected WithEvents CustomValidator1 As CustomValidator
   Protected Sub Page_Load(ByVal sender As object, _
      ByVal e As System.EventArgs)
      AddHandler CustomValidator1.ServerCondition = _
         AddressOf CoordinateDataTypeCheck
   End Sub

   Protected Sub CoordinateDataTypeCheck( _
      ByVal sourceCondition As IBaseCondition, _
      ByVal args As IConditionEventArgs)

         ' shows typecasting
      Dim vCondition As CustomCondition = _
         CType(sourceCondition, CustomCondition)
      Dim vArgs As ConditionTwoFieldEventArgs =_
         CType(args, ConditionTwoFieldEventArgs)

         ' cannot evaluate when blank or ControlToEvaluate is unassigned
      If ((vArgs.Value = "") Or _
         (vArgs.ControlToEvaluate Is Nothing)) Then
         vArgs.CannotEvaluate = True
      Else  ' evaluate
         Dim vParts() As string = vArgs.Value.Split(",")
         If (vParts.Length < 2) Then
            vArgs.IsMatch = False   ' bad format
         Else
            Try
               Convert.ToInt32(vParts(0)) ' throws exceptions if not ints
               Convert.ToInt32(vParts(1))
               vArgs.IsMatch = True
            Catch e As Exception
               vArgs.IsMatch = False
            End Try
         End If
      End If
   End Sub  ' CoordinateDataTypeCheck
End Class
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Changing The Error Message On The Server Side

Your server-side condition event handler can modify the error message of any Validator that uses it. See "Server-Side Condition".

The **ServerCondition** event handler passes you the ConditionEventArgs object. It contains the **Validator** property, which is a reference to the Validator control. Your event handler can modify any property on the Validator control based on your rules for the condition. Most likely, you will change **ErrorMessage**, **ErrorMessageLookupID**, **SummaryErrorMessage**, and **SummaryErrorMessageLookupID**.

Be aware of these rules when using the **ConditionEventArgs.Validator** property:

- Changes you make will only apply to the currently generated page. When the page is posted back the next time, the properties will return to the values assigned in the ASP.NET Markup or `Page_Load()` method. *You can override this by using the ViewStateMgr property on a Validator. Call Validator.ViewStateMgr.TrackProperty("propertyname") in Page_Load().*
- The **Validator** property will be `null`/`nothing` when your Condition event handler is used in a FieldStateController.

This is an example of updating the **ErrorMessage** property within the evaluation method. It replaces the property with the value "One is not supported" when the user enters a value of "One":

[C#]

```
using PeterBlum.DES;
…
 protected void MyCondition(IBaseCondition sourceCondition,
    IConditionEventArgs args)
 {
    ConditionTwoFieldEventArgs vArgs =
             (ConditionTwoFieldEventArgs) args;
    ... condition evaluation code here. Set IsMatch ...

    // assuming args.Value = "One", change the message
    if ((vArgs.Value == "One") && (vArgs.Validator != null))
       vArgs.Validator.ErrorMessage = "One is not supported";

 }  // MyCondition
```

[VB]

```
Imports PeterBlum.DES
…
 Protected Sub MyCondition(ByVal sourceCondition As IBaseCondition, _
    ByVal args As IConditionEventArgs)
    Dim vArgs As ConditionTwoFieldEventArgs = _
       CType(args, ConditionTwoFieldEventArgs)
    ... condition evaluation code here. Set IsMatch ...

    ' assuming args.Value = "One", change the message
    If (vArgs.Value = "One") And Not (vArgs.Validator Is Nothing)
       vArgs.Validator.ErrorMessage = "One is not supported"
    End If
 End Sub  ' MyCondition
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Client-Side Condition

The client-side can use an "evaluation function" to implement the logic of your Condition, much like it uses on the server-side. Each Validation control except for CustomValidator and IgnoreConditionValidator supplies the evaluation function for you. You can supply the evaluation function with these two properties:

- **CustomEvalFunctionName** – When you write your own function, specify its name here. The remainder of this topic explains how to set up your custom function.
- **OverrideClientSideEvaluation** – When you only want server side validation, you can still offer a client-side user interface with this property. It's interface hides the error message after the field is changed or submit it clicked. It can also invoke a callback to let server side validation update the error message after the Control to Evaluate has changed. See the **OverrideClientSideEvaluation** property.

If neither of these properties are set up, the CustomValidator and CustomCondition will continue to provide server-side validation, but no client-side validation.

Click on any of these topics to jump to them:

- The Client-Side Evaluation Function
- Accessing Data On the Client-Side
- Client-Side Condition Example
- Changing The Error Message On The Client Side

The Client-Side Evaluation Function

You write the function in JavaScript, either adding it directly to the page or through the `Page.RegisterClientScriptBlock()` method. See "Adding Your JavaScript to the Page".

The evaluation function takes one parameter, the client-side representation of the Condition class. This is a JavaScript object with numerous properties. For a complete look at the client-side Condition object, see the Developer's Guide.

The client-side function must return an integer with one of these values: 1 = success; 0 = failed; -1 = cannot evaluate.

Here is the basic structure of the function:

```
function MyConditionName(cond)
{
   // evaluate the condition from the fields on the page
   if (cannotevaluate logic)
      return -1;
   else if (success logic)
      return 1;
   else
      return 0;
}
```

Here is a sample CustomValidator using MyConditionName:

```
<des:CustomValidator id="CustomValidator1" runat="server"
  CustomEvalFunctionName="MyConditionName" other properties />
```

On the next few pages, you will see some JavaScript functions that assist in writing your function and an example.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Accessing Data On the Client-Side Condition object

DES supplies a number of methods and properties on the Condition object (the *cond* parameter). It also supplies functions to evaluate some of those properties. Here are a few that will help you build your evaluation function:

- The property *cond*.IDToEval contains the ID to the **ControlToEvaluate**. You can access the element that represents this control using `DES_GetByID(id)` like this:

  ```
  var vFld = DES_GetById(cond.IDToEval);
  ```

- The property *cond*.IDToEval2 contains the ID to the **SecondControlToEvaluate**. You can access the element that represents this control using `DES_GetByID(id)` like this:

  ```
  var vFld2 = DES_GetById(cond.IDToEval2);
  ```

- The `DES_GetTextValue(id, trim, cond.GetText)` function retrieves the textual value of an control, given its ID and a flag indicating if you want to trim the text or not. (cond.**Trim** contains the same value found in the Validator's **Trim** property.)

  ```
  var vVal = DES_GetTextValue(cond.IDToEval, cond.Trim, cond.GetText);
  var vVal2 = DES_GetTextValue(cond.IDToEval2, cond.Trim, cond.GetText);
  ```

- The `DES_GetSelIdx(id, cond.GetSelIdx)` function retrieves the index of a list or DropDownList control, given its ID and the property **GetSelIdx** on the *cond* parameter. (**GetSelIdx** may be `null`.) It returns -1 for no selection, 0 for the first element selected, etc.

  ```
  var vIdx = DES_GetSelIdx(cond.IDToEval, cond.GetSelIdx);
  var vIdx2 = DES_GetSelIdx(cond.IDToEval2, cond.GetSelIdx2);
  ```

- When using DES's numeric, date or time textboxes, use the `DES_GetDTTBValue(id)` function to return their value in its native format (integer, floating point, JavaScript Date or integer representing a number of seconds).

  ```
  var vNativeValue = DES_GetDTTBValue(cond.IDToEval);
  ```

- Checkboxes and RadioButtons, except those in CheckBoxLists or RadioButtonLists, can be evaluated like this:

  ```
  var vChecked = DES_GetById(cond.IDToEval).checked;
  ```

- Checkboxes and RadioButtons within CheckBoxLists or RadioButtonLists can be evaluated like this:

  ```
  var vChecked = DES_GetById(cond.IDToEval + "_" + pos).checked;
  ```

  pos is the offset into the list where 0 is the first item.

- If the condition is part of a Validator, you can override the error message and summary error message. See "Changing The Error Message On The Client Side".

- See "JavaScript Support Functions" for additional functions.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Client-Side Condition Example

This example is the same as the Server-Side Condition Example above. The JavaScript function will evaluate one textbox field to determine that it represents coordinates with the format "##,##". The user is expected to assign **ControlIDToEvaluate** to the textbox. It cannot evaluate when the **ControlIDToEvaluate** is unassigned or the textual value is blank.

```
<script type='text/javascript' language='javascript'>
function CoordinateCondition(cond)
{
   if (cond.IDToEval == "")   // ControlIDToEvaluate is unassigned
      return -1;
   var vVal1 = DES_GetTextValue(cond.IDToEval, cond.Trim, cond.GetText);
   if (vVal1 == "")     // blank
      return -1;
   var vParts = vVal1.split(",");
   if (vParts.length < 2)
      return 0;   // failed – bad format
   else if ((DES_ParseInt(vParts[0]) != NaN) &&
      (DES_ParseInt(vParts[1]) != NaN))
      return 1;   // success
   else
         return 0;      // failed – could not convert to numbers
}
</script>
```

Here is the associated CustomValidator:

```
<des:CustomValidator id="CustomValidator1" runat="server"
   CustomEvalFunctionName="CoordinateCondition" ControlIDToEvaluate="TextBox1" />
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Changing The Error Message On The Client Side

You can modify the error message and summary error message on any condition that you define in the **CustomEvalFunctionName** property. See "Client-Side Condition".

When you write your evaluation function, the function is passed the client-side Condition object. That object contains a property called '**Action**' which contains the error message in its **ErrMsg** property and the summary error message in the **SumMsg** property. You can change each of these properties in your custom condition function, based on the rules of your Condition.

Be aware of these rules when using the **Action** property.

- If you decide to change the **ErrMsg** or **SumMsg** property values, do it for every case your function is called. After all, you are overriding the original error message from when the page is loaded. So assign **ErrMsg** to its original value and new values as the condition dictates.
- Always test for the presence of the **Action** property before getting its properties. There are cases where Action is `null` (doesn't exist.)

This is an example of updated both the **ErrMsg** and **SumMsg** properties. This function gets text from a text box. If they enter "One", use the error message "One is not supported". Otherwise, use the message "Type something else".

```
function MyCondition(cond)
{
   if (cond.IDToEval == "")
      return -1;
   var vValue = DES_GetTextValue(cond.IDToEval, cond.Trim, cond.GetText);
   // always set the error message when the Action exists
   if (cond.Action != null)
      if (vValue == "One")
      {
         cond.Action.ErrMsg = "One is not supported";
         cond.Action.SumMsg = "One is not supported in the first field.";
      }
      else
      {
         cond.Action.ErrMsg = "Type something else";
         cond.Action.SumMsg = "Type something else in the first field.";
      }
   ... evaluate the condition and return 1, 0 or -1 ...
}
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Passing around data into your evaluation function

Sometimes your evaluation function needs extra data to make its evaluation. You can have that data assigned to the CustomValidator control through its `SetUserData()` method. Your server side method can retrieve that data by calling the `GetUserData<T>()` method on the Condition object you were passed. Your client side function can have extra properties assigned to its cond parameter by applying the `PeterBlum.DES.Web.IUserDataHtmlClient` interface to the data object.

Click on any of these topics to jump to them:

- Defining your data
- Attaching the data to your CustomValidator
- Using your data inside the server-side evaluation method
- Using your data inside the client-side evaluation function

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Defining your data

The `SetUserData()` method takes a single parameter, an object. That object can be anything. If you want to pass along several values, consider creating a class with properties for each of the values. If you intend to expose this data to the client-side function, you must create a class because you will need to implement the IUserDataHtmlClient interface.

The IUserDataHtmlClient interface has two methods:

- `AddCode(FormDirector)` – Add script files and script blocks to the page. Use the FormDirector's ClientScripts class which has functions to write out javascript. Primarily use `RegisterClientScriptBlock()` to add code near the top of the form and `RegisterStartupBlock()` to add code near the bottom of the form. These functions are similar to the same-named functions of the System.Web.UI.ClientScriptsManager class on the Page object.
- `AddData(writer, FormDirector)` – The writer parameter is a class that is collecting properties for the *cond* parameter of the client-side evaluation function. Just call its `RegisterValue("propertyname", value)` method to add a property.

Example

The MyCustomValidatorData class will provide two values to both server and client side evaluation functions.

[C#]

```
public class MyCustomValidatorData : PeterBlum.DES.Web.IUserDataHtmlClient
{
  public string MyString { get; set; }

  public int MyInt { get; set; }

  public void AddCode(PeterBlum.DES.Web.IBaseHtmlFormDirector formDirector)
  {
    // nothing needed, but it could add scripts here
  }

  public void AddData(JavaScriptWriter writer,
      PeterBlum.DES.Web.IBaseHtmlFormDirector formDirector)
  {
    writer.RegisterValue("MyString", MyString);
    writer.RegisterValue("MyInt", MyInt);
  }
}
```

[VB]

```
Public Class MyCustomValidatorData
    Implements PeterBlum.DES.Web.IUserDataHtmlClient

  Public Property MyString() As String  …
  End Property

  Public Property MyInt() As Integer …
  End Property

  Public Sub AddCode(formDirector As PeterBlum.DES.Web.IBaseHtmlFormDirector)
    ' nothing needed, but it could add scripts here
  End Sub

  Public Sub AddData(writer As JavaScriptWriter, _
      formDirector As PeterBlum.DES.Web.IBaseHtmlFormDirector)
    writer.RegisterValue("MyString", MyString)
    writer.RegisterValue("MyInt", MyInt)
  End Sub
End Class
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Attaching the data to your CustomValidator

Assign the data before its needed by calling CustomValidator's SetDataValue() method with your data. When not in a postback, you only need to assign the data once in the Page_Init, Page_Load, or Page_PreRender event handlers of your Page. Our recommendation is in the Page_Load method, because that helps when setting up code for postback.

However with postback, there are two phases. If it uses server side validation, your code must be invoked before that happens, in Page_Load. Then if the data changes as a result of the postback, it needs to be reassigned later, in Page_PreRender.

Example

[C#]

```
public void Page_Load(object sender, EventArgs e)
{
   CustomValidator1.SetUserData(new MyCustomValidatorData("test", 100));
}

public void Page_PreRender(object sender, EventArgs e)
{
// this code is only needed if the user data may have changed,
// perhaps due to databinding
   if (IsPostBack)
      CustomValidator1.SetUserData(new MyCustomValidatorData("test", 100));
}
```

[VB]

```
Public Sub Page_Load(sender As object, e As EventArgs)
   CustomValidator1.SetUserData(New MyCustomValidatorData("test", 100))
End Sub

Public Sub Page_PreRender(sender As object, e As EventArgs)
' this code is only needed if the user data may have changed,
' perhaps due to databinding
   If IsPostBack Then
      CustomValidator1.SetUserData(New MyCustomValidatorData("test", 100))
   End If
End Sub
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Using your data inside the server-side evaluation method

Typecast the first parameter, the condition object, to `PeterBlum.DES.ICustomCondition` and call its `GetUserData()` method. This method has a generic parameter to indicate the type you demand. This allows you to avoid typecasting the data it returns.

Example

[C#]

```
void MyServerCondition(IBaseCondition sourceCondition, IConditionEventArgs args)
{
  ICustomCondition vCondition = (ICustomCondition) sourceCondition;
  MyCustomValidatorData vUserData =
    vCondition.GetUserData<MyCustomValidatorData>();
  // use the data
}
```

[VB]

```
Private Sub MyServerCondition(_
  sourceCondition As IBaseCondition, args As IConditionEventArgs)

  Dim vCondition As ICustomCondition = _
    DirectCast(sourceCondition, ICustomCondition)
  Dim vUserData As MyCustomValidatorData = _
    vCondition.GetUserData(Of MyCustomValidatorData)()
  ' use the data
End Sub
```

Using your data inside the client-side evaluation function

When implementing PeterBlum.DES.Web.IUserDataHtmlClient on your data, your AddData() method defines new properties on the *cond* parameter. They have the same name (case sensitive) as you passed to the writer.RegisterValue() method. Their values have been correctly typed. For strings, the value is a string. For numbers, the value is a number. All you need to do is use these properties.

Example

Assume the server side code looked like this in AddData():

```
    writer.RegisterValue("MyString", MyString);
    writer.RegisterValue("MyInt", MyInt);
```

Here is a function to use MyString and MyInt within the error message.

```
<script type='text/javascript' language='javascript'>
function MyCondition(cond)
{
   // the actual validation code

   // insert values into the error message using MyString and MyInt
   cond.Action.ErrMsg = "[" + cond.MyString + "] needs " + cond.MyInt + "entries";
}
</script>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Condition Properties for CustomValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, **ValueToEvaluate**, **ReportErrorsAfter**, **Trim**, **NotCondition** and **ExtraControlsToRunThisAction** – See "Properties Common to most Conditions".
- **SecondControlIDToEvaluate** (string) – Identifies an optional second control. It is passed to your evaluation function on the client and server side.

  If you want to connect to a control that is inside of a usercontrol, see "Attaching Validators to controls inside of a User Control".
- **SecondControlToEvaluate** (System.Web.UI.Control) – An alternative to **SecondControlIDToEvaluate**. Use it when the control is not in the same or ancestor naming container. It must be assigned programmatically. For example, if you have a Validator instance in the variable "Val2" and a textbox instance in the variable "TextBox2", write code like this: `Val2.SecondControlToEvaluate = TextBox2`.

  When programmatically assigning properties to a Validator control or Condition class, if you have access to the control object that will be evaluated, it is better to assign it here than assign its ID to the **SecondControlIDToEvaluate** property because DES operates faster using **SecondControlToEvaluate**.
- **SecondValueToEvaluate** (object) – Allows the Validator or Condition to be used without a control attached. Instead, data comes from the value assigned to this property. When used, it replaces **SecondControlIDToEvaluate** and **SecondControlToEvaluate**.

  See "Using Validator controls to evaluate data not from Controls".
- **CustomEvalFunctionName** (string) – Specify the function name of the client-side evaluation function as described in "The Client-Side Evaluation Function".
- **OverrideClientSideEvaluation** (PeterBlum.DES.OverrideClientSideEvaluationType) – Use this when you cannot write a JavaScript evaluation function but want some interaction in your user interface. This property sets up a predefined evaluation function that hides the error message your server side evaluation function set during post back. It hides the error message when the user changes the field(s) or submits the page.

  Choose this when your server side function's logic cannot be reproduced on the client-side, such as a database lookup. This property will allow your server side function to supply the error message and the client-side to hide it and to blink it if blinking is established in `Globals.WebFormDirector.BlinkOnSubmit`.

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

  - `No` – Do not override. Use the function identified by **CustomEvalFunctionName** property or no client-side user interface. This is the default.
  - `Hide` – On post back, remove the error message after the user changes a field or submits the page. On a new page, the error message is always hidden. It monitors changes on **ControlIDToEvaluate**, **SecondControlIDToEvaluate** or any control in **ExtraControlsToRunThisAction**.
  - `HideOnSubmit` – On post back, remove the error message only after the user submits the page. If there are other errors on the page, the user will notice its removal. On a new page, the error message is always hidden.
  - `Callback` –A change to the **ControlIDToEvaluate** will cause a behind-the-scenes callback to the server to run your page and validate the Validator. This way, your page appears to be using client-side validation even though the server does the actual work. This system uses the ASP.NET Client Callbacks architecture which has these issues:
    - Some browsers may not support it. In this case, the Validator will only update on a traditional post back. Consider using the `CallbackOrHide` setting to offer some kind of user interface for these browsers.
    - The process of callbacks may take some time. After all, it has to make a request of the server. Once the callback completes, the error message is either shown or hidden.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- Only one Validator should use this setting per data entry control to be validated. Unfortunately, the ASP.NET Callback architecture is designed for one event at a time.

When you use this feature, your page should be prepared a little differently. The Callback system will run your OnInit and Page_Load events. ASP.NET supplies the Page.IsCallback property to help you optimize your code within these methods. The goal is to avoid doing work that is not necessary for the Validator to do its job. For example, if you normally query the database on post back, do not do this unless that query is used by the Validator.

- `CallbackOrHide` –Works the same as `CallBack` except when the browser doesn't support the ASP.NET 2.0 client callback architecture, it works like the `Hide` setting.

Note: CustomConditions are supported in the Enabler property and FieldStateControllers. OverrideClientSideEvaluationType should not be used in the Enabler or FieldStateController.

Example of Responding to a Callback

Suppose you have a CustomValidator that uses either OverrideClientSideEvaluation=`Callback` or `CallbackOrHide`. This is how you intercept the callback and set the Validator's **IsValid** property.

This code is within CustomValidator's `ServerCondition` evaluation method. When you evaluation method is called the ASP.NET page has assigned the data values of all web controls that are created as `Page_Load()` finishes. You use those web controls to determine the value of **IsValid**.

In this example, TextBox1 must contain the name "Peter" (case insensitive).

[C#]

```
if (Page.IsCallBack)
{
   // Determine if the data is valid here
   IgnoreConditionValidator1.IsValid =
      CaseInsensitiveComparer.Default.Compare(TextBox1.Text, "Peter") == 0;
}
```

[VB]

```
If Page.IsCallBack Then
' Determine if the data is valid here
   IgnoreConditionValidator1.IsValid = _
      CaseInsensitiveComparer.Default.Compare(TextBox1.Text, "Peter") = 0
End If
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

IgnoreConditionValidator

Condition	n/a
License Required	Peter's Professional Validation
Supported controls	n/a
Can evaluate blank fields	n/a

The IgnoreConditionValidator is similar to the CustomValidator. Both Validators are designed for you to write your own code to determine if the Condition is valid or not. They go about it in different ways:

- CustomValidator supports both client and server side validation. It requires your evaluation logic to be in its own event handler method.
- IgnoreConditionValidator supports only server side validation. You usually use it when your validation code is part of some other logic that should not be separated out into the CustomValidator's evaluation method. For example, you let your business layer run some validation against the data. If it's good, that layer saves the data; if it's bad, that layer returns an error message.

The IgnoreConditionValidator allows you to write code in one place and simply set the **IsValid** property to `false` on the Validator when an invalid condition is found.

- You cannot create a client-side evaluation function. This is because it is designed for cases that are server specific, such as database queries and calling web services. However, the **OverrideClientSideEvaluation** property can provide client-side code to automatically hide the error message.
- Generally you are writing a post back event handler for a button that determines something is invalid.
- **IsValid** is always `true` after calling `PeterBlum.DES.Globals.WebFormDirector.Validate()`. To indicate the Validator is invalid, simply set **IsValid** to `false`.

For example, suppose you had a textbox where the user enters a Tracking ID code. You want to show a validation error message when the user submits and there is no match for the Tracking ID code in your database. You would add the IgnoreConditionValidator to show the error message. Then write the submit button's Click event handler to run the database lookup against the user's Tracking ID code and either set the IgnoreConditionValidator.**IsValid** property to `false` (failed to match) or take some other action when a matching record is found.

Click on any of these topics to jump to them:

- Adding Validation to a WebForm
- Using This Validator
- Condition Properties for IgnoreConditionValidator
  - Example of Responding to a Callback
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Changing When the Validator is Evaluated
- Other Properties
- Additional Validation Topics
- Online example

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Using This Validator

Set the **IsValid** property to `false` on post back after detecting an error. Optionally set the **ErrorMessage** property to the desired message.

*ALERT: **Always** set up server-side validation by testing **PeterBlum.DES.Globals.WebFormDirector.IsValid** is `true` in your post back event handler before using the data from the page. See "Properties and Methods to Validate the Page".*

If there is a control on the page which it uses, set the **RelatedControlID** property to that control. **RelatedControlID** is supported by features like setting focus to the field with the error, changing the style of the field with the error, and ValidationSummary's **HyperLinkToFields** property.

This code shows the Click event handler that detects an illegal Tracking ID code in TextBox1. The stored procedure "LookupCode" is passed the code and returns a row from a table with a column ID that contains the record ID. If that ID is found, some action is taken. Otherwise, the IgnoreConditionValidator (ID=IgnoreConditionValidator1) has its **IsValid** property set to false and the method is exited.

[C#]

```
private void Submit_Click(object sender, System.EventArgs e)
{
   if (PeterBlum.DES.Globals.WebFormDirector.IsValid)
   {
      // connection string is in Application["ConnStr"]
      SqlConnection vConnection =
         new SqlConnection((string) Application["ConnStr"]);
      SqlCommand vCommand = new SqlCommand("LookupCode", vConnection);
      vCommand.Parameters.Add("@Code", TextBox1.Text.Trim());
      vCommand.CommandType = CommandType.StoredProcedure;
      try
      {
         vConnection.Open();
         SqlDataReader vReader = vCommand.ExecuteReader();
         if (vReader.Read())
         {
         // do something here with the record found
         }
         else  // did not match.
         {
         // Set the IsValid property to false and exit the method
            IgnoreConditionValidator1.IsValid = false;
            return;
         }
      }
      finally
      {
         vConnection.Close();
      }
   }  // if
}
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

[VB]

```
Private Sub Submit_Click(ByVal sender As Object, ByVal e As System.EventArgs)
   If PeterBlum.DES.Globals.WebFormDirector.IsValid Then
      ' connection string is in Application["ConnStr"]
      Dim vConnection As SqlConnection = _
         New SqlConnection(Application["ConnStr"].ToString())
      Dim vCommand As SqlCommand = New SqlCommand("LookupCode", vConnection)
      vCommand.Parameters.Add("@Code", TextBox1.Text.Trim())
      vCommand.CommandType = CommandType.StoredProcedure
      Try
         vConnection.Open()
         Dim vReader As SqlDataReader = vCommand.ExecuteReader()
         If vReader.Read() Then
            ' do something here with the record found
         Else  ' did not match.
            ' Set the IsValid property to false and exit the method
            IgnoreConditionValidator1.IsValid = False
            Exit Sub
         End If
      Finally
         vConnection.Close()
      End Try
   End If
End Sub
```

The IgnoreConditionValidator is often used to adding a list of error messages to the ValidationSummary control. Here's how:

- Create one IgnoreConditionValidator for each error message.
- Set the **ErrorMessage** property.
- Set **IsValid** to `false`.
- Set **ErrorFormatter.Display** property to `None`.
- Add the IgnoreConditionValidator to a PlaceHolder or other container control.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Condition Properties for IgnoreConditionValidator

Each Validator control includes a Condition with several properties that require setup. The Properties Editor shows them in the "Condition" section.

The following list are properties specific to this Condition:

- **Trim**, **NotCondition** and **ExtraControlsToRunThisAction** – See "Properties Common to most Conditions".
- **RelatedControlID** (string) – Identifies a control that contains data used by this Validator. It is optional. When assigned, it supports several features in DES:
  - **ValidationSummary.HyperLinkToFields** property will establish a hyperlink to this control.
  - All of these properties on **PeterBlum.DES.Globals.WebFormDirector.ValidationManager** will use this control: **FocusAfterAlert** and **ChangeStyleOnControlsWithError**.
- **RelatedControl** (System.Web.UI.Control) – An alternative to **RelatedControlID**. Use it when the control is not in the same or ancestor naming container. It must be assigned programmatically. For example, if you have a Validator instance in the variable "Val2" and a textbox instance in the variable "TextBox2", write code like this: `Val2.RelatedControl = TextBox2`.

  When programmatically assigning properties to this control, if you have access to the control object that will be evaluated, it is better to assign it here than assign its ID to the **RelatedControlID** property because DES operates faster using **RelatedControl**.
- **OverrideClientSideEvaluation** (PeterBlum.DES.OverrideClientSideEvaluationType) – Use this when you want some interaction in your user interface. This property provides client-side behavior that hides the error message your server side evaluation function set during post back. It hides the error message when the user changes the field(s) or submits the page.

  The enumerated type `PeterBlum.DES.OverrideClientSideEvaluationType` has these values:
  - `No` – No client-side behaviors. This is the default.
  - `Hide` – On post back, remove the error message after the user changes a field specified by **RelatedControlID** or submits the page. On a new page, the error message is always hidden.
  - `HideOnSubmit` – On post back, remove the error message only after the user submits the page. If there are other errors on the page, the user will notice its removal. On a new page, the error message is always hidden.
  - `Callback` – A change to the **ControlIDToEvaluate** will cause a behind-the-scenes callback to the server to run your page and validate the Validator. This way, your page appears to be using client-side validation even though the server does the actual work. This system uses the ASP.NET 2.0 Client Callbacks architecture which has these issues:
    - Some browsers may not support it. In this case, the Validator will only update on a traditional post back. Consider using the `CallbackOrHide` setting to offer some kind of user interface for these browsers.
    - The process of callbacks may take some time. After all, it has to make a request of the server. Once the callback completes, the error message is either shown or hidden.
    - Only one Validator should use this setting per data entry control to be validated. Unfortunately, the ASP.NET 2.0 Callback architecture is designed for one event at a time.

    When you use this feature, your page should be prepared a little differently. The Callback system will run your OnInit and Page_Load events. ASP.NET supplies the Page.IsCallback property to help you optimize your code within these methods. The goal is to avoid doing work that is not necessary for the Validator to do its job. For example, if you normally query the database on post back, do not do this unless that query is used by the Validator. See the example below.
  - `CallbackOrHide` – Works the same as `CallBack` except when the browser doesn't support the ASP.NET 2.0 client callback architecture, it works like the `Hide` setting.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Example of Responding to a Callback

Suppose you have an IgnoreConditionValidator that uses either OverrideClientSideEvaluation=`Callback` or `CallbackOrHide`. This is how you intercept the callback and set the Validator's **IsValid** property.

This code is within `Page_Load()` and must be able to execute every time the page loads. (In otherwords, not buried inside an IF statement.) During Page_Load, the ASP.NET page has assigned the data values of all web controls that are created as Page_Load starts. You use those web controls to determine the value of **IsValid**.

In this example, TextBox1 must contain the name "Peter" (case insensitive).

[C#]

```
if (Page.IsCallBack)
{
   // Determine if the data is valid here
   IgnoreConditionValidator1.IsValid =
      CaseInsensitiveComparer.Default.Compare(TextBox1.Text, "Peter") == 0;
}
```

[VB]

```
If Page.IsCallBack Then
' Determine if the data is valid here
   IgnoreConditionValidator1.IsValid = _
      CaseInsensitiveComparer.Default.Compare(TextBox1.Text, "Peter") = 0
End If
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Non-Data Entry Conditions

Conditions play several roles in each Validator control making them an important part of the framework.

- Each Validator control employs a Condition object to determine the rules of evaluating the data on the page.
- Each Validator control offers the **Enabler** property where a Condition object determines if the control will evaluate.
- The MultiConditionValidator builds its Boolean expressions using Conditions.
- The FieldStateController uses a Condition object to select whether the state changes are from its **ConditionFalse** or **ConditionTrue** properties. The FieldStateController offers an **Enabler** property too. See the **Peter's Interactive Pages User's Guide** for details.

The Validator controls identify the Conditions that they employ to evaluate data fields on the page. In addition, the **Enabler** property, MultiConditionValidator, and FieldStateController can evaluate non-data attributes of any HTML tag on the page using **Non-Data Entry Conditions**. Non-Data Entry Conditions answer questions like "is this field visible?", "is this field enabled", "what value does the attribute [attributename] contain?". They are particularly useful in **Enablers** so you can shut off Validators whose associated control is invisible or disabled.

Non-Data Entry Conditions can evaluate any HTML tag on the page, not just data entry controls. When referring to an HTML tag, it must have `runat=server` and an ID.

Click on any of these topics to jump to them:

- Non-Data Entry Condition Classes
- Using Non-Data Entry Conditions
- Properties for VisibleCondition
- Properties for EnabledCondition
- Properties for ClassNameCondition
- Properties for ReadOnlyCondition
- Properties for CompareToValueAttributeCondition

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Non-Data Entry Condition Classes

The following are the Non-Data Entry Conditions that look at other attributes of fields beyond their user-assign data values:

- VisibleCondition – Determines if a field is visible. See "Properties for VisibleCondition".

  Visibility is determined by the styles "visibility" and "display" (`<tagname style='visibility:hidden;display:none'/>`). When the **Visible** property on any control is `false`, the control is entirely turned off, generating no HTML. So do not use the VisibleCondition to detect it. Instead, Validators and Conditions automatically turn themselves off when the associated control has **Visible**=`false`.

  You do not have to use this in a Validator's **Enabler** property unless you have set **PeterBlum.DES.Globals.WebFormDirector.ValidationManager.AutoDisableValidators** to `false`.

- EnabledCondition – Determines if a field is enabled. See "Properties for EnabledCondition".

  The HTML attribute 'disabled' is on most form input tags: `<input>`, `<select>`, and `<textarea>`. Browsers support the 'disabled' attribute on most data entry controls. Internet Explorer for Windows provides it on many other tags. Only use it on non-data entry tags when your site is limited to IE browsers.

  You do not have to use this in a Validator's **Enabler** property unless you have set **PeterBlum.DES.Globals.WebFormDirector.ValidationManager.AutoDisableValidators** to `false`.

- ClassNameCondition – Determines if a field has a style sheet class name that matches your setting. See "Properties for ClassNameCondition".

- ReadOnlyCondition – Determines if a field is read-only. See "Properties for ReadOnlyCondition".

  The HTML attribute 'readonly' is only on TextBox controls.

- CompareToValueAttributeCondition – Compares the value of any attribute or style on the field to a value that you supply. You determine the operator for comparison. For example, attribute "tagName" equals "textarea". See "Properties for CompareToValueAttributeCondition".

For example, suppose you have a textbox with a RequiredTextValidator. However, your textbox is invisible until the user clicks on a checkbox (using DES's FieldStateController or your own JavaScript code). When the page is submitted, the invisible textbox should not be evaluated. You would use the VisibleCondition within the RequiredTextValidator's **Enabler** property to avoid problems.

```
<des:RequiredTextValidator id="RequiredTextValidator1" runat="server"
   ControlIDToEvaluate="TextBox1" ErrorMessage="This field is required">
   <EnablerContainer>
      <des:VisibleCondition Name="VisibleCondition"
         ControlIDToEvaluate="TextBox1" IsVisible="True" />
   </EnablerContainer>
</des:RequiredTextValidator>
```

*Note: The tag "<EnablerContainer>" is used by the **Enabler** property. Normally you would expect to see the tag with the same name as the property (<Enabler>). In this case, DES is working around a limitation of the .Net framework that only supports polymorphic properties by nesting them into a collection class. (EnablerContainer is a hidden property that is a collection class.)*

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Using Non-Data Entry Conditions

Non-Data Entry Conditions can be assigned to these properties: Validator.**Enabler**, MultiConditionValidator.**Conditions**, MultiCondition.**Conditions**, CountTrueConditionsValidator.**Conditions**, CountTrueConditions.**Conditions**, FieldStateController.**Condition**, and FieldStateController.**Enabler**.

Start by assigning the **ControlIDToEvaluate** property to the control whose attribute or style you are evaluating. Non-Data Entry Conditions can evaluate any HTML tag on the page, not just data entry controls. When referring to an HTML tag, it must have `runat=server` and an ID. For example, a `<span>` tag can be used if it is set up this way:

```
<span runat="server" id="id" >
```

When using VisibleCondition, set the **IsVisible** property to indicate whether the field is visible or not.

When using EnabledCondition, set the **IsEnabled** property to indicate whether the field is enabled or not.

When using ReadOnlyCondition, set the **IsReadOnly** property to indicate whether the field is read only or not.

When using ClassNameCondition, set the **ClassName** property to that style sheet class name that you want to match. If you want any class name except the name you supply, also set **NotCondition** to `true`.

When using CompareToValueAttributeCondition, set the attribute or style name in the **AttributeName** property. Set the value to compare in the **Value** property. Set the data type of the attribute with the **DataType** property. Select whether you are looking at an attribute or style with the **AttributeType** property. Finally, set the **Operator** property to compare the attribute value to the **Value** property.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Properties for VisibleCondition

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, and **NotCondition** – See "Properties Common to most Conditions".
- **IsVisible** (Boolean) – When `true`, the Condition indicates success if the control determined by **ControlIDToEvaluate** is visible. When `false`, the Condition indicates success when the control is invisible. A control can be nested within other controls that are invisible. This condition will always look through the ancestor controls before determining if it is visible. It defaults to `true`.

  On the client side, visibility is determined by the 'style.visibility ' and 'style.display' properties. The field is considered invisible only when style.visibility = "hidden" or style.display = "none".

  On the server side, visibility is determined by looking in the **Style** property for control.Style["Visibility"] = "hidden" or control.Style["Display"] = "none". (These comparisons are case insensitive.)

  When the **Visible** property on the ControlIDToEvaluate is `false`, the control is entirely turned off, generating no HTML. So do not use the VisibleCondition to detect it. Instead, Validators and Conditions automatically turn themselves off when the associated control has **Visible**=`false`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Properties for EnabledCondition

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, and **NotCondition** – See "Properties Common to most Conditions".
- **IsEnabled** (Boolean) – When `true`, the condition indicates success if the control determined by **ControlIDToEvaluate** is enabled. When `false`, the condition indicates success when the control is disabled. It defaults to `true`.

  Not all controls offer an enabled state on the client-side. Internet Explorer 6 supports this state on most of its controls. Other browsers only support it on `<input>`, `<select>`, and `<textarea>` tags. When the browser doesn't support this state, this condition cannot evaluate on the client-side. However, the server-side always compares to the **Enabled** property on the Control object.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Properties for ClassNameCondition

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, and **NotCondition** – See "Properties Common to most Conditions".
- **ClassName** (string) – Performs a case insensitive match between this name and the current name for the style sheet class used by the **ControlIDToEvaluate**. When it matches, the condition indicates success. It defaults to an empty string.

  On the client-side, it looks at the attribute 'className' on the control. On the server-side it looks at the **CssClass** property on WebControls and the **Attributes**["Class"] on HtmlControls.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Properties for ReadOnlyCondition

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, and **NotCondition** – See "Properties Common to most Conditions".

  This condition only applies to TextBox and HtmlInputText controls. All other controls selected from **ControlIDToEvaluate** cannot be evaluated.

- **IsReadOnly** (Boolean) – When `true`, the condition indicates success when the control is Read Only. When `false`, the condition indicates success when the control is not ReadOnly. It defaults to `false`.

  On the client-side, it evaluates the attribute 'readonly'. On the server side, it evaluates the property **ReadOnly** on TextBoxes and the **Attributes**["readonly"] on HtmlInputText.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Properties for CompareToValueAttributeCondition

Use the CompareToValueAttributeCondition to compare a value that you supply to the value of any attribute or style on the field identified by the **ControlIDToEvaluate** property. This is a power-users tool, letting you make decisions based on non-data attributes of fields. *You should use the preceding Non-Data Entry Conditions if they support the attribute you have in mind.*

While it can look at the values of any attribute or style, you must have some knowledge of the attributes and styles. Here are some guidelines:

- You must supply the DOM or DHTML name of the property using the correct case. JavaScript is case sensitive to the names. If the name you supply doesn't exactly match an attribute name in the field, the client-side cannot evaluate and CompareToValueAttributeCondition will act as if its disabled.

  This link is to Microsoft's online DHTML Reference: http://msdn2.microsoft.com/en-us/library/ms533050.aspx. DHTML is used by Internet Explorer for Windows and Macintosh. As you read topics on each attribute and style, note the section "Standards Information". If it indicates support from the "World Wide Web Consortium", it is supported by most DOM browsers, including FireFox, Netscape 7+, Opera 7+, Safari, and Mozilla.

- If you intend to evaluate on the server side, the **ControlToEvaluate** must have its **Attributes** or **Styles** property containing the name and value. If there is no value associated with the attribute name that you supply, the condition cannot evaluate.

- You must identify the correct data type of the attribute from this list: string, integer, or boolean. If you supply an incorrect data type, an exception may be thrown on either the client or server side. At minimum, the Condition will evaluate incorrectly.

The following list are properties specific to this Condition:

- **ControlIDToEvaluate**, **ControlToEvaluate**, and **NotCondition** – See "Properties Common to most Conditions".

- **AttributeName** (string) – The name of the attribute or style found on the control to evaluate.

  On the server side, it is used in **ControlToEvaluate.Attributes** or **ControlToEvaluate.Style** collections. If these collections do not contain the **AttributeName**, the condition will not evaluate on the server side.

  On the client side, the name must be an attribute or style of the field. Otherwise, the client-side will not evaluate the condition. Its up to you to identify the correct attribute and style names available to any browser. Browsers vary greatly, offering many specialized attributes and styles. You can programmatically detect the browser with the TrueBrowser class to determine the appropriate name.

  Browsers are case sensitive to attribute and style names. When "", the condition cannot be evaluated. It defaults to "".

- **Value** (string) – The value that is compared to in the attribute or style. While it is a string here, it is converted to the data type specified by the **DataType** property. It defaults to "". Booleans demand "true" or "false". Integers demand digits and not to be blank.

- **AttributeType** (enum PeterBlum.DES.AttributeType) – Determines whether the **AttributeName** is an attribute or style name. The enumerated type `PeterBlum.DES.AttributeType` has these values:

  - `Attribute` – This is the default.
  - `Style`

- **DataType** (enum PeterBlum.DES.AttributeDataType) – You must set this to be the data type that the DOM or DHTML has defined for the attribute. This property is used to convert the string in **AttributeValue**. If you fail to correctly set it, expect anything from incorrect evaluation to exceptions being thrown on the server side or JavaScript errors on the client side. The enumerated type `PeterBlum.DES.AttributeDataType` as these values:

  - `String` – **AttributeValue** can be any value, including blank. This is the default.
  - `Integer` – **AttributeValue** must contain digits only. It cannot be blank.
  - `Boolean` – **AttributeValue** must contain 'false' or 'true'. It cannot be blank.

- **Operator** (enum PeterBlum.DES.ConditionOperator) – Determines how the attribute or style is compared to **Value**. The enumerated type `PeterBlum.DES.ConditionOperator` has the following values:

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- `Equal` (This is the default.) ControlIDToEvaluate = Value
- `NotEqual` ControlIDToEvaluate <> Value
- `GreaterThan` ControlIDToEvaluate > Value
- `GreaterThanEqual` ControlIDToEvaluate >= Value
- `LessThan` ControlIDToEvaluate < Value
- `LessThanEqual` ControlIDToEvaluate <= Value

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Extending Existing Validators and Conditions

Sometimes you need to take advantage of a Validator or Condition class but modify its evaluation function. For example, you want to use the SelectedIndexCondition, but always report "success" if the textual value of the selection contains a specific phrase if the original evaluation method reports "failure".

There are three approaches you can use:

- Subclass the object. This is the most powerful approach, allowing you to add new controls to the Visual Studio.net toolbox, complete with new properties. Please see the Developer's Kit for details.
- Combine a CustomCondition with existing Conditions using the MultiConditionValidator. This is the most common approach since the MultiConditionValidator is so powerful. Design a Boolean expression consisting of Conditions from various Validators. Create your own CustomCondition with the evaluation function for anything not already defined in DES. For example, you want a Validator that requires text in a textbox and verifies its contents against a database. You would use the logic: RequiredTextCondition AND CustomCondition, where the CustomCondition handles the database query. See "MultiConditionValidator" and "CustomValidator".
- Replace the evaluation functions in any existing Validator or Condition, potentially calling the original evaluation function from your own. This technique is the topic of this section.

The CustomValidator and CustomCondition classes use the properties **ServerCondition** and **CustomEvalFunctionName** to define their evaluation functions, on server and client-side respectively. These same properties are found on all Validator and Condition classes, allowing you to treat them all as the basis for a custom Validator. This technique has two advantages over CustomValidators and CustomConditions:

- You can continue to use the original evaluation function supplied by the class, simply by calling it from within your own evaluation function.
- You can take advantage of the properties found on the class. For example, suppose your evaluation function will evaluate the **Checked** property of a checkbox. If you start with the CheckedStateValidator, it offers this property. Now your evaluation function can develop its own logic with access to the user's setting for **Checked** and **ControlIDToEvaluate**.

Click on any of these topics to jump to them:

- Extending the Server Side
- Extending The Client-Side

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Extending the Server Side

Write your server-side evaluation method and attach it to the **ServerCondition** delegate. See "Server-Side Condition" for setting up that method, taking into account the following details about the second parameter.

The second parameter passed to your evaluation method is one of these types:

- PeterBlum.DES.Web.WebControls.ConditionEventArgs – Includes **IsMatch**, **CannotEvaluate**, and **Validator** properties. Passed when there is no **ControlIDToEvaluate** property on your Validator.
- PeterBlum.DES.Web.WebControls.ConditionOneFieldEventArgs – Includes **IsMatch**, **CannotEvaluate**, **Validator**, **Value**, **ValueAsNative**, and **ControlToEvaluate** properties. Passed when there is a **ControlIDToEvaluate** property but no **SecondIDToEvaluate** property on your Validator.
- PeterBlum.DES.Web.WebControls.ConditionTwoFieldEventArgs – Includes **IsMatch**, **CannotEvaluate**, **Validator**, **Value**, **ControlToEvaluate**, **SecondValue**, **SecondValueAsNative**, and **SecondControlToEvaluate** properties. Passed when there are **ControlIDToEvaluate** and **SecondIDToEvaluate** properties on your Validator.

If you want to call the original evaluation method from your method, call the condition's `EvaluateCondition()` method within your event handler.

The `EvaluateCondition()` method takes no parameters. It returns an integer with one of these values:

- 1 – Evaluated successfully
- 0 – Failed evaluation (error was detected)
- -1 – Cannot evaluate. The data was not set up in a way that the method could evaluate it at all.

This design allows you to call `EvaluateCondition()` in any order with your own condition logic: before, within, and after. It also lets you take action based on either success or failure results. If you want to impose further limits on the evaluation rules of the condition, test for 1 (success) and perform your own rules. If you want to loosen the limits of the evaluation rules, test for 0 (failed) and perform your own rules on the same data.

This is an example where the `EvaluateCondition()` method is called before the custom logic. The custom logic is only applied when the method returns 1 (success):

[C#]

```
using PeterBlum.DES;
…
 protected void MyCondition(IBaseCondition sourceCondition,
    IConditionEventArgs args)
 {
    switch (sourceCondition.EvaluateCondition())
    {
       case 1:  // success
       // your code goes here. Set args.IsMatch
          break;
       case 0:  // failed
          args.IsMatch = false;
          break;
       case -1: // cannot evaluate
          args.CannotEvaluate = true;
          break;
    }
 }  // MyCondition
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

[VB]

```
Imports PeterBlum.DES;
…
 Protected Sub MyCondition( _
    ByVal sourceCondition As IBaseCondition, _
    ByVal args As IConditionEventArgs)
    Select sourceCondition.EvaluateCondition()
       Case 1   ' success
          ' your code goes here. Set args.IsMatch
       Case 0   ' failed
          args.IsMatch = False
       Case -1  ' cannot evaluate
          args.CannotEvaluate = True
    End Select
 End Sub  ' MyCondition
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Extending The Client-Side

Write your client-side evaluation function in JavaScript and assign its name to the **CustomEvalFunctionName** property. See "The Client-Side Evaluation Function" for details.

Your function is passed one parameter, *cond*, which is JavaScript object representing the Condition. The *cond* parameter will have the property **IDToEval** only when the Validator has the **ControlIDToEvaluate** property. The **IDToEval2** property will only be available when the Validator has the **SecondControlIDToEvaluate** property.

If your function needs to call the original client-side evaluation function, you need to know the name of the original function. It is found in the Condition's `GetClientSideEvalFunctionName()` method. It is easiest if you write your client-side code on the server side by building a string that you output using `Page.RegisterClientScriptBlock()`. The function returned by `GetClientSideEvalFunctionName()` works exactly the way your own client-side evaluation function works: it takes the *cond* parameter and returns 1 for success, 0 for failed, and -1 for cannot evaluate.

This example shows script created as a StringBuilder object and output. Its logic first calls the parent's evaluation function. If it returns 1 (for success), it continues with your own logic.

[C#]

```
System.Text.StringBuilder vScript = new System.Text.StringBuilder(500);
vScript.Append("<script type='text/javascript' language='JavaScript'><!--");
vScript.Append("function MyCond(cond)");
vScript.Append("{");
vScript.Append("var vResult = ");
vScript.Append(base.GetClientSideEvalFunctionName());
vScript.Append("(cond);");
vScript.Append("if (vResult == 1)");
vScript.Append("do your own evaluation here, setting vResult");
vScript.Append("return vResult;");
vScript.Append("}");
vScript.Append("-->");
vScript.Append("</script>");
Page.RegisterClientScriptBlock("MyCond", vScript.ToString());
```

[VB]

```
Dim vScript As System.Text.StringBuilder = New System.Text.StringBuilder(500)
vScript.Append("<script type='text/javascript' language='JavaScript'><!--")
vScript.Append("function MyCond(cond)")
vScript.Append("{")
vScript.Append("var vResult = ")
vScript.Append(base.GetClientSideEvalFunctionName())
vScript.Append("(cond);")
vScript.Append("if (vResult == 1)")
vScript.Append("do your own evaluation here, setting vResult")
vScript.Append("return vResult;")
vScript.Append("}")
vScript.Append("-->")
vScript.Append("</script>")
Page.RegisterClientScriptBlock("MyCond", vScript.ToString())
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Changing When the Validator is Evaluated

Validators normally evaluate their condition and show the error when you change the field or attempt to submit the page. Error notification occurs at two points:

- When a field is changed. This happens on the client side, using JavaScript, so that the user doesn't have to submit the form to see the error. When the field is textual, a listbox or dropdownlist, validation is performed as the focus leaves the field. When the field is a checkbox or radiobutton, validation is performed as soon as the user changes the mark in the button.

  Client-side validation is offered on browsers that support DOM or DHTML well, including Internet Explorer 4+ for Windows, FireFox for Windows and Mac, Safari for Windows and Mac, Opera 7+ for Windows and Mac, Internet Explorer 5 for Macintosh, Netscape 6+ for Windows and Mac, and browsers created from Mozilla. Other browsers require the server to detect and display errors. In addition, you can elect to turn off client-side validation on individual Validator controls if their validation logic requires functionality only found on the server side, such as a database lookup.

  DES will automatically detect if JavaScript is disabled on a browser that supports JavaScript. When it does, all further page requests will have the server-side only user interface. This makes for a much cleaner experience. JavaScript is detected to be disabled after the first post back on any page using DES controls.

- When the page is submitted. This action asks all Validator controls to validate the fields on the page. When one or more report an error, the page is redisplayed with errors and the server side should not attempt to use the data from the page. Submission also has a client-side validation process that can update the page without posting back to the server. If the client-side validation is disabled (because you disabled it or the browser doesn't support it), the page will be posted to the server and server-side validation code will run.

  You can separate Validators into groups where a particular **submit control** will only validate those controls within the group. For example, suppose your page includes a login area and a survey form, each with its own Validators and submit buttons. You assign the group names "login" and "survey" to the respective controls (including the buttons). When the user submits, validation is limited to the group of the button.

There are a number of circumstances where you don't want the Validator to evaluate. They are:

- Use **Validation Groups** to make a submit control validate a specific list of Validators. This is often used on pages with multiple submit buttons that server different purposes.
- Use the **Enabler Property** when one field's state affects whether the Validator evaluates its data or not. It intelligently enables the Validator depending on conditions of the page.
- The Validator control includes numerous properties that disable validation in special situations: **Enabled**, **EnableClientScript**, **EventsThatValidate**, **EvaluateOnClickOrChange**, and **SupportClientSideLookupByID**.

Click on any of these topics to jump to them:

- Validation Groups
- The Enabler Property
- Other Properties That Disable Validation

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Validation Groups

Online example

Often, your pages will include separate sections, each with its own data entry fields and their Validators. Usually each of these sections has its own submit control. When it's clicked, you don't want Validators outside of the section to be evaluated. DES allows you to define a **Group name** on each submit control and the associated controls it acts upon: Validators and ValidationSummary. When the user hits a submit control, it will only evaluate Validators that share the same group name. The ValidationSummary will only appear when its group name matches. This behavior is called **validation groups**.

Using Validation Groups

Set the group name on the **Group** property, which is found on Validator controls, the ValidationSummary control and all of the submit controls supplied with DES. In fact, if you leave the **Group** property empty, the Submit control is still matching its group name (empty) to all Validators with the same name.

When you use the validation group feature, it is easiest of you use DES's submit controls because they include a **Group** property. However, you can continue using Microsoft's controls to submit a page by assigning a NativeControlExtender to them. The NativeControlExtender provides a **Group** property that supports these features.. See "Using Native Button Controls to Submit the Page" for details.

Special Characters In Group Names

Group names are normally terms that are meaningful to the design of the page, such as "Login" and "Panel1". They should not include any of these characters: pipe (|), plus (+) or asterisk (*). Those characters have special meanings. They are:

- The asterisk (*) means match to all groups. It can be used on a submit control to run all Validators on the page. It can be used on the ValidationSummary to display Validators from any group. It cannot be used in the Validator control.
- The pipe (|) is a delimiter for defining a list of groups. Each group name is separated by a pipe (|). Validators and ValidationSummary support it; submit controls do not. They will match to any group name on the submit control that matches to the list. For example, "Group1|Group2" will match submit controls whose group is "Group1" or "Group2". If you are using an empty string in a Group property and want it to be in the list, simply define your list with the | as the first character, like this: "|Group2" (matches empty string and "Group2").
- The plus (+) helps when Validators and submit controls are in a row of a grid or Repeater. It also sometimes applies to UserControls. When using a grid or Repeater, the same row is repeated. If that row contains the same Validator, each row's Validator will share the same **Group** property name. That means when the user attempts to submit a single row, it will validate all groups. The hack around this is to rename the **Group** property with the naming container's UniqueID as the row is created. That gives each row a separate group name.

  Similarly, if you are using the same UserControl multiple times on the page, it reuses the same validators. If each should be its own group name, use the + character in the **Group** property of each validator and button on that User Control.

  The plus character simplifies the process. Add a plus character as the first character of the group name. It will automatically create a group name of NamingContainer.UniqueID + Group.

  This applies to Validators, submit controls, and ValidationSummary. The string in the **Group** property is automatically changed as each of these controls are added to the control tree (the OnLoad phase).

Special Character Support By Control Type			
Control type	*	|	+
Validators		√	√
ValidationSummary	√	√	√
Buttons (submit controls)	√		√

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

The Enabler Property

Online example

There are times when a Validator control should be disabled. For example, don't validate a textbox because an associated checkbox is unmarked or a DropDownList does not have the right value. Use the **Enabler** property on the Validator control to set up rules to enable and disable it. The rules are actually Conditions, like those inside of each Validator and used in the MultiConditionValidator. See "Evaluation rules for Validators: Condition".

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. Often you set the **NotCondition** property to `true` to reverse the logic of these conditions. For a list of available conditions, see "**Error! Reference source not found.**" and "Non-Data Entry Condition Classes".

Some commonly used Conditions are:

- RequiredTextCondition – From the RequiredTextValidator, it determines if another textbox has text.
- CheckStateCondition – From the CheckStateValidator, it determines the mark in a checkbox or radiobutton.
- SelectedIndexCondition – From the SelectedIndexValidator, it determines the SelectedIndex on a list-type control.
- VisibleCondition – A Non Data Entry Condition that determines if a field is visible. See "Non-Data Entry Conditions".
- MultiCondition – From the MultiConditionValidator, use it to build Boolean logic from all available Conditions.

Be aware of these issues when using the **Enabler** property:

- By default, Validators will not evaluate when the associated data entry control is hidden or disabled. If you need to allow validation when hidden or disabled, you must set the page-level property **AutoDisableValidators** to `false`. Then assign the Enabler to the appropriate conditions: VisibleCondition, EnabledCondition, or a MultiCondition that contains both.
- Most Conditions have a property called **EvaluateOnClickOrChange** which defaults to `true`. Change it to `false` when using it in an Enabler. If its left `true`, the control it evaluates will cause the validator to evaluate and it will have its appearance changed when there is an error and the page-level property **ChangeStyleOnControlsWithError** is used.
- 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 Condition's **Enabled** property to false when the control is **Visible**=`false`.

Click on any of these topics to jump to them:

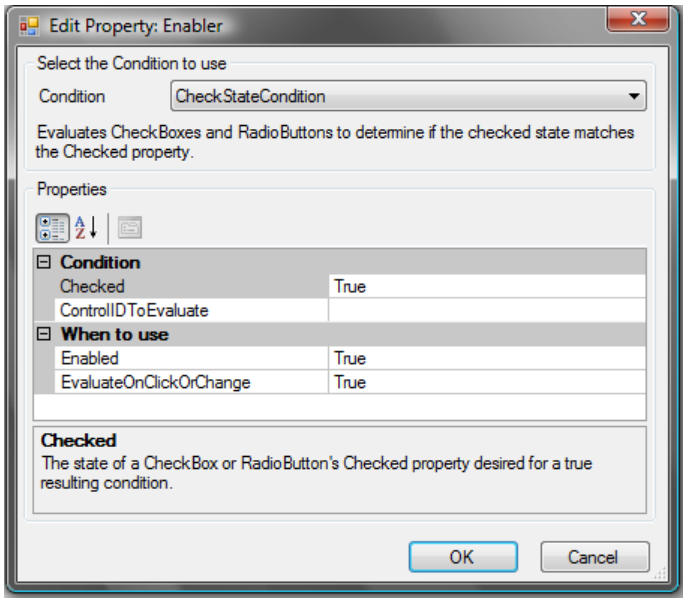
- Adding the Enabler in Design Mode
- Adding an Enabler with ASP.NET Markup
- Adding an Enabler Programmatically

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Adding the Enabler in Design Mode

The Properties Editor offers the window shown below to select a Condition and to edit its properties.



1. Select the Condition from the DropDownList. For a description of conditions, see "**Error! Reference source not found.**" and "Non-Data Entry Condition Classes".
2. Establish the properties in the Properties grid.

   *Reminder: Be sure to set **EvaluateOnClickOrChange** to false on this Condition and any of its child Conditions.*
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 unusual, in part because the .Net framework doesn't support changing the class of a property (polymorphism) without an interesting hack.

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

```
<des:RequiredTextValidator id="RequiredTextValidator1" runat="server"
   ControlIDToEvaluate="TextBox1" ErrorMessage="This field is required" >
   <EnablerContainer>
      <des:CheckStateCondition ControlIDToEvaluate="CheckBox1"
         EvaluateOnClickOrChange="false" />
   </EnablerContainer>
</des:RequiredTextValidator>
```

*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:RequiredTextValidator>` tag. (*It will be added when using the Property Editor but its completely meaningless.*) Instead, the `<EnablerContainer>` tag is a child of the Validator control tag. That tag has no attributes. The child to `<EnablerContainer>` 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.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Adding an Enabler Programmatically

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

1. Create an instance of the desired Condition. There is a constructor that takes no parameters. For a list of conditions, see "**Error! Reference source not found.**" and "Non-Data Entry Condition Classes".

   Note: There are also constructors that take parameters. Each demands an "owner" in the first parameter. That value must be the Validator 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 Validator1.

[C#]

```
PeterBlum.DES.Web.WebControls.CheckStateCondition vCond =
   new PeterBlum.DES.Web.WebControls.CheckStateCondition();
vCond.ControlToEvaluate = CheckBox1;
vCond.EvaluateOnClickOrChange = false;
Validator1.Enabler = vCond;
```

[VB]

```
Dim vCond As PeterBlum.DES.Web.WebControls.CheckStateCondition = _
   New PeterBlum.DES.Web.WebControls.CheckStateCondition()
vCond.ControlToEvaluate = CheckBox1
vCond.EvaluateOnClickOrChange = False
Validator1.Enabler = vCond
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Other Properties That Disable Validation

The following properties are on each Validator control and Condition except where noticed. Within the Properties Editor, they are in the "Behavior" category.

- **Enabled** (Boolean) – When `false`, the Validator is entirely disabled. It will never validate nor generate any HTML into the page. Generally you use this when you have a Validator control on the page which you programmatically need to enable and disable based on other conditions. It defaults to `true`.

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

  You can set the **Enabled** property on the client side using the javascript function `DES_SetEnabled()`, published in the DES scripts. See "Changing The Properties of a Validator With Client-Side Scripts".

- **EnableClientScript** (Boolean) - Determines if client-side validation runs.

  *ALERT: You are strongly recommended to use **OverrideClientSideEvaluation** instead of **EnableClientScript**. **EnableClientScript** turns off ALL client-side features, including the ability to use some ErrorFormatters that depend on javascript. **OverrideClientSideEvaluation** only disables the evaluation aspect of the Validator control.*

  When `true`, it sets up client-side scripting for this control to validate. When `false`, this control never validates on the client-side. This property doesn't prevent the server side validation from occurring. It defaults to `true`.

  If you set up a Validator whose validation data you don't want to expose to the user, such as comparing to password, set this to `false`. Otherwise, the page will contain the private text and the user can simply view the HTML of the page to find it.

  When you set up a CustomValidator, you may only be able to supply a server-side evaluation function (perhaps you desire functionality that is only available at the server.) When **CustomValidator.CustomEvalFunctionName** is blank, it will automatically act as if **EnableClientScript** is `false`.

  Some browsers do not support DES's JavaScript code. DES automatically sets this to `false` when it detects such a browser. Some browsers have turned off their JavaScript code. DES detects this and disables client-side validation. See "PeterBlum.DES.Globals.WebFormDirector.JavaScriptEnabled property".

  *ALERT: **Always** set up server-side validation by testing **PeterBlum.DES.Globals.WebFormDirector.IsValid** is `true` in your post back event handler before using the data. See "Properties and Methods to Validate the Page".*

- **EventsThatValidate** (enum PeterBlum.DES.ValidationEvents) - Determines which client-side events are fired to validate. There are two events: when the field is changed and when the page is submitted. By default, both fire to validate. Here are the values of the `PeterBlum.DES.ValidationEvents` enumerated type:

  - `All` – Fire both events. This is the default.
  - `OnSubmit` – Only fire when the page is submitted. Use this when you want error messages to wait until the page is submitted or the validation logic requires so many fields to be set that it doesn't make sense after each field is changed.

    For example, if you have 3 TextBoxes where a MultiConditionValidator reports an error if any are blank, as the user changes the first textbox, the MultiConditionValidator will report an error because the next two are blank.
  - `OnChange` – Only fire when the field is changed.

    Note: When using OnChange, the error message will not appear in the ValidationSummary. It will also be removed when the page is submitted.
  - `WarningMode` - This option works well as a **warning message** that appears once the user has changed the field. When they submit, the Validator may report an error but will not stop postback. On the server-side, the condition always indicates success. (It never sets **PeterBlum.DES.Globals.WebFormDirector.IsValid** to `false`.)

    A warning message communicates a condition that is likely to be an error although it is still valid. For example, you accept the quantities above 1000 for a Quantity field although such an entry may be an error. Your Validator's error message could read: "Are you sure that this quantity should be {TEXTVALUE}?."

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

You can further assist the user by showing a confirmation message with the current warnings listed as they attempt to submit the page. The message lets them know of problems and optionally cancel submitting the page. See "Warning Mode".

- **EvaluateOnClickOrChange** (Boolean) – This property is only on Condition classes. When `true`, the Condition will hookup the client-side onclick or onchange event handlers of any controls to evaluate. When a user clicks or changes the control to evaluate, it will run the Validator. When `false`, clicks and edits do not run validation. It defaults to `true`.

  This can be used in several ways:

  - If you have multiple controls in the condition and want to prevent firing them the user has edited the last one, set this to false on any condition that includes the initial controls and set it to true on a condition that only involves the last control. For example, suppose you have two TextBoxes that use a CompareTwoFieldValidator and RequiredTextValidators. Set the CompareTwoFieldValidator.EvaluateOnClickOrChange = `false`; the first RequiredTextValidator.EvaluateOnClickOrChange = `false`; and the last RequiredTextValidator.EvaluateOnClickOrChange = `true`.
  - When using the **Enabler** property, its conditions often should not cause validation to run. Always set this to `false`.
  - When a Condition is used in a FieldStateController, it allows a master control to run it. All conditions within the FieldStateController can set this to `false`.

- **SupportClientSideLookupByID** (Boolean) - Allows JavaScript programmers to get to the client-side representation of the Validator object by the ClientID of the owner control. See "Invoking Validation With Client-Side Scripts" and "Changing The Properties of a Validator With Client-Side Scripts".

  Use the client-side function `DES_FindAOById()` 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 modify it, such as changing the **Enabled** property. When **SupportClientSideLookupByID** is `true`, two things happen:

  - 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.)
  - 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. Note that client-side code is *never* generated when the browser doesn't support it or the **EnableClientScript** property is `false`.

  It defaults to `false`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Defining the Error Message and Associated Labels

The Error Message communicates the problem found by the condition to the user. There are actually two error messages that you can define per Validator. One appears directly on the Validator control through its **ErrorFormatter** property (see "ErrorFormatters: Customizing the Appearance of the Error Message"). The other appears in the ValidationSummary control. This allows you to provide a different level of details on each message, perhaps keeping the message next to the field shorter.

You set the **ErrorMessage** property on each Validator control to establish its error message. If you have a ValidationSummary and want a different error message, also set the **SummaryErrorMessage** property. The ValidationSummary uses the **ErrorMessage** property when **SummaryErrorMessage** is blank.

See "Properties for Error Messages".

```
<des:RequiredTextValidator id="RequiredTextValidator1" runat="server"
   ControlIDToEvaluate="ProductName" ErrorMessage="This field is required"
   SummaryErrorMessage="Product Name is required" />
```

Error messages support tokens – terms embedded into brackets – which get replaced by properties or runtime values determined when the condition is evaluated. This allows you to build standard messages where the programmer selects the right message for the situation, complete with tokens, and it applies the properties and runtime values automatically. By having runtime values, you can provide the user with specifics about what is wrong with their entry, such as the count of the number of characters they typed and how much that exceeds your limits. There are tokens to show the labels of the controls. They use the **Label**, **SecondLabel**, **Label2**, **Label3** and **Label4** properties to determine the text of the label.

See "Tokens in Error Messages" and "Applying Styles To Tokens".

```
<des:RequiredTextValidator id="RequiredTextValidator1" runat="server"
   ControlIDToEvaluate="ProductName" ErrorMessage="This field is required"
   SummaryErrorMessage="{LABEL} is required"
   Label-LabelControlID="ProductNameLabel" />
```

The **ErrorMessage** and **SummaryErrorMessage** properties support the String Lookup System where enter your strings are in a .resx file, database or other data source. Use it if you want to localize strings or develop a standardized collection of messages. For example, you reuse the RequiredTextValidator with the message "This field requires a value." If you put it into the String Lookup System, programmers will no longer enter that string. They just enter a "Lookup ID". That guarantees that the exact same error message appears throughout the site. It also allows one person to maintain the actual text and update all cases at once. See "String Lookup System" in the **General Features Guide**.

Click on any of these topics to jump to them:

- Properties for Error Messages
- Tokens in Error Messages
- Applying Styles To Tokens
- Online example

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Properties for Error Messages and Associated Labels

The Properties Editor includes a section titled "Error Messages" containing all of these properties.

- **ErrorMessage** (string) – The text that appears in the Validator control's **ErrorFormatter**. If there is a ValidationSummary control and the **SummaryErrorMessage** property is blank, this will be used in the ValidationSummary as well.

  Error messages can contain HTML tags. HTML lets you customize the appearance of the error message. However, remember that when you set them up, they can appear in the ValidationSummary, tooltips, and alert boxes. You may want different formatting the ValidationSummary control. Tooltips and alert boxes do not support HTML. The **ErrorFormatter** property has a rich set of tools to customize the appearance of the **ErrorMessage** without embedding HTML. You are strongly encouraged to use ErrorFormatters instead of embedded HTML.
  See "ErrorFormatters: Customizing the Appearance of the Error Message".

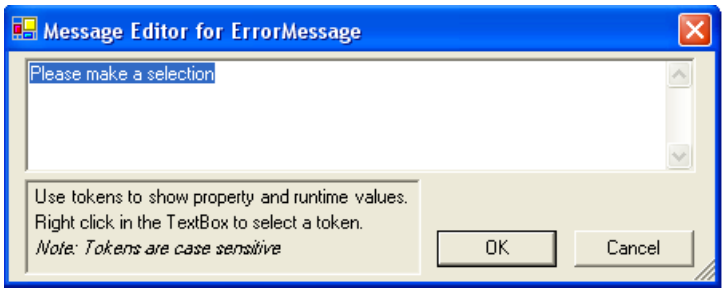
  Note: If you want to show the characters "<", ">" and "&", be sure to type in "<", ">" and "&". Alerts will automatically strip out any HTML tags.

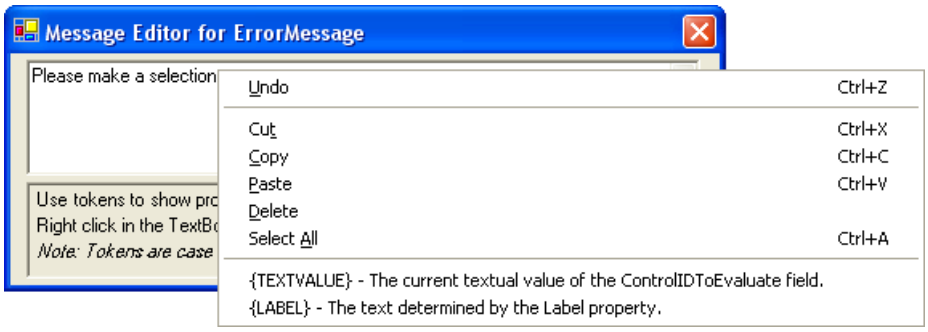
  Reminder: If your page should follow the XHTML standard, make sure your tags conform to XHTML.

  Each Validator class supports some tokens, that are replaced at runtime, to allow standardized messages with variations in the tokens or live data like the current text length in the TextLengthValidator's **ErrorMessage**.
  See "Tokens in Error Messages".

  *Note: If you want to apply styles to any of the tokens in the **ErrorMessage**, use these properties on the PeterBlum.DES.Globals.WebFormDirector.ValidationManager object: LabelTokenCssClass, PropertyTokenCssClass and RuntimeTokenCssClass. See "Applying Styles To Tokens" for details.*

  The Properties Editor includes an editor on this property that gives you plenty of space to enter your message and offers a way to quickly insert tokens from the context menu when you right-click on the textbox.





- **ErrorMessageLookupID** (string) – Gets the value for **ErrorMessage** 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 ErrorMessage. If no match is found OR this is blank, **ErrorMessage** 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 "".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- **SummaryErrorMessage** (string) – Supplies the error message for the ValidationSummary control and the alert shown when using **PeterBlum.DES.Globals.WebFormDirector.ValidationManager.ShowAlertOnSubmit** and **ShowAlertAfterPostBackError**. You can leave it blank if the value of **ErrorMessage** is the message you want in the ValidationSummary and alert.

  SummaryErrorMessage supports the same tokens and editor in the Properties Editor as **ErrorMessage** does.

  *Note: The **SummaryErrorMessage** property accepts HTML tags. If you want to show the characters "<", ">" and "&", be sure to type in "<", ">" and "&". Alerts will automatically strip out any HTML tags.*

  Reminder: If your page should follow the XHTML standard, make sure your tags conform to XHTML.

  *Note: If you want to apply styles to any of the tokens in the **SummaryErrorMessage**, use these properties on the PeterBlum.DES.Globals.WebFormDirector.ValidationManager object: SummaryLabelTokenCssClass, SummaryPropertyTokenCssClass and SummaryRuntimeTokenCssClass. See "Applying Styles To Tokens" for details.*

- **SummaryErrorMessageLookupID** (string) – Gets the value for **SummaryErrorMessage** 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 ErrorMessage. If no match is found OR this is blank, **SummaryErrorMessage** 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 "".

- **Label** (PeterBlum.DES.Web.WebControls.LabelText) – Provides text for the "{LABEL}" token that can appear in **ErrorMessage** and **SummaryErrorMessage** properties. Use it to separate the label from your standardized and localized error messages. This lets you use the same error message for differently named fields.

  See "Properties of the PeterBlum.DES.Web.WebControls.LabelText Class".

  The **Label** property can also cause the Label control to change its style when an error is shown. Simply set either the **LabelControlID** or **LabelControl** properties and set up the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager.HiliteFieldsNearbyError** property.

  When defining properties of any Label property into ASP.NET, embed each within the beginning tag using the style "LabelName-propertyname". For example:

```
<des:CompareTwoFieldsValidator id="CompVal1" runat="server"
   Label-Text="Start Date" SecondLabel-LabelControlID="EndDateLabel"
   ControlIDToEvaluate="DateTextBox1" ControlIDToEvaluate="DateTextBox2"/>
```

- **SecondLabel** (PeterBlum.DES.Web.WebControls.LabelText) – Provides text for the "{LABEL2}" token that can appear in **ErrorMessage** and **SummaryErrorMessage** properties. This property appears on controls with a **SecondControlIDToEvaluate** property and should be used as a label for the control assigned to **SecondControlIDToEvaluate**.

  See "Properties of the PeterBlum.DES.Web.WebControls.LabelText Class".

  The **SecondLabel** property can also cause the Label control to change its style when an error is shown. Simply set either the **LabelControlID** or **LabelControl** properties and set up the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager.HiliteFieldsNearbyError** property.

  When defining properties of any Label property into ASP.NET, embed each within the beginning tag using the style "LabelName-propertyname". For example:

```
<des:CompareTwoFieldsValidator id="CompVal1" runat="server"
   Label-Text="Start Date" SecondLabel-LabelControlID="EndDateLabel"
   ControlIDToEvaluate="DateTextBox1" ControlIDToEvaluate="DateTextBox2"/>
```

- **Label2**, **Label3**, and **Label4** (PeterBlum.DES.Web.WebControls.LabelText) – Each of these is associated with the tokens "{LABEL2}", "{LABEL3}", and "{LABEL4}" that you can insert into the **ErrorMessage** and

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

SummaryErrorMessage on MultiConditionValidator and CountTrueConditionsValidator. These two Validators do allow you to creatively define the fields that they evaluate. So four Label properties are available to define up to four tokens. See the **Label** property above for details on the `PeterBlum.DES.Web.WebControls.LabelText` class.

See "Properties of the PeterBlum.DES.Web.WebControls.LabelText Class".

The **Label** property can also cause the Label control to change its style when an error is shown. Simply set either the **LabelControlID** or **LabelControl** properties and set up the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager.HiliteFieldsNearbyError** property.

When defining properties of any Label property into ASP.NET, embed each within the beginning tag using the style "LabelName-propertyname". For example:

```
<des:MultiConditionValidator id="MSVal1" runat="server"
   Label-Text="First name" Label2-LabelControlID="LastNameLabel"
   [other properties] >
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Properties of the PeterBlum.DES.Web.WebControls.LabelText Class

The `PeterBlum.DES.Web.WebControls.LabelText` class provides several possible sources for the text and can alter the format by changing case and stripping trailing characters like colon.

- **LabelControlID** (string) – If you have a `System.Web.UI.WebControl.Label` or DES's LocalizableLabel control on the page with the text that you want to show in the {LABEL} token, assign its ID here. You will be able to use the **Case** and **TrimTrailingSpaces** properties to perform minor modifications to the text from this control. Otherwise, consider using the **Text** property. It defaults to "".
- **LabelControl** (System.Web.UI.Control) – If you have a Label or LocalizableLabel control with the desired text but its not in the same or an ancestor naming container, assign it to this property programmatically.

  When programmatically assigning properties to a Validator control, if you have access to the Label control object, it is better to assign it here than assign its ID to the **LabelControlID** property because DES operates faster using **LabelControl**.
- **Text** (string) – If you don't have the desired text available from a Label or LocalizableLabel control on the page, enter the text here.
- **TextLookupID** (string) – Gets the value for **Text** 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 Labels. If no match is found OR this is blank, **Text** 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 "".
- **Case** (enum PeterBlum.DES.CaseConversion) – When the **LabelControlID** or **LabelControl** property is used, the text may be correct except for the case. You can use this to adjust the case in these ways:
  - `None` – no conversion. This is the default.
  - `Lowercase`
  - `Uppercase`
  - `SentenceCase` – Only the first letter is capitalized
  - `TitleCase` – The first letter of each word is capitalized
- **TrimTrailingSymbol** (Boolean) – When the **LabelControlID** or **LabelControl** property is used, the text many be correct except for trailing spaces or a colon. When this is `true`, it removes all non-alphanumeric characters appearing at the end of the text. For example, "Birthdate:" is converted to "Birthdate". It defaults to `false`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Tokens in Error Messages

For a demo, see http://www.peterblum.com/DES/DemoTokens.aspx.

Here are the tokens that can be entered into the **ErrorMessage** and **SummaryErrorMessage** properties.

Note: All tokens are case sensitive.

Click on any of these topics to jump to them:

- {NEWLINE}
- {LABEL}, {LABEL2}, {LABEL3}, and {LABEL4}
- {TEXTVALUE}, and {TEXTVALUE2}
- {MINIMUM} and {MAXIMUM}
- {VALUETOCOMPARE}
- {DIFFERENCEVALUE} and {DIFFRESULT}
- {COUNT} and {COUNT:singular:plural}
- {EXCEEDS} and {EXCEEDS:singular:plural}
- {UNWANTED}

{NEWLINE}

Supporting Validators: All

This is a special token that is replaced by the appropriate character or symbol to represent a linebreak. The same text from the **ErrorMessage** and **SummaryErrorMessages** are used in several locations: the web page, alerts, and tooltips. The {NEWLINE} token lets the web page use a "<br />" while alerts and tooltips use specific ASCII characters. *Note: Only Internet Explorer permits linebreaks in it's tooltip. This token is replaced by a SPACE character on other browsers.*

It is valid to use the HTML tag "<br />" directly in these properties. However, when used in an alert or tooltip, its simply stripped out. The {NEWLINE} token assures that a linebreak will be used if possible.

Example

1. The date is incorrect{NEWLINE}2. The date must be later than July 1, 2003.

{LABEL}

*Supporting Validators: All that use the **ControlIDToEvaluate** property and DuplicateEntryValidator*

From: Server-side property

Uses the text from the **Label** property.

Example

The {LABEL} contains an invalid date.

{LABEL2}

*Supporting Validators: All that use the **SecondControlIDToEvaluate** and **Label2** property. Also DuplicateEntryValidator*

From: Server-side property

Uses the text from the **SecondLabel** or **Label2** property.

Example

{LABEL1} cannot differ from {LABEL2} by more than 100.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

{LABEL3}

*Supporting Validators: All that use the **Label3** property.*

From: Server-side property

Uses the text from the **Label3** property.

Example

{LABEL3} contains an error.

{LABEL4}

*Supporting Validators: All that use the **Label4** property.*

From: Server-side property

Uses the text from the **Label4** property.

Example

{LABEL4} has an invalid format.

{TEXTVALUE}

*Supporting Validators: All that use the **ControlIDToEvaluate** property.*

From: Runtime property

Retrieves the textual value of the control specified by **ControlIDToEvaluate** when the control is evaluated.

Example

The date you entered, {TEXTVALUE}, cannot be earlier than January 1, 2003.

{TEXTVALUE2}

*Supporting Validators: All that use the **SecondControlIDToEvaluate** property.*

From: Runtime property

Retrieves the textual value of the control specified by **SecondControlIDToEvaluate** when the control is evaluated.

Example

The value {TEXTVALUE2} must be less than {TEXTVALUE}.

{MINIMUM}

Supporting Validators: RangeValidator, TextLengthValidator, WordCountValidator, CountSelectionsValidator, and CountTrueConditionsValidator.

From: Server-side property

Uses the value of the **Minimum** property.

Example

Please do not enter a value lower than {MINIMUM}.

{MAXIMUM}

Supporting Validators: RangeValidator, TextLengthValidator, WordCountValidator, CountSelectionsValidator, and CountTrueConditionsValidator.

From: Server-side property

Uses the value of the **Maximum** property.

Example

You have exceeded the limit of {MAXIMUM} checkmarks.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

{VALUETOCOMPARE}

Supporting Validators: CompareToValueValidator

From: Server-side property

Use the value of the **ValueToCompare** property.

Example

{TEXTVALUE} exceeds {VALUETOCOMPARE}. Please reduce this value.

{DIFFERENCEVALUE}

Supporting Validators: DifferenceValidator

From: Server-side property

Uses the value of the **DifferenceValue** property.

Example

These dates should be no more than {DIFFERENCEVALUE} days apart.

{DIFFRESULT}

Supporting Validators: DifferenceValidator

From: Runtime property

Uses the difference calculated by the condition. It is a positive integer or decimal number.

Example

These dates should be no more than {DIFFERENCEVALUE} days apart. They are {DIFFRESULT} days apart now.

{COUNT}

Supporting Validators: TextLengthValidator, WordCountValidator, CountSelectionsValidator, and CountTrueConditionsValidator.

From: Runtime property

Uses the count determined by the condition: number of characters, words, selections, or conditions, depending on the Validator.

Example

You entered {COUNT} characters.

{COUNT:singular:plural}

Supporting Validators: TextLengthValidator, WordCountValidator, CountSelectionsValidator, and CountTrueConditionsValidator.

From: Runtime property

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.

Example

You entered {COUNT} {COUNT:character:characters}.

You entered {COUNT} character{COUNT::s}.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

{EXCEEDS}

Supporting Validators: TextLengthValidator, WordCountValidator

From: Runtime property

How much the {COUNT} exceeds the minimum or maximum.

Example

You have entered {EXCEEDS} too many characters.

{EXCEEDS:singular:plural}

Supporting Validators: TextLengthValidator, WordCountValidator

From: Runtime property

Helps build sentences where singular and plural forms are needed when you use the {EXCEEDS} token. For example, "You are over by 1 character." and "You are over by 2 characters."

You replace the term "singular" with the singular form of the word. You replace the term "plural" with the plural form of the word.

Example

You are over by {EXCEEDS} {EXCEEDS:character:characters}.

You are over by {EXCEEDS} character{EXCEEDS::s}.

{UNWANTED}

Supporting Validators: UnwantedWordsValidator

From: Runtime property

The text found to be unwanted.

Example

Do not enter '{UNWANTED}'.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Applying Styles To Tokens

Tokens can be automatically assigned to a style sheet class to make their text stand out. There are three types of tokens for which there are style sheet classes: labels, server side properties, and runtime properties.

The following are the predefined in the **DES\Appearance\Validation\Validation.css** style sheet file and used as defaults in the properties shown below.

```
/* Property:
PeterBlum.DES.Globals.WebFormDirector.ValidationManager.LabelTokenCssClass
{LABEL} tokens in the ErrorMessage
*/
.DESVALLabelToken
{
   font-weight:bold;
}

/* Property:
PeterBlum.DES.Globals.WebFormDirector.ValidationManager.RuntimeTokenCssClass
Tokens showing runtime values, like {COUNT} and {TEXTVALUE} in the ErrorMessage
*/
.DESVALRuntimeToken
{
}

/* Property:
PeterBlum.DES.Globals.WebFormDirector.ValidationManager.PropertyTokenCssClass
Tokens showing property values, like {MINIMUM} and {DIFFVALUE} in the ErrorMessage
*/
.DESVALPropertyToken
{
}

/* Property:
PeterBlum.DES.Globals.WebFormDirector.ValidationManager.LabelTokenCssClass
{LABEL} tokens in the SummaryErrorMessage
*/
.DESVALSummaryLabelToken
{
   font-weight:bold;
}

/* Property:
PeterBlum.DES.Globals.WebFormDirector.ValidationManager.RuntimeTokenCssClass
Tokens showing runtime values, like {COUNT} and {TEXTVALUE} in the
SummaryErrorMessage
*/
.DESVALSummaryRuntimeToken
{
}

/* Property:
PeterBlum.DES.Globals.WebFormDirector.ValidationManager.PropertyTokenCssClass
Tokens showing property values, like {MINIMUM} and {DIFFVALUE} in the
SummaryErrorMessage
*/
.DESVALSummaryPropertyToken
{
}
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

You set up the class names on properties of **PeterBlum.DES.Globals.WebFormDirector.ValidationManager**, which are page specific, or globally using the **Global Settings Editor** program.

You assign properties to **PeterBlum.DES.Globals.WebFormDirector.ValidationManager** programmatically in the `Page_Load()` method.

- **LabelTokenCssClass** (string) – The style sheet class name of tokens for Labels within the **ErrorMessage** property. No style is applied when "".

  In **Global Settings Editor**, the property is **DefaultLabelTokenCssClass**. It defaults to "`DESVALLabelToken`" and is predefined in the **DES\Appearance\Validation\Validation.css** style sheet file as shown on the previous page.

  *Note: If you want to change the style of the label control itself, set up the Label property on the Validator to reference the label. Then use the style sheet class defined by the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager.TextHiliteFieldsCssClass** property which defaults to "DESVALTextHiliteFields".*

- **PropertyTokenCssClass** (string) – The style sheet class name of tokens associated with server-side properties within the **ErrorMessage** property. No style is applied when "".

  In **Global Settings Editor**, the property is **DefaultPropertyTokenCssClass**. It defaults to "`DESVALPropertyToken`" and is predefined in the **DES\Appearance\Validation\Validation.css** style sheet file as shown on the previous page.

- **RuntimeTokenCssClass** (string) – The style sheet class name of tokens associated with runtime properties within the **ErrorMessage** property. No style is applied when "".

  In **Global Settings Editor**, the property is **DefaultRuntimeTokenCssClass**. It defaults to "`DESVALRuntimeToken`" and is predefined in the **DES\Appearance\Validation\Validation.css** style sheet file as shown on the previous page.

- **SummaryLabelTokenCssClass** (string) – The style sheet class name of tokens for Labels within the **SummaryErrorMessage** property. No style is applied when "".

  In **Global Settings Editor**, the property is **DefaultSummaryLabelTokenCssClass**. It defaults to "`DESVALSummaryLabelToken`" and is predefined in the **DES\Appearance\Validation\Validation.css** style sheet file as shown on the previous page.

- **SummaryPropertyTokenCssClass** (string) – The style sheet class name of tokens associated with server-side properties within the **SummaryErrorMessage** property. No style is applied when "".

  In **Global Settings Editor**, the property is **DefaultSummaryPropertyTokenCssClass**. It defaults to "`DESVALSummaryPropertyToken`" and is predefined in the **DES\Appearance\Validation\Validation.css** style sheet file as shown on the previous page.

- **SummaryRuntimeTokenCssClass** (string) – The style sheet class name of tokens associated with runtime properties within the **SummaryErrorMessage** property. No style is applied when "".

  In **Global Settings Editor**, the property is **DefaultSummaryRuntimeTokenCssClass**. It defaults to "`DESVALSummaryRuntimeToken`" and is predefined in the **DES\Appearance\Validation\Validation.css** style sheet file as shown on the previous page.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

ErrorFormatters: Customizing the Appearance of the Error Message

Validator Controls use **ErrorFormatters** to display the error message. An ErrorFormatter installs the HTML that formats the error message in the same location as the Validator control is placed into your ASP.NET web form. That HTML can do anything you want, including using JavaScript to make the message more interactive.

DES includes these ErrorFormatters:

- Text (PeterBlum.DES.Web.WebControls.TextErrorFormatter) – Displays the error message on the page, much like a Label control. It has numerous style properties although using a style sheet class with its **CssClass** property is preferred. It supports optional HTML text that encloses the error message, to customize its appearance. It can display an image to the left of the error message. See "Text".
- Hyperlink with Alert (PeterBlum.DES.Web.WebControls.HyperLinkErrorFormatter) – Displays a hyperlink with a short label that you supply. When clicked, an alert appears with the error message. See "Hyperlink with Alert".
- Image with Tooltip (PeterBlum.DES.Web.WebControls.TooltipImageErrorFormatter) – Displays an image. When the user points to it, a tooltip shows the error message. See "Image with Tooltip".
- Image with Alert (PeterBlum.DES.Web.WebControls.AlertImageErrorFormatter) – Displays an image. When the user clicks on it, an alert appears with the error message. See "Image with Alert".
- PopupView (PeterBlum.DES.Web.WebControls.PopupErrorFormatter) – Only show an image. When the user clicks on the image, show a PopupView, which is a floating message. In addition, when focus moves into a control being evaluated, it automatically pops up. See "PopupView". *ALERT: Requires a license that covers Peter's More Validators.*

Note: You can define other Error Formatters through subclassing. See the Developer's Guide for details.

Click on any of these topics to jump to them:

- Using ErrorFormatters
- ErrorFormatter Skins
- Properties Common To All ErrorFormatter Classes
- Text (TextErrorFormatter)
  - Using The TextErrorFormatter
  - Properties for TextErrorFormatter ("Text")
- Hyperlink with Alert(HyperLinkErrorFormatter)
  - Using This ErrorFormatter
  - Properties for HyperLinkErrorFormatter ("HyperLink with Alert")
- Image with Tooltip (TooltipImageErrorFormatter)
  - Using This ErrorFormatter
  - Properties for TooltipImageErrorFormatter ("Image with ToolTip")
- Image with Alert (AlertImageErrorFormatter)
  - Using This ErrorFormatter
  - Properties for AlertImageErrorFormatter ("Image with Alert")
- PopupView (PopupErrorFormatter)
  - Using This ErrorFormatter
  - Properties for PopupErrorFormatter ("PopupView")
  - Working With PopupViews
- Online example

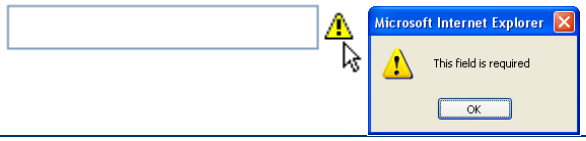
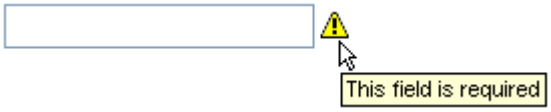
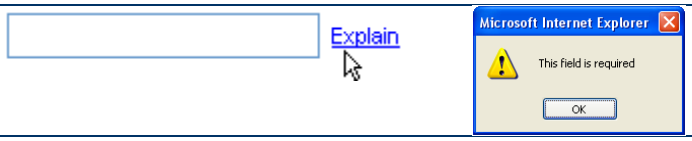
Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Using ErrorFormatters

You set up an ErrorFormatter on the Validator's **ErrorFormatter** property. The property holds an object, one of the ErrorFormatter classes. It defaults to the TextErrorFormatter ("Text"). If that is the desired ErrorFormatter, assign the properties of the TextErrorFormatter object.

Otherwise, assign the desired ErrorFormatter class to **ErrorFormatter**. Then assign its properties.

You can also predefine ErrorFormatters as "Skins". These are globally defined ErrorFormatter objects that are given a unique name. You assign the name to the Validator's **ErrorFormatterSkinID** property. See "ErrorFormatter Skins".

Name	Class	Example
Text	TextErrorFormatter	
Image with Alert	AlertImageErrorFormatter	
Image with Tooltip	ToolTipImageErrorFormatter	
Hyperlink with Alert	HyperLinkErrorFormatter	
PopupView	PopupErrorFormatter	

All Error Formatters supply HTML to the page. Before the error is detected, the HTML is invisible. Browsers permit invisible HTML to take up space or not. Use the **Display** property to control the preferred appearance. When `Static`, it takes up space. This is the default. When `Dynamic`, it uses no space until shown. You will find that when there are several Validators associated with one control, the `Dynamic` setting works better. *Note: The CombinedErrorMessages control also helps reduce the space used by multiple Validators in one location. See "CombinedErrorMessages Control".*

If you want to show the error only in the ValidationSummary control when the page is submitted, set **Display** to `None`.

Click on any of these topics to jump to them:

- Setting the Appearance
- Editing the ErrorFormatter in Design Mode
- ASP.NET Markup for the ErrorFormatter Property
- Creating an ErrorFormatter Programmatically
- Online example

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Setting the Appearance

ErrorFormatters provides most of the style properties found on any `System.Web.UI.WebControls.WebControl` subclass like **ForeColor** and **Font**. DES recommends that you use style sheets and sets the **CssClass** property to "`DESVALErrorText`", a predefined style sheet class that sets the font to red. It is defined in the style sheet file **\DES\Appearance\Validation\Validation.css**. You will find it easiest to edit the "`DESVALErrorText`" style instead of all the properties here and such a change is global to your Validators. See step 1 of "Adding Validation to a WebForm".

You can make your error blink on the page to get the user's attention. See "Blinking the ErrorFormatter".

When you choose a formatter with an image, the default image is this: ⚠. The file is located in **[web application root]\DES\Appearance\Validation\valerroricon.gif**. You can set your own image with the **ImageURL** property. DES supplies an animated GIF of the same image only it fades and returns. It is named "valerroricon_animated.GIF" in the **[web application root]\DES\Appearance** folder.

When you place the Validator on the page, it is often right next to another field, such as the textbox. You can insert a " " before it, but that character will not be hidden when the Validator is hidden. Use the **GapBefore** property to insert a gap before the Validator that gets hidden.

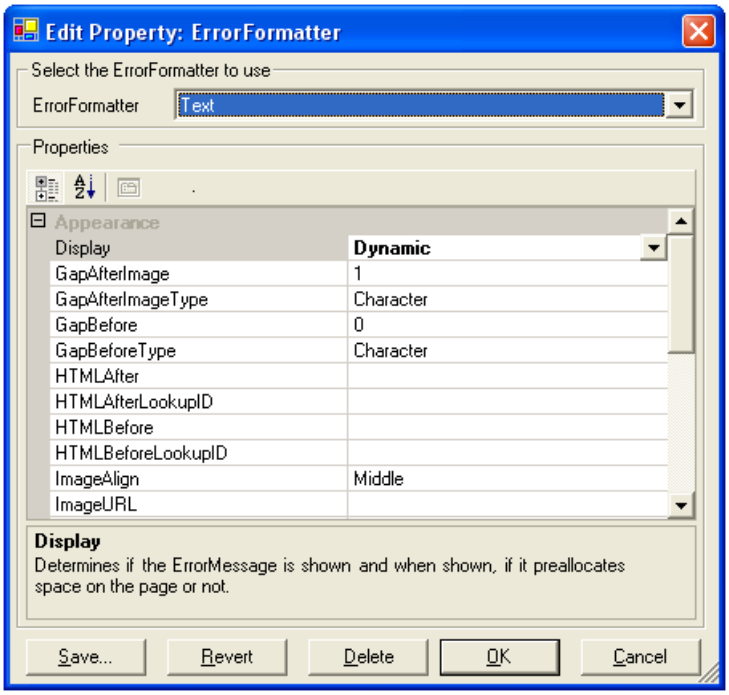
For the remaining properties, see the topics on each ErrorFormatter class that follow.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Editing the ErrorFormatter in Design Mode

The Properties Editor provides an editor on the **ErrorFormatter** property where you can change both the type of ErrorFormatter and its properties.



Start by selecting the type of ErrorFormatter you want from the ErrorFormatter field at the top of this window. Edit the properties and click **OK**.

You can customize the list by editing properties with these buttons:

- **Save** – Writes the properties and ErrorFormatter class to the **custom.des.config** file. It first prompts for a name. If you provide an existing name, you can overwrite that name. For example, if you use the name "Text", the factory defaults for the "Text" item in the list will now use the properties that you have supplied. However, any changes made to the saved names will not be applied to any controls that you already have on the page. You will need to go to each control and use the **Revert** button to change it to your new settings.

  Each saved ErrorFormatter – called a "skin" to follow the terminology from ASP.NET's "Themes and Skins" – is available for assignment to the **ErrorFormatterSkinID** property. This lets you select global ErrorFormatter configurations instead of having a fixed setup defined in the individual validator. A change to the skin will be reflected on all validators that use the same skin in their **ErrorFormatterSkinID** properties. See "ErrorFormatter Skins".

  Note: DES writes all changes to the custom.des.config file.

- **Revert** – Use this when you want to restore to the default properties of the name in the list. Remember that the default properties may be those you have previously stored with the **Save** button.

- **Delete** – Use this to remove the ErrorFormatter currently named in the DropDownList from DES. While you most likely will use this to remove your own saved definitions, you can delete DES's own definitions. That doesn't mean the objects are fully removed. They are simply no longer selectable in this interface. Any Validator that was associated with an Error Formatter that you delete here will continue to use the Error Formatter.

  Note: You can undelete DES's own definitions. Open the custom.des.config file. Locate the <ErrorFormatters> section and remove the node <ErrorFormatter name="name" enabled="false" />

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

ASP.NET Markup for the ErrorFormatter Property

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

Here is the RequiredTextValidator with the TextErrorFormatter and the **Display** property set to `Dynamic`.

```
<des:RequiredTextValidator id="RequiredTextValidator1" runat="server"
   ControlIDToEvaluate="TextBox1" ErrorMessage="This field is required" >
   <ErrorFormatterContainer>
      <des:TextErrorFormatter Display="Dynamic">
      </des:TextErrorFormatter>
   </ErrorFormatterContainer>
</des:RequiredTextValidator>
```

Notice that the **ErrorFormatter** property never appears as an attribute of the `<des:RequiredTextValidator>` tag. Instead, the `<ErrorFormatterContainer>` tag is a child of the Validator control. That tag has no attributes. The child to `<ErrorFormatterContainer>` defines the class, textual name and all properties of the ErrorFormatter:

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

- des:classname – Use "des:TextErrorFormatter", "des:TooltipImageErrorFormatter", "des:AlertImageErrorFormatter", "des:HyperLinkErrorFormatter", "des:PopupErrorFormatter" or the namespace and class of other ErrorFormatters that you made.
- [all properties] – Enter the properties into the tag the same way you do for any other control.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Creating an ErrorFormatter Programmatically

By default, the **ErrorFormatter** property contains a TextErrorFormatter. If you want to use it, typecast the **Validator.ErrorFormatter** property to `PeterBlum.DES.Web.WebControls.TextErrorFormatter` and edit its properties.

To change the ErrorFormatter, use these steps:

1. Create a new instance of your selected ErrorFormatter object. ErrorFormatters have numerous constructors including one with no parameters.
2. Assign the object to the **ErrorFormatter** property of the Validator control.
3. Assign any properties on your ErrorFormatter object.

Example

This example creates a TextErrorFormatter with the **Display** property set to `Dynamic`. It is assigned to the Validator Val1.

[C#]

```
PeterBlum.DES.Web.WebControls.TextErrorFormatter vEF = new
TextErrorFormatter();
Val1.ErrorFormatter = vEF;
vEF.Display = PeterBlum.DES.ValidatorDisplay.Dynamic;
```

[VB]

```
Dim vEF As PeterBlum.DES.Web.WebControls.TextErrorFormatter = New
TextErrorFormatter()
Val1.ErrorFormatter = vEF
vEF.Display = PeterBlum.DES.ValidatorDisplay.Dynamic
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

ErrorFormatter Skins

When you define an ErrorFormatter as described above, it becomes part of the individual Validator control. DES provides "skins" – a term borrowed from ASP.NET 's "Themes and Skins" – to let you set up named templates for your ErrorFormatters. You assign the name of a skin to the **ErrorFormatterSkinID** property and immediately that skin is used instead of the settings directly in the **ErrorFormatter** property of the individual Validator. Any change to the skin definition is applied to all Validators that have that skin name assigned to the **ErrorFormatterSkinID** property.

It is important to understand that once the **ErrorFormatterSkinID** property is assigned, it completely overrides the **ErrorFormatter** property. If you are using the **ErrorFormatterSkinID** and want to make a change to the **ErrorFormatter** property, you must clear the **ErrorFormatterSkinID** property to use the **ErrorFormatter** property settings.

Using ErrorFormatter Skins

Start by defining skins. They are ErrorFormatter objects with a name. They are either stored in the **custom.des.config** file or created programmatically as the application starts up.

Click on any of these topics to jump to them:

- Defining a Skin in Design Mode
- Defining a Skin in a Text Editor
- Defining a Skin Programmatically
- Online example

When skins are defined, each time your web application starts up, it will have a list of ErrorFormatters to choose from by a name. The name is the *SkinID*. DES also predefines several skin names. See "Predefined Skins".

You can assign the SkinID on an individual validator or for most of the validators on the page. You can also declare a property list along with Skin to customize the skin on-the-fly.

Assigning to an individual Validator

Set the Validator's **ErrorFormatterSkinID** property to the name of the skin. The name is case insensitive.

```
<des:validator id="validator1" runat="server" ErrorFormatterSkinID="SkinID" />
```

At this point, the ErrorFormatter property will be ignored as the skin will be used.

Using the skin on most of the Validators on the page

The page has a default SkinID. It is found in the **DefaultErrorFormatterSkinID** property on the PageManager control and programmatically on **PeterBlum.DES.Globals.WebFormDirector.ValidationManager**. When set, any validator whose **ErrorFormatterSkinID** property contains "{DEFAULT}" will use the default **ErrorFormatterSkinID** if it is assigned. (The validator will use its own ErrorFormatter if the default is blank.)

```
<des:PageManager id="PageManager1" runat="server"
   DefaultErrorFormatterSkinID="SkinID" />
```

Because every validator's **ErrorFormatterSkinID** defaults to "{DEFAULT}", a change to the **DefaultErrorFormatterSkinID** will immediately affect all validators, overriding their **ErrorFormatter** properties. For any whose **ErrorFormatter** should be used, you must change the validator's **ErrorFormatterSkinID** to "".

```
<des:validator id="validator1" runat="server" ErrorFormatterSkinID="" />
```

Customize the Skin on-the-fly

The **ErrorFormatterSkinID** and **DefaultErrorFormatterSkinID** allow you to add a list of properties that will override the skin's values. For example, you want to use the skin named "PopupRed" but demand the ErrorFormatter has its the Display property set to "Dynamic". The SkinID will have this text:

```
ErrorFormatterSkinID="PopupRed {Display='Dynamic'}"
```

The overall syntax is a space delimited list of PropertyName='value' pairs. The property name is case senstive. The value must be in quotes (usually single quotes because the entire property is enclosed in double-quotes.)

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

```
ErrorFormatterSkinID="SkinID {PropertyName1='Value' PropertyName2='Value'}"
```

Predefined Skins

DES already knows of several skins, which create the default version of each ErrorFormatter type unless you redefine it in the **custom.des.config** file. The skin names and their ErrorFormatters are here:

Skin name	ErrorFormatter type
"Text"	TextErrorFormatter
"TextErrorFormatter"	TextErrorFormatter
"Image with Alert"	AlertImageErrorFormatter
"AlertImageErrorFormatter"	AlertImageErrorFormatter
"Image with Tooltip"	TooltipImageErrorFormatter
"TooltipImageErrorFormatter"	TooltipImageErrorFormatter
"Hyperlink with Alert"	HyperLinkErrorFormatter
"HyperLinkErrorFormatter"	HyperLinkErrorFormatter
"PopupView"	PopupErrorFormatter
"PopupErrorFormatter"	PopupErrorFormatter

These are most effective when combined with the ability to customize the properties on the fly (see the previous topic).

For example:

```
ErrorFormatterSkinID="Text {Display='Dynamic'}"
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Defining a Skin in Design Mode

1. Open the Properties Editor window.
2. Click on the button in the **ErrorFormatter** property to open the ErrorFormatter dialog.
3. Set up this dialog with the desired ErrorFormatter class and property settings.
4. Click Save.
5. Enter the name of the skin and click OK. If you are changing an existing skin, enter the same name as the one you are changing.
6. Click OK to dismiss the ErrorFormatter dialog.

Note: The skin has not been assigned to the Validator. Instead, the Validator now has the configuration you have set up. You still need to assign the skin to the ErrorFormatterSkinID property.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Defining a Skin in a Text Editor

1. Open the **custom.des.config** file in a text editor. By default, it is located in the **[web application]\DES** folder.
2. Locate the `<ErrorFormatters>` section.
3. The syntax is:

```
<ErrorFormatter name="[Skin name]" type="[ErrorFormatter class]">
   <Property name="[property name]" value="[value]" />
   <Property name="[property name]" value="[value]" />
</ErrorFormatter>
```

The type attribute must reflect the full class name:
`PeterBlum.DES.Web.WebControls.TextErrorFormatter`,
`PeterBlum.DES.Web.WebControls.AlertImageErrorFormatter`,
`PeterBlum.DES.Web.WebControls.TooltipImageErrorFormatter`,
`PeterBlum.DES.Web.WebControls.HyperLinkErrorFormatter`,
`PeterBlum.DES.Web.WebControls.PopupErrorFormatter`.

Every tag, attribute, type, and property name is case sensitive.

For example, the skin called "Fancy Text" which uses the TextErrorFormatter to change the style sheet using the **CssClass** property looks like this:

```
   <ErrorFormatter name="Fancy Text"
type="PeterBlum.DES.Web.WebControls.TextErrorFormatter">
      <Property name="CssClass" value="SuperClass" />
   </ErrorFormatter>
```

If you need to change a sub property of the **Font** property, use this syntax:

```
   <ErrorFormatter name="[Skin name]" type="[ErrorFormatter class]">
      <Property name="[property name]" value="[value]" />
      <Property name="[property name]" value="[value]" />
      <Object name="Font">
        <Property name="[property name]" value="[value]" />
      </Object>
   </ErrorFormatter>
```

For example:

```
   <ErrorFormatter name="Italic Error"
type="PeterBlum.DES.Web.WebControls.TextErrorFormatter">
      <Object name="Font">
        <Property name="Italic" value="True" />
      </Object>
   </ErrorFormatter>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Defining a Skin Programmatically

Skins are globally defined. ASP.NET has you setup global values in the `Application_Start()` method of the **Global.asax** file. Follow these steps to add ErrorFormatters as skins programmatically.

1. Open the **Global.asax** file and locate the `Application_Start()` method.
2. Add this line:

   [C#]

   ```
   PeterBlum.DES.ConfigFile.ConfigFileLoaded +=
     new PeterBlum.DES.ConfigFilesLoadedEventHandler(MyConfigFilesLoaded);
   ```

   [VB]

   ```
   AddHandler PeterBlum.DES.ConfigFile.ConfigFileLoaded _
     AddressOf MyConfigFilesLoaded
   ```

3. Add this method to **Global.asax**:

   [C#]

   ```
   protected void MyConfigFilesLoaded(PeterBlum.DES.ConfigFile pConfigFile)
   {
   }
   ```

   [VB]

   ```
   Protected Sub MyConfigFilesLoaded(ByVal pConfigFile As PeterBlum.DES.ConfigFile)
   End Sub
   ```

4. Create your ErrorFormatter object in the `MyConfigFilesLoaded()` method then add it to `PeterBlum.DES.Globals.ErrorFormatters.Add()`. The function takes two parameters: the skin name (a string), and the ErrorFormatter instance.

Example

Adds a skin named "Popup to Left" that uses a PopupErrorFormatter whose **PopupViewName** is set to "LtYellow_Small ToLeft". *The popupview name is assumed to be defined in the custom.des.config file.*

[C#]

```
protected void MyConfigFilesLoaded(PeterBlum.DES.ConfigFile pConfigFile)
{
  PeterBlum.DES.Web.WebControls.PopupErrorFormatter vPEF =
    new PeterBlum.DES.Web.WebControls.PopupErrorFormatter();
  vPEF.PopupViewName = "LtYellow_Small ToLeft";
  PeterBlum.DES.Globals.ErrorFormatters.Add("Popup to Left", vPEF);
}
```

[VB]

```
Protected Sub MyConfigFilesLoaded(ByVal pConfigFile As PeterBlum.DES.ConfigFile)
  Dim vPEF As PeterBlum.DES.Web.WebControls.PopupErrorFormatter = _
    New PeterBlum.DES.Web.WebControls.PopupErrorFormatter()
  vPEF.PopupViewName = "LtYellow_Small ToLeft"
  PeterBlum.DES.Globals.ErrorFormatters.Add("Popup to Left", vPEF)
End Sub
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Properties Common To All ErrorFormatter Classes

These properties are shown on any ErrorFormatter. They handle much of the formatting. Typically you assign the **Display** property and customize the style sheet associated with **CssClass** property.

For properties specific to each Errormatter Class, click on any of these topics to jump to them:

- Properties for TextErrorFormatter ("Text")
- Properties for HyperLinkErrorFormatter ("HyperLink with Alert")
- Properties for TooltipImageErrorFormatter ("Image with ToolTip")
- Properties for AlertImageErrorFormatter ("Image with Alert")
- Properties for PopupErrorFormatter ("PopupView")

- **Display** (enum PeterBlum.DES.ValidatorDisplay) – Determines whether the error is visible at all and if so, does it take up space on the page when its invisible. The enumerated type `PeterBlum.DES.ValidatorDisplay` has the following values:
  - `None` – The error message is not shown where the Validator control is located. There are several other ways to show the error message without using up valuable space near the field that reflects the error. Use the ValidationSummary control or show an alert box. See "Show an Alert".
  - `Static` – The error message will show where the Validator control is located. When invisible, it will still use space on the page as if the error message was shown. When error messages use runtime tokens, like "{COUNT}" and "{TEXTVALUE}", the space may differ from the actual error message as those tokens are replaced. This is the default.

    This setting limits the amount of rearrangement that goes on when the error message appears.
  - `Dynamic` – The error message will show where the Validator control is located. When invisible, it does not use up any space on the page. This is a good choice when there are several Validator controls, side-by-side, because when one appears, it will have a gap where other Validators are located before it.

    This setting causes other elements on the page to move as the ErrorFormatter is shown and hidden.
- **CssClass** (string) – Gets and sets the class name used with the control. A style class can encompass all of the other properties associated with styles here. So edit the style sheet file to update the look of all your Validators at once.

  It defaults to "`DESVALErrorText`".

  The style is declared in **DES\Appearance\Validation\Validation.css**:

```
.DESVALErrorText
{
   color: Red !important;
}
```

- **GapBefore** (int) – When you place the Validator on the page, it is often right next to another field, such as the textbox. You can insert a " " before it but that character will not be hidden when the Validator is hidden. Use this to insert a gap before the Validator that gets hidden. This sets the number of characters or pixels that will form the gap, depending on **GapBeforeType**. When 0, there is no gap. It defaults to 0.
- **GapBeforeType** (enum PeterBlum.DES.GapType) – Associated with **GapBefore** to determine if the number specified is in characters or pixels. The enumerated type `PeterBlum.DES.GapType` has these values:
  - `Character` – " " is used to form the gap
  - `Pixels` – an `<img>` with a stretched 1x1 pixel image is used to form the gap
- **ControlStyle** (System.Web.UI.Style) – Gets and sets the style of the ErrorFormatter. Programmers generally use this property. It provides the internal storage for the remaining properties listed here.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

- **BackColor** (System.Drawing.Color) – Gets and sets the background color. It defaults to `Color.Empty`. *Recommendation: Use style sheets through the CssClass property.*
- **BorderColor** (System.Drawing.Color) – Gets and sets the border color, if a border is enabled with BorderStyle. It defaults to `Color.Empty`. *Recommendation: Use style sheets through the CssClass property.*
- **BorderStyle** (System.Web.UI.WebControls.BorderStyle) – Gets and sets the style of the border. It defaults to `BorderStyle.None`. *Recommendation: Use style sheets through the CssClass property.*
- **BorderWidth** (string) – Gets and sets the width of the border. See BorderWidth. *Recommendation: Use style sheets through the CssClass property.*
- **Font** (System.Web.UI.WebControls.FontInfo) – Gets and sets numerous properties describing the font including the family, size, italics, bold, and underline. See Font. *Recommendation: Use style sheets through the CssClass property.*
- **ForeColor** (System.Drawing.Color) – Gets and sets the font color. It defaults to `Color.Empty`. *Recommendation: Use style sheets through the CssClass property.*
- **Height** (System.Web.UI.WebControls.Unit) – Gets and sets the height of the error message area. The default is `Unit.Empty`. See Height. *Recommendation: Use style sheets through the CssClass property.*
- **Width** (System.Web.UI.WebControls.Unit) – Gets and sets the width of the error message area. The default is `Unit.Empty`. See Width. *Recommendation: Use style sheets through the CssClass property.*
- **EnclosedBy** (enum PeterBlum.DES.EFEnclosedBy) – By default, the Validator is enclosed in a `<span>` tag. When using XHTML syntax, "block"-style tags like `<div>` and `<table>` are not allowed inside a `<span>` tag. When you use block style tags inside the **ErrorMessage**, **HTMLBefore** or **HTMLAfter** properties, set this to DIV to enclose the Validator in a `<div>` tag. This enumerated type has these values:
    - `SPAN` – Creates a `<span>` tag. This is the default.
    - `DIV` – Creates a `<div>` tag.

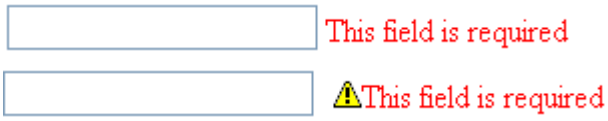
Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Text

Class: PeterBlum.DES.Web.WebControls.TextErrorFormatter

The TextErrorFormatter class shows the error message as text on the form. It supports styles. It is similar to using the `System.Web.UI.WebControls.Label` control to show text. It can optionally show an image to the left of the error message and be wrapped in HTML to heavily customize its appearance.



Using The TextErrorFormatter

Generally you set the **Display** property and customize the style sheet class "`DESVALErrorText`" in **DES\Appearance\Validation\Validation.css**:

```
.DESVALErrorText
{
   color: Red;
}
```

Tokens embedded in the error messages can have a different font style. See "Applying Styles To Tokens".

Insert an image before the text with the **ImageURL** property. Use the associated properties **ImageAlign** and **GapAfterImage** to perfect its appearance.

Use the **HTMLBefore** and **HTMLAfter** properties to insert HTML before and after the error message so that you can create almost any appearance that you want.

Online example

Properties for TextErrorFormatter ("Text")

Here are the properties for this class.

- **Display**, **CssClass**, **GapBefore**, **GapBeforeType**, **ControlStyle**, **BackColor**, **BorderColor**, **BorderStyle**, **BorderWidth**, **Font**, **ForeColor**, **Height**, and **Width** – See "Properties Common To All ErrorFormatter Classes".
- **TextCssClass** (string) – While **CssClass** provides a style sheet class for the overall HTML, this provides the style sheet class to the `<span>` tag directly enclosing the error message itself. It defaults to "".

  ```
  <span class='CssClass'>content before error message
     <span class='TextCssClass'>error message</span>
  content after error message</span>
  ```

- **ImageURL** (string) – An optional image that can appear to the left of the error message text. Provide a URL to an image file. When "", no image is used. It defaults to "". If a tool tip is set up, it will appear when the user points to the image.

  Use the token "{DEFAULT}" to use the global setting defined in the **DefaultImageErrorFormatterImageURL** property of the **Global Settings Editor**.

  #### 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.

  Note: If you want to blink the image but not the text, set BlinkCssClass = "". See "Blinking the ErrorFormatter".

- **ImageAlign** (enum System.Web.UI.WebControls.ImageAlign) – When using **ImageURL**, this aligns the image to the error message text and the rest of the row. It defaults to `ImageAlign.Middle`.
- **GapAfterImage** (int) – When you use the **ImageURL** property, use this to establish a gap between the image and the error message text. This sets the number of characters or pixels that will form the gap, depending on **GapAfterImageType**. When 0, there is no gap. It defaults to 0.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- **GapAfterImageType** (enum PeterBlum.DES.GapType) – Associated with **GapAfterImage** to determine if the number specified is in characters or pixels. The enumerated type `PeterBlum.DES.GapType` has these values:
  - `Character` – " " is used to form the gap
  - `Pixels` – an <img> with a stretched 1x1 pixel image is used to form the gap
- **HTMLBefore** (string) – Add HTML that appears before the error message and optional image. This gives great flexibility in formatting, often avoiding having to build a new ErrorFormatter class. If the HTML needs ending tags after the error message, use **HTMLAfter**.

  When entering HTML directly into the ASP.NET page, be sure to enter any tags using "<" for "<" and ">" for ">". For example, "" should become HTMLBefore="`<i>`".

  Example

  Suppose these formatting properties are set:

  ```
  HTMLBefore = "&lt;i&gt;"
  HTMLAfter = "&lt;/i&gt;"
  ImageURL = "/Images/Graphic.gif"
  ```

  When the page is rendered to the browser, it will contain this HTML:

  ```
  <i><img src="/Images/Graphic.gif" align="Middle" />[error message tags]</i>
  ```

  Reminder: If your page should follow the XHTML standard, make sure your tags conform to XHTML.
- **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 ValidationMisc. 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) – Add HTML that appears after the error message. This gives great flexibility in formatting, often avoiding having to build a new ErrorFormatter class.

  When entering HTML directly into the ASP.NET page, be sure to enter any tags using "<" for "<" and ">" for ">". For example, "" should become HTMLBefore="`</i>`".

  Reminder: If your page should follow the XHTML standard, make sure your tags conform to XHTML.
- **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 ValidationMisc. 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 "".
- **ToolTip** (string) – The tool tip that appears when the user points to the error message. It defaults to "".
- **ToolTipLookupID** (string) – Gets the value for **ToolTip** 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 Hints. If no match is found OR this is blank, **ToolTip** 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.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

To use it, define a LookupID and associated textual value in your data source (resource, database, etc). Assign the same LookupID to this property.

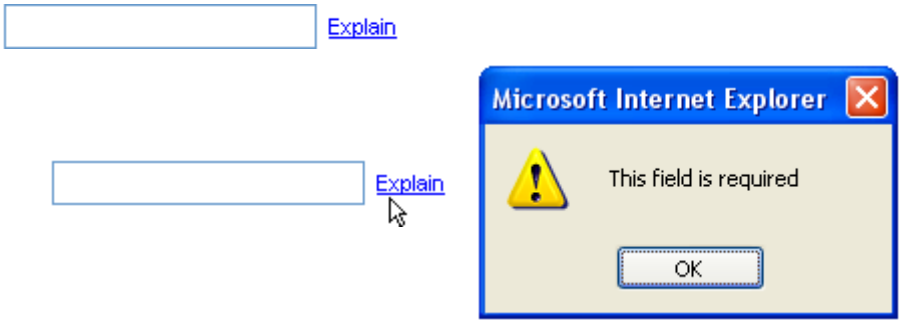
It defaults to "".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Hyperlink with Alert

Class: PeterBlum.DES.Web.WebControls.HyperLinkErrorFormatter

The HyperLinkErrorFormatter class shows a hyperlink. When the user clicks on the hyperlink, an alert shows the error message. The hyperlink may be easier for users to understand so that they click on it to get the error. The text of the hyperlink is static, so it can take up very little room on the page. The default text is "Explain". This ErrorFormatter is based on the TextErrorFormatter and inherits its capabilities to show an image and have HTML before and after the hyperlink.



Suggestion: If you want to show an image where the user clicks to show an alert, use the "Image with Alert".

Using This ErrorFormatter

Set the **Display** property. Assign the text for the hyperlink to **HyperLinkText** and its optional style sheet class to **HyperLinkCssClass**.

A tooltip is recommended so that the user knows what to expect when they click. Set the **ToolTip** property. For example, use "Show the error".

You can add an image that appears to the left of the hyperlink with the **ImageURL** property.

This ErrorFormatter supports the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager.FocusAfterAlert** property that sets focus to the field with the error after the alert is dismissed.

Online example

Properties for HyperLinkErrorFormatter ("HyperLink with Alert")

Here are the properties for this class.

- **Display**, **CssClass**, **GapBefore**, **GapBeforeType**, **ControlStyle**, **BackColor**, **BorderColor**, **BorderStyle**, **BorderWidth**, **Font**, **ForeColor**, **Height**, and **Width** – See "Properties Common To All ErrorFormatter Classes".
- **TextCssClass** (string) – While **CssClass** provides a style sheet class for the overall HTML, this provides the style sheet class to the `<span>` tag directly enclosing the error message itself. It defaults to "".

```
<span class='CssClass'>content before error message
   <span class='TextCssClass'><a>error message</a></span>
content after error message</span>
```

- **ImageURL**, **ImageAlign**, **GapAfterImage**, **GapAfterImageAlign**, **HTMLBefore**, **HTMLBeforeLookupID**, **HTMLAfter**, **HTMLAfterLookupID**, **ToolTip**, and **ToolTipLookupID** – See "Properties for TextErrorFormatter ("Text")".
- **HyperLinkText** (string) – The text label of the HyperLink. It defaults to "Explain".
- **HyperLinkTextLookupID** (string) – Gets the value for **HyperLinkText** 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 ValidationMisc. If no match is found OR this is blank, **HyperLinkText** 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.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

It defaults to "".

- **HyperLinkCssClass** (string) - A style sheet class name assigned to the HyperLink. When blank, the page's style sheet rules for hyperlinks are applied. It defaults to "".

  When working with hyperlinks, here is how to set up the 4 states – normal, visited, hover, and active – in your style sheet file.

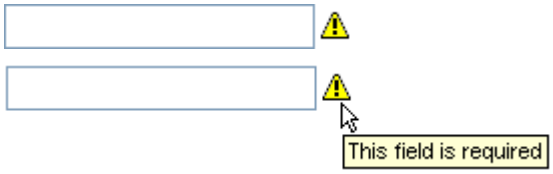
```
A.ClassName
{ attributes }
A.ClassName:link
{ attributes }
A.ClassName:active
{ attributes }
A.ClassName:hover
{ attributes }
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Image with Tooltip

Class: PeterBlum.DES.Web.WebControls.TooltipImageErrorFormatter

The TooltipImageErrorFormatter class shows an image. When the user points to the image, a tool tip exposes the error message. This format usually takes up much less room than when text is shown directly on the page. DES supplies a default image (shown below), but you can substitute your own.



Using This ErrorFormatter

Set the **Display** property. Set the image with the **ImageURL** property.

Online example

Properties for TooltipImageErrorFormatter ("Image with ToolTip")

Here are the properties for this class.

- **Display**, **CssClass**, **GapBefore**, **GapBeforeType**, **ControlStyle**, **BackColor**, **BorderColor**, **BorderStyle**, **BorderWidth**, **Font**, **ForeColor**, **Height**, and **Width** – See "Properties Common To All ErrorFormatter Classes".
- **ImageURL** (string) – The URL to the image.

  When assigned to "{DEFAULT}", it retrieves a value from the global **DefaultImageErrorFormatterImageURL** property that you set with the **Global Settings Editor**. By using the global value, your site will maintain a consistent look and changes are applied in one place. You can also assign a URL directly to this property.

  By default, the global setting **DefaultImageErrorFormatterImageURL** uses the URL "`{APPEARANCE}/Validation/ValErrorIcon.gif`" which looks like this: . DES provides an animated version of this image when you change to this URL: "`{APPEARANCE}/Validation/valerroricon_animated.GIF`"

  ImageURL 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.

- **ImageAltText** (string) – The alternative text displayed by the `<img>` tag when it cannot load the image.

  When assigned to "{DEFAULT}", it retrieves a value from the global **DefaultImageErrorFormatterImageAltText** property that you set with the **Global Settings Editor**. By using the global value, your site will maintain a consistent look and changes are applied in one place.

  By default, the global **DefaultImageErrorFormatterImageAltText** uses "".

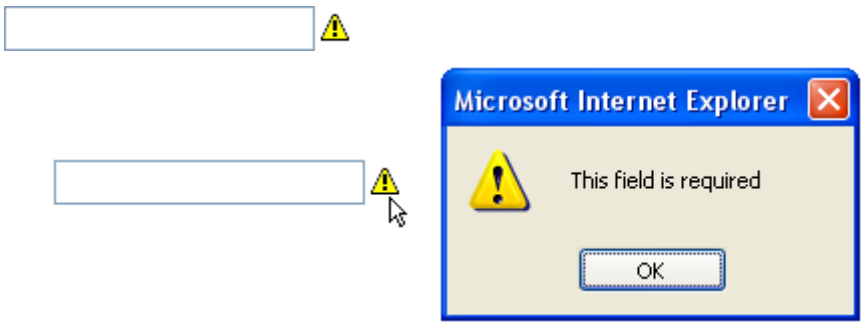
  ImageAltText defaults to "{DEFAULT}".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Image with Alert

Class: PeterBlum.DES.Web.WebControls.AlertImageErrorFormatter

The AlertImageErrorFormatter class shows an image. When the user clicks on the image, an alert shows the error message. The image has an optional tool tip to tell the user to "click for more details". DES supplies a default image (shown below), but you can substitute your own.



Using This ErrorFormatter

Set the **Display** property. Set the image with the **ImageURL** property. Set the optional tooltip with the **ToolTip** property.

This ErrorFormatter supports the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager.FocusAfterAlert** property that sets focus to the field with the error after the alert is dismissed.

Online example

Properties for AlertImageErrorFormatter ("Image with Alert")

Here are the properties for this class.

- **Display**, **CssClass**, **GapBefore**, **GapBeforeType**, **ControlStyle**, **BackColor**, **BorderColor**, **BorderStyle**, **BorderWidth**, **Font**, **ForeColor**, **Height**, and **Width** – See "Properties Common To All ErrorFormatter Classes".
- **ImageURL** (string) – The URL to the image.

  When assigned to "{DEFAULT}", it retrieves a value from the global **DefaultImageErrorFormatterImageURL** property that you set with the **Global Settings Editor**. By using the global value, your site will maintain a consistent look and changes are applied in one place. You can also assign a URL directly to this property.

  By default, the global setting **DefaultImageErrorFormatterImageURL** uses the URL "`{APPEARANCE}/Validation/ValErrorIcon.gif`" which looks like this: ⚠. DES provides an animated version of this image when you change to this URL: "`{APPEARANCE}/Validation/valerroricon_animated.GIF`"

  ImageURL 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.

- **ImageAltText** (string) – The alternative text displayed by the `<img>` tag when it cannot load the image.

  When assigned to "{DEFAULT}", it retrieves a value from the global **DefaultImageErrorFormatterImageAltText** property that you set with the **Global Settings Editor**. By using the global value, your site will maintain a consistent look and changes are applied in one place.

  By default, the global **DefaultImageErrorFormatterImageAltText** uses "".

  ImageAltText defaults to "{DEFAULT}".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- **ToolTip** (string) – The ToolTip that appears when the user points to the image. The purpose it to tell the user to click to get the full message. It defaults to “Click here for details”.
- **ToolTipLookupID** (string) – Gets the value for **ToolTip** 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 Hints. If no match is found OR this is blank, **ToolTip** 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 "".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

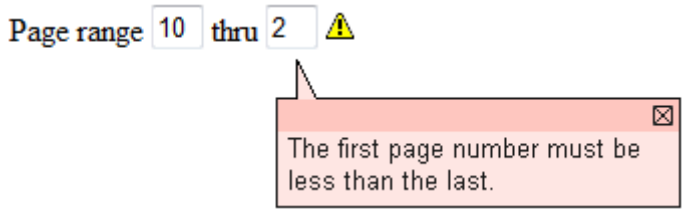
PopupView

Class: PeterBlum.DES.Web.WebControls.PopupErrorFormatter

ALERT: Requires a license that covers Peter's More Validators.

The PopupErrorFormatter class shows an image. When the user clicks on the image, a PopupView exposes the error message. Optionally, it can popup simply by passing the mouse over the image. In addition, when focus enters the control, it also opens the 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.



Using This ErrorFormatter

Set the **Display** property. Set the image with the **ImageURL** property. Set the PopupView definition in the **PopupViewName** property. It will default to a global setting, **DefaultErrorMessagePopupViewName**.

To add, edit, rename and delete PopupView definitions, see "Working With PopupViews".

If you are using the PopupView.HelpBehavior feature, set the appropriate text in the **ErrorMessageHelp** property.

By default, the PopupView appears when the mouse passes over the image. If you want to modify this behavior, use the **PopupOnMouseOver** and **PopupOnMouseOverDelay** properties.

By default, the PopupView appears when focus is established on the control with the error. If you want to disable this feature, set **PopupOnFocus** to `false`.

The PopupView will point to the last control specified by the list of ControlIDs to evaluate on a validator. For example, if the MultiRequiredFieldValidator is associated with 5 textboxes, the PopupView points to the 5th textbox. You can override this in two ways. Specify the desired control to display the PopupView in the **TargetControlID** property or have it point to the validator's image by setting **TargetIsValidator** to `true`.

Online example

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Properties for PopupErrorFormatter ("PopupView")

Here are the properties for this class.

- **Display**, **CssClass**, **GapBefore**, **GapBeforeType**, **ControlStyle**, **BackColor**, **BorderColor**, **BorderStyle**, **BorderWidth**, **Font**, **ForeColor**, **Height**, and **Width** – See "Properties Common To All ErrorFormatter Classes".

- **ImageURL** (string) – The URL to the image.

  When assigned to "{DEFAULT}", it retrieves a value from the global **DefaultImageErrorFormatterImageURL** property that you set with the **Global Settings Editor**. By using the global value, your site will maintain a consistent look and changes are applied in one place. You can also assign a URL directly to this property.

  By default, the global setting **DefaultImageErrorFormatterImageURL** uses the URL "`{APPEARANCE}/Validation/ValErrorIcon.gif`" which looks like this: ⚠. DES provides an animated version of this image when you change to this URL: "`{APPEARANCE}/Validation/valerroricon_animated.GIF`"

  ImageURL 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.

- **ImageAltText** (string) – The alternative text displayed by the `<img>` tag when it cannot load the image.

  When assigned to "{DEFAULT}", it retrieves a value from the global **DefaultImageErrorFormatterImageAltText** property that you set with the **Global Settings Editor**. By using the global value, your site will maintain a consistent look and changes are applied in one place.

  By default, the global **DefaultImageErrorFormatterImageAltText** uses "".

  ImageAltText defaults to "{DEFAULT}".

- **PopupViewName** (string) – Determines which globally defined PopupView is used. Specify the name or use "{DEFAULT}" to select the name defined globally. See "Working With PopupViews".

  Use the **Global Settings Editor** to edit PopupView definitions. See "Using the Global Settings Editor to define PopupViews".

  Here are the predefined values:

  `LtRed-Small`, `LtRed-Medium`, `LtRed-Large`.

  All of these are light red (actually "mistyrose". 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 **DefaultErrorMessagePopupViewName**, 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 you change the name of a global value and forget to change the name in this property.

  It defaults to "{DEFAULT}".

CONTINUED ON THE NEXT PAGE

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

- **OverriddenPopupView** (PeterBlum.DES.Web.WebControls.ErrorFormatterPopupView) – Overrides the value in **PopupViewName** with an instance of your own `PeterBlum.DES.Web.WebControls.ErrorFormatterPopupView` class to establish the appearance of the popup error message 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.ErrorFormatterPopupView Class".

  It defaults to `null`.

  If you want to start with one of the ErrorMessage 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.ErrorFormatterPopupView vPV =
   new PeterBlum.DES.Web.WebControls.ErrorFormatterPopupView();
vPV.HelpBehavior = PeterBlum.DES.HelpBehavior.ButtonReplaces;
vPV.Width = new Unit("350px");
vPV.DefaultPosition = PeterBlum.DES.DefaultViewPosition.BottomCentered;
vErrorFormatter.OverriddenPopupView = vPV;
```

  [VB]

```
Dim vPV As PeterBlum.DES.Web.WebControls.ErrorFormatterPopupView = _
   New PeterBlum.DES.Web.WebControls.ErrorFormatterPopupView()
vPV.HelpBehavior = PeterBlum.DES.HelpBehavior.ButtonReplaces
vPV.Width = New Unit("350px")
vPV.DefaultPosition = PeterBlum.DES.DefaultViewPosition.BottomCentered
vErrorFormatter.OverriddenPopupView = vPV
```

  Example: Using GlobalToOverriddenPopupView() method

  The `PopupErrorFormatter.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.ErrorFormatterPopupView vPV =
   vPopupErrorFormatter.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.ErrorFormatterPopupView = _
   vPopupErrorFormatter.GlobalToOverriddenPopupView("MyPopupView")
vPV.HelpBehavior = PeterBlum.DES.HelpBehavior.ButtonReplaces
vPV.Width = New Unit("350px")
vPV.DefaultPosition = PeterBlum.DES.DefaultViewPosition.BottomCentered
```

CONTINUED ON THE NEXT PAGE

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

- **ErrorMessageHelp** (string) – 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 **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 **ErrorMessageHelp** based on **PopupView.HelpBehavior**:

  - `None` - Do not show a Help Button. The **ErrorMessageHelp** property is not used.
  - `ButtonAppends` - Add the text from **ErrorMessageHelp** 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 **ErrorMessageHelp**. 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 **ErrorMessageHelp** 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 **ErrorMessageHelp** text will appear in the "{0}" token of **PopupView.HyperlinkUrlForHelpButton**.
  - `ButtonRunsScript` - Runs the script supplied in **PopupView.ScriptForHelpButton**. The **ErrorMessageHelp** text will replace the token "{0}" in that script.

  This defaults to "".

- **ErrorMessageHelpLookupID** (string) – Gets the value for **ErrorMessageHelp** 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 ErrorMessages. If no match is found OR this is blank, **ErrorMessageHelp** 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 "".

- **PopupOnMouseOver** (Boolean) – When `true`, the user can point to the image. After a short delay, the PopupView will automatically appear. It defaults to `true`.

  The delay is defined in **PopupOnMouseOverDelay**.

  It will popdown when:

  - PopdownOnMouseExit is true and you move the mouse off the image.
  - Focus is lost
  - Another PopupView, whether a PopupErrorFormatter, Hint, or Extended Tooltip, appears.

- **PopupOnMouseOverDelay** (integer) - When **PopupOnMouseOver** is `true`, this is the time delay between when the mouse moves over the image and the PopupView appears. The value is in milliseconds.

  If 0, it pops up immediately.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

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.

- **PopdownOnMouseExit** (Boolean) – Determines if **PopupOnMouseOver** will pop down the PopupView once the mouse moves off the error formatter.

  It has no effect when **PopupOnMouseOver** is `false`.

  When `true`, the PopupView will be hidden as the mouse moves off the error formatter.

  When `false`, the PopupView remains visible when the mouse moves off the error formatter.

  It defaults to `false`.

- **PopupOnFocus** (Boolean) – When `true`, if the focus enters any data entry control assigned to the validator. It pops down when focus is lost.

  It defaults to `true`.

- **PopupOnFocusDelay** (Integer) – When **PopupOnFocus** is `true`, this is the time delay between when the focus enters the control and the PopupView appears.

  The value is in milliseconds. If 0, it pops up immediately.

  It defaults to 350 (>1/3 second).

- **TargetControlID** (string) – Specify the exact control which targets where the popup appears. If not specified, it uses the last control it was assigned from the various **ControlIDToEvaluate** and **SecondControlIDToEvaluate** properties.

  If you want the target to be the validator's own image, use **TargetIsValidator** instead.

  It defaults to "".

  When the target control is not in the same naming container, assign the control reference programmatically to the **TargetControl** property (below).

- **TargetControl** (Control) – A reference to the target control. It is an alternative to **TargetControlID** that allows the control to be anywhere on the page instead of the same naming container as the Validator. You must assign it programmatically.

  When assigned, it overrides the value of **TargetControlID**.

- **TargetIsValidator** (Boolean) – If you want the PopupErrorFormatter to point to the image from the errorformatter instead of any other field, set this to `true`.

  Overrides **TargetControlID** and **TargetControl** properties.

  When `true`, the validator is the target.

  When `false`, the **TargetControl** or if not assigned, the last ControlID to Evaluate is the target.

  It defaults to `false`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Working With PopupViews

A PopupView is similar to a ToolTip, created with HTML and JavaScript to float near the control.

Click on any of these topics to jump to them:

- PopupView Features
- Using the Global Settings Editor to define PopupViews
- View an existing definition
  - Fields on the Initial View
- 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 Features

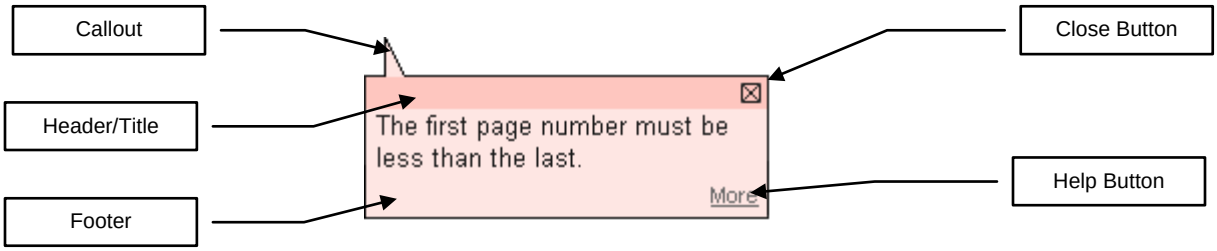
- Create as many PopupView definitions as needed in the **Global Settings Editor**.
- Style sheets control much of the appearance, including colors, borders, and fonts. There are predefined style sheets with yellow, red, blue and grey color schemes in the **DES\Appearance\Interactive Pages\PopupErrorFormatter.css** 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 **ErrorMessageHelp** property. It also has an optional close box.
- It can be dragged to expose other controls that it is covering. While dragging, its opacity increases 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 more 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 **ErrorMessageHelp** 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 **ErrorMessageHelp** property, offering the user expanded directions.
  - Run javascript that can use the text from **ErrorMessageHelp** to customize it
  - Go to a URL with the text from the **ErrorMessageHelp** 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 PopupView can enclose the error message in HTML, such as for showing an image.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Using the Global Settings Editor to define PopupViews

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 **PopupViewFormatter.PopupViewName** property by setting its name in the **DefaultErrorMessagePopupViewName** property.

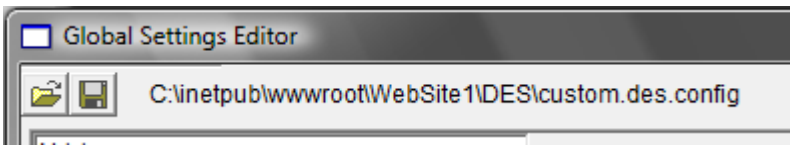


Here is how to define PopupViews.

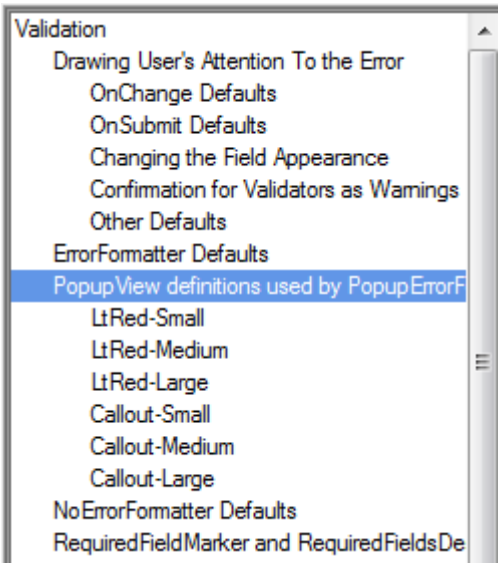
1. Open the **Global Settings Editor**.

   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 PopupErrorFormatters** 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.ErrorFormatterPopupView 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.

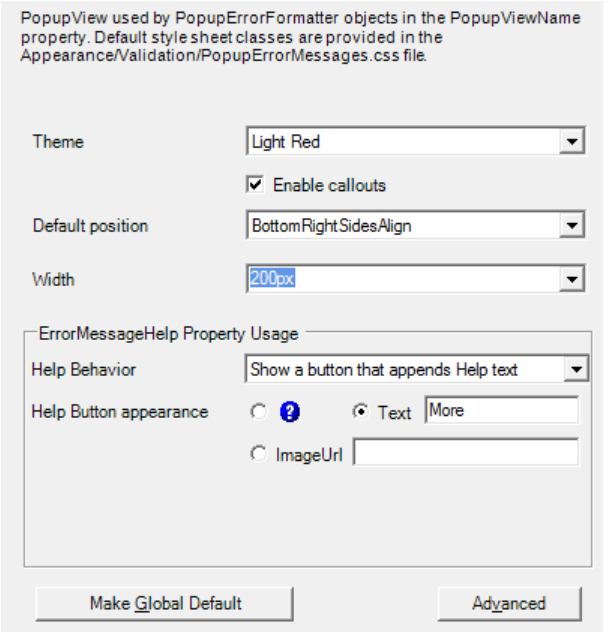


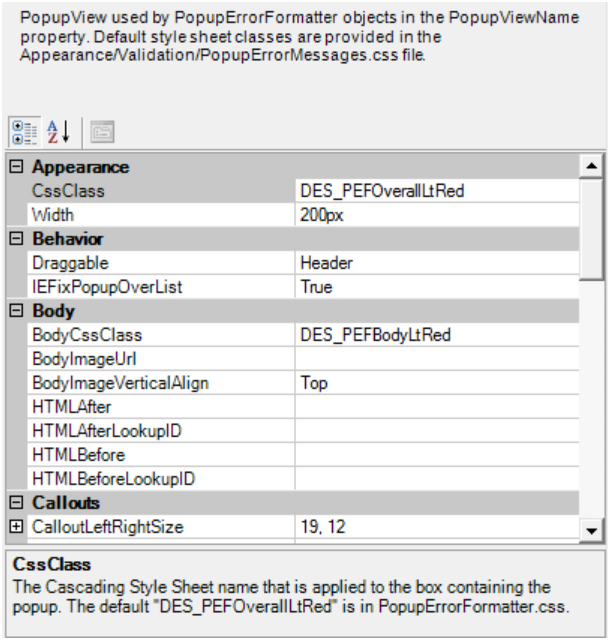
Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

View an existing definition

To view an existing definition, click on its name in the items below the **PopupView definitions used by PopupErrorFormatter** 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

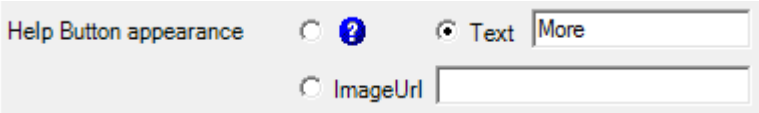
For details, see "Properties for the PeterBlum.DES.Web.WebControls.ErrorFormatterPopupView Class".

- **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 **ErrorMessageHelp** 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 **ErrorMessageHelp**. Do not show a Help Button.
    - `Show a button that appends the help text` – Use the Help Button. When clicked, redraw with the **ErrorMessageHelp** text appended to the current text. The value of **PopupView.AppendHelpSeparator** is inserted between the original hint and the text of **ErrorMessageHelp**.
    - `Show an` **ErrorMessageHelp** `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 **ErrorMessageHelp** text appears in the header as the title. It is used instead of the **PopupView.HeaderText** property value. There is no Help Button.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- `Show a button that hyperlinks` – Use the Help Button that acts as a hyperlink. Define the URL in the **URL** field. The **ErrorMessageHelp** 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 **ErrorMessageHelp** text will appear in the "{0}" token.
- `Show a button that runs a script` – Runs the script supplied in the **Script** field. The **ErrorMessageHelp** 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.



- **Make Global Default** – When clicked, this PopupView definition will become the default for all PopupErrorFormatters whose **PopupViewName** property is "{DEFAULT}". It updates the setting **DefaultErrorFormatterPopupViewName** in the topic "ErrorFormatter Defaults" of the **Global Settings Editor**.
- **Advanced** – Switch to the Advanced view, where you have access to every PopupView property using a Properties Editor.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Edit a definition

To edit a definition, click on its name in the items below the **PopupView definitions used by PopupErrorFormatters** 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.ErrorFormatterPopupView 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 PopupErrorFormatters** 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.

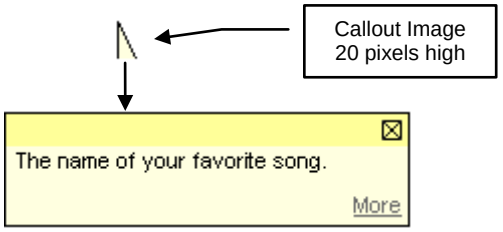
Delete a definition

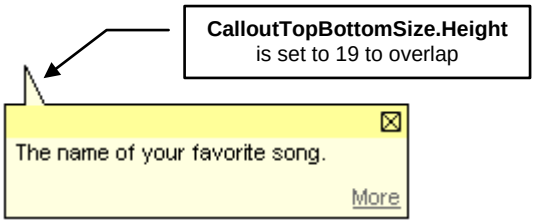
Click on the name of the PopupView definition in the list and click the **Delete** button at the bottom of the window.

Creating your own Callouts

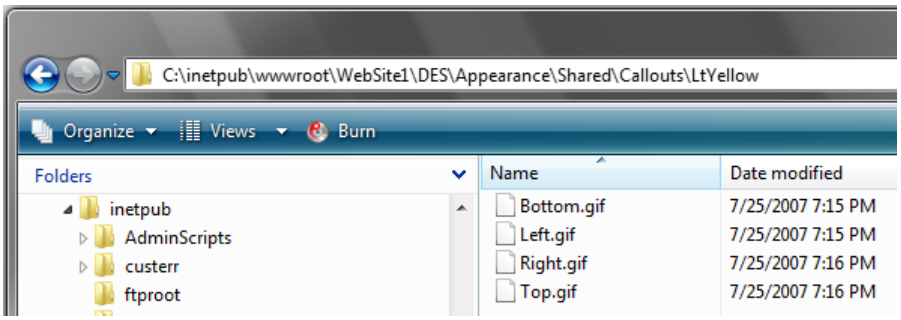
The Callout is an image file. In fact, there are 4 of these image files, one for each direction: left, right, top, and bottom. They have these characteristics:

- 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**.



Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

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. The suggested 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.

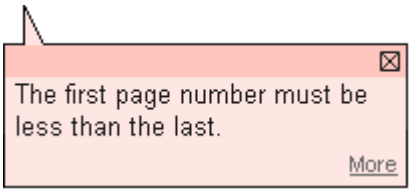
Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Properties for the PeterBlum.DES.Web.WebControls.ErrorFormatterPopupView Class

The `PeterBlum.DES.Web.WebControls.ErrorFormatterPopupView` class contains a PopupView definition. You normally edit these in the **Global Settings** Editor. You can also create them for use with the **PopupErrorFormatter.OverriddenPopupView** property.

See "Working With 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

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

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_PEFOverallLtRed`".

  These styles are declared in **DES/Appearance/Validation/PopupErrorFormatter.css**:

```
.DES_PEFOverallLtRed
{
   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: #ffe4e1; /* mistyrose */
}
/* default font for all nested tables in the control */
.DES_PEFOverallLtRed TABLE
{
   font-family: Arial;
   font-size: 8pt;
}

/* prevent external img styles from affecting these styles */
.DES_PEFOverallLtRed 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_PEFOverallLtRed`

  `DES_PEFOverallLtBlue`

  `DES_PEFOverallLtGray`

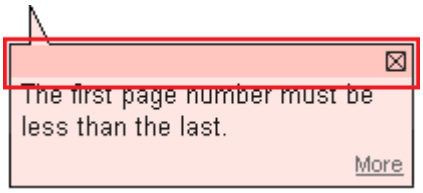
- **Width** (System.Web.UI.WebControls.Unit) – The width of the PopupView (excluding any callouts). The width is a fixed value. The height varies based on hint text.

  Create different width PopupView definitions for any appearance you want.

  It defaults to `200px`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

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_PEFHeaderLtRed`".

  This style is declared in **DES/Appearance/Validation/PopupErrorFormatter.css**:

```
.DES_PEFHeaderLtRed
{
   background-color: #ffc1b9; /* darker version of mistyrose */
   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_PEFHeaderLtYellow`

  `DES_PEFHeaderLtBlue`

  `DES_PEFHeaderLtGray`

- **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`".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

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, "myimage*pressed*.gif". Create the mouseover version and give it the same name, with "MouseOver" added before the extension. For example, myimage*mouseover*.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 "".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- **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 overrided properties in this, but not the entire style.

  If blank, it is not used.

  It defaults to "`DES_CloseButtonLtRed`".

  These styles are declared in **DES/Appearance/Validation/PopupErrorFormatter.css**:

```
.DES_CloseButtonLtRed
{
   cursor: default;
   color: #696969; /* dimgray */
   font-size:8pt;
   background-color:White;
}
.DES_CloseButtonLtRedPressed
{
   color: black;
}
.DES_CloseButtonLtRedMouseOver
{
   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_PEFCloseButtonLtYellow`

  `DES_PEFCloseButtonLtBlue`

  `DES_PEFCloseButtonLtGray`

- **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.

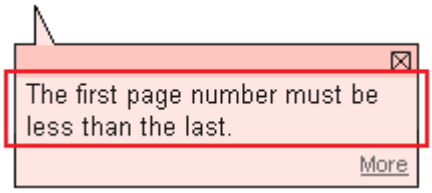
  The String Lookup System lets you define a common set of terms so the programmer doesn't uniquely define them each time. It also provides localization based on the current culture.

  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 "".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

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_PEFBodyLtRed`".

  These styles are declared in **DES/Appearance/Validation/PopupErrorFormatter.css**:

```
.DES_PEFBodyLtRed
{
   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_PEFBodyLtRed hr
{
}
```

  In addition, the style sheet file contains these alternatives (which have identical attributes but are selected depending to the desired color scheme):

  `DES_PEFBodyLtYellow`

  `DES_PEFBodyLtBlue`

  `DES_PEFBodyLtGray`

- **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 **DefaultErrorMessagePopupViewBodyImageUrl** property of the **Global Settings Editor**. Assign BodyImageUrl to "{DEFAULT}" to use the global default. It has the default value of "{APPEARANCE}/Validation/BigValErrorIcon.GIF". DES supplies these image files in the **[WebApplicationRoot]\DES\Appearance\Validation** folder that are useful here:

	ValErrorIcon.GIF
(animated)	ValErrorIcon_animated.GIF
	ValErrorIcon.GIF (from the AJAX Control Toolkit)
	BigValErrorIcon.GIF (This is the default.)
(animated)	BigValErrorIcon_animated.GIF
	BigValErrorIcon2.GIF

  It defaults to "{DEFAULT}".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

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, "myimage*pressed*.gif". Create the mouseover version and give it the same name, with "MouseOver" added before the extension. For example, myimage*mouseover*.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.

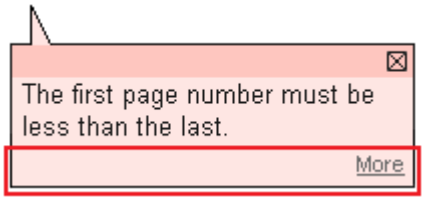
  The String Lookup System lets you define a common set of terms so the programmer doesn't uniquely define them each time. It also provides localization based on the current culture.

  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 "".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

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, "myimage*pressed*.gif". Create the mouseover version and give it the same name, with "MouseOver" added before the extension. For example, myimage*mouseover*.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 overrided properties in this, but not the entire style.

  If blank, it is not used.

  It defaults to "`DES_PEFHelpButtonLtRed`".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

These styles are declared in **DES/Appearance/Validation/PopupErrorFormatter.css**:

```
.DES_PEFHelpButtonLtRed
{
   cursor: default;
   color: #696969; /* dimgray */
   font-size:8pt;
   text-decoration:underline;
}
.DES_PEFHelpButtonLtRedPressed
{
   color: black;
   text-decoration:underline;
}
.DES_PEFHelpButtonLtRedMouseOver
{
   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_PEFHelpButtonLtYellow`

`DES_PEFHelpButtonLtBlue`

`DES_PEFHelpButtonLtGray`

- **FooterCssClass** (string) – The Cascading Style Sheet name that is applied to the footer.

  It defaults to "`DES_PEFFooterLtRed`".

  This style is declared in **DES/Appearance/Validation/PopupErrorFormatter.css**:

```
.DES_PEFFooterLtRed
{
}
```

  In addition, the style sheet file contains these alternatives which just change the background-color attribute:

  `DES_PEFFooterLtYellow`

  `DES_PEFFooterLtBlue`

  `DES_PEFFooterLtGray`

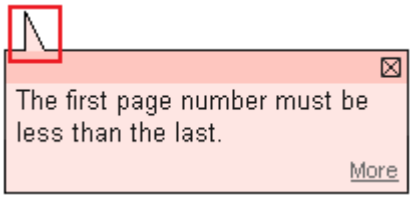
- **FooterHorizontalAlign** (string) – (enum System.Web.UI.WebControls.HorizontalAlign) – The alignment of contents of the footer.

  It defaults to `HorizontalAlign.Right`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

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 messagebox.

  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/Validation/PopupErrorFormatter.css**. They are:

  ```
  {APPEARANCE}/Shared/Callouts/AliceBlue
  {APPEARANCE}/Shared/Callouts/LtRed
  {APPEARANCE}/Shared/Callouts/LtBlue
  {APPEARANCE}/Shared/Callouts/LtRed
  {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 predefine callout files predefined in these properties: 20 tall and 12 wide.

  It defaults to "`{APPEARANCE}/Shared/Callouts/LtMistyRose`" 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 its slightly larger, the entire callout will be moved away from the target. If its 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.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- **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 its slightly larger, the entire callout will be moved away from the target. If its 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).

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

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.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Other Properties

- **HelpBehavior** (enum PeterBlum.DES.HelpBehavior) – Determines how the **ErrorMessageHelp** property (on the control with the hint) is used.

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

  - `None` – Do not use **ErrorMessageHelp**. Do not show a Help Button.
  - `ButtonAppends` – Use the Help Button. When clicked, redraw with the **ErrorMessageHelp** text appended to the current text. The value of **PopupView.AppendHelpSeparator** is inserted between the original hint and the text of **ErrorMessageHelp**.
  - `ButtonReplaces` – Use the Help Button. When clicked, redraw with the **ErrorMessageHelp** text replacing the current text.
  - `Title` – The **ErrorMessageHelp** 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 **ErrorMessageHelp** 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 **ErrorMessageHelp** text will appear in the "{0}" token.
  - `Script` – Runs the script supplied in the **ScriptForHelpButton** property. The **ErrorMessageHelp** 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 ErrorMessageHelp 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 **ErrorMessageHelp** 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 **ErrorMessageHelp** 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 **ErrorMessageHelp** 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 `<br />` and `<hr />`.

  It defaults to "`<hr />`".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- **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`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

- **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.

  When the ASP.NET SmartNavigation feature is enabled on the page, it installs an IFrame and causes the same problem.

  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`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Other Validator Properties that Customize the Appearance

The Validator primarily uses the **ErrorFormatter** property to display itself. Here are a few more properties to customize the appearance.

- **ShowRequiredFieldMarker** (Boolean) – When `true`, the RequiredFieldMarker Control will be inserted to the left of the Validator control automatically. It defaults to `false`.

  It uses the global defaults for the text, image, and tool tip that are set up in the **Global Settings Editor** with these properties described in "Setting up the Global Defaults".

  When the Validator uses the **Enabler** property or is associated with controls that will be hidden and shown by FieldStateControllers, the RequiredFieldMarker normally remains visible. If you prefer it to be hidden, you have these options:

  - Set **PeterBlum.DES.Globals.WebFormDirector.ValidationManager.AutoHideRequiredFieldMarker** to `true`.
  - Substitute the RequiredFieldMarker Control for this property. Then set up a FieldStateController that shows and hides the RequiredFieldMarker control as needed.
  - In any script that changes the condition evaluated by the **Enabler**, call this function:

    ```
    DES_UpdateRFM();
    ```

- **HiliteFields** (PeterBlum.DES.Web.WebControls.ControlConnectionCollection) – Identifies other controls on the page whose style will change when this Validator detects an error. It works in conjunction with "Change the Style of Other Fields Nearby the Error".

  Validators show their error message in the ErrorFormatter and optional ValidationSummary control. Since the goal is to attract the user to the error, this property can identify other controls on the page. These controls will have their style sheet changed when there is an error and return to the default when the error is fixed.

  Choose this to change an enclosing control like a `<div>` or `<table>` to a different background color. Also consider changing the color of the field label.

  By default, **HiliteFields** is not used. You must enable it by setting **HiliteFieldsNearbyError** to `true` on **PeterBlum.DES.Globals.WebFormDirector.ValidationManager** in `Page_Load()` or within the PageManager control. See "Change the Style of Other Fields Nearby the Error".

  When you add textual controls like a Label, TextBox or anything that is formed with a tag, the following style sheet class is used:

  ```
  .DESVALTextHiliteFields
  {
     font-weight: bold;
  }
  ```

  The rest use this style sheet class:

  ```
  .DESVALNonTextHiliteFields
  {
     background-color: #ffebcd; /* blanchedalmond */
  }
  ```

  Note that any **Label** property assigned to a control is automatically included in this list at runtime. So you don't need to set up the labels twice.

  Do not include the actual control being validated here because it is handled automatically elsewhere. See "Change the Style of the Field With the Error".

  Elements of the **HiliteFields** collection must be `PeterBlum.DES.Web.WebControls.ControlConnection` objects. They can reference a control by its ID or an object reference.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Adding ControlConnections with ASP.NET Markup

```
<des:ValidatorClass id="[id]" runat="server" >
  <HiliteFields>
     <des:ControlConnection ControlID="[controlid1]" />
     <des:ControlConnection ControlID="[controlid2]" />
  </HiliteFields>
</des:ValidatorClass>
```

Adding ControlConnections Programmatically

The **HiliteFields** property will create `PeterBlum.DES.Web.WebControls.ControlConnections` for you by calling its `Add()` method. `HiliteFields.Add()` takes a reference to the control object or its ID. (The reference is recommended for performance.)

[C#]

```
Validator1.HiliteFields.Add(Label1);      // reference to control
Validator1.HiliteFields.Add("Table1");    // ID of a control
```

[VB]

```
Validator1.HiliteFields.Add(Label1)       ' reference to control
Validator1.HiliteFields.Add("Table1")     ' ID of a control
```

- **OtherStyles** (PeterBlum.DES.OtherStyles) – Determines the source for style sheets associated with two features that are not controlled by the **ErrorFormatter** property.
  - The **PeterBlum.DES.Globals.WebFormDirector.ValidationManager.ChangeStyleOnControlsWithError** property assigns a style sheet to the control to evaluate when there is an error.
  - The ValidationSummary Control shows the validator's error messages using a specific style sheet defined in its **ErrorMessageCssClass** property.

  Use the **OtherStyles** property to use a second set of style sheets for those features to differentiate them from other controls. For example, differentiating required field errors from other errors.

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

  - `Normal` - **ChangeStyleOnControlsWithError** uses these styles in the **DES\Appearance\Validation\Validation.css** style sheet file: `DESVALFieldWithError`, `DESVALListWithError`, and `DESVALCheckBoxWithError`.

    The ValidationSummary control uses `DESVALSummaryErrors`.

    Here are the defaults for those styles:

```
.DESVALFieldWithError
{
   background-color: #ffb6c1 !important; /* lightpink */
}
.DESVALListWithError
{
   color: red !important;
}
.DESVALCheckBoxWithError
{
   color: red !important;
}
.DESVALSummaryErrors
{
   color: red;
   margin-top:6px;
}
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- `Alternative` - **ChangeStyleOnControlsWithError** uses these styles in the **DES\Appearance\Validation\Validation.css** style sheet file: `DESVALFieldWithErrorAlt`, `DESVALListWithErrorAlt`, and `DESVALCheckBoxWithErrorAlt`.

  The ValidationSummary control uses `DESVALSummaryErrorsAlt`.

  Here are the defaults for those styles:

```
.DESVALFieldWithErrorAlt
{
   background-color: #ffff99 !important; /* light yellow */
}
.DESVALListWithErrorAlt
{
   color: #ff8c00 !important; /* dark orange */
}
.DESVALCheckBoxWithErrorAlt
{
   color: #ff8c00 !important; /* dark orange */
}
.DESVALSummaryErrorsAlt
{
   color: #ff8c00; /* dark orange */
   margin-top:6px;
}
```

When there are two or more validators assigned to the same control to evaluate, so long as any has **OtherStyles**=`Alternative` and is showing an error, the control to evaluate will use the alternative style.

It defaults to `OtherStyles.Normal`.

- **NoErrorFormatter** (PeterBlum.DES.Web.WebControls.NoErrorFormatter) – Shows text and/or an image when there is no error to display. It is commonly used to clearly state that the error has been fixed or there is no error. For example, when the user corrects the error, an image like appears. (That image is supplied with DES. You can use your own as well.) It can also show an indicator that a field has yet to be edited.

  See below for its properties.

  When setup, it appears in the location of the Validator. It can show an image using the **ImageURL** property and text using the **Text** property. It has its own style sheet set in the **CssClass** property. Each can have the text "{DEFAULT}" to tell DES to lookup a global setting that you made with the **Global Settings Editor**. That way, you have one standard appearance for this feature.

  Online example

PeterBlum.DES.Web.WebControls.NoErrorFormatter Class

The `PeterBlum.DES.Web.WebControls.NoErrorFormatter` class is used by the Validator's **NoErrorFormatter** property, described above. It has the following properties.

- **Mode** (enum PeterBlum.DES.Web.WebControls.NoErrorFormatterMode) – Determines when to show the text and image. The enumerated type `PeterBlum.DES.Web.WebControls.NoErrorFormatterMode` has these values:

  - `Off` – Do not use the **NoErrorFormatter**.
  - `Default` – Use the value assigned to the **DefaultNoErrorFormatterMode** property in the **Global Settings Editor**.

    DefaultNoErrorFormatterMode also uses the enumerated type `PeterBlum.DES.Web.WebControls.NoErrorFormatterMode`. It can be set to `Off`, `Always`, `Corrected`, `Validated` and `FirstTime`. Do not set **DefaultNoErrorFormatterMode** to `Default`.
  - `Always` – Always show when there is no error. It will appear as the page is first displayed. It will be removed as an error is revealed. It will return after the error is fixed.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

- `Corrected` – Show after the user fixes the error. It appears after the Validator displays the error and the user corrects it.
- `Validated` – Show after the field has been validated and has no errors. The user does not have to edit to cause this. For example, if a field requires text and opens with text in the textbox, when the user clicks Submit, the RequiredTextValidator will validate and the **NoErrorFormatter** appears.
- `FirstTime` - Show only on a new page, prior to any edits. It is hidden after validation occurs. Never show on a post back page because validation has already occurred. It lets you attract users to fields that have not been edited yet.

It defaults `NoErrorFormatterMode.Default`.

- **CssClass** (string) – The style sheet class name applied to the text and image. When "", the Validator's style sheet class is used.

  When set to "{DEFAULT}", it uses the value assigned to the **DefaultNoErrorFormatterCssClass** property in the **Global Settings Editor**. The **DefaultNoErrorFormatterCssClass** property defaults to "".

  CssClass defaults to "{DEFAULT}".

- **Text** (string) – Assign when you want to display text or HTML.

  When set to "{DEFAULT}", it uses the value assigned to the **DefaultNoErrorFormatterText** property in the **Global Settings Editor**. The **DefaultNoErrorFormatterText** property defaults to "".

  When using the ImageUrl property, you can position the image within the text by using the token "{IMAGE}" here. For example: "`Looks good {IMAGE}.`". If the token is not found, the image will appear before the text.

  Text defaults to "{DEFAULT}".

  Reminder: If your page should follow the XHTML standard, make sure your tags conform to XHTML.

- **TextLookupID** (string) – Gets the value for **Text** 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 ValidationMisc. If no match is found OR this is blank, **Text** 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.

  When "", no string lookup is performed. When set to "{DEFAULT}", it uses the value assigned to the **DefaultNoErrorFormatterTextLookupID** property in the **Global Settings Editor**. The **DefaultNoErrorFormatterTextLookupID** defaults to "", which indicates no lookup.

  TextLookupID defaults to "{DEFAULT}".

- **ImageURL** (string) - The URL to an image file. If assigned, an image will appear before the Text or at the location of the "{IMAGE}" token within the **Text** property. If unassigned, no image will be shown.

  When set to "{DEFAULT}", it uses the global setting **DefaultNoErrorFormatterImageURL** property in the **Global Settings Editor** which defaults to "".

  ImageURL defaults to "{DEFAULT}".

  DES provides this graphic: when you use "`{APPEARANCE}/Validation/ValidIcon.gif`".

  You can assign the alternative text used by the `<img>` tag when the image cannot be loaded to the **RequiredNoErrorFormatterImageAltText** property in the **Global Settings Editor** which defaults to "".

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.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Other Properties

- **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**, **ControlToChange**, **SecondControlIDToEvaluate**, and **SecondControlToEvaluate**
  - Controls in **ControlConnections**.
  - Nested conditions of MultiConditionValidator and CountTrueConditionsValidator. Look at the **ControlIDToEvaluate** and **SecondControlIDToEvaluate** properties of the child conditions.
  - **Enabler**. Look at the **ControlIDToEvaluate** and **SecondControlIDToEvaluate**.
  - **ExtraControlsToRunThisAction**.

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

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

- **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 for PostBack" 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").

- **IsValid** (Boolean) – Determines if validation found errors or not. Test it only after the `Validate()` method has run. Otherwise, it will be a meaningless `true`. After validation runs, it is `true` when there are no errors and `false` when there are errors.

  *Note: Validator controls may not evaluate their conditions for many reasons including the **Enabled** and **Enabler** properties turned off validation, the validation group name did not match the **Group** property, and the Validator's Condition did not have the settings or data to evaluate. In these cases, the Validator control reports that it is valid.*

- **HasValidated** (Boolean) – Determines if the `Validate()`method has been called. It returns `true` when it has.

If you did not find the desired property above, click on any of these topics to jump to them:

- Available Validator controls
- Finding the Validator by the Data Entry Control
- Properties Common to most Conditions
- Changing When the Validator is Evaluated
- Properties for Error Messages
- ErrorFormatters: Customizing the Appearance of the Error Message
- Other Validator Properties that Customize the Appearance
- Other Properties

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Drawing The User's Attention To The Error

The ErrorFormatter is one way to attract the user to the error. However, the message may not be noticed because it's small, off the page, or simply turned off (ErrorFormatter.**Display** property is set to `None`.)

DES provides numerous ways to get the user's attention.

Click on any of these topics to jump to them:

- Set Focus to the Control
- Show an Alert
- Change the Style of the Field With the Error
- Change the Style of Other Fields Nearby the Error
- Blinking the ErrorFormatter
- Warning Mode
- Use the ValidationSummary Control
- Online examples

In addition, there are several hooks available where you can run your own script after validation occurs. These are found on the PageManager control and **PeterBlum.DES.Globals.WebFormDirector.ValidationManager** object:

- **PostPageValidationFunctionName** – Your function is called after page-level validation. It is passed the validation group and validation results. Use it to communicate an error on page submission.
- **PostValidationUpdateScript** – Script called after any validation, whether the field is edited or the page is submitted.
- **CustomSubmitFunctionName** – Your function is called during page submission. It may be called before or after validation depending on the **SubmitOrder** property. This function offers a way to cancel page submission. For example, if you want to display a confirmation message that has a Cancel button, use it here.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Set Focus to the Control

DES can automatically set the focus back to the field containing the error. It offers the following settings:

- **PageManager.FocusOnChange** – Set the focus when the error is detected immediately after a change is made. It is also available on the page level property **PeterBlum.DES.Globals.WebFormDirector.ValidationManager**.
- **PageManager.FocusOnSubmit** – Set the focus when the error is detected as the page is submitted. It sets focus to the first control with an error. It is also available on the page level property **PeterBlum.DES.Globals.WebFormDirector.ValidationManager**.
- **PageManager.FocusAfterAlert** – Set focus after the alert is displayed when using the ErrorFormatters that display alerts: Image with Alert (AlertImageErrorFormatter) and Hyperlink with Alert (HyperLinkErrorFormatter). It is also available on the page level property **PeterBlum.DES.Globals.WebFormDirector.ValidationManager**.
- **ValidationSummary.HyperLinkToFields** – The ValidationSummary control offers the **HyperLinkToFields** property to let the user click on an error message to set the focus to the field associated with that error.

You set the page-level properties in the `Page_Load()` method, the PageManager control, or in the **Global Settings Editor**.

This example sets **FocusOnChange** and **FocusOnSubmit** properties to `true`.

```
<des:PageManager id="PageManager1" runat="server"
   FocusOnChange="true" FocusOnSubmit="true" />
```

Click on any of these topics to jump to them:

- Page-Level Properties
- Online example

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Page-Level Properties

The following properties can be set in the PageManager control or on the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager** object in `Page_Load()`. You can also set global default values for these properties using the **Global Settings Editor**.

- **FocusOnChange** (Boolean) - Set the focus to the field after a change is made when there is a validation error. When `true`, focus is set on the field (unless the field or browser doesn't support setting focus.)

  It defaults to the **DefaultFocusOnChange** property in the **Global Settings Editor**, which defaults to `false`.

- **FocusOnSubmit** (Boolean) – Set focus to the first field that generates a validation error when the page is submitted. When `true`, focus is set on the field (unless the field or browser doesn't support setting focus.)

  It defaults to the **DefaultFocusOnSubmit** property in the **Global Settings Editor**, which defaults to `false`.

- **FocusAfterAlert** (Boolean) – Set focus to a field with an error after the alert is displayed when using the ErrorFormatters that display alerts. Supported by "Image with Alert" (AlertImageErrorFormatter) and "HyperLink with Alert" (HyperLinkErrorFormatter). When `true`, focus is set to the field after the alert is dismissed.

  It defaults to the **DefaultFocusAfterAlert** property in the **Global Settings Editor**, which defaults to `false`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Show an Alert

DES offers four ways to show an alert with error messages.

1. If you want the user to click on a button to show the alert, use the "Image with Alert" ErrorFormatter in the **ErrorFormatter** property of the Validator control.
2. If you want the user to click on a hyperlink to show the alert, use the "Hyperlink with Alert" ErrorFormatter in the **ErrorFormatter** property of the Validator control.
3. To show an alert automatically after the user changes the field, set the **PageManager.ShowAlertOnChange** property to `true`. It will consolidate all error messages associated with the field that was edited.
4. To show an alert automatically when the user submits the page, set the **PageManager.ShowAlertOnSubmit** property to `true`. It will also show an alert if server side validation found an error. In this case, it shows the alert after the page is reloaded. It will consolidate all error messages associated with the validation group that is being submitted. This alert can be delayed until the user attempted to submit the page several times with an error by using **PageManager.DelayAlertOnSubmitCount**.
5. To show an alert after server side validation finds an error, set the **PageManager.ShowAlertAfterPostBackError** property to true. The alert will be shown as the page is reloaded. While **PageManager.ShowAlertOnSubmit** also does this, **ShowAlertAfterPostBackError** does not show the alert before postback. Users will appreciate this as they have cleaned up all client-side errors using the ValidationSummary control and the alert draws there attention to new errors found by server side validation.

ShowAlertOnChange and **ShowAlertOnSubmit** can be formatted using these PageManager control properties: **AlertTemplate**, **AlertErrorListStyle**, and **AlertErrorLeadText**.

This example sets **ShowAlertOnChange** and **ShowAlertOnSubmit** properties to `true`.

```
<des:PageManager id="PageManager1" runat="server"
   ShowAlertOnChange="true" ShowAlertOnSubmit="true" />
```

Click on any of these topics to jump to them:

- Page-Level Properties
- Online example

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Page-Level Properties

The following properties can be set in the PageManager control or on the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager** object in `Page_Load()`. You can also set global default values for these properties using the **Global Settings Editor**.

- **ShowAlertOnChange** (Boolean) – Show an alert after a validation error caused by an onchange event. The alert includes all the error messages associated with the field that was changed. When `true`, the alert is shown. It uses the property **AlertTemplate** to apply further formatting. Messages are formatted in list or sentence style using **AlertErrorListStyle** and given **AlertErrorLeadText** in front of each message.

  If you don't assign it on the page, it uses the value from the **DefaultShowAlertOnChange** property in the **Global Settings Editor**, which defaults to `false`.

  Validators have two sources for error messages: the **ErrorMessage** and **SummaryErrorMessage** properties. The error message shown depends on the **AlertShowsSummaryMessage** property.

- **ShowAlertOnSubmit** (Boolean) – Show an alert when the page is submitted and errors are found. The alert includes all of the error messages associated with the submit control's validation group. When `true`, the alert is shown. It uses the property **AlertTemplate** to apply further formatting. Messages are formatted in list or sentence style using **AlertErrorListStyle** and given **AlertErrorLeadText** in front of each message.

  An alert can show either before or after post back. When Validators can evaluate on the client-side, they appear on an alert prior to post back. Any Validators found to have errors after post back will be displayed in an alert that appears immediately after post back generates the page. If you don't want the alert to appear prior to postback, use the **ShowAlertAfterPostBackError** property instead.

  If you don't assign it on the page, it uses the value from the **DefaultShowAlertOnSubmit** property in the **Global Settings Editor**, which defaults to `false`.

  This alert can be delayed until the user attempted to submit the page several times with an error by using **DelayAlertOnSubmitCount**. Use it when you normally wouldn't show an alert because it helps the user when they click the submit button several times without noticing the errors that are preventing the page from being submitted.

- **DelayAlertOnSubmitCount** (int) – When using **ShowAlertOnSubmit**, you can delay the alert from appearing until the user has attempted to submit several times. Use it when you normally wouldn't show an alert because it helps the user when they click the submit button several times without noticing the errors that are preventing the page from being submitted.

  When 0, the alert shows each time. When 1, it shows on the second attempt to submit. When 2, it shows on the third attempt, etc. Make sure **ShowAlertOnSubmit** is `true` or this feature is not used.

  If you don't assign it on the page, it uses the value from the **DefaultDelayAlertOnSubmitCount** property in the **Global Settings Editor**, which defaults to `0`.

- **ShowAlertAfterPostBackError** (Boolean) - Shows an alert after server side validation finds an error. The alert will appear as the page is reloaded and draw the user's attention to new errors. While **ShowAlertOnSubmit** also does this, **ShowAlertAfterPostBackError** does not show the alert before postback. This helps when you build a user interface that users the ValidationSummary to draw users to errors. After they have cleaned up all client-side errors, they are not expecting new errors. The alert draws there attention to new errors found by server side validation.

  If you don't assign it on the page, it uses the value from the **DefaultShowAlertAfterPostBackError** property in the **Global Settings Editor**, which defaults to `false`.

- **AlertTemplate** (string) – Used in the alerts shown by the **ShowAlertOnChange** and **ShowAlertOnSubmit** properties. It provides a textual framework for building a single description from the list of error messages associated with the fields that have errors. If it is blank, you will get only a list of error messages. Otherwise, use the token "{0}" in this string. It will be replaced by the error messages.

  The string from **AlertErrorLeadText** will precede each error message. The messages are shown in a list or sentence style based on **AlertErrorListStyle**.

  You can use these tokens in the text:

  {COUNT} – The number of error messages listed.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

{COUNT:singular:plural} – Helps build sentences where singular and plural forms are needed when you use the {COUNT} token. For example, "There is 1 error." and "There are 2 errors.". You replace the term "singular" with the singular form of the word. You replace the term "plural" with the plural form of the word. For example, "{COUNT:error:errors}".

If you don't assign it on the page, it uses the value from the **DefaultAlertTemplate** property in the **Global Settings Editor**, which defaults to "`Please correct these errors:\n{0}`". (The symbol "\n" represents a carriage return in JavaScript strings.)

- **AlertTemplateLookupID** (string) – Gets the value for **AlertTemplate** 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 ValidationMisc. If no match is found OR this is blank, **AlertTemplate** 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 "".

- **AlertErrorLeadText** (string) – Used in the alerts shown by the **ShowAlertOnChange**, **ShowAlertOnSubmit**, and **ShowAlertAfterPostBackError** properties. This text is inserted before each error message. Use it to denote a new message. For example, "-" or "*". If you use a "#" symbol, that character will be replaced by a number that represents the error message's position in the list.

  If you don't assign it on the page, it uses the value from the **DefaultAlertErrorLeadText** property in the **Global Settings Editor**, which defaults to "#. ".

- **AlertErrorListStyle** (Boolean) – Used in the alerts shown by the **ShowAlertOnChange**, **ShowAlertOnSubmit**, and **ShowAlertAfterPostBackError** properties. When `true`, error messages are listed on separate lines. When `false`, error messages are listed in a single paragraph style.

  If you don't assign it on the page, it uses the value from the **DefaultAlertErrorListStyle** property in the **Global Settings Editor**, which defaults to `true`.

- **AlertShowsSummaryMessage** (Boolean) – Used by the **ShowAlertOnChange** property to determine the source of the error message on a validator: from its **ErrorMessage** or **SummaryErrorMessage** property. When `True`, **SummaryErrorMessage** is used. (If not assigned, **ErrorMessage** is used.) When `False`, the **ErrorMessage** is used.

  If you don't assign it on the page, it uses the value from the **DefaultAlertShowsSummaryMessage** property in the **Global Settings Editor**, which defaults to `true`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Change the Style of the Field With the Error

A very effective way to get the users attention is to change the background or text color of a data entry field with an error.

Click on any of these topics to jump to them:

- Alternative Styles
- Using This Feature
- Page-Level Properties
- Online example

Since textboxes, checkboxes, radiobuttons, listboxes and dropdownlists all have different looks, you can set up to 3 style sheet class names:

- For listboxes and dropdownlists
- For checkboxes and radiobuttons. With these you can also determine if the input field, label or both have the style applied.
- For all remaining controls including textboxes.

DES predefines style sheets for you in **DES\Appearance\Validation\Validation.css**:

```
.DESVALListWithError
{
   color: red;
}
.DESVALCheckBoxWithError
{
   color: red;
}
.DESVALFieldWithError
{
   background-color: #ffb6c1; /* lightpink */
}
```

They are automatically used by DES, unless you change the values of **ControlErrorCssClass**, **ListErrorCssClass**, and **CheckBoxErrorCssClass**. You only need to enable the feature by setting **ChangeStyleOnControlsWithError** to `true`. All of these properties are set either on the PageManager control or **PeterBlum.DES.Globals.WebFormDirector.ValidationManager** property in `Page_Load()`.

These style sheet classes are merged with the existing style sheet class of the control. So only edit the style sheet attributes that should change, not the overall look.

This example enables the feature using the default style sheet classes:

```
<des:PageManager id="PageManager1" runat="server"
   ChangeStyleOnControlsWithError="true" />
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Alternative Styles

Sometimes you want a validator to use an alternative style to differentiate it. For example, it is common to differentiate required validators from others. This feature supports an alternative style by setting the **OtherStyles** property to `Alternative` on the Validator control.

```
<des:ValidatorClass id="Validator1" runat="server"
   OtherStyles="Alternative" [other properties] />
```

Once done, it uses these style sheet classes in **DES\Appearance\Validation\Validation.css**:

```
.DESVALListWithErrorAlt
{
   color: #ff8c00; /* dark orange */
}

.DESVALCheckBoxWithErrorAlt
{
   color: #ff8c00; /* dark orange */
}

.DESVALFieldWithErrorAlt
{
   background-color: #ffff99; /* light yellow */
}
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Using This Feature

1. Enable the feature by setting **ChangeStyleOnControlsWithError** to `true` on the PageManager control or the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager** property in `Page_Load()`.

```
<des:PageManager id="PageManager1" runat="server"
 ChangeStyleOnControlsWithError="true" />
```

2. Evaluate how the default styles look. If not OK, continue.
3. Edit the style sheet classes in **DES\Appearance\Validation\Validation.css**. These styles will merge with the control's current style sheet. So only provide style sheet attributes that differentiate the control from its basic style.

Control type	Style sheet class
Listboxes and dropdownlists	`DESVALListWithError`
Checkboxes and radiobuttons	`DESVALCheckBoxWithError`
All remaining controls including textboxes	`DESVALFieldWithError`

4. If desired, you can create your own style sheet classes. Set the following properties to your classes. They are on the PageManager control or the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager** property in `Page_Load()`.

Control type	PropertyName
Listboxes and dropdownlists	ListErrorCssClass
Checkboxes and radiobuttons	CheckBoxErrorCssClass
All remaining controls including textboxes	ControlErrorCssClass

If your style sheet class merges with the control's current style sheet class, put a plus ("+") character first. For example, "+MyClassName".

5. Test the appearance. If your checkboxes and radiobuttons don't have the desired look, see if the **CheckBoxECCMode** property can help. It is set on the PageManager control or the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager** property in `Page_Load()`.
6. For any validator that should use alternative style sheet classes, set its **OtherStyles** property to `Alternative`. It will use these style sheet classes:

Control type	Style sheet class
Listboxes and dropdownlists	`DESVALListWithErrorAlt`
Checkboxes and radiobuttons	`DESVALCheckBoxWithErrorAlt`
All remaining controls including textboxes	`DESVALFieldWithErrorAlt`

```
<des:ValidatorClass id="Validator1" runat="server"
  OtherStyles="Alternative" [other properties] />
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Page-Level Properties

The following properties can be set in the PageManager control or on the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager** object in `Page_Load()`. You can also set global default values for these properties in the "Changing the Field Appearance" topic of the **Global Settings Editor**.

- **ChangeStyleOnControlsWithError** (Boolean) – Determines if controls change their appearance to show an error. By default, the style sheet classes "`DESVALFieldWithError`", "`DESVALListWithError`", and "`DESVALCheckBoxWithError`" determine their appearance. When the validator's **OtherStyles** property is set to `Alternative`, it uses these style sheet classes by default: "`DESVALFieldWithErrorAlt`", "`DESVALListWithErrorAlt`", and "`DESVALCheckBoxWithErrorAlt`".

  Usually you edit these to change their font color or background, although its up to you to determine the desired effect.

  If you want to choose alternative style sheet class names, set them in these properties: **ControlErrorCssClass**, **ListErrorCssClass**, and **CheckBoxErrorCssClass**. (See below.)

  These styles are declared in **DES\Appearance\Validation\Validation.css**:

```
.DESVALFieldWithError
{
   background-color: #ffb6c1; /* lightpink */
}
.DESVALListWithError
{
   color: red;
}
.DESVALCheckBoxWithError
{
   color: red;
}
.DESVALFieldWithErrorAlt
{
   background-color: #ffff99; /* light yellow */
}
.DESVALListWithErrorAlt
{
   color: #ff8c00; /* dark orange */
}
.DESVALCheckBoxWithErrorAlt
{
   color: #ff8c00; /* dark orange */
}
```

  These style sheet classes are merged with the existing style sheet class of the control. So only edit the style sheet attributes that should change, not the overall look.

  When `true`, the style is changed. If you do not assign it, its initial value comes from the global setting **DefaultChangeStyleOnControlsWithError** which starts at `false`.

- **ControlErrorCssClass** (string) – When **ChangeStyleOnControlsWithError** is `true`, the style sheet class name assigned to all of the controls that generate a validation error. The original class is restored when the error is fixed. No style sheet changes are made when this property is "".

  To merge your style sheet with the current style sheet class, put a plus ("+") character first. For example, "+MyClassName". Your style sheet class's attributes will override any attributes that it shares with the original class.

  It is overridden by the **ListErrorCssClass** and **CheckBoxCssClass** properties, when they are assigned to a class name. Those properties cover ListBox, DropDownList, CheckBox, and RadioButton controls. For all other controls, including textboxes, **ControlErrorCssClass** is always used.

  Unless you edit this property, **ControlErrorCssClass** uses the global setting **DefaultControlErrorCssClass**, which defaults to "+`DESVALFieldWithError`".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

The default style is declared in **DES\Appearance\Validation\Validation.css**:

```
.DESVALFieldWithError
{
   background-color: #ffb6c1; /* lightpink */
}
```

- **ListErrorCssClass** (string) – When **ChangeStyleOnControlsWithError** is `true`, the style sheet class name assigned to ListBox and DropDownList controls that generate a validation error. The original class is restored when the error is fixed. No style sheet changes are made when this property is "".

  To merge your style sheet with the current style sheet class, put a plus ("+") character first. For example, "+MyClassName". Your style sheet class's attributes will override any attributes that it shares with the original class.

  When "", the class name in **ControlErrorCssClass** is used on ListBoxes and DropDownLists. If both **ListErrorCssClass** and **ControlErrorCssClass** are "", these controls do not change their styles on error.

  Unless you edit this property, **ListErrorCssClass** uses the global setting **DefaultListErrorCssClass**, which defaults to "+`DESVALListWithError`".

  The default style is declared in **DES\Appearance\Validation\Validation.css**:

```
.DESVALListWithError
{
   color: red;
}
```

- **CheckBoxErrorCssClass** (string) – When **ChangeStyleOnControlsWithError** is `true`, the style sheet class name assigned to CheckBox and RadioButton controls that generate a validation error. The original class is restored when the error is fixed. No style sheet changes are made when this property is "".

  To merge your style sheet with the current style sheet class, put a plus ("+") character first. For example, "+MyClassName". Your style sheet class's attributes will override any attributes that it shares with the original class.

  When "", the class name in **ControlErrorCssClass** is used on CheckBoxes and RadioButtons. If both **CheckBoxErrorCssClass** and **ControlErrorCssClass** are "", these controls do not change their styles on error.

  Note: CheckBoxErrorCssClass does not apply to CheckBoxList or RadioButtonList controls. Their container tag is always assigned the style sheet of ControlErrorCssClass.

  Checkboxes and RadioButtons have two parts: their input field and label. Use the **CheckBoxECCMode** property to determine which of them have the class applied.

  Unless you edit this property, **CheckBoxErrorCssClass** uses the global setting **DefaultCheckBoxErrorCssClass**, which defaults to "+`DESVALCheckBoxWithError`".

  The default style is declared in **DES\Appearance\Validation\Validation.css**:

```
.DESVALCheckBoxWithError
{
   color: red;
}
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- **CheckBoxECCMode** (integer) – Checkboxes and RadioButtons have two parts: their input field and label. Use this to determine which of those have the class name applied when using the **CheckBoxErrorCssClass** and **ControlErrorCssClass** properties.

  It has these values:

  - 0 = Both input field and label
  - 1 = Input field
  - 2 = Label

  Unless you edit this property, **CheckBoxECCMode** uses the global setting **DefaultCheckBoxECCMode**, which defaults to 0.

  CheckBoxECCMode only affects client-side code that updates the classes. ASP.NET does not provide enough information to assign the class on the two parts (input and label). After post back detects errors and applies the class, the entire CheckBox and RadioButton control will be assigned to the class. The input field of HtmlInputCheckBox and HtmlInputRadioButton will be assigned to the class.

- **ControlErrorCssClass_Alt** (string) – When **ChangeStyleOnControlsWithError** is `true`, the style sheet class name assigned to all of the controls that generate a validation error when their **OtherStyles** property is set to `Alternative`. The original class is restored when the error is fixed. No style sheet changes are made when this property is "".

  To merge your style sheet with the current style sheet class, put a plus ("+") character first. For example, "+MyClassName". Your style sheet class's attributes will override any attributes that it shares with the original class.

  It is overridden by the **ListErrorCssClass_Alt** and **CheckBoxCssClass_Alt** properties, when they are assigned to a class name. Those properties cover ListBox, DropDownList, CheckBox, and RadioButton controls. For all other controls, including textboxes, **ControlErrorCssClass_Alt** is always used.

  Unless you edit this property, **ControlErrorCssClass_Alt** uses the global setting **DefaultControlErrorCssClass_Alt**, which defaults to "+`DESVALFieldWithErrorAlt`".

  The default style is declared in **DES\Appearance\Validation\Validation.css**:

  ```
  .DESVALFieldWithErrorAlt
  {
     background-color: #ffff99; /* light yellow */
  }
  ```

- **ListErrorCssClass_Alt** (string) – When **ChangeStyleOnControlsWithError** is `true`, the style sheet class name assigned to ListBox and DropDownList controls that generate a validation error. The original class is restored when the error is fixed. No style sheet changes are made when this property is "".

  To merge your style sheet with the current style sheet class, put a plus ("+") character first. For example, "+MyClassName". Your style sheet class's attributes will override any attributes that it shares with the original class.

  When "", the class name in **ControlErrorCssClass_Alt** is used on ListBoxes and DropDownLists. If both **ListErrorCssClass_Alt** and **ControlErrorCssClass_Alt** are "", these controls do not change their styles on error.

  Unless you edit this property, **ListErrorCssClass_Alt** uses the global setting **DefaultListErrorCssClass_Alt**, which defaults to "+`DESVALListWithErrorAlt`".

  The default style is declared in **DES\Appearance\Validation\Validation.css**:

  ```
  .DESVALListWithErrorAlt
  {
     color: #ff8c00; /* dark orange */
  }
  ```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- **CheckBoxErrorCssClass_Alt** (string) – When **ChangeStyleOnControlsWithError** is `true`, the style sheet class name assigned to CheckBox and RadioButton controls that generate a validation error. The original class is restored when the error is fixed. No style sheet changes are made when this property is "".

  To merge your style sheet with the current style sheet class, put a plus ("+") character first. For example, "+MyClassName". Your style sheet class's attributes will override any attributes that it shares with the original class.

  When "", the class name in **ControlErrorCssClass_Alt** is used on CheckBoxes and RadioButtons. If both **CheckBoxErrorCssClass_Alt** and **ControlErrorCssClass_Alt** are "", these controls do not change their styles on error.

  Note: CheckBoxErrorCssClass_Alt does not apply to CheckBoxList or RadioButtonList controls. Their container tag is always assigned the style sheet of ControlErrorCssClass_Alt.

  Checkboxes and RadioButtons have two parts: their input field and label. Use the **CheckBoxECCMode** property to determine which of them have the class applied.

  Unless you edit this property, **CheckBoxErrorCssClass_Alt** uses the global setting **DefaultCheckBoxErrorCssClass_Alt**, which defaults to "+`DESVALCheckBoxWithErrorAlt`".

  The default style is declared in **DES\Appearance\Validation\Validation.css**:

```
.DESVALCheckBoxWithErrorAlt
{
   color: #ff8c00; /* dark orange */
}
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Change the Style of Other Fields Nearby the Error

When a Validator detects an error, it can change the style of the label and other controls around the field with the error.

Click on any of these topics to jump to them:

- Using This Feature
- Page-Level Properties
- Online example

For example, suppose you have a textbox with a label and its enclosed in a `<div>` like this:

```
<div width="100%">
  <asp:Label id="Label1" runat="server">Name</asp:Label>
  <asp:TextBox id="TextBox1" runat="server"></asp:TextBox>
  <des:RequiredTextValidator id="RequiredTextValidator1" runat="server"
    ControlIDToEvaluate="TextBox1" ErrorMessage="Required">
  </des:RequiredTextValidator>
</div>
```

With DES, you can use the "HiliteFields" feature to do the following:

- Assign a style, such as a font color, to the label.
- Assign a style, such as a background color, to the enclosing `<div>`
- Assign a style to any other control

DES will switch to your style when the error is shown and switch back when the error is corrected.

Normal:

Name

When the error is shown:

Name Required

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Using This Feature

Here is how to set up the HiliteFields feature.

1. Enable the feature by setting **HiliteFieldsNearbyError** to `true` on the PageManager control or on the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager** property in `Page_Load()`.

```
<des:PageManager id="PageManager1" runat="server"
 HiliteFieldsNearbyError="true" />
```

2. To set up label controls, edit the Validator's **Label** property to reference the Label control. (This feature supports **Label**, **SecondLabel**, **Label2**, **Label3**, and **Label4** properties.) Specifically, the Label's **LabelControlID** or **LabelControl** property must be assigned. Labels will automatically use the style sheet class defined in **TextHiliteFieldsCssClass**. See "Properties for Error Messages".

```
<div width="100%">
  <asp:Label id="Label1" runat="server">Name</asp:Label>
  <asp:TextBox id="TextBox1" runat="server"></asp:TextBox>
  <des:RequiredTextValidator id="RequiredTextValidator1" runat="server"
     ControlIDToEvaluate="TextBox1" ErrorMessage="Required"
     Label-LabelControlID="Label1">
  </des:RequiredTextValidator>
</div>
```

3. For each additional control whose styles you want to change, be sure that the control has `runat="server"` and an `ID=` property.

```
<div id="divTextBox1" runat="server" width="100%">
  <asp:Label id="Label1" runat="server">Name</asp:Label>
  <asp:TextBox id="TextBox1" runat="server"></asp:TextBox>
  <des:RequiredTextValidator id="RequiredTextValidator1" runat="server"
     ControlIDToEvaluate="TextBox1" ErrorMessage="Required"
     Label-LabelControlID="Label1">
  </des:RequiredTextValidator>
</div>
```

4. Add each additional control into the **HiliteFields** property on the Validator control. You can add as many as you like. The Properties Editor provides an easy interface for adding controls.

Adding to HiliteFields using ASP.NET Markup

```
<des:RequiredTextValidator id="RequiredTextValidator1" runat="server"
      ControlIDToEvaluate="TextBox1" ErrorMessage="Required"
      Label-LabelControlID="Label1">
   <HiliteFields>
      <des:ControlConnection ControlID="divTextBox1" />
   </HiliteFields>
</des:RequiredTextValidator>
```

Adding to HiliteFields Programmatically

Pass either the control ID or its object into the `HiliteFields.Add()` method like this:

```
Validator1.HiliteFields.Add("ID")
Validator1.HiliteFields.Add(Control)
```

STEPS CONTINUE ON THE NEXT PAGE

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

5. Run the form and evaluate how the default styles look. If changes are needed, continue with the next step.
6. Edit the style sheet classes in **DES\Appearance\Validation\Validation.css**. These styles will merge with the control's current style sheet. So only provide style sheet attributes that differentiate the control from its basic style.

Control type	Style sheet class
Labels, TextBoxes, CheckBoxes, and RadioButtons	`DESVALTextHiliteFields`
All remaining controls including ListBoxes	`DESVALNonTextHiliteFields`

Alternatively, create your own style sheet classes. Set the following properties to your classes. They are on the PageManager control or on **PeterBlum.DES.Globals.WebFormDirector.ValidationManager** property in `Page_Load()`.

Control type	PropertyName
Labels, TextBoxes, CheckBoxes, and RadioButtons	**TextHiliteFieldsCssClass**
All remaining controls including ListBoxes	**NonTextHiliteFieldsCssClass**

If your style sheet class merges with the control's current style sheet class, put a plus ("+") character first. For example, "+MyClassName".

Complete Example

Here is the HTML from above with HiliteFields support for a Label control and `<div>` tag.

```
<des:PageManager id="PageManager1" runat="server"
  HiliteFieldsNearbyError="true" />

<div id="divTextBox1" runat="server" width="100%">
  <asp:Label id="Label1" runat="server">Name</asp:Label>
  <asp:TextBox id="TextBox1" runat="server"></asp:TextBox>
  <des:RequiredTextValidator id="RequiredTextValidator1" runat="server"
    ControlIDToEvaluate="TextBox1" ErrorMessage="Required"
    Label-LabelControlID="Label1">
     <HiliteFields>
        <des:ControlConnection ControlID="divTextBox1" />
     </HiliteFields>
  </des:RequiredTextValidator>
</div>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Page-Level Properties

The following properties can be set in the PageManager control or on the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager** object in `Page_Load()`. You can also set global default values for these properties in the "Changing the Field Appearance" topic of the **Global Settings Editor**.

- **HiliteFieldsNearbyError** (Boolean) – Determines if fields identified by the validator's **Label** and **HiliteFields** properties will change their appearance when the Validator detects an error.

  By default, the style sheet classes "`DESVALTextHiliteFields`" and "`DESVALNonTextHiliteFields`" are where you set their appearance. The first handles textual controls such as Label, TextBox, CheckBox, and RadioButton. The other handles the rest (ListBox, DropDownList, etc).

  Usually you edit these to change their font color or background, although its up to you to determine the desired effect.

  If you want to choose alternative style sheet class names, set them in these properties: **TextHiliteFieldsCssClass** and **NonTextHiliteFieldsCssClass**.

  These styles are declared in **DES\Appearance\Validation\Validation.css**:

```
.DESVALTextHiliteFields
{
   font-weight: bold !important;
}
.DESVALNonTextHiliteFields
{
   background-color: #ffebcd !important; /* blanchedalmond */
}
```

  These style sheet classes are merged with the existing style sheet class of the control. So only edit the style sheet attributes that should change, not the overall look.

  When `true`, the style is changed. If you do not assign it, its initial value comes from the global setting **DefaultHiliteFieldsNearbyError** which starts at `false`.

- **TextHiliteFieldsCssClass** (string) – The style sheet class name assigned to textual controls that appear in the **HiliteFields** and **Label** properties on Validator controls when there is a validation error. The original class is restored when no error is detected.

  The following control classes and their subclasses are considered textual fields: System.Web.UI.WebControls.Label, System.Web.UI.WebControls.TextBox, System.Web.UI.WebControls.CheckBox, System.Web.UI.HtmlControls.HtmlInputText, System.Web.UI.HtmlControls.HtmlInputCheckBox, System.Web.UI.HtmlControls.HtmlInputRadioButton, and the tag. (DES's TextBoxes and LocalizableLabel are subclasses and considered textual.) All other controls are non-textual.

  When "", it does not cause any style changes to the textual controls in **HiliteFields** or **Label** properties.

  To merge your style sheet with the current style sheet class, put a plus ("+") character first. For example, "+MyClassName". Your style sheet class's attributes will override any attributes that it shares with the original class.

  Unless you edit this property, **TextHiliteFieldsCssClass** uses the global setting **DefaultTextHiliteFieldsCssClass**, which defaults to "+`DESVALTextHiliteFields`".

  The default style is declared in **DES\Appearance\Validation\Validation.css**:

```
.DESVALTextHiliteFields
{
   font-weight: bold !important;
}
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- **NonTextHiliteFieldsCssClass** (string) – The style sheet class name assigned to non-textual controls that appear in the **HiliteFields** properties on Validator controls when there is a validation error. The original class is restored when no error is detected.

  When "", it does not cause any style changes to the non-textual controls in **HiliteFields** properties.

  To merge your style sheet with the current style sheet class, put a plus ("+") character first. For example, "+MyClassName". Your style sheet class's attributes will override any attributes that it shares with the original class.

  Unless you edit this property, **NonTextHiliteFieldsCssClass** uses the global setting **DefaultNonTextHiliteFieldsCssClass**, which defaults to "+`DESVALNonTextHiliteFields`".

  The default style is declared in **DES\Appearance\Validation\Validation.css**:

```
.DESVALNonTextHiliteFields
{
   background-color: #ffebcd !important; /* blanchedalmond */
}
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Blinking the ErrorFormatter

You can make your ErrorFormatter blink on the page to get the user's attention. DES generates the blinking effect by switching between two style sheet classes.

Blinking is established on the **BlinkOnChange** and **BlinkOnSubmit** properties in the PageManager control or on the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager** object in `Page_Load()`.

This example sets **BlinkOnChange** to blink three times and **BlinkOnSubmit** to blink continuously.

```
<des:PageManager id="PageManager1" runat="server"
  BlinkOnChange="Three" BlinkOnChange="On" />
```

Each Validator can control blinking with the **BlinkAllowed** property and dictate the style sheet used with the **BlinkCssClass** properties. **BlinkAllowed** defaults to `true`. Generally you use it to turn off blinking on an individual validator. BlinkCssClass is only changed when the appearance of the error message looks incorrect and could use a different style sheet to make it blink.

Click on any of these topics to jump to them:

- Page-Level Properties
- Properties on the Validator
- Online example

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Page-Level Properties

They can be set in the PageManager control or on the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager** object in `Page_Load()`. You can also set global default values for these properties in the "Changing the Field Appearance" topic of the **Global Settings Editor**.

- **BlinkOnChange** (enum PeterBlum.DES.Web.BlinkMode) - Enables the error to blink to catch the user's attention after a change event. It can blink continuously or a specific number of times. The enumerated type `PeterBlum.DES.Web.BlinkMode` has these values:
    - `Off` – It does not blink.
    - `On` – It blinks continuously
    - `One` – It blinks once
    - `Two` – It blinks twice
    - `Three` – It blinks three times
    - `Five` – It blinks five times
    - `Ten` – It blinks ten times

  If you do not assign it, its initial value comes from the **DefaultBlinkOnChange** property in the **Global Settings Editor**, which defaults to `BlinkMode.Off`.

- **BlinkOnSubmit** (enum PeterBlum.DES.Web.BlinkMode) - Enables the error to blink to catch the user's attention after a submit event. (See above for the values of this enumerated type.)

  If you do not assign it, its initial value comes from the **DefaultBlinkOnSubmit** property in the **Global Settings Editor**, which defaults to `Off`.

- **BlinkTime** (Integer) - The number of milliseconds to hold each state of a blinking object.

  If you do not assign it, its initial value comes from the **DefaultBlinkTime** property in the **Global Settings Editor**, which defaults to 1000 (1 second).

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Properties on the Validator

- **BlinkAllowed** (Boolean) – When `true`, blinking is allowed on this error message. Generally you set this to `false` when blinking does not apply to the individual error message. It defaults to `true`.

  Note: The hyperlink text shown in the HyperLinkErrorFormatter does not blink. However, the rest of it – BeforeHTML, AfterHTML and ImageUrl – do blink.

- **BlinkCssClass** (string) – A style sheet class name for the blinked state of the error message. When blinking, the client side switches HTML 'class' attribute of the `<span>` tags that enclose the error message between this and its original value (in **CssClass**). If this is blank, the client side switches between transparent text and its original text.

  It defaults to "`DESVALBlinkText`".

  The default style is declared in **DES\Appearance\Validation\Validation.css**:

  ```
  .DESVALBlinkText
  {
     color: White;
  }
  ```

  Note: When using the Text ErrorFormatter, if you want to blink the image but not the text, set BlinkCssClass = "".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Warning Mode on Validators

Validators can be used as warnings instead of errors when you set their **EventsThatValidate** property to `WarningMode`. A warning looks like a validator error, but it does not prevent submitting or saving the page. For example, you may prefer the user to enter a birthdate that is at least 18 years old and if they don't tell them they need a parent to come to the appointment with them.

```
<des:CompareToValueValidator id="ValidatorControl1" runat="server"
   ControlIDToEvaluate="Birthdate"
   Operator="LessThan" ValueToCompare="<%# DateTime.Today.AddYears(18) %>"
   ErrorMessage="Parent is required for visit"
   EventsThatValidate="WarningMode" />
```

To remind the user of these warnings, you can show a confirmation message that lists the warnings on the page. When **WarningConfirmOnSubmit** is `true`, the confirmation appears when the user attempts to submit the page. The user can click OK to submit the page or Cancel to prevent submission. The warnings will be on the page at that time for their review.

The message is a combination of a few things. Text before and after the list of warnings is established with the **WarningConfirmTemplate** property. By default, it has a header of "Are you sure you want to submit the page with these issues remaining?". Since multiple warnings can appear, they are each preceded by the text from the **WarningConfirmLeadText** property, which by default shows incremental numbers.

This example sets **WarningConfirmOnSubmit** and defines a message:

```
<des:PageManager id="PageManager1" runat="server"
   WarningConfirmOnSubmit="true"
   WarningConfirmTemplate="Do you want to fix these issues now?\n\n{0}" />
```

Submission has an order of actions between validating, showing confirm messages, and running custom scripts as determined by the **SubmitOrder** property. This confirmation feature is run along immediately after the confirmation message established with the **ConfirmMessage** property.

Page-Level Properties

They can be set in the PageManager control or on the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager** object in `Page_Load()`. You can also set global default values for these properties in the "Confirmation for Validators as Warnings" topic of the **Global Settings Editor**.

- **WarningConfirmOnSubmit** (Boolean) – Determines if any warning messages from validators (where **Validator.EventsThatValidate**=`OnChange`) will show in a confirm message.

  The confirmation message is only shown when the user attempts to submit the page. It occurs in the **SubmitOrder** immediately after the normal confirm message does.

  Use the **WarningsConfirmTemplate** to define text appearing before and after the warning messages. It inserts the warning messages into the "`{0}`" token of the **WarningsConfirmTemplate** property, or at the end if no `{0}` token exists. It uses the Validator's SummaryErrorMessage if defined. If not, it uses the ErrorMessage property.

  Warning messages are preceded by **WarningsConfirmLeadText** and followed by a linefeed.

  When not assigned on the page, it uses the global setting **DefaultWarningConfirmOnSubmit**, which defaults to `false`.

- **WarningsConfirmTemplate** (string) – The text of the Warning Confirmation message that appears before and after the warning messages. Use the token "`{0}`" to indicate where the warning messages go. If omitted, the warnings appear at the end of this property's text.

  When not assigned on the page, it uses the global setting **DefaultWarningConfirmTemplate**, which defaults to "`Are you sure you want to submit the page with these issues remaining?\n\n{0}`".

- **WarningsConfirmTemplateLookupID** (string) – Gets the value for **WarningsConfirmTemplate** 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, **WarningsConfirmTemplate** will be used.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

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 "".

When not assigned on the page, it uses the global setting **DefaultWarningConfirmTemplateLookupID**, which defaults to "".

- **WarningsConfirmLeadText** (string) – Text that appears before each warning message. It denotes a new message. For example, "-" or "*".

  If you use a # symbol, that character will be replaced by a number that represents the error message's position in the list.

  When not assigned on the page, it uses the global setting **DefaultWarningsConfirmLeadText**, which starts at "#. ".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Submitting the Page: Server-Side Validation

Note: This section provides a significant feature to DES. When users do not follow the directions here, ***server side validation does not happen****.*

When the user submits a page, validation should take place across the page or the validation group. If validation fails, the user should get an updated page with error messages from the Validators that reported errors.

If the browser supports client-side validation, DES will validate on the client-side and prevent the page from being posted back when errors are detected. This requires each control that submits the page to be hooked into DES's client-side validation code. See "Submitting the Page: Client-Side Validation".

When the page is posted back, DES must validate again, to handle any case not captured by the client-side. Server side validation is essential because client-side validation is not always available for these reasons:

- User's browser isn't supported by DES for client-side validation. DES scales down gracefully but expects server side validation to be in place.
- User has turned off JavaScript on their browser. Again DES handles this gracefully, but needs server side validation to be in place.
- Hackers anticipate that the user has not setup server-side validation and turn off javascript to bypass the client-side validation. Without server side validation, they can launch SQL Injection and Cross Site Scripting attacks on your site. See the **Input Security User's Guide** for details.

Click on any of these topics to jump to them:

- Setting Up Server-Side Validation
- Preventing Validation Or Evaluating Individual Validators
- Properties and Methods to Validate the Page
- Properties and Methods to Evaluate a Validator Control
- DES's Button, LinkButton, and ImageButton Controls
- DES's Submit Controls For The GridView and DetailsView
- DES's Submit Controls For The DataGrid
- Using Native Button Controls to Submit the Page
- Using the Menu Control to Validate the Page
- Using the BulletedList Control
- Using AJAX Callback Controls
- Using Validation with AutoPostBack
- Submitting the Page: Client-Side Validation

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Setting Up Server-Side Validation

Click on any of these topics to jump to them:

- DES Submit Control PostBack Event Handlers when CausesValidation=true
- PostBack Event Handlers when CausesValidation=false
- Validating With All Other PostBack Event Handlers

1. Make your submit controls run DES validation on the client and server side.
   - For buttons, use DES's Button, LinkButton, and ImageButton controls. See "DES's Button, LinkButton, and ImageButton Controls". Alternatively, you can attach the NativeControlExtender to native buttons. See "Using Native Button Controls to Submit the Page".
   - For the DataGrid, use DES's EditCommandColumn and ButtonColumn controls. See "DES's Submit Controls For The DataGrid".
   - For the GridView and DetailView control, use DES's ButtonField and CommandField controls. See "DES's Submit Controls For The GridView and DetailsView".
   - For autopostback in non-DES controls, see "Using Validation with AutoPostBack".
   - For any other control or action that needs validation when it submits, see "Using the Menu Control to Validate the Page".
2. If a submit control should not validate, set its **CausesValidation** property to `false`. For example, a Cancel or Back button. In addition, if you need to evaluate individual validators instead of those in a validation group, set **CausesValidation** to `false`.
3. If you are using the **Group** property on Validators, assign the group name to the **Group** property on the submit control. See "Validation Groups".
4. All submit controls have a postback event handler event. For example, the **Click** event handler on a Button and **TextChanged** event handler on a TextBox. When you use that event, add server side validation code to it, as shown below.

DES Submit Control PostBack Event Handlers when CausesValidation=true

Your postback event handler method must check that **PeterBlum.DES.Globals.WebFormDirector.IsValid** is `true` before saving or otherwise using the data.

*Note: When their **SkipPostBackEventsWhenInvalid** property is true, DES buttons will skip calling their Click and Command postback event handler methods. Yet, its wise to add the code shown below in case that setting gets inadvertently changed.*

[C#]

```
protected void Button1_Click(object sender, EventArgs e)
{
   if (PeterBlum.DES.Globals.WebFormDirector.IsValid)
   {
      // code that saves or uses the data of the web form
   }
}
```

[VB]

```
Protected Sub Button1_Click(ByVal sender As Object, ByVal e As EventArgs) _
   Handles Button1.Click

   If PeterBlum.DES.Globals.WebFormDirector.IsValid Then
      ' code that saves or uses the data of the web form
   End If
End Sub
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

PostBack Event Handlers when CausesValidation=false

If you set the **CausesValidation** property to `false` with the intent on evaluating individual validators, use this technique. Otherwise no changes are needed to your postback event handler method.

Your postback event handler method must call Validate() on each validator that you need to evaluate. After check that all validators IsValid properties are `true` before saving or otherwise using the data.

This example uses three validators.

[C#]

```
protected void Button1_Click(object sender, EventArgs e)
{
   RequiredTextValidator3.Validate();
   RequiredTextValidator4.Validate();
   RequiredTextValidator5.Validate();
   if (RequiredTextValidator3.IsValid &&
      RequiredTextValidator4.IsValid &&
      RequiredTextValidator5.IsValid)
   {
      // code that saves or uses the data of the web form
   }
}
```

[VB]

```
Protected Sub Button1_Click(ByVal sender As Object, ByVal e As EventArgs) _
   Handles Button1.Click

   RequiredTextValidator3.Validate()
   RequiredTextValidator4.Validate()
   RequiredTextValidator5.Validate()
   If RequiredTextValidator3.IsValid And _
      RequiredTextValidator4.IsValid And _
      RequiredTextValidator5.IsValid Then
      ' code that saves or uses the data of the web form
   End If
End Sub
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Validating With All Other PostBack Event Handlers

Your post back event handler method must call `PeterBlum.DES.Globals.WebFormDirector.Validate()`. Then check **PeterBlum.DES.Globals.WebFormDirector.IsValid** is `true` before saving or otherwise using the data. See the logic below. If you are not using any validation groups, pass "" to the `Validate()` method.

[C#]

```
protected void Button1_Click(object sender, EventArgs e)
{
   PeterBlum.DES.Globals.WebFormDirector.Validate("validation group name");
   if (PeterBlum.DES.Globals.WebFormDirector.IsValid)
   {
      // code that saves or uses the data of the web form
   }
}
```

[VB]

```
Protected Sub Button1_Click(ByVal sender As Object, ByVal e As EventArgs) _
   Handles Button1.Click

   PeterBlum.DES.Globals.WebFormDirector.Validate("validation group name")
   If PeterBlum.DES.Globals.WebFormDirector.IsValid Then
      ' code that saves or uses the data of the web form
   End If
End Sub
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Preventing Validation Or Evaluating Individual Validators

Every submit control should be connected to DES's system. This mostly allows the client-side validation system to determine how to handle each button. If you do not want submit control to validate, such as with a Cancel button, set the **CausesValidation** property to `false` on the submit control.

When `false`, the **CausesValidation** property indicates that automatic validation will not occur when the button is clicked. It will not occur on the client-side or server-side. However, on the server-side, you can elect to validate within your **Click** post back event handler method. You can either use the `PeterBlum.DES.Globals.WebFormDirector.Validate()` method or the `Validate()` methods on individual validators.

See "PostBack Event Handlers when CausesValidation=false".Properties and Methods to Validate the Page

Validating a group of validators

These properties and methods are on **PeterBlum.DES.Globals.WebFormDirector** and operate on a group or all Validators. They can only be used programmatically. Generally these are used in the postback event handler method.

- `Validate(groupname)` (method) – Runs the validation process on Validator controls whose **Group** property matches the group name passed in. If the supplied group is "*", it matches to all Validators. Any Validator control found to be invalid will show its error message for the page that is returned to the user. It returns `true` when there are no errors and `false` when there are errors.

  [C#]

  ```
  public bool Validate(string groupName);
  ```

  [VB]

  ```
  Public Function Validate(ByVal groupName As string) As Boolean
  ```

  Call it within your post back event handler method, prior to saving data. Then test the **PeterBlum.DES.Globals.WebFormDirector.IsValid** property is `true`. See "Validating With All Other PostBack Event Handlers".

  Note: It is safe to call it in Page_Load(), even when the page is new (IsPostback=false). However, only do it in special cases. Normally Page_Load() is reserved to setting up the structure of the page and post back event handlers are designed to handle the posted back data.

  When run, it updates the **PeterBlum.DES.Globals.WebFormDirector.IsValid** property and sets **PeterBlum.DES.Globals.WebFormDirector.HasValidated** to `true`.

  If you want to run some code before validation occurs, use the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager.BeforeValidation** event handler.

- **IsValid** (Boolean) – Determines if validation found errors or not. Test it only after the `Validate()` method has run. Otherwise, it will be a meaningless `true`. After validation runs, it is `true` when there are no errors and `false` when there are errors.

  For examples, see "DES Submit Control PostBack Event Handlers when CausesValidation=true" and " Validating With All Other PostBack Event Handlers".

  *Note: Validator controls may not evaluate their conditions for many reasons including the **Enabled** and **Enabler** properties turned off validation, the validation group name did not match the **Group** property, and the Validator's Condition did not have the settings or data to evaluate. In these cases, the Validator control reports that it is valid.*

- **HasValidated** (Boolean) – Determines if the `Validate()`method has been called. It returns `true` when it has.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Properties and Methods to Evaluate a Validator Control

These properties and methods are on Validator controls. If you need to explicitly validate a list of validators, use these instead of "Properties and Methods to Validate the Page". Usually, you set the submit control's **CausesValidation** property to `false` so the submit control does not automatically call `PeterBlum.DES.Globals.WebFormDirector.Validate()`. Then use these methods and properties in the post back event handler method, prior to saving the data.

For example, you have a menu with a command that should only validate two of the 10 Validators on the page and a validation group does not work for you. Call the `Validate()` method on the two Validators then check if both are valid by testing their **IsValid** property to be `true`.

See "PostBack Event Handlers when CausesValidation=false" for an example.

- `Validate()` (method) – Runs the validation process on the Validator control. If it is found to be invalid, it will show its error message for the page that is returned to the user. It sets the **IsValid** property to `true` or `false` and **HasValidated** property to `true`. It returns `true` when there are no errors and `false` when there are errors.

  [C#]

  ```
  public bool Validate();
  ```

  [VB]

  ```
  Public Function Validate() As Boolean
  ```

- **IsValid** (Boolean) – Determines if validation found errors or not. Test it only after the `Validate()` method has run. Otherwise, it will be a meaningless `true`. After validation runs, it is `true` when there are no errors and `false` when there are errors.

  *Note: Validator controls may not evaluate their conditions for many reasons including the **Enabled** and **Enabler** properties turned off validation, the validation group name did not match the **Group** property, and the Validator's Condition did not have the settings or data to evaluate. In these cases, the Validator control reports that it is valid.*

- **HasValidated** (Boolean) – Determines if the `Validate()`method has been called. It returns `true` when it has.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

DES's Button, LinkButton, and ImageButton Controls

DES provides replacements to the native Button, LinkButton, and ImageButton controls. When using Validator controls on a page, you should use these replacements or use the technique described in "Using Native Button Controls to Submit the Page" to use Microsoft's and third party buttons.

Each of these controls has been extended to handle client-side and server-side validation with the DES Validation Framework. They have several new properties built around validation. If you have a license for the **Peter's Interactive Pages** module, there are several new properties beyond those needed for validation. See "Properties on DES's Button, LinkButton, and ImageButton Controls".

Click on any of these topics to jump to them:

- Using DES's Button, LinkButton, and ImageButton Controls
- Properties on DES's Button, LinkButton, and ImageButton Controls

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Using DES's Button, LinkButton, and ImageButton Controls

Each DES submit control is a direct subclass of Microsoft's controls and has the same name. However, the namespace is different. All controls use the namespace of `PeterBlum.DES.Web.WebControls`. The following are DES's submit controls:

- Button – subclassed from System.Web.UI.WebControls.Button.
- LinkButton – subclassed from System.Web.UI.WebControls.LinkButton.
- ImageButton – subclassed from System.Web.UI.WebControls.ImageButton.

Converting An Existing Page

If you already have a page with the native button controls, use the **Web Application Updater** program with the option **Convert native controls to their DES equivalents**. See the **Installation Guide**.

Adding the Control In Design Mode

1. Drag the control from the toolbox. DES's controls are in a separate area from those supplied with ASP.NET. Be sure that you use DES's controls.
2. Use the Properties Editor to assign:
   - ID – The unique identifier of this control
   - Text – the label shown on the button
   - Group – If you are using validation groups, this is the group name. See "Validation Groups".
   - Click event – Set up the **Click** event method where you will use the data submitted.

   See "Properties on DES's Button, LinkButton, and ImageButton Controls" for other properties.

Adding the Control in ASP.NET Markup

Each of these controls have this format:

```
<des:controlclass id="your_id" runat="server" [properties] />
```

Some examples:

```
<des:Button id="Button1" runat="server" Text="Submit" />
<des:LinkButton id="LinkButton1" runat="server" Text="Submit" Group="Data" />
```

See "Properties on DES's Button, LinkButton, and ImageButton Controls" for properties.

Adding the Control Programmatically

1. Identify a container control where the submit control will be added. This is usually a PlaceHolder, UserControl, Panel, row in a DataGrid, or TableCell. If you want to add to the main page, add a PlaceHolder where it will be attached.
2. Instantiate the control. Its constructor takes no parameters.
3. Assign the **ID** property. The **Text** and **Group** properties are commonly assigned too.
4. Add the control object to the **Controls** property of the container control.
5. Attach an event handler method to its **Click** event.

Here is an example which creates a Button and attaches it to the Submit_Click event method, defined elsewhere.

[C#]

```
PeterBlum.DES.Web.WebControls.Button vButton = new
PeterBlum.DES.Web.WebControls.Button();
vButton.ID = "SubmitButton";
vButton.Text = "Submit";
PlaceHolder1.Controls.Add(vButton);
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

```
vButton.Click += new System.EventHandler(Submit_Click);
```

[VB]

```
Dim vButton As PeterBlum.DES.Web.WebControls.Button = New
PeterBlum.DES.Web.WebControls.Button()
vButton.ID = "SubmitButton"
vButton.Text = "Submit"
PlaceHolder1.Controls.Add(vButton)
' on your Submit_Click() method, use Handles SubmitButton.Click
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Properties on DES's Button, LinkButton, and ImageButton Controls

For properties inherited from the native buttons, click on each of these links: Button, LinkButton, ImageButton.

- **CausesValidation** (Boolean) – When `true`, the button performs client-side and server-side validation. When `false`, it does not. See "Submitting the Page: Server-Side Validation" and "Submitting the Page: Client-Side Validation".

  Even when `false`, on the server side, you can still manually perform validation either on the page (call `PeterBlum.DES.Globals.WebFormDirector.Validate()`) or individual Validators (see "PostBack Event Handlers when CausesValidation=false").

  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 "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.

- **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. See "Validation Groups" for details.

  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 "".

  *Note: The **ValidationGroup** property, inherited from the base class, also works the same way. If you assign both, **Group** overrides **ValidationGroup**.*

- **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`.

- **PostBackUrl** (string) – This is documented here. You should not use the **PostBackUrl** property when the button invokes validation. See "Validation and the PostBackUrl Property" for details.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

You can use this property when not validating with this button.

- **MayMoveOnClick** (Boolean) – When `true`, it installs some extra scripts to prevent the following situation:

  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 `false`, no additional code is added.

  It defaults to `false`.

  Note: Mozilla-based browsers (FireFox, Netscape, etc) do not support this capability on the LinkButton. A browser must support the DHTML click() method on the <a> tag.

- **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`.

For the following properties, please see the "Enhanced Buttons" section of the **Interactive Pages User's Guide** as they all require a license that covers the **Peter's Interactive Pages** module.

- **ConfirmMessage** (string) – When assigned, this button will display a Confirm message box with this text. The user can click OK or Cancel. If OK, the page submission process continues. If Cancel, it stops.

- **DisableOnSubmit** (Boolean) – When `true`, the submit control disables itself visually after the user clicks on it and the page is submitted. (Validation may stop submission, which leaves the control enabled.)

- **ChangeMonitorEnables** (enum PeterBlum.DES.ChangeMonitorEnablesSubmitControl) – Determines if the button switches its state between disabled and enabled.

- **ChangeMonitorUsesConfirm** (enum PeterBlum.DES.ChangeMonitorUsesConfirm) – When the button uses a confirmation message from its **ConfirmMessage** property, with the ChangeMonitor, you can make it display based on the changed state of the page.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

DES's Submit Controls For The GridView and DetailsView

The GridView and DetailsView work well with all DES controls. However, their System.Web.UI.WebControls.CommandField and System.Web.UI.WebControls.ButtonField classes do not set up client or server side validation with DES's framework. They create buttons using Microsoft's Button or LinkButton classes. DES needs buttons support its client side and server side validation.

DES supplies new versions of CommandField and ButtonField that automatically validate and offer other useful features.

Using DES's CommandField Control

DES provides a replacement to CommandField, `PeterBlum.DES.Web.WebControls.CommandField`. It is a direct subclass of `System.Web.UI.WebControls.CommandField` that provides DES validation and additional properties.

To use `PeterBlum.DES.Web.WebControls.CommandField` through design mode or HTML view in your editor:

1. Add the native CommandField control to the GridView or DetailsView.

   ```
   <asp:CommandField [properties]></asp:CommandField>
   ```

2. Replace the "asp" namespace with "des".

   ```
   <des:CommandField [properties]></des:CommandField>
   ```

3. If you want the **Group** property assigned to something other than "", add it directly:

   ```
   <des:CommandField Group="GroupName" [properties]></des:CommandField>
   ```

   If you make each row GridView editable and only want a Button to evaluated its own row, add a plus (+) character as the first character of the **Group** property on the CommandField and all Validators.

   For example, **Group**="+GridView".

   Note: The CommandField already has a ValidationGroup property which is basically the same as the Group property. You can use either. If the Group property is assigned, it always overrides the ValidationGroup property.

4. If you have set up a RowUpdating event handler and want to validate on the server side, you must call `PeterBlum.DES.Globals.WebFormDirector.Validate(`*`validationgroupname`*`)` before testing `PeterBlum.DES.Globals.WebFormDirector.IsValid`. (Validation will happen on the client-side so long as **CausesValidation**=`true`.) If **IsValid** is `false`, set the args parameter's **Cancel** property to `true` and do not save.

   See "Validating With All Other PostBack Event Handlers".

5. For additional properties, see "Properties on DES's Button, LinkButton, and ImageButton Controls".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Using DES's ButtonField Control

DES provides a replacement to ButtonField, `PeterBlum.DES.Web.WebControls.ButtonField`. It is a direct subclass of `System.Web.UI.WebControls.ButtonField` that provides DES validation and additional properties.

To use `PeterBlum.DES.Web.WebControls.ButtonField` through design mode or HTML view in your editor:

1. Add the native ButtonField control to the GridView or DetailsView.

```
<asp:ButtonField [properties]></asp:ButtonField>
```

2. Replace the "asp" namespace with "des".

```
<des:ButtonField [properties]></des:ButtonField>
```

3. If you want the button to perform validation, set the **CausesValidation** property to `true`. (It defaults to `false`.)
4. If you want the **Group** property assigned to something other than "", add it directly:

```
<des:ButtonField Group="GroupName" [properties]></des:ButtonField>
```

   If you make each row GridView editable and only want a Button to evaluated its own row, add a plus (+) character as the first character of the **Group** property on the ButtonField and all Validators.

   For example, **Group**="+GridView".

   Note: The ButtonField already has a ValidationGroup property which is basically the same as the Group property. You can use either. If the Group property is assigned, it always overrides the ValidationGroup property.

5. If you have set up a RowUpdating event handler and want to validate on the server side, you must call `PeterBlum.DES.Globals.WebFormDirector.Validate(`*`validationgroupname`*`)` before testing `PeterBlum.DES.Globals.WebFormDirector.IsValid`. (Validation will happen on the client-side so long as **CausesValidation**=`true`.) If **IsValid** is `false`, set the args parameter's **Cancel** property to `true` and do not save.

   See "Validating With All Other PostBack Event Handlers".

6. For additional properties, see "Properties on DES's Button, LinkButton, and ImageButton Controls".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

DES's Submit Controls For The DataGrid

The DataGrid works well with all DES controls. However, its System.Web.UI.WebControls.EditCommandColumn and System.Web.UI.WebControls.ButtonColumn classes do not set up client or server side validation with DES's framework. They create buttons for a DataGrid row using Microsoft's Button or LinkButton classes. DES needs buttons to support its client side and server side validation.

DES supplies new versions of EditCommandColumn and ButtonColumn that automatically validate and offer other useful features.

Using the EditCommandColumn Control

DES provides a replacement to EditCommandColumn, `PeterBlum.DES.Web.WebControls.EditCommandColumn`. It is a direct subclass of `System.Web.UI.WebControls.EditCommandColumn` that provides DES validation and additional properties.

To use `PeterBlum.DES.Web.WebControls.EditCommandColumn` through design mode or HTML view in your editor:

1. Add the native EditCommandColumn to the DataGrid.

```
<asp:EditCommandColumn ButtonType="PushButton" UpdateText="Update"
CancelText="Cancel" EditText="Edit"></asp:EditCommandColumn>
```

2. Replace the "asp" namespace with "des".

```
<des:EditCommandColumn ButtonType="PushButton" UpdateText="Update"
CancelText="Cancel" EditText="Edit"></des:EditCommandColumn>
```

3. If you want the **Group** property assigned to something other than "", add it directly:

```
<des:EditCommandColumn Group="GroupName" ButtonType="PushButton"
UpdateText="Update" CancelText="Cancel" EditText="Edit">
</des:EditCommandColumn>
```

   If you make each row DataGrid editable and only want a Button to evaluated its own row, add a plus (+) character as the first character of the **Group** property on the EditCommandColumn and all Validators.

   For example, **Group**="+Row".

4. If you have set up an **UpdateCommand** event handler and want to validate on the server side, you must call `PeterBlum.DES.Globals.WebFormDirector.Validate(`*`validationgroupname`*`)` before testing `PeterBlum.DES.Globals.WebFormDirector.IsValid`. (Validation will happen on the client-side so long as **CausesValidation**=`true`.) If **IsValid** is `false`, do not save and do not change the value of **EditItemIndex** to -1.

   See "Validating With All Other PostBack Event Handlers".

5. For additional properties, see "Properties on DES's Button, LinkButton, and ImageButton Controls".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Using the ButtonColumn Control

DES provides a replacement to ButtonColumn, `PeterBlum.DES.Web.WebControls.ButtonColumn`. It is a direct subclass of `System.Web.UI.WebControls.ButtonColumn` that provides DES validation and additional properties.

To use `PeterBlum.DES.Web.WebControls.ButtonColumn` through design mode or HTML view in your editor:

1. Add the native ButtonColumn to the DataGrid.

```
<asp:ButtonColumn ButtonType="PushButton" Text="Delete"></asp:ButtonColumn>
```

2. Replace the "asp" namespace with "des".

```
<des:ButtonColumn ButtonType="PushButton" Text="Delete"></des:ButtonColumn>
```

3. If you want the button to perform validation, set the **CausesValidation** property to `true`. (It defaults to `false`.)
4. If you want the **Group** property assigned to something other than "", add it directly:

```
<des:ButtonColumn Group="GroupName" ButtonType="PushButton" Text="Delete">
</des:ButtonColumn>
```

   If you make each row DataGrid editable and only want a Button to evaluated its own row, add a plus (+) character as the first character of the **Group** property on the ButtonColumn and all Validators.

   For example, **Group**="+Row".

5. If you have set up an **ItemCommand** event handler and want to validate on the server side, you must call `PeterBlum.DES.Globals.WebFormDirector.Validate(`*`validationgroupname`*`)` before testing `PeterBlum.DES.Globals.WebFormDirector.IsValid`. (Validation will happen on the client-side so long as **CausesValidation**=`true`.) If **IsValid** is `false`, do not save and do not change the value of **EditItemIndex** to -1.

   See "Validating With All Other PostBack Event Handlers".

6. For additional properties, see "Properties on DES's Button, LinkButton, and ImageButton Controls".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Using Native Button Controls to Submit the Page

The ASP.NET framework includes a number of controls that can submit the page with ASP.NET. They do not have built in support to validate on the client or server side when clicked. If you intend them to validate on the client-side, you must take one of these actions:

- Replace them with equivalent controls supplied in DES. See "DES's Button, LinkButton, and ImageButton Controls".
- Assign a NativeControlExtender control to them. See the **General Features Guide** for details. This only provides client-side validation. You must also setup server side validation within the button's **Click** or **Command** event handler method. See "Validating With All Other PostBack Event Handlers".

  Example

  ```
  <asp:Button id="Button1" runat="server" Text="Submit" />
  <des:NativeControlExtender id="ExtendButton1" runat="server"
     ControlIDToExtend="Button1" Group="Group1" />
  ```

- When writing code:
  - For Button, LinkButton, and ImageButton, use the `RegisterSubmitControl()` method. See below.
  - For composite controls that contain buttons, use the `RegisterChildSubmitControls()` method. See below.

  This only provides client-side validation. You must also setup server side validation within the button's **Click** or **Command** event handler method. See "Validating With All Other PostBack Event Handlers".

Click on any of these topics to jump to them:

- The RegisterSubmitControl method
- The PeterBlum.DES.SubmitBehavior Class
- The RegisterChildSubmitControls method
- The PeterBlum.DES.ChildSubmitBehavior Class

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

The RegisterSubmitControl method

Use `RegisterSubmitControl()` to register a native Button, LinkButton, or ImageButton control. It establishes client-side validation for the control with an associated validation group name. You still must set up server side validation in your **Click** or **Command** event handler methods. It also establishes other properties if you have a license that covers the **Peter's Interactive Pages** module.

`RegisterSubmitControl()` is a method on **PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager**. Generally it is called in `Page_Load()` because validation happens immediately after `Page_Load()`. If you create the button programmatically, call `RegisterSubmitControl()` at the same time.

[C#]

```
void RegisterSubmitControl(PeterBlum.DES.SubmitBehavior pSubmitBehavior)
```

[VB]

```
Sub RegisterSubmitControl(ByVal pSubmitBehavior As PeterBlum.DES.SubmitBehavior)
```

Use these steps to attach your native button.

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()`.
4. Set up server side validation within the button's **Click** or **Command** event handler method. See "Validating With All Other PostBack Event Handlers".

Example

Adds validation group "group1" and sets **DisableOnSubmit** to true on Button1.

[C#]

```
PeterBlum.DES.SubmitBehavior vSubmitBehavior =
     new PeterBlum.DES.SubmitBehavior(Button1);
vSubmitBehavior.Group = "group1";
vSubmitBehavior.DisableOnSubmit = true;
PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager.RegisterSubmitControl(vSubmitB
ehavior);

// also set up server side validation in Button1_Click
```

[VB]

```
Dim vSubmitBehavior As PeterBlum.DES.SubmitBehavior = _
     New PeterBlum.DES.SubmitBehavior(Button1)
vSubmitBehavior.Group = "group1"
vSubmitBehavior.DisableOnSubmit = True
PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager.RegisterSubmitControl(vSubmitB
ehavior)

' also set up server side validation in Button1_Click
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

The PeterBlum.DES.SubmitBehavior Class

Properties

- **SubmitControl** (object) – 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 must still setup server side validation within the button's **Click** or **Command** event handler method. See "Validating With All Other PostBack Event Handlers".
- **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.ValidationManager.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.ValidationManager.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 <img>, 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, 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`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- **ChangeMonitorEnables** (enum PeterBlum.DES.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.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.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.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)
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

The RegisterChildSubmitControls method

Use `RegisterChildSubmitControls()` when you have a composite control that contains one or more native submit controls. This method will find any controls of the following classes and set up client-side validation: System.Web.UI.WebControls.Button, System.Web.UI.WebControls.LinkButton, and System.Web.UI.WebControls.ImageButton. You still must set up server side validation in your **Click** or **Command** event handler methods. It also establishes other properties if you have a license that covers the **Peter's Interactive Pages** module.

`RegisterChildSubmitControls()` is a method on **PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager**. Generally it is called in `Page_Load()` because validation happens immediately after `Page_Load()`. If you create the containing control programmatically, call `RegisterChildSubmitControls()` at the same time.

In order for a third party control to be supported, it must create child controls of the above classes before you call `RegisterChildSubmitControls()`. If you use `RegisterChildSubmitControls()` and it doesn't work, the third party control cannot be used with client-side validation. You can still support server side validation by calling the `PeterBlum.DES.Globals.WebFormDirector.Validate()` method in a post back event handler of the third party control.

[C#]

```
void RegisterChildSubmitControls(Control pParentControl,
   PeterBlum.DES.ChildSubmitBehavior pSubmitBehavior,
   bool pUnused, bool pShallowSearch,
   string pDeleteID)
```

[VB]

```
Sub RegisterChildSubmitControls(ByVal pParentControl As Control,
   ByVal pSubmitBehavior As PeterBlum.DES.ChildSubmitBehavior,
   ByVal pUnused As Boolean, ByVal pShallowSearch As Boolean,
   ByVal pDeleteID As String)
```

Parameters

pParentControl

The container control whose children will be scanned for buttons.

pSubmitBehavior

The rules to apply to each child button found, including validation, confirm message (only on those that cause validation), disable on submit, and change monitor.

pUnused

Pass `false`.

pShallowSearch

When true, only update the immediate children; when false, update all children in the tree.

pDeleteID

If you need to delete this button using `PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager.DeleteSubmitControls()`, assign this to the name that you will use with DeleteSubmitControls. If not used, pass "".

Use these steps to attach your native button.

1. Create a `PeterBlum.DES.ChildSubmitBehavior` 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.RegisterChildSubmitControls()`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

4. Set up server side validation within the event handler methods used by this control. See "Validating With All Other PostBack Event Handlers".

Example

Adds validation group "group1" and sets **DisableOnSubmit** to on the child buttons of Panel1.

[C#]

```
PeterBlum.DES.ChildSubmitBehavior vSubmitBehavior =
     new PeterBlum.DES.ChildSubmitBehavior();
vSubmitBehavior.Group = "group1";
vSubmitBehavior.DisableOnSubmit = true;
PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager.RegisterChildSubmitControls
(
  Panel1, vSubmitBehavior, false, true, "");

// also set up server side validation in Button1_Click
```

[VB]

```
Dim vSubmitBehavior As PeterBlum.DES.ChildSubmitBehavior = _
     New PeterBlum.DES.ChildSubmitBehavior()
vSubmitBehavior.Group = "group1"
vSubmitBehavior.DisableOnSubmit = True
PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager.RegisterChildSubmitControls
(
  Panel1, vSubmitBehavior, False, True, "");

' also set up server side validation in Button1_Click
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

The PeterBlum.DES.ChildSubmitBehavior Class

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.

  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 validators 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 Peter's 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.ValidationManager.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.ValidationManager.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 <img>, it changes the opacity of the control to dim it.

  It defaults to `false`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- **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`.

- **ChangeMonitorEnables** (enum PeterBlum.DES.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.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.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.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#]

```
ChildSubmitBehavior()

ChildSubmitBehavior(string pValidationGroup)

ChildSubmitBehavior(string pValidationGroup,
   string pConfirmMessage)

ChildSubmitBehavior(string pValidationGroup,
   string pConfirmMessage, bool pDisableOnSubmit,
   bool pMayMoveOnClick, bool pInAJAXUpdate)
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

[VB]

```
ChildSubmitBehavior()

ChildSubmitBehavior(ByVal pValidationGroup As String)

ChildSubmitBehavior(ByVal pValidationGroup As String, _
  ByVal pConfirmMessage As String)

ChildSubmitBehavior(ByVal pValidationGroup As String, _
  ByVal pConfirmMessage As String, ByVal pDisableOnSubmit As Boolean,
  ByVal pMayMoveOnClick As Boolean, ByVal pInAJAXUpdate As Boolean)
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Using the Menu Control to Validate the Page

Note: For third party Menu controls, see the ***Using Third Party Controls*** *guide.*

The System.Web.UI.WebControls.Menu control runs commands which often submit the page. Often those commands should validate the page before allowing a postback. Here's how to set it up.

1. Assign the token "{SUBMIT}" to **NavigateUrl** property of any MenuItem that needs validation.

   The "{SUBMIT}" token takes additional parameters to define the validation group and elect to use the confirmation message. The parameters are in this colon delimited format:

   ```
   {SUBMIT:group=[validationgroupname]:confirm}
   ```

   When `:group=` is defined, validation is used with the validation group name specified. It supports a single group name or "*". For the blank group name, just use `group=`.

   When `:confirm` is used, show the confirmation message. If validation is also used, the confirmation message will appear in the order determined by the **SubmitOrder** property on the PageManager control or **PeterBlum.DES.Globals.WebFormDirector.ValidationManager**. *Note: the ConfirmMessage feature requires a license that covers the* ***Peter's Interactive Pages*** *module.*

   ```
   <asp:Menu ID="Menu1" runat="server" OnMenuItemClick="Menu1_MenuItemClick">
     <Items>
       <asp:MenuItem Text="Confirm and validate" Value="Confirm and validate"
         NavigateUrl="{SUBMIT:group=:confirm}"></asp:MenuItem>
       <asp:MenuItem Text="Confirm" Value="Confirm"
         NavigateUrl="{SUBMIT:confirm}"></asp:MenuItem>
       <asp:MenuItem Text="New Item3" Value="New Item3">
         <asp:MenuItem Text="Validate group 1" Value="New Item 3.1"
           NavigateUrl="{SUBMIT:group=group1}"></asp:MenuItem>
         <asp:MenuItem Text="Validate group 1 and confirm" Value="New Item 3.2"
           NavigateUrl="{SUBMIT:group=group1:confirm}"></asp:MenuItem>
         <asp:MenuItem Text="Nothing" Value="Nothing" ></asp:MenuItem>
       </asp:MenuItem>
     </Items>
   </asp:Menu>
   ```

2. Attach NativeControlExtender control to the Menu control or call `PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager.PrepareMicrosoftMenuControl()`. See below.

   ### Using the NativeControlExtender

   Attach the NativeControlExtender to the Menu control. If you want the confirmation message, set the message in the **ConfirmMessage** property of the NativeControlExtender. See the **General Features Guide** for the NativeControlExtender.

   Here is an example with the NativeControlExtender attached to the above Menu control:

   ```
   <des:NativeControlExtender id="ExtendMenu1" runat="server"
   ControlIDToExtend="Menu1" ConfirmMessage="Continue?" />
   ```

   ### Using the PrepareMicrosoftMenuControl() method

   The `PrepareMicrosoftMenuControl()` method is on **PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager**. Call it in `Page_Load()`. Here is a usage example with the same menu definition shown above and with the confirmation message of "Continue?":

   ```
   PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager.PrepareMicrosoftMenuC
   ontrol
      Menu1, "Continue?")
   ```

3. Set up server side validation within the MenuItemClick event handler method. See "Validating With All Other PostBack Event Handlers".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

The PrepareMicrosoftMenuControl() method

This method on **PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager** takes a System.Web.UI.WebControls.Menu control and applies DES client-side validation to any of its MenuItems whose **NavigateUrl** contains the {SUBMIT} token. See the steps above for details on the {SUBMIT} token.

[C#]

```
void PrepareMicrosoftMenuControl(System.Web.UI.WebControls.Menu pMenu,
  string pConfirmMessage)
```

[VB]

```
Sub PrepareMicrosoftMenuControl(ByVal pMenu As System.Web.UI.WebControls.Menu,
  ByVal pConfirmMessage As String)
```

Parameters

pMenu

The System.Web.UI.WebControls.Menu with its MenuItems already setup with the {SUBMIT} token.

pConfirmMessage

An optional confirmation message. It requires the {SUBMIT} token has the `:confirm` parameter and a license covering **Peter's Interactive Pages** module. Otherwise, it is ignored. If you don't want a confirmation message, pass "".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Using the BulletedList Control

The System.Web.UI.WebControls.BulletedList control can be set up to submit the page by setting its **DisplayMode** property to `LinkButton`. Since it posts back, you can have it invoke DES validation and optionally prompt with a confirmation message.

Steps to set up the BulletedList Control

1. Set up the BulletedList be sure to set these properties to their desired values:

   - **CausesValidation** – Set to `true` when validation should occur and `false` when it should not.
   - **ValidationGroup** – The Validation group name. If **CausesValidation** is `false`, this is ignored.
   - **DisplayMode** – It must be set `LinkButton` to establish validation.

   ```
   <asp:BulletedList id="BulletedList1" runat="server" DisplayMode="ListButton"
      CausesValidation="true" ValidationGroup="Group1" />
   ```

2. Attach NativeControlExtender control to the Menu control or call `PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager.RegisterBulletedListControl()`. See below.

   #### Using the NativeControlExtender

   Attach the NativeControlExtender to the BulletedList control. If you want the confirmation message, set the message in the **ConfirmMessage** property of the NativeControlExtender. See the **General Features Guide** for the NativeControlExtender.

   Here is an example with the NativeControlExtender attached to the above BulletedList control:

   ```
   <des:NativeControlExtender id="ExtendBulletedList1" runat="server"
   ControlIDToExtend="BulletedList1" ConfirmMessage="Continue?" />
   ```

   #### Using the RegisterBulletedListControl() method

   The `RegisterBulletedListControl()` method is on **PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager**. Call it in `Page_Load()`. Here is a usage example with the same menu definition shown above and with the confirmation message of "Continue?":

   ```
   PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager.RegisterBulletedListC
   ontrol
      BulletedList1, "Continue?", false)
   ```

3. Set up server side validation within the MenuItemClick event handler method. See "Validating With All Other PostBack Event Handlers".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

The RegisterBulletedListControl() method

This method on **PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager** takes a System.Web.UI.WebControls.BulletedList control and applies DES client-side validation to all of its items.

[C#]

```
void RegisterBulletedListControl(
  System.Web.UI.WebControls.BulletedList pBulletedList,
  string pConfirmMessage, bool pInPreRender)
```

[VB]

```
Sub RegisterBulletedListControl(
  ByVal pBulletedList As System.Web.UI.WebControls.BulletedList,
  ByVal pConfirmMessage As String, ByVal pInPreRender As Boolean)
```

Parameters

pBulletedList

The System.Web.UI.WebControls.BulletedList control.

pConfirmMessage

An optional confirmation message. It requires a license covering **Peter's Interactive Pages** module. Otherwise, it is ignored. If you don't want a confirmation message, pass "".

pInPreRender

If this method is called from the PreRender method, pass `true`. Otherwise pass `false`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Using AJAX Callback Controls

AJAX is a technology that allows the browser to request information from the server without a post back. This is called a "callback". There are numerous products which support AJAX, including Microsoft ASP.NET AJAX, Telerik's radAjax, and Magic AJAX.

There are a few types of controls that can make a callback. They include "update panels", buttons and other controls. Each of these are addressed in the following sections.

Update panels

Each of these AJAX frameworks has a control that identifies a group of web controls whose HTML is replaced on the callback. Microsoft ASP.NET AJAX calls its control an UpdatePanel; radAjax offers its radAjaxManager and radAjaxPanel; Magic AJAX calls its control an AjaxPanel. When any of these "update panels" are on the page, you must do some extra set up to assist DES. See "Using these controls with AJAX" in the **General Features Guide**.

Buttons

Buttons that invoke a callback do not run the <form onsubmit=> event that is normally used to submit a page. That's where DES sets up its validation code. For these controls, you need to use the The RegisterCallbackControl() method, described below.

Note: If the Button is involved with an "update panel", do not use RegisterCallbackControl(). Instead use the directions in "Using these controls with AJAX" in the ***General Features Guide****.*

Other controls

Some vendors provide callback features in other controls, including textboxes, lists, and images. If you want validation to run prior to the callback, use the The RegisterCallbackControl() method.

The RegisterCallbackControl() method

The `RegisterCallbackControl()` method is on **PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager**.

Use `RegisterCallbackControl()` to register a control that invokes a callback, when you want validation to occur first. This method optionally sets up a confirmation message prompt. If validation fails or the user clicks Cancel on the confirmation message, the callback will not be invoked.

ALERT: Do not use this method with an AJAX Update Panel control.

When you pass a control subclassed from Microsoft's TextBox, DropDownList or ListBox, it will hook up its code to the onchange event of that control. For all other cases, it will add code to the onclick event.

When the callback is received on the server side, you should still validate the data merely as a defense against hackers who modify the scripts of your page to work around client-side validation. See "Properties and Methods to Validate the Page".

[C#]

```
public void RegisterCallbackControl(Control pControl,
   string pGroup,
   string pConfirmMessage);
```

[VB]

```
Public Sub RegisterCallbackControl(ByVal pControl As Control, _
   ByVal pGroup As String,  _
   ByVal pConfirmMessage As String)
```

Parameters

pControl

The control to be registered. It must have its ID property assigned or it will throw an exception. (The ID is required to get the client-side to locate the control on the page.)

ALERT: Do not use this method with an AJAX Update Panel control.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

pGroup

The group name to be associated with this submit control. It can be "".

pConfirmMessage

It establishes the text of a confirm messagebox that will appear when the submit control is clicked. It will show this message along with buttons for OK and Cancel. If the user clicks Cancel, it will not submit the page.

This feature is only supported with a license for the Peter's Interactive Pages. If you do not have a license, pass "".

If you need a carriage return within the message, use the token "{NEWLINE}" (all capital letters) where the return should go.

Example

This example connects a button control from Telerik's radCallback to the group "Login".

[C#]

```
protected System.Web.UI.WebControls.Button Button1;

protected void Page_Load(object sender, System.EventArgs e)
{
   PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager.RegisterCallbackControl(
      CallbackButton1, "Login", "");
}
```

[VB]

```
Protected WithEvents Button1 As System.Web.UI.WebControls.Button

Protected Sub Page_Load(ByVal sender As object, ByVal e As System.EventArgs)
   PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager.RegisterCallbackControl(
_
      CallbackButton1, "Login", "")
End Sub
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Using Validation with AutoPostBack

AutoPostBack is when an edit to a data entry control immediately posts back. DES's TextBox controls can validate on AutoPostBack by setting the **AutoPostBackValidates** property to `true`. With the addition of the NativeControlExtender, other data entry controls can also validate on AutoPostBack by setting the NativeControlExtender's **AutoPostBackValidates** property to `true`.

Click on any of these topics to jump to them:

- How to Add DES Validation to AutoPostBack
- AutoPostBack with Peter's Date Package TextBoxes
- Overcoming a bug in Google Chrome 1

TextBoxes, Lists, CheckBoxes, and most other data entry controls have the **ValidationGroup** and **CausesValidation** properties to implement validation by validation group prior to AutoPostBack. These properties extend DES's validation scope.

- To validate on the validators assigned to the data entry control, set **AutoPostBackValidates** to `true`. Set **CausesValidation** to `false`.
- To validate all validators within a validation group, set **AutoPostBackValidates** and **CausesValidation** to `true`. Set the validation group name in **ValidationGroup**.

Note: These features only provide client-side validation. The intent is to block a postback if validation fails.

How to Add DES Validation to AutoPostBack

Follow these steps to set up AutoPostBack on a data entry control with client-side DES validation:

1. Set the control's **AutoPostBack** property to `true`.
2. If you are using any of DES's TextBox controls, set its **AutoPostBackValidates**, **CausesValidation**, and **ValidationGroup** properties as needed. You are finished.
3. If you are using any other control with the **AutoPostBack** property, add the NativeControlExtender to the page and assign its **ControlIDToExtend** property to your data entry control. Set these NativeControlExtender properties as needed: **AutoPostBackValidates** and **Group**. You are finished. *The NativeControlExtender ignores the CausesValidation property found on the data entry control.*
4. If you are using any of "Peter's Date Package" TextBox controls, see "AutoPostBack with Peter's Date Package TextBoxes".
5. For all remaining controls (those that don't have their own autopostback property), establish the onchange= or onclick= attribute to use `PeterBlum.DES.Globals.WebFormDirector.SubmitManager.GetPostBackEventReference()`. TextBoxes, ListBoxes, and DropDownLists support onchange=. CheckBoxes and RadioButtons support onclick=.

   Only run the Validators assigned to the Data Entry Control

   [C#]

   ```
   TextBox1.Attributes.Add("onchange",
      PeterBlum.DES.Globals.WebFormDirector.SubmitManager.GetPostBackEventRefere
   nce(
         TextBox1, "groupname");
   ```

   [VB]

   ```
   TextBox1.Attributes.Add("onchange", _
      PeterBlum.DES.Globals.WebFormDirector.SubmitManager.GetPostBackEventRefere
   nce( _
         TextBox1, "groupname")
   ```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Validate an entire Validation Group

This uses the System.Web.UI.PostBackOptions class to describe the auto post back options and the ability to track focus.

[C#]

```
TextBox1.Attributes.Add("onchange",
   PeterBlum.DES.Globals.WebFormDirector.SubmitManager.GetPostBackEventRefere
nce(
      new PostBackOptions(TextBox1,
      "",   /* argument */
      "",   /* actionUrl */
      true, /*AutoPostBack*/
      false, /* requiresJavascriptProtocal */
      true, /* trackFocus – can be true or false as you like */
      true, /* clientSubmit */
      true, /* performvalidation */
      "groupname"));    /* validation group name */
```

[VB]

```
TextBox1.Attributes.Add("onchange", _
   PeterBlum.DES.Globals.WebFormDirector.SubmitManager.GetPostBackEventRefere
nce( _
      New PostBackOptions(TextBox1, _
      "",   ' argument  _
      "",   ' actionUrl  _
      True, ' AutoPostBack _
      False, ' requiresJavascriptProtocal  _
      True, ' trackFocus – can be True or False as you like  _
      True, ' clientSubmit  _
      True, ' performvalidation  _
      "groupname"))     ' validation group name
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

AutoPostBack with Peter's Date Package TextBoxes

The TextBoxes within Peter's Date Package (www.PeterBlum.com) require a slightly more complex onchange= attribute.

For DateTextBox, MonthYearTextBox and AnniversaryTextBox, use:

[C#]

```
DateTextBox1.xAutoPostBackB = false;
DateTextBox1.Attributes.Add("onchange",
   "javascript: if (DTB_OnChange('" + DateTextBox1.ClientID + "')) { " +
   PeterBlum.DES.Globals.WebFormDirector.SubmitManager.GetPostBackEventRefere
nce(
   DateTextBox1,"groupname") + "; };");
```

[VB]

```
DateTextBox1.xAutoPostBackB = False
DateTextBox1.Attributes.Add("onchange", _
   "javascript: if (DTB_OnChange('" + DateTextBox1.ClientID + "')) { " + _
   PeterBlum.DES.Globals.WebFormDirector.SubmitManager.GetPostBackEventRefere
nce(_
   DateTextBox1,"groupname") + "; };");
```

For TimeOfDayTextBox and DurationTextBox, use:

[C#]

```
TimeOfDayTextBox1.xAutoPostBackB = false;
TimeOfDayTextBox1.Attributes.Add("onchange",
   "javascript: if (TMTB_OnChange('" + TimeOfDayTextBox1.ClientID +
   "','[DateTextBoxClientID]')) { " +
   PeterBlum.DES.Globals.WebFormDirector.SubmitManager.GetPostBackEventRefere
nce(
   TimeOfDayTextBox1,"groupname") + "; };");
```

[VB]

```
TimeOfDayTextBox1.xAutoPostBackB = False
TimeOfDayTextBox1.Attributes.Add("onchange", _
   "javascript: if (TMTB_OnChange('" + TimeOfDayTextBox1.ClientID + _
   "','[DateTextBoxClientID]')) { " + _
   PeterBlum.DES.Globals.WebFormDirector.SubmitManager.GetPostBackEventRefere
nce(_
   TimeOfDayTextBox1,"groupname") + "; };");
```

[DateTextBoxClientID] is an empty string when **TimeOfDayTextBox.DateTextBoxControlID** is unassigned and the ClientID of the DateTextBox specified in **TimeOfDayTextBox.DateTextBoxControlID** otherwise. DurationTextBox always uses an empty string for *[DateTextBoxClientID]*.

Overcoming a bug in Google Chrome 1

When a DropDownList has **AutoPostBack** set to `true` and you assign a DES validator to that DropDownList, Google Chrome v1 will generate a JavaScript error. Use the NativeControlExtender to avoid this. Add it to any DropDownList whose **AutoPostBack** property is set to `true` and has validators.

Internally the NativeControlExtender swaps the original AutoPostBack scripts with alternatives that work with all browsers.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Submitting the Page: Client-Side Validation

Client-side validation is attached to a few objects on the page: Validators and submit controls. DES automatically sets up client-side validation on Validators unless:

- The **OverrideClientSideEvaluation** property is assigned.
- The **EnableClientScript** property is `false`.
- The **Enabled** property is `false`.
- The **PeterBlum.DES.Globals.WebFormDirector.JavaScriptEnabled** property is `false`. This happens when the user has turned off JavaScript on their browser or the browser does not support client-side validation.
- The **PeterBlum.DES.Globals.WebFormDirector.Browser.SupportsClientSideValidators** property is `false`. This happens when the browser does not support DES's scripts. See "The TrueBrowser Class" in the **General Features Guide**.

Submit controls automatically run validation on a validation group prior to submitting the page unless:

- The **CausesValidation** property is `false`. This property is on Button, LinkButton, and ImageButton controls. Even in this case, client-side code is generated to tell the page submission process to bypass the validation logic. See "Preventing Validation Or Evaluating Individual Validators".
- Any control that submits the page has been incorrectly setup. DES's submit controls always setup correctly. See "DES's Button, LinkButton, and ImageButton Controls". Native buttons need to use the techniques described in "Using Native Button Controls to Submit the Page". Third party controls are described in the **Using Third Party Controls** guide.
- The **PeterBlum.DES.Globals.WebFormDirector.JavaScriptEnabled** property or the **PeterBlum.DES.Globals.WebFormDirector.Browser.SupportsClientSideValidators** property is `false`.

DES provides a number of page-level properties to customize how submitting a page behaves. See "Page-Level Properties used by Client-Side Validation".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Page-Level Properties used by Client-Side Validation

These properties are set in the PageManager control or on the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager** property in `Page_Load()` They also have global default values that are set up in the **Global Settings Editor**, where indicated.

- **ConfirmMessage**, **ConfirmMessageLookupID**, and **ConfirmMessageGroup** – 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. See the **Interactive Pages User's Guide** for details.

  ALERT: These properties require a license that covers the Peter's Interactive Pages module.

  If you need a carriage return within the message, use the token "{NEWLINE}" (all capital letters) where the return should go.

  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.

  When unassigned, these properties get their default from these global settings: **DefaultConfirmMessage**, **DefaultConfirmMessageID**, and **DefaultConfirmMessageGroup**. These global settings are all "" by default.

```
<des:PageManager id="PageManager1" runat="server"
   ConfirmMessage="Save this page?" />
```

- **CustomSubmitFunctionName** (string) – Use this to add your own JavaScript code into the page submission process.

  Before selecting this function, please consider several other available functions. **CustomSubmitFunctionName** may not be called if its **SubmitOrder** is after page-level validation occurs and page-level validation fails. If you need a function that is always called after page-level validation occurs, use **PostPageValidationFunctionName**. If you need to run code after any validation occurs, including after field edits and page submission, use **PostValidationUpdateScript**.

  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 a boolean value. Return `true` to continue submitting or `false` to stop submitting the page. This property should only contain the name of the function.

  Here is what your function should look like:

```
function functionname(group)
{
   // do your work
   if (continue)
      return true;
   else
      return false;
}
```

  Your function should appear on the page that is generated. See "Adding Your JavaScript to the Page" for several ways to do this.

  Note: The group parameter's value 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 **SubmitOrder** property to determine the order of these actions.

  Define the name of the function in this property. If "", no function is defined.

  It defaults to "". There is no global setting for this property.

```
<des:PageManager id="PageManager1" runat="server"
   CustomSubmitFunctionName="MyFunction" />
```

  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.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- **SubmitOrder** (enum PeterBlum.DES.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.

  Use this enumerated type to determine the order of these actions. Its values are:
  - `ConfirmCustomValidate` – Confirm message, Custom submit function, Validate.
  - `ConfirmValidateCustom`
  - `CustomConfirmValidate`
  - `CustomValidateConfirm`
  - `ValidateConfirmCustom`
  - `ValidateCustomConfirm`

  When unassigned, its uses the value from the **DefaultSubmitOrder** property of the **Global Settings Editor**, which defaults to `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.

- **PostPageValidationFunctionName** (string) – Use this to execute your own JavaScript code immediately after page-level validation occurs, whether or not the page was valid. This property should only contain the name of the function.

  Your function should take these two parameters in the order shown:
  - Group (an uppercase string) – The Validation Group name that was applied. When you are not using validation groups, this will be passed "".
  - IsValid (boolean) – If true, the controls are valid for the given group. If false, there is a validation error.

  Your function should not return a value.

  Here is what your function should look like:

```
function functionname(group, isValid)
{
   // do your work
}
```

  Your function should appear on the page that is generated. See "Adding Your JavaScript to the Page" for several ways to do this.

  Note: The group parameter's value 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.

  Define the name of the function in this property. If "", no function is defined.

  It defaults to "". There is no global setting for this 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.

SEE AN EXAMPLE ON THE NEXT PAGE

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Example

This function changes the entire background color of the page when there is an error.

```
<script type='text/javascript'>
function ChangeBackColorToRed(group, isValid)
{
   document.body.style.backgroundColor = isValid ? "" : "red";
}
</script>

<des:PageManager id="PageManager1" runat="server"
   PostPageValidationFunctionName="ChangeBackColorToRed" />
```

- **PostValidationUpdateScript** (string) – Assign any JavaScript code that you want executed after validation occurs. Its primary intent is to allow you to run your own client-side code that relocates absolutely positioned elements after Validator error messages or ValidationSummary have caused the page to reposition its elements. However, you can use it for anything you want.

  You can enter any JavaScript statements you want into this string. Your string will be executed by using the JavaScript `eval()` function.

  This function will be called even if nothing visually changed on the page.

  It defaults to "".

```
<des:PageManager id="PageManager1" runat="server"
   PostValidationUpdateScript="alert('validation ran');" />
```

- **DefaultGroup** (string) – Defines the group name used if the user hits Enter to submit the page before using any buttons that have a group. Once a submit control is hit, its group becomes the default. The Enter key usually hits a default button, determined by the browser. But it is unreliable, especially on Internet Explorer when there is only one <input type=text> field. It defaults to "".

  Note: The Peter's Interactive Pages offers other techniques to map the Enter key to a particular button.

- **SubmitHidesOtherValidationGroups** (Boolean) – When using validation groups, errors can be displayed for multiple groups at the same time. If the user clicks a submit button, this determines if the validation groups not on the same group as the button will be hidden or remain in their current state.

  When they are hidden, it becomes clearer to the user which errors to fix.

  When `true`, they are hidden. When `false`, they remain in their current state.

  It defaults to `true`.

  This property exists only because in versions prior to 4.0.5, submit buttons kept the messages and there needs to be a way to implement the previous behavior.

- **RequiredFieldsReportErrorsOnBlur** (Boolean) - Determines if required oriented validators will fire if the user just tabs through the control they evaluate without changing it.

  Validators supported include RequiredTextValidator, RequiredListValidator, MultipleRequiredControlsValidator, and several others that include the **IgnoreBlankText** property if its set to `true`.

  In addition, MultiConditionValidator is included when a child Condition is a required type.

  When `true`, the validation is invoked on the onblur event.

  When unassigned, its uses the value from the **DefaultRequiredFieldsReportErrorsOnBlur** property of the **Global Settings Editor**, which defaults to `false`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Additional Validation Topics

This section covers a variety of special cases when using the Validator controls.

Click on any of these topics to jump to them:

- Validation Best Practices
- Using the DES Validation Framework with each ASP.NET web control
- Using Validators with Third Party Controls
- Analyzing the Page's Validation configuration
- Supporting a Reset or Clear Button
- The ViewState and Preserving Properties for PostBack
- Validation and the PostBackUrl Property
- Invoking Validation With Client-Side Scripts
- Running a Script when validation occurs
- Changing The Properties of a Validator With Client-Side Scripts
- Adding Client-Side Validation Scripts Within Your Server Side Code
- Support for the Native Validation Framework
- Using Validator controls to evaluate data not from Controls
- Attaching Validators to controls inside of a User Control
- Submitting the Page: Server-Side Validation
- Submitting the Page: Client-Side Validation
- ValidationSummary Control
- CombinedErrorMessages Control
- Required Field Marker Controls

These topics are found in the General Settings Guide:

- Using these Controls with AJAX
- 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

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Validation Best Practices

Here are some things to remember to create good sites with validation.

- Client-side validation should never be assumed to run. Always have validators evaluate the same data on the server side. Make sure the `Validate()` method is called on the page (**PeterBlum.DES.Globals.WebFormDirector**) or individual Validators within your post back event method. Then test the **IsValid** property on **PeterBlum.DES.Globals.WebFormDirector** or the individual Validators. See "Setting Up Server-Side Validation".
- Hackers can attack a site through your data entry controls and other inputs using SQL Injection and Script Injection (Cross-site scripting). They can access your database and modify your site when your inputs are not protected. Validation plays a huge role in preventing this. Use the **Peter's Input Security** module to add validators that protect you against these attacks. Every input on your page should be reviewed for security holes. The Page Security Analysis report within **Peter's Input Security** can identify them. Some of your Validators on each page already provide the necessary protection. For others, you will add the PageSecurityValidator and FieldSecurityValidator.
- Make error messages clear. Instead of keeping them short, use error formatters that popup the error message: AlertImageErrorFormatter, HyperLinkErrorFormatter, ToolTipImageErrorFormatter, or PopupErrorFormatter. See "ErrorFormatters: Customizing the Appearance of the Error Message".
- Often users mix together several Validators under a single error message using the MultiConditionValidator. Instructions to the user are clearer if individual Validators are set up for each situation. The user is then shown the exact problem to fix. For example, "Fill this in and make it a date format" is less clear than two separate messages, "Please fill this in" and "Incorrect date format."
- Some Validators simply should not run on the client side. For example, a password validator, using the CompareToValueValidator. Otherwise it will expose the password that you are comparing in the client-side. For these, set the **OverrideClientSideEvaluation** property to something other than `No`.
- Users often use an "*" character for the ErrorMessage while using a ValidationSummary control. It saves space and gets the user's attention. A better solution is to use the AlertImageErrorFormatter, TooltipImageErrorFormatter, or PopupErrorFormatter with the complete error message. The user will be able to see the error message next to the field without it using up valuable screen real estate. See "ErrorFormatters: Customizing the Appearance of the Error Message".
- On a tabbed or paneled interface, often there are data entry controls on hidden panels. Set up Validators either with a validation group name for each panel or set the Enabler to the VisibleCondition that looks at the same control as you set up in **ControlIDToEvaluate**.
- Some developers want to disable the submit button when there is a validation error to inspire the user to fix the validation problem. Avoid this because:
    - It prevents DES from taking actions built into the submit to remind the user they need to fix the validators. It cannot show the ValidationSummary, an alert, set focus to fields with the error, etc.
    - It does not communicate the issue that disables the button. If the issue is not absolutely obvious, a good user interface will keep the command (button, menu item, etc) enabled and give the user a message when they use that command to tell them of a problem. DES does this for you (ValidationSummary, alert, etc.)

  If you are not using anything to communicate a validation error when the page is submitted, see "Drawing The User's Attention To The Error".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Using the DES Validation Framework with each ASP.NET web control

This section lists each ASP.NET webcontrol supplied by Microsoft in alphabetic order. Each control identifies how to use it with DES. You only need to follow these directions when you add DES's validators to the page.

Click on any of these topics to jump to them:

- AdRotator
- BulletedList
- Button
- Calendar
- ChangePassword
- CheckBox
- CheckBoxList
- CompareValidator
- CreateUserWizard
- CustomValidator
- DataGrid
- DataList
- DetailsView
- DropDownList
- FileUpload
- FormView
- GridView
- HiddenField
- HyperLink
- Image
- ImageButton
- ImageMap
- Label
- LinkButton
- ListBox
- Literal
- Localize
- Login
- LoginName
- LoginStatus
- LoginView
- Menu
- MultiView
- Panel
- PasswordRecovery
- PlaceHolder
- RadioButton
- RadioButtonList
- RangeValidator
- RegularExpressionValidator
- RequiredFieldValidator
- Repeater
- SiteMapPath
- Substitution
- Table
- TextBox
- TreeView
- ValidationSummary
- WebZone and WebParts
- Wizard
- XML

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

AdRotator

DES does not interact with this control. Use it normally.

BulletedList

When the **DisplayMode** property is `LinkButton`, validation must be set up. See "Using the BulletedList Control".

Button

Every Button that should invoke validation needs additional code from DES's validation system. You should either use the Button supplied in DES (see "DES's Button, LinkButton, and ImageButton Controls") or assign the NativeControlExtender to the Button. If you do not, validation will not fire on either the client- or server-side.

ALERT: Always set up Server side validation: Validating With All Other PostBack Event Handlers.

You should not use the **PostBackUrl** property when the button invokes validation. See "Validation and the PostBackUrl Property" for details.

Calendar

If you require a selection in the calendar, add a CustomValidator with the following for its ServerCondition event. Here's an example of the ServerCondition event evaluating the value of the Calendar control called Calendar1. See "Using This Condition" for more.

[C#]

```
using PeterBlum.DES;
…
 protected void MyCondition(IBaseCondition sourceCondition,
    IConditionEventArgs args)
 {
    args.IsMatch = Calendar1.SelectedDate != DateTime.MinValue;
 }
```

[VB]

```
Imports PeterBlum.DES
…
 Protected Sub MyCondition(ByVal sourceCondition As IBaseCondition, _
    ByVal args As IConditionEventArgs)
    args.IsMatch = Calendar1.SelectedDate <> DateTime.MinValue
 End Sub
```

ChangePassword

The ChangePassword control requires changing all validators and buttons to DES's controls. If there are no other data entry controls on the page, it is not necessary to do anything. However, using DES, you can enhance this control to validate the length, character set, and other requirements of your new passwords.

To use the ChangePassword control with DES, you must change its contents into a Template so that they appear in the <ChangePasswordTemplate> node. A complete conversion is shown on the next page. You can copy it from this document and paste it into your web form. If you prefer to start from scratch, here are the steps to convert.

1. Use Visual Studio 2005 to add the ChangePassword control onto your web form.
2. In design mode, use the **Tasks** menu on the ChangePassword control to select **Convert To Template**.
3. Save the document.
4. Run the **Web Application Updater** program on this page with its **Convert native controls to their DES equivalents** option. This will change all validators and buttons for you.

On the next page you will find a completely converted page with several extensions beyond what is described above:

- RequiredTextValidators use the TooltipImageErrorFormatter instead of the "*" character.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- TextLengthValidator is available to set a minimum size. To use it, set its **Enabled** property to `true` and its **Minimum** property to the desired minimum. You can add a **Maximum** too.
- CharacterValidator is available to limit the character set. To use it, set its **Enabled** property to `true`. Then set the various properties, especially the **OtherCharacters** property to non-alphanumeric characters like underscore and period, if allowed. If spaces are allowed, set the **Space** property to `true`.
- Several validators are combined using a CombinedErrorMessages control. This makes it easier to change the formatting so all have a shared look. You can remove it if you like.

Here are some suggested enhancements to consider:

- Change the error message text and error formatter type.
- Set the **ShowRequiredFieldMarker** property to `true` on the RequiredTextValidators
- Add validators to confirm other patterns required for your passwords, such as a demand for at least 2 digits. If you add these rules, also add them into the CombinedErrorMessages control.

Example

Note: Adobe Acrobat Reader permits copying text.

```
<asp:ChangePassword ID="ChangePassword1" runat="server">
 <ChangePasswordTemplate>
    <table border="0" cellpadding="1" cellspacing="0" style="border-collapse: collapse">
       <tr>
          <td>
             <table border="0" cellpadding="0">
                <tr>
                   <td align="center" colspan="2">
                      Change Your Password</td>
                </tr>
                <tr>
                   <td align="right">
                      <asp:Label ID="CurrentPasswordLabel" runat="server"
                        AssociatedControlID="CurrentPassword">Password:</asp:Label>
                   </td>
                   <td>
                      <asp:TextBox ID="CurrentPassword" runat="server" TextMode="Password">
                      </asp:TextBox>
                      <des:RequiredTextValidator ID="CurrentPasswordRequired" runat="server"
                         ControlIDToEvaluate="CurrentPassword"
                         ErrorMessage="{LABEL} is required." Group="ChangePassword1"
                         Label-LabelControlID="CurrentPasswordLabel" Label-TrimTrailingSymbol="true">
                         <ErrorFormatterContainer>
                           <des:TooltipImageErrorFormatter />
                         </ErrorFormatterContainer>
                       </des:RequiredTextValidator>
                   </td>
                </tr>
                <tr>
                   <td align="right">
                      <asp:Label ID="NewPasswordLabel" runat="server"
                        AssociatedControlID="NewPassword">New Password:</asp:Label>
                   </td>
                   <td>
                      <asp:TextBox ID="NewPassword" runat="server" TextMode="Password">
                      </asp:TextBox>
                      <des:RequiredTextValidator ID="NewPasswordRequired" runat="server"
                         ControlIDToEvaluate="NewPassword"
                         ErrorMessage="{LABEL} is required." Group="ChangePassword1"
                         Label-LabelControlID="NewPasswordLabel" Label-TrimTrailingSymbol="true">
                         <ErrorFormatterContainer>
                           <des:TooltipImageErrorFormatter />
                         </ErrorFormatterContainer>
                       </des:RequiredTextValidator>
                   </td>
                </tr>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

```
<tr>
   <td align="right">
      <asp:Label ID="ConfirmNewPasswordLabel" runat="server"
        AssociatedControlID="ConfirmNewPassword">Confirm New Password:</asp:Label>
   </td>
   <td>
      <asp:TextBox ID="ConfirmNewPassword" runat="server" TextMode="Password">
      </asp:TextBox>
      <des:RequiredTextValidator ID="ConfirmNewPasswordRequired" runat="server"
         ControlIDToEvaluate="ConfirmNewPassword"
         ErrorMessage="{LABEL} is required." Group="ChangePassword1"
         Label-LabelControlID="ConfirmNewPasswordLabel" Label-TrimTrailingSymbol="true">
         <ErrorFormatterContainer>
           <des:TooltipImageErrorFormatter />
         </ErrorFormatterContainer>
     </des:RequiredTextValidator>
   </td>
</tr>
<tr>
   <td align="center" colspan="2">
      <des:CompareTwoFieldsValidator ID="NewPasswordCompare" runat="server"
         ControlIDToEvaluate="NewPassword"
         SecondControlIDToEvaluate="ConfirmNewPassword"
         ErrorMessage="The {LABEL2} must match the {LABEL} entry."
         Label-LabelControlID="NewPasswordLabel" Label-TrimTrailingSymbol="true"
         SecondLabel-LabelControlID="ConfirmNewPasswordLabel"
         SecondLabel-TrimTrailingSymbol="true"
         Group="ChangePassword1">
      </des:CompareTwoFieldsValidator>
      <%-- If you want a minimum text length,
        set Enabled=true and set the Minimum property --%>
      <DES:TextLengthValidator ID="PasswordLength" runat="server"
        Enabled="false"
        Minimum="0"
        ControlIDToEvaluate="NewPassword"
        ErrorMessage ="The {LABEL} requires a minimum of {MINIMUM} characters."
        Label-LabelControlID="NewPasswordLabel" Label-TrimTrailingSymbol="true"
        Group="ChangePassword1" />
      <%-- If you want to limit the characters used,
         set Enabled=true and any other properties as needed.
         Update the ErrorMessage. --%>
      <DES:CharacterValidator ID="ValidCharacters" runat="server"
        Enabled="false"
        ControlIDToEvaluate="NewPassword"
        LettersLowercase="true" LettersUppercase="true" Digits="true"
        Space="false" OtherCharacters=""
        ErrorMessage ="The {LABEL} only allows these characters:[fill in]."
        Label-LabelControlID="NewPasswordLabel" Label-TrimTrailingSymbol="true"
        Group="ChangePassword1" />
      <%-- NOTE: If you add more validators to this section,
       add their IDs to the Validators node.
      This will create a single error message section. --%>
      <DES:CombinedErrorMessages ID="CombinedErrorMessages1" runat="server">
         <ErrorFormatterContainer>
           <des:TextErrorFormatter Display="Dynamic" />
         </ErrorFormatterContainer>
         <Validators>
           <DES:ValidatorControlConnection ControlID="NewPasswordCompare" />
           <DES:ValidatorControlConnection ControlID="PasswordLength" />
           <DES:ValidatorControlConnection ControlID="ValidCharacters" />
         </Validators>
      </DES:CombinedErrorMessages>
   </td>
</tr>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

```
                <tr>
                   <td align="center" colspan="2" style="color: red">
                      <asp:Literal ID="FailureText" runat="server"
                              EnableViewState="False"></asp:Literal>
                   </td>
                </tr>
                <tr>
                   <td align="right">
                      <des:Button ID="ChangePasswordPushButton" runat="server"
                         CommandName="ChangePassword" Text="Change Password" Group="ChangePassword1" />
                   </td>
                   <td>
                      <des:Button ID="CancelPushButton" runat="server"
                         CausesValidation="False" CommandName="Cancel" Text="Cancel" />
                   </td>
                </tr>
             </table>
          </td>
       </tr>
    </table>
 </ChangePasswordTemplate>
</asp:ChangePassword>
```

CheckBox

The CheckBox control can be validated with the CheckStateValidator.

Use CheckStateCondition in Enablers, the MultiConditionValidator, CountTrueConditionsValidator, etc.

When building a list of CheckBoxes where at least one must be checked, see the section "Requiring one mark amongst Checkboxes on DataGrid Rows" in the **Tutorials.pdf** document.

When using the **AutoPostBack** property, you can provide validation prior to postback by using the NativeControlExtender. Set the NativeControlExtender's **AutoPostBackValidates** property to `Control` or `ValidationGroup`.

Note: The CheckBox has its own properties for validation on postback: CausesValidation and ValidationGroup. These are not used by DES.

CheckBoxList

The CheckBoxList control can be validated with RequiredListValidator, SelectedIndexValidator, and SelectedIndexRangesValidator.

Use RequiredListCondition and SelectedIndexCondition in Enablers, the MultiConditionValidator, CountTrueConditionsValidator, etc are the RequiredListCondition and SelectedIndexCondition.

When using the **AutoPostBack** property, you can provide validation prior to postback by using the NativeControlExtender. Set the NativeControlExtender's **AutoPostBackValidates** property to `Control` or `ValidationGroup`.

Note: The CheckBoxList has its own properties for validation on postback: CausesValidation and ValidationGroup. These are not used by DES.

CompareValidator

Replace it with a DES validator depending on the situation:

- For validating the data type format where the **Operator** property is `DataTypeCheck`, use the DataTypeCheckValidator.
- For comparing to a value, use the CompareToValueValidator.
- For comparing two data entry controls, use the CompareTwoFieldsValidator.

To quickly convert the native validators, run the **Web Application Updater** program on this page with its **Convert native controls to their DES equivalents** option. This will change all validators and buttons for you.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

CreateUserWizard

The CreateUserWizard control requires changing all validators and buttons to DES's controls. If there are no other data entry controls on the page, it is not necessary to do anything. However, using DES, you can enhance this control to validate the length, character set, and other requirements of your new passwords.

To use the CreateUserWizard control with DES, you must change its contents into a Template so that they appear in the <ContentTemplate> node of the <asp:CreateUserWizardStep>. A complete conversion is shown starting on the next page. You can copy it from this document and paste it into your web form. If you prefer to start from scratch, here are the steps to convert.

1. Use Visual Studio 2005 to add the CreateUserWizard control onto your web form.
2. In design mode, use the **Tasks** menu on the ChangePassword control to select the following commands:
    - **Customize Create User Step**
    - **Convert to StartNavigationTemplate**
    - **Convert to StepNavigationTemplate**
    - **Convert to FinishNavigationTemplate**
3. Save the document.
4. Run the **Web Application Updater** program on this page with its **Convert native controls to their DES equivalents** option. This will change all validators and buttons for you.

In the example below, you will find a completely converted page with several extensions beyond what is described above:

- RequiredTextValidators use the TooltipImageErrorFormatter instead of the "*" character.
- TextLengthValidator is available to set a minimum size. To use it, set its **Enabled** property to `true` and its **Minimum** property to the desired minimum. You can add a **Maximum** too.
- CharacterValidator is available to limit the character set. To use it, set its **Enabled** property to `true`. Then set the various properties, especially the **OtherCharacters** property to non-alphanumeric characters like underscore and period, if allowed. If spaces are allowed, set the **Space** property to `true`.
- Several validators are combined using a CombinedErrorMessages control. This makes it easier to change the formatting so all have a shared look. You can remove it if you like.

Here are some suggested enhancements to consider:

- Change the error message text and error formatter type.
- Set the **ShowRequiredFieldMarker** property to `true` on the RequiredTextValidators
- Add validators to confirm other patterns required for your passwords, such as a demand for at least 2 digits. If you add these rules, also add them into the CombinedErrorMessages control.

Example

Note: Adobe Acrobat Reader permits copying text.

```
<asp:CreateUserWizard ID="CreateUserWizard1" runat="server">
 <WizardSteps>
    <asp:CreateUserWizardStep runat="server">
       <ContentTemplate>
          <table border="0">
             <tr>
                <td align="center" colspan="2">
                   Sign Up for Your New Account</td>
             </tr>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

```
<tr>
   <td align="right">
      <asp:Label ID="UserNameLabel" runat="server"
        AssociatedControlID="UserName">User Name:</asp:Label>
   </td>
   <td>
      <asp:TextBox ID="UserName" runat="server"></asp:TextBox>
      <des:RequiredTextValidator ID="UserNameRequired" runat="server"
         ControlIDToEvaluate="UserName"
         ErrorMessage="{LABEL} is required." Group="CreateUserWizard1"
         Label-LabelControlID="UserNameLabel" Label-TrimTrailingSymbol="true">
         <ErrorFormatterContainer>
           <des:TooltipImageErrorFormatter />
         </ErrorFormatterContainer>
     </des:RequiredTextValidator>
   </td>
</tr>
<tr>
   <td align="right">
      <asp:Label ID="PasswordLabel" runat="server"
        AssociatedControlID="Password">Password:</asp:Label>
   </td>
   <td>
      <asp:TextBox ID="Password" runat="server" TextMode="Password"></asp:TextBox>
      <des:RequiredTextValidator ID="PasswordRequired" runat="server"
         ControlIDToEvaluate="Password"
         ErrorMessage="{LABEL} is required." Group="CreateUserWizard1"
         Label-LabelControlID="PasswordLabel" Label-TrimTrailingSymbol="true">
         <ErrorFormatterContainer>
           <des:TooltipImageErrorFormatter />
         </ErrorFormatterContainer>
     </des:RequiredTextValidator>
   </td>
</tr>
<tr>
   <td align="right">
      <asp:Label ID="ConfirmPasswordLabel" runat="server"
           AssociatedControlID="ConfirmPassword">Confirm Password:</asp:Label>
   </td>
   <td>
      <asp:TextBox ID="ConfirmPassword" runat="server" TextMode="Password"></asp:TextBox>
      <des:RequiredTextValidator ID="ConfirmPasswordRequired" runat="server"
         ControlIDToEvaluate="ConfirmPassword"
         ErrorMessage="{LABEL} is required." Group="CreateUserWizard1"
         Label-LabelControlID="ConfirmPasswordLabel" Label-TrimTrailingSymbol="true">
         <ErrorFormatterContainer>
           <des:TooltipImageErrorFormatter />
         </ErrorFormatterContainer>
     </des:RequiredTextValidator>
   </td>
</tr>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

```
<tr>
   <td align="right">
      <asp:Label ID="EmailLabel" runat="server"
          AssociatedControlID="Email">E-mail:</asp:Label>
   </td>
   <td>
      <asp:TextBox ID="Email" runat="server"></asp:TextBox>
      <des:RequiredTextValidator ID="EmailRequired" runat="server"
         ControlIDToEvaluate="Email"
         ErrorMessage="{LABEL} is required." Group="CreateUserWizard1"
         Label-LabelControlID="EmailLabel" Label-TrimTrailingSymbol="true">
         <ErrorFormatterContainer>
           <des:TooltipImageErrorFormatter Display="Dynamic" />
         </ErrorFormatterContainer>
     </des:RequiredTextValidator>
     <DES:EmailAddressValidator ID="EmailValidator1" runat="server"
         ControlIDToEvaluate="Email"
         ErrorMessage="{LABEL} is not a valid email address." Group="CreateUserWizard1"
         Label-LabelControlID="EmailLabel" Label-TrimTrailingSymbol="true">
         <ErrorFormatterContainer>
           <des:TooltipImageErrorFormatter Display="Dynamic" />
         </ErrorFormatterContainer>
     </des:EmailAddressValidator>
   </td>
</tr>
<tr>
   <td align="right">
      <asp:Label ID="QuestionLabel" runat="server"
        AssociatedControlID="Question">Security Question:</asp:Label>
   </td>
   <td>
      <asp:TextBox ID="Question" runat="server"></asp:TextBox>
      <des:RequiredTextValidator ID="QuestionRequired" runat="server"
         ControlIDToEvaluate="Question"
         ErrorMessage="{LABEL} is required." Group="CreateUserWizard1"
         Label-LabelControlID="QuestionLabel" Label-TrimTrailingSymbol="true">
         <ErrorFormatterContainer>
           <des:TooltipImageErrorFormatter Display="Dynamic" />
         </ErrorFormatterContainer>
     </des:RequiredTextValidator>
   </td>
</tr>
<tr>
   <td align="right">
      <asp:Label ID="AnswerLabel" runat="server"
        AssociatedControlID="Answer">Security Answer:</asp:Label>
   </td>
   <td>
      <asp:TextBox ID="Answer" runat="server"></asp:TextBox>
      <des:RequiredTextValidator ID="AnswerRequired" runat="server"
         ControlIDToEvaluate="Answer"
         ErrorMessage="{LABEL} is required." Group="CreateUserWizard1"
         Label-LabelControlID="AnswerLabel" Label-TrimTrailingSymbol="true">
         <ErrorFormatterContainer>
           <des:TooltipImageErrorFormatter Display="Dynamic" />
         </ErrorFormatterContainer>
     </des:RequiredTextValidator>
   </td>
</tr>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

```
            <tr>
               <td align="center" colspan="2">
                  <des:CompareTwoFieldsValidator ID="PasswordCompare" runat="server"
                     ControlIDToEvaluate="Password"
                     SecondControlIDToEvaluate="ConfirmPassword"
                     ErrorMessage="The {LABEL2} must match the {LABEL} entry."
                     Label-LabelControlID="PasswordLabel" Label-TrimTrailingSymbol="true"
                     SecondLabel-LabelControlID="ConfirmPasswordLabel"
                     SecondLabel-TrimTrailingSymbol="true"
                     Group="CreateUserWizard1">
                  </des:CompareTwoFieldsValidator>
                  <%-- If you want a minimum text length,
                    set Enabled=true and set the Minimum property --%>
                  <DES:TextLengthValidator ID="PasswordLength" runat="server"
                    Enabled="false"
                    Minimum="0"
                    ControlIDToEvaluate="Password"
                    ErrorMessage ="The {LABEL} requires a minimum of {MINIMUM} characters."
                    Label-LabelControlID="PasswordLabel" Label-TrimTrailingSymbol="true"
                    Group="ChangePassword1" />
                  <%-- If you want to limit the characters used,
                     set Enabled=true and any other properties as needed.
                     Update the ErrorMessage. --%>
                  <DES:CharacterValidator ID="ValidCharacters" runat="server"
                    Enabled="true"
                    ControlIDToEvaluate="Password"
                    LettersLowercase="true" LettersUppercase="true" Digits="true"
                    Space="false" OtherCharacters=""
                    ErrorMessage ="The {LABEL} only allows these characters:[fill in]."
                    Label-LabelControlID="PasswordLabel" Label-TrimTrailingSymbol="true"
                    Group="ChangePassword1" />
                  <%-- NOTE: If you add more validators to this section,
                  add their IDs to the Validators node.
                  This will create a single error message section. --%>
                  <DES:CombinedErrorMessages ID="CombinedErrorMessages1" runat="server">
                     <ErrorFormatterContainer>
                       <des:TextErrorFormatter Display="Dynamic" />
                     </ErrorFormatterContainer>
                     <Validators>
                       <DES:ValidatorControlConnection ControlID="PasswordCompare" />
                       <DES:ValidatorControlConnection ControlID="PasswordLength" />
                       <DES:ValidatorControlConnection ControlID="ValidCharacters" />
                     </Validators>
                  </DES:CombinedErrorMessages>
               </td>
            </tr>
            <tr>
               <td align="center" colspan="2" style="color: red">
                  <asp:Literal ID="ErrorMessage" runat="server" EnableViewState="False"></asp:Literal>
               </td>
            </tr>
         </table>
      </ContentTemplate>
      <CustomNavigationTemplate>
         <table border="0" cellspacing="5" style="width: 100%; height: 100%;">
            <tr align="right">
               <td align="right" colspan="0">
                  <des:Button ID="StepNextButton" runat="server"
                     CommandName="MoveNext" Text="Create User"
                     Group="CreateUserWizard1" />
               </td>
            </tr>
         </table>
      </CustomNavigationTemplate>
   </asp:CreateUserWizardStep>
   <asp:CompleteWizardStep ID="CompleteWizardStep1" runat="server">
   </asp:CompleteWizardStep>
</WizardSteps>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

```
 <StartNavigationTemplate>
    <des:Button ID="StartNextButton" runat="server"
       CommandName="MoveNext" Text="Next" Group="CreateUserWizard1" />
 </StartNavigationTemplate>
 <StepNavigationTemplate>
    <des:Button ID="StepPreviousButton" runat="server"
       CausesValidation="False" CommandName="MovePrevious" Text="Previous" />
    <des:Button ID="StepNextButton" runat="server"
       CommandName="MoveNext" Text="Next" Group="CreateUserWizard1" />
 </StepNavigationTemplate>
 <FinishNavigationTemplate>
    <des:Button ID="FinishPreviousButton" runat="server"
       CausesValidation="False" CommandName="MovePrevious" Text="Previous" />
    <des:Button ID="FinishButton" runat="server"
       CommandName="MoveComplete" Text="Finish" Group="CreateUserWizard1" />
 </FinishNavigationTemplate>
</asp:CreateUserWizard>
```

CustomValidator

Many situations where you used your own custom code can now be handled by DES's existing validators. If the logic only involves the controls on the page and does not have to query your business objects, its likely that DES's validators will work. Choose the MultiConditionValidator when you have a boolean expression in your logic.

If you cannot use the other validators, replace it with DES's CustomValidator.

To quickly convert the native validators, run the **Web Application Updater** program on this page with its **Convert native controls to their DES equivalents** option. This will change all validators and buttons for you.

DataGrid

When you want to validate data entry controls in a DataGrid, you must use a TemplateColumn where you define the data entry controls and their validators. You will have to abandon the convenience of the BoundColumn and write your own data assignment code.

When there are any DES validators on the page, whether in the DataGrid or elsewhere, always replace the ButtonColumn and EditCommandColumn controls with DES's equivalents. See "DES's Submit Controls For The DataGrid".

If you are programmatically assigning properties on any of DES's controls, do so in the ItemCreated event, not the ItemDataBound event. (ItemCreated is called each time the page is requested and happens before the attempt to validate. ItemDataBound is only called when you invoke the `DataBind()` method.)

When using AJAX, if any of the controls are not shown until a callback occurs because they are not on the initial view, you must be sure to set up AJAX properly. For most types of controls, you will use the `PeterBlum.DES.AJAXManager.PreregisterForAJAX()` method or the **PreLoadForAJAX** property found on the PageManager control. For controls in the **Peter's Date and Time** module, you will also need to create a dummy control that is preregistered.

Details for all of these issues are in the steps for setting up AJAX in the **General Features Guide**.

DataList

When there are any DES validators on the page, whether in the DataList or elsewhere, always replace the submit controls with DES's equivalents. See "DES's Button, LinkButton, and ImageButton Controls".

If you are programmatically assigning properties on any of DES's controls, do so in the ItemCreated event, not the ItemDataBound event. (ItemCreated is called each time the page is requested and happens before the attempt to validate. ItemDataBound is only called when you invoke the `DataBind()` method.)

Its SelectedIndex property can be validated with the RequiredSelectionValidator.

DetailsView

When you want to validate data entry controls in a DetailsView , you must use a TemplateField where you define the data entry controls and their validators. You will have to abandon the convenience of the BoundField and write your own data assignment code.

If you are programmatically assigning properties on any of DES's controls, do so in the ItemCreated event.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

When there are any DES validators on the page, whether in the DetailsView or elsewhere, always replace the ButtonField and CommandField controls with DES's equivalents. See "DES's Submit Controls For The GridView and DetailsView".

DropDownList

The DropDownList control can be validated with RequiredListValidator, SelectedIndexValidator, and SelectedIndexRangesValidator. They all validate the selected index. You can also evaluate the textual value selected by the DropDownList using most of the validators that support text including CompareToValueValidator, RangeValidator, and RegexValidator.

The Conditions that you can use in Enablers, the MultiConditionValidator, CountTrueConditionsValidator, etc are the RequiredListCondition and SelectedIndexCondition.

When using the **AutoPostBack** property, you can provide validation prior to postback by using the NativeControlExtender. Set the NativeControlExtender's **AutoPostBackValidates** property to `Control` or `ValidationGroup`.

Note: The DropDownList has its own properties for validation on postback: CausesValidation and ValidationGroup. These are not used by DES.

Overcoming a bug in Google Chrome 1

When a DropDownList has **AutoPostBack** set to `true` and you assign a DES validator to that DropDownList, Google Chrome v1 will generate a JavaScript error. Use the NativeControlExtender to avoid this. Add it to any DropDownList whose **AutoPostBack** property is set to `true` and has validators.

Internally the NativeControlExtender swaps the original AutoPostBack scripts with alternatives that work with all browsers.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

FileUpload

The FileUpload control creates the HTML tag `<input type='file'>`. Users often expect that they can add client-side validation to prevent uploading files of the wrong type. Because the browser defines the functionality of the `<input type='file'>` tag, this is not available. In fact, it's a little tricky even to require text in its textbox. This section will give you some suggested solutions.

Checking the FilePath

If the field is required, add a RequiredTextValidator to this control.

If you want to confirm a file extension, use the CompareToStringsValidator with its **MatchTextRule** property set to `EndsWith` and the **Items** collection identifying the valid file extensions. This example checks for the following file extensions: **txt**, **html**, and **htm**.

```
<DES:CompareToStringsValidator ID="CompareToStringsValidator1" runat="server"
      ControlIDToEvaluate="FileUpload1" CaseInsensitive="true"
      MatchTextRule="EndsWith"
      ErrorMessage="Please select only files of type 'txt', 'html' or 'htm'.">
      <Items>
            <DES:CompareToStringsItem Value="txt" />
            <DES:CompareToStringsItem Value="html" />
            <DES:CompareToStringsItem Value="htm" />
      </Items>
</DES:CompareToStringsValidator>
```

You cannot validate that the path points to a real file on the user's hard disk. The browser is designed to provide the user with security that prevents you from evaluating the contents of their hard disk.

Server Side Validation

There are several actions you should take on the server side to confirm that you received a file and it meets your requirements. They are:

- Check the file type from the **FileUpload.PostedFile.ContentType** property
- Check the file length in case the file is too small or large from the **FileUpload.PostedFile.ContentLength** property.
- Check the bytes of the file to make sure they match patterns expected for the file type. This way, users cannot pass you an unwanted file by changing its file extension to match your valid file extensions. You have to acquire the code necessary to evaluate any file format.

Use an IgnoreConditionValidator to report any errors you find with the file.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Example

Here is the HTML for this form. It includes three validators: RequiredTextValidator, CompareToStringsValidator, and IgnoreConditionValidator.

```
<asp:FileUpload ID="FileUpload1" runat="server" />
<DES:RequiredTextValidator ID="RequiredTextValidator1" runat="server"
        ControlIDToEvaluate="FileUpload1" ErrorMessage="Required">
   <ErrorFormatterContainer>
      <DES:TextErrorFormatter Display="Dynamic" />
   </ErrorFormatterContainer>
</DES:RequiredTextValidator>
<DES:CompareToStringsValidator ID="CompareToStringsValidator1" runat="server"
        ControlIDToEvaluate="FileUpload1" CaseInsensitive="true"
        MatchTextRule="EndsWith"
        ErrorMessage="Please select only files of type 'txt'.">
   <Items>
      <DES:CompareToStringsItem Value="txt" />
   </Items>
   <ErrorFormatterContainer>
      <DES:TextErrorFormatter Display="Dynamic" />
   </ErrorFormatterContainer>
</DES:CompareToStringsValidator>
<DES:IgnoreConditionValidator ID="IgnoreConditionValidator1" runat="server">
   <ErrorFormatterContainer>
      <DES:TextErrorFormatter Display="Dynamic" />
   </ErrorFormatterContainer>
</DES:IgnoreConditionValidator>
```

In the Click post back event of your submit control, add this code:

[C#]

```
if (FileUpload1.HasFile)
{
   // test the file content type. This example uses a text file.
   if (FileUpload1.PostedFile.ContentType != "text/plain")
   {
      IgnoreConditionValidator1.IsValid = false;
      IgnoreConditionValidator1.ErrorMessage = "Please provide a text file.";
      return;
   }
   // if desired, test the file length. This example limits it to 10K
   if (FileUpload1.PostedFile.ContentLength > 10 * 1024)
   {
      IgnoreConditionValidator1.IsValid = false;
      IgnoreConditionValidator1.ErrorMessage =
             "The file is too large. Select files that are smaller than 10K.";
      return;
   }
   // perform additional testing on the file bytes to be sure
   // the user isn't trying to sneak an illegal file under a valid content type
   // The developer should acquire the logic that evaluates the file bytes.
/* if (!TestFileBytes())
   {
      IgnoreConditionValidator1.IsValid = false;
      IgnoreConditionValidator1.ErrorMessage = "The file is not a text file.";
      return;
   }      */
}
else
{
   IgnoreConditionValidator1.IsValid = false;
   IgnoreConditionValidator1.ErrorMessage = "Please assign a file";
}
if (PeterBlum.DES.Globals.WebFormDirector.IsValid)
{
   // save the file
}
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

[VB]

```
If FileUpload1.HasFile Then
   ' test the file content type. This example uses a text file.
   If FileUpload1.PostedFile.ContentType <> "text/plain" Then
      IgnoreConditionValidator1.IsValid = False
      IgnoreConditionValidator1.ErrorMessage = "Please provide a text file."
      Return
   End If

   ' if desired, test the file length. This example limits it to 10K
   If FileUpload1.PostedFile.ContentLength > 10 * 1024 Then
      IgnoreConditionValidator1.IsValid = False
      IgnoreConditionValidator1.ErrorMessage = _
             "The file is too large. Select files that are smaller than 10K."
      Return
   End If

   ' perform additional testing on the file bytes to be sure
   ' the user isn't trying to sneak an illegal file under a valid content type
   ' The developer should acquire the logic that evaluates the file bytes.

'    If Not TestFileBytes() Then
'       IgnoreConditionValidator1.IsValid = False
'       IgnoreConditionValidator1.ErrorMessage = "The file is not a text file."
'       Return
'    End If
      */
Else
   IgnoreConditionValidator1.IsValid = False
   IgnoreConditionValidator1.ErrorMessage = "Please assign a file"
End If
If PeterBlum.DES.Globals.WebFormDirector.IsValid Then
   ' save the file
End If
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

FormView

The FormView uses templates to define the entire layout of the control. You should add DES validators and submit controls instead of the originals.

When you define a button that does not require validation, like "Cancel", set its **CausesValidation** property to `false`.

If you are programmatically assigning properties on any of DES's controls, do so in the ItemCreated event.

When using AJAX, if any of the controls are not shown until a callback occurs because they are not on the initial view, you must be sure to set up AJAX properly. For most types of controls, you will use the `PeterBlum.DES.AJAXManager.PreregisterForAJAX()` method or the **PreLoadForAJAX** property found on the PageManager control. For controls in the **Peter's Date and Time** module, you will also need to create a dummy control that is preregistered.

Details for all of these issues are in the steps for setting up AJAX in the **General Features Guide**.

GridView

When you want to validate data entry controls in a GridView , you must use a TemplateField where you define the data entry controls and their validators. You will have to abandon the convenience of the BoundField and write your own data assignment code.

When there are any DES validators on the page, whether in the GridView or elsewhere, always replace the ButtonField and CommandField controls with DES's equivalents. See "DES's Submit Controls For The GridView and DetailsView".

If you are programmatically assigning properties on any of DES's controls, do so in the RowCreated event, not the RowDataBound event. (RowCreated is called each time the page is requested and happens before the attempt to validate. RowDataBound is only called when you invoke the `DataBind()` method.)

When using AJAX, if any of the controls are not shown until a callback occurs because they are not on the initial view, you must be sure to set up AJAX properly. For most types of controls, you will use the `PeterBlum.DES.AJAXManager.PreregisterForAJAX()` method or the **PreLoadForAJAX** property found on the PageManager control. For controls in the **Peter's Date and Time** module, you will also need to create a dummy control that is preregistered.

Details for all of these issues are in the steps for setting up AJAX in the **General Features Guide**.

Its **SelectedIndex** property can be validated with the RequiredSelectionValidator.

HiddenField

Many of DES's validators can evaluate the HiddenField. They are triggered only when the form is submitted since the HTML tag `<input type='hidden'>` does not fire an event when its contents are changed.

HyperLink

DES does not interact with this control. Use it normally.

If you want a HyperLink to validate before moving to another page, use a DES LinkButton instead. It will post back where server side validation will be applied (which you should demand in case the browser doesn't support client-side validation.) To go to another page from the server side, call `Response.Redirect("URL")`.

Image

DES does not interact with this control. Use it normally.

ImageButton

Every ImageButton that should invoke validation needs additional code from DES's validation system. You should either use the ImageButton supplied in DES (see "DES's Button, LinkButton, and ImageButton Controls") or assign the NativeControlExtender to the ImageButton. If you do not, validation will not fire on either the client- or server-side.

ALERT: Always set up Server side validation: Validating With All Other PostBack Event Handlers.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

You should not use the **PostBackUrl** property when the button invokes validation. See "Validation and the PostBackUrl Property" for details.

ImageMap

DES does not interact with this control. Use it normally.

Label

DES does not interact with this control. Use it normally.

DES offers an alternative control called LocalizableLabel, available if you are using the String Lookup System for localization. See the **General Features Guide**.

LinkButton

Every LinkButton that should invoke validation needs additional code from DES's validation system. You should either use the LinkButton supplied in DES (see "DES's Button, LinkButton, and ImageButton Controls") or assign the NativeControlExtender to the LinkButton. If you do not, validation will not fire on either the client- or server-side.

ALERT: Always set up Server side validation: Validating With All Other PostBack Event Handlers.

You should not use the **PostBackUrl** property when the button invokes validation. See "Validation and the PostBackUrl Property" for details.

ListBox

The ListBox control can be validated with RequiredListValidator, SelectedIndexValidator, and SelectedIndexRangesValidator. They all validate the selected index. You can also evaluate the textual value selected by the ListBox using most of the validators that support text including CompareToValueValidator, RangeValidator, and RegexValidator. You can also validate the number of items in the list with the ListSizeValidator.

The Conditions that you can use in Enablers, the MultiConditionValidator, CountTrueConditionsValidator, etc are the RequiredListCondition, SelectedIndexCondition, and SelectedIndexRangesCondition.

When using the **AutoPostBack** property, you can provide validation prior to postback by using the NativeControlExtender. Set the NativeControlExtender's **AutoPostBackValidates** property to `Control` or `ValidationGroup`.

Note: The ListBox has its own properties for validation on postback: CausesValidation and ValidationGroup. These are not used by DES.

Literal

DES does not interact with this control. Use it normally.

Localize

DES does not interact with this control. Use it normally.

Login

The Login control requires changing all validators and buttons to DES's controls. If there are no other data entry controls on the page, it is not necessary to do anything.

To use the Login control with DES, you must change its contents into a Template so that they appear in the `<LayoutTemplate>` node. A complete conversion is shown on the next page. You can copy it from this document and paste it into your web form. If you prefer to start from scratch, here are the steps to convert.

1. Use Visual Studio to add the Login control onto your web form.
2. In design mode, use the **Tasks** menu on the Login control to select **Convert To Template**.
3. Save the document.
4. Run the **Web Application Updater** program on this page with its **Convert native controls to their DES equivalents** option. This will change all validators and buttons for you.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

WARNING: When using Forms Authentication, users have found that their login page's `Page_Load()` method is invoked several times. DES needs files from the **[webapplication]\DES** folder to support your login page. The problem occurs when those files require authentication to access them. The DES folder must be setup with anonymous access for your login page to operate. For details, see the topic "Images and style sheets do not load when the site uses Forms Authentication" in the Troubleshooting section of the **General Features Guide**.

Below is a completely converted page with several extensions beyond what is described above:

- RequiredTextValidators use the TooltipImageErrorFormatter instead of the "*" character.

Here are some suggested enhancements to consider:

- Always test the **PeterBlum.DES.Globals.WebFormDirector.IsValid** property is `true` inside your postback event handler method. For the Login control, use its **LoggingIn** event. If **IsValid** is `false`, set **e.Cancel** to `true`.
- Change the error message text and error formatter type.
- Set the **ShowRequiredFieldMarker** property to `true` on the RequiredTextValidators.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Example

Note: Adobe Acrobat Reader permits copying text.

```
<asp:Login ID="Login1" runat="server">
   <LayoutTemplate>
      <table border="0" cellpadding="1" cellspacing="0" style="border-collapse: collapse">
         <tr>
            <td>
               <table border="0" cellpadding="0">
                  <tr>
                     <td align="center" colspan="2">
                        Log In</td>
                  </tr>
                  <tr>
                     <td align="right">
                        <asp:Label ID="UserNameLabel" runat="server"
                              AssociatedControlID="UserName">User Name:</asp:Label>
                      </td>
                     <td>
                        <asp:TextBox ID="UserName" runat="server"></asp:TextBox>
                        <des:RequiredTextValidator ID="UserNameRequired" runat="server"
                           ControlIDToEvaluate="UserName"
                           ErrorMessage="{LABEL} is required." Group="Login1"
                           Label-LabelControlID="UserNameLabel" Label-TrimTrailingSymbol="true">
                           <ErrorFormatterContainer>
                           <des:TooltipImageErrorFormatter />
                           </ErrorFormatterContainer>
                     </des:RequiredTextValidator>
                     </td>
                  </tr>
                  <tr>
                     <td align="right">
                        <asp:Label ID="PasswordLabel" runat="server"
                              AssociatedControlID="Password">Password:</asp:Label>
                     </td>
                     <td>
                        <asp:TextBox ID="Password" runat="server" TextMode="Password"></asp:TextBox>
                        <des:RequiredTextValidator ID="PasswordRequired" runat="server"
                           ControlIDToEvaluate="Password"
                           ErrorMessage="{LABEL} is required." Group="Login1"
                           Label-LabelControlID="PasswordLabel" Label-TrimTrailingSymbol="true">
                           <ErrorFormatterContainer>
                           <des:TooltipImageErrorFormatter />
                           </ErrorFormatterContainer>
                       </des:RequiredTextValidator>
                     </td>
                  </tr>
                  <tr>
                     <td colspan="2">
                        <asp:CheckBox ID="RememberMe" runat="server" Text="Remember me next time." />
                     </td>
                  </tr>
                  <tr>
                     <td align="center" colspan="2" style="color: red">
                        <asp:Literal ID="FailureText" runat="server" EnableViewState="False">
                         </asp:Literal>
                     </td>
                  </tr>
                  <tr>
                     <td align="right" colspan="2">
                        <des:Button ID="LoginButton" runat="server"
                          CommandName="Login" Text="Log In" Group="Login1" />
                     </td>
                  </tr>
               </table>
            </td>
         </tr>
      </table>
   </LayoutTemplate>
</asp:Login>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

LoginName

DES does not interact with this control. Use it normally.

LoginStatus

DES does not interact with this control. Use it normally.

LoginView

DES does not interact with this control. Use it normally.

Menu

See "Using the Menu Control to Validate the Page".

MultiView

Use the DES validators and submit controls instead of the originals.

When using AJAX, if any of the controls are not shown until a callback occurs because they are not on the initial view, you must be sure to set up AJAX properly. For most types of controls, you will use the `PeterBlum.DES.AJAXManager.PreregisterForAJAX()` method or the **PreLoadForAJAX** property found on the PageManager control. For controls in the **Peter's Date and Time** module, you will also need to create a dummy control that is preregistered.

Details for all of these issues are in the steps for setting up AJAX in the **General Features Guide**.

Panel

DES does not interact with this control. Use it normally.

The Panel has a property **DefaultButton**. This is very similar to DES's **TextBox.EnterSubmitsControlID** property and the "Direct Keystrokes to Click Buttons" features described in the **Interactive Pages User's Guide**. You can use either system to achieve the desired effect. Also, the Panel has the method `Focus()` which sets focus to a control as the page is loaded. It is very similar to the property **PeterBlum.DES.Globals.WebFormDirector.InitialFocusControl**. You can use either system but not both at the same time.

PasswordRecovery

The PasswordRecovery control requires changing all validators and buttons to DES's controls. If there are no other data entry controls on the page, it is not necessary to do anything.

To use the Login control with DES, you must change its contents into a Template so that they appear in the `<UserNameTemplate>` node. A complete conversion is shown below. You can copy it from this document and paste it into your web form. If you prefer to start from scratch, here are the steps to convert.

1. Use Visual Studio to add the Login control onto your web form.
2. In design mode, use the **Tasks** menu on the Login control to select **Convert To Template**.
3. Save the document.
4. Run the **Web Application Updater** program on this page with its **Convert native controls to their DES equivalents** option. This will change all validators and buttons for you.

On the next page you will find a completely converted page with several extensions beyond what is described above:

- RequiredTextValidators use the TooltipImageErrorFormatter instead of the "*" character.

Here are some suggested enhancements to consider:

- Change the error message text and error formatter type.
- Set the **ShowRequiredFieldMarker** property to `true` on the RequiredTextValidator.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Example

Note: Adobe Acrobat Reader permits copying text.

```
<asp:PasswordRecovery ID="PasswordRecovery1" runat="server">
   <UserNameTemplate>
      <table border="0" cellpadding="1" cellspacing="0" style="border-collapse: collapse">
         <tr>
            <td>
               <table border="0" cellpadding="0">
                  <tr>
                     <td align="center" colspan="2">
                        Forgot Your Password?</td>
                  </tr>
                  <tr>
                     <td align="center" colspan="2">
                        Enter your User Name to receive your password.</td>
                  </tr>
                  <tr>
                     <td align="right">
                        <asp:Label ID="UserNameLabel" runat="server"
                        AssociatedControlID="UserName">User Name:</asp:Label>
                     </td>
                     <td>
                        <asp:TextBox ID="UserName" runat="server"></asp:TextBox>
                        <des:RequiredTextValidator ID="UserNameRequired" runat="server"
                           ControlIDToEvaluate="UserName"
                           ErrorMessage="{LABEL} is required." Group="PasswordRecovery1"
                           Label-LabelControlID="UserNameLabel" Label-TrimTrailingSymbol="true">
                           <ErrorFormatterContainer>
                             <des:TooltipImageErrorFormatter />
                           </ErrorFormatterContainer>
                        </des:RequiredTextValidator>
                  </td>
                  </tr>
                  <tr>
                     <td align="center" colspan="2" style="color: red">
                        <asp:Literal ID="FailureText" runat="server" EnableViewState="False">
                        </asp:Literal>
                     </td>
                  </tr>
                  <tr>
                     <td align="right" colspan="2">
                        <des:Button ID="SubmitButton" runat="server"
                          CommandName="Submit" Text="Submit" Group="PasswordRecovery1" />
                     </td>
                  </tr>
               </table>
            </td>
         </tr>
      </table>
   </UserNameTemplate>
</asp:PasswordRecovery>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

PlaceHolder

DES does not interact with this control. Use it normally.

RadioButton

The RadioButton control can be validated with the CheckStateValidator.

Use the CheckStateCondition in Enablers, the MultiConditionValidator, CountTrueConditionsValidator.

When building a list of RadioButtons where at least one must be checked, see the section "Requiring one mark amongst Checkboxes on DataGrid Rows" in the **Tutorials.pdf** document. It works for both checkboxes and radiobuttons.

When using the **AutoPostBack** property of the RadioButton, see "Using Validation with AutoPostBack".

RadioButtonList

The RadioButtonList control can be validated with RequiredListValidator, SelectedIndexValidator, and SelectedIndexRangesValidator. They all validate the selected index. You can also evaluate the textual value selected by the RadioButtonList using most of the validators that support text including CompareToValueValidator, RangeValidator, and RegexValidator.

Use the RequiredListCondition and SelectedIndexCondition in Enablers, the MultiConditionValidator, CountTrueConditionsValidator, etc.

When using the **AutoPostBack** property, you can provide validation prior to postback by using the NativeControlExtender. Set the NativeControlExtender's **AutoPostBackValidates** property to `Control` or `ValidationGroup`.

Note: The RadioButtonList has its own properties for validation on postback: CausesValidation and ValidationGroup. These are not used by DES.

RangeValidator

Replace it with DES's RangeValidator.

To quickly convert the native validators, run the **Web Application Updater** program on this page with its **Convert native controls to their DES equivalents** option. This will change all validators and buttons for you.

RegularExpressionValidator

Replace it with DES's RegexValidator.

To quickly convert the native validators, run the **Web Application Updater** program on this page with its **Convert native controls to their DES equivalents** option. This will change all validators and buttons for you.

RequiredFieldValidator

Replace it with DES's RequiredTextValidator. If you are validating the ListBox, DropDrownList, CheckBoxList, or RadioButtonList, also consider the RequiredListValidator.

To quickly convert the native validators, run the **Web Application Updater** program on this page with its **Convert native controls to their DES equivalents** option. This will change all validators and buttons for you.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Repeater

Use the DES validators and submit controls instead of the originals.

If you are programmatically assigning properties on any of DES's controls, do so in the ItemCreated event, not the ItemDataBound event. (ItemCreated is called each time the page is requested and happens before the attempt to validate. ItemDataBound is only called when you invoke the `DataBind()` method.)

When using AJAX, if any of the controls are not shown until a callback occurs because they are not on the initial view, you must be sure to set up AJAX properly. For most types of controls, you will use the `PeterBlum.DES.AJAXManager.PreregisterForAJAX()` method or the **PreLoadForAJAX** property found on the PageManager control. For controls in the **Peter's Date and Time** module, you will also need to create a dummy control that is preregistered.

Details for all of these issues are in the steps for setting up AJAX in the **General Features Guide**.

SiteMapPath

DES does not interact with this control. Use it normally.

Substitution

DES does not interact with this control. Use it normally.

Table

DES does not interact with this control. Use it normally.

TextBox

Most of DES's validators are designed to use this class. In addition, DES supplies several expanded versions of this control in its **Peter's TextBoxes** and **Peter's Date and Time** modules.

When using the **AutoPostBack** property, you can provide validation prior to postback by using the NativeControlExtender. Set the NativeControlExtender's **AutoPostBackValidates** property to `Control` or `ValidationGroup`.

Note: The TextBox has its own properties for validation on postback: CausesValidation and ValidationGroup. These are not used by DES.

TreeView

The TreeView optionally provides checkboxes. If you want to evaluate that a certain number have been marked, use a CustomValidator with this server-side evaluation method assigned to its **ServerCondition** event. It evaluates the number of checkboxes in the **CheckedNodes.Count** property. In this example, it demands between 1 and 5 checkboxes.

[C#]

```
protected void TreeViewCountSelections(PeterBlum.DES.IBaseCondition pCondition,
   PeterBlum.DES.IConditionEventArgs pArgs)
{
 // must have 1 - 5 checkboxes marked
    pArgs.IsMatch =
   TreeView1.CheckedNodes.Count > 0 && TreeView1.CheckedNodes.Count < 5;
}
```

In `Page_Load()`, set the **ServerCondition** like this:

```
CustomValidator1.ServerCondition +=
   new PeterBlum.DES.ServerConditionEventHandler(TreeViewCountSelections);
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

[VB]

```
Protected Sub TreeViewCountSelections( _
  ByVal pCondition As PeterBlum.DES.IBaseCondition, _
  ByVal pArgs As PeterBlum.DES.IConditionEventArgs)
 ' must have 1 - 5 checkboxes marked
   pArgs.IsMatch = _
     TreeView1.CheckedNodes.Count > 0 And TreeView1.CheckedNodes.Count < 5
End Sub
```

In `Page_Load()`, set the **ServerCondition** like this:

```
AddHandler CustomValidator1.ServerCondition = AddressOf TreeViewCountSelections
```

ValidationSummary

Replace with the DES ValidationSummary.

To quickly convert the native ValidationSummary, run the **Web Application Updater** program on this page with its **Convert native controls to their DES equivalents** option. This will change all validators, the ValidationSummary, and buttons for you.

WebZone and WebParts

DES does not interact with this control. Use it normally.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Wizard

As you define data entry controls in the Wizard pages, use DES's validators. If you add a submit control into the page, it should be a DES submit control.

The navigation controls – Next, Previous, and Finish – default to the native buttons. They need to be converted to DES buttons. Convert the three navigation control sections – StartNavigationTemplate, StepNavigationTemplate, and FinishNavigationTemplate – to their respective templates and add the DES buttons. Here is a wizard with the templates for buttons.

```
<asp:Wizard ID="Wizard1" runat="server" ActiveStepIndex="2"
   OnNextButtonClick="Wizard1_NextButtonClick"
   OnPreviousButtonClick="Wizard1_PreviousButtonClick">
   <WizardSteps>
      <asp:WizardStep runat="server" Title="Step 1">
         This is step 1</asp:WizardStep>
      <asp:WizardStep runat="server" Title="Step 2">
         This is step 2<br />
      </asp:WizardStep>
      <asp:TemplatedWizardStep runat="server" Title="Blah">
         <ContentTemplate>
            This is the an example using a template
         </ContentTemplate>
      </asp:TemplatedWizardStep>
   </WizardSteps>
   <StartNavigationTemplate>
      <des:Button ID="StartNextButton" runat="server"
         CommandName="MoveNext" Text="Next" />
   </StartNavigationTemplate>
   <StepNavigationTemplate>
      <des:Button ID="StepPreviousButton" runat="server"
         CausesValidation="False" CommandName="MovePrevious" Text="Previous" />
      <des:Button ID="StepNextButton" runat="server"
         CommandName="MoveNext" Text="Next" />
   </StepNavigationTemplate>
   <FinishNavigationTemplate>
      <des:Button ID="FinishPreviousButton" runat="server"
         CausesValidation="False" CommandName="MovePrevious" Text="Previous" />
      <des:Button ID="FinishButton" runat="server"
         CommandName="MoveComplete" Text="Finish" />
   </FinishNavigationTemplate>
</asp:Wizard>
```

On buttons whose **CausesValidation** property is `true`, make sure their Click event handler – NextButtonClick, PreviousButtonClick, FinishButtonClick – tests **PeterBlum.DES.Globals.WebFormDirector.IsValid** is `true`. If not, it should assign **e.Cancel** to `true` so the wizard remains on the current page.

When using AJAX, if any of the controls are not shown until a callback occurs because they are not on the initial wizard step, you must be sure to set up AJAX properly. For most types of controls, you will use the `PeterBlum.DES.AJAXManager.PreregisterForAJAX()` method or the **PreLoadForAJAX** property found on the PageManager control. For controls in the **Peter's Date and Time** module, you will also need to create a dummy control that is preregistered.

Details for all of these issues are in the steps for setting up AJAX in the **General Features Guide**.

XML

DES does not interact with this control. Use it normally.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Using Validators with Third Party Controls

When a third party control subclasses from a native ASP.NET control that is already supported by the DES Validation Framework, it will automatically support validation.

For any other control, see the **Using Third Party Controls** guide.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Analyzing the Page's Validation configuration

When validation isn't working correctly, there are a number of factors to look at. Did you setup the button or validators correctly? Do the validation group names match? Did server side validation run? Was a validator disabled?

DES has a built in report that shows exactly how validation is setup on any page. There are two ways to activate this feature: using a querystring parameter or adding a control to the page.

Querystring parameter

This technique only works when the page is requested the first time. It cannot evaluate the results of a postback.

To use it, add the `desdebug=` parameter to the querystring of your URL. You will get a menu. Select the Validation option.

Note: the desdebug= parameter has security restrictions based on IP address. If you are not running from http://localhost, you will either have to expand the IP addresses allowed, or define a password. See "Running a Page from when the Server is not local" in the General Features Guide.

Add a control to the page

This technique takes a little more work but gives you the analysis on every page request, including after the callback.

- Add a Literal or Label control to the page. *Don't worry, it can be removed when you are done.* It will be inserting an extensive element to your page so be prepared for the page to look dramatically different.
- In `Page_Load()`, call this code passing the Literal or Label control:

  [C#]

  ```
  PeterBlum.DES.Globals.WebFormDirector.ValidationReport.OutputDescription(control);
  ```

  [VB]

  ```
  PeterBlum.DES.Globals.WebFormDirector.ValidationReport.OutputDescription(control)
  ```

- When done, remove the control and line of code, or at least set the control to **Visible**=`false`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Supporting a Reset or Clear Button

When you add a Reset button to the page, you expect that the data entry fields will return to their original contents if the user clicks the button. You should also expect the Validators to return to their original state too. When the page is first shown, the Validators should not show any messages. When the page is returned from a post back, the validation ErrorFormatters associated with the original data should show.

DES makes this easy by automatically installing a client-side event handler for the Reset button. Specifically, the `<form>` tag supports an onreset event and DES connects to it. If you have already assigned JavaScript code to this event handler, DES will insert its code first with a trailing semicolon so that your code will continue to run.

If you want to turn off this feature, programmatically set the **PeterBlum.DES.Globals.WebFormDirector.UseOnResetEventHandler** property to `false` in the `Page_Load()` method. (It defaults to `true`.)

```
PeterBlum.DES.Globals.WebFormDirector.UseOnResetEventHandler = false
```

A Reset button must be this type of tag: `<input type='reset'>`.

Some users also add a non-submitting button (`<input type='button'>`) with an onclick event to run some JavaScript code. For example, that code may clear a group of text fields. Those changes will not cause any validation error message to be updated. So DES provides some additional JavaScript for your button's onclick event handler to call. Use the method `PeterBlum.DES.Globals.WebFormDirector.ValidationManager.GetValidateGroupScript()` to return a string that contains the JavaScript that runs validation on all Validator controls of a particular group.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

The ViewState and Preserving Properties for PostBack

DES attempts to limit the use of the ViewState. It preserves the states of **Visible**, **Enabled**, **ReadOnly**, and **Text** properties for you. The rest you must elect to store.

To include a property on these control in the ViewState, do this in `Page_Load()`:

```
Control.ViewStateMgr.TrackProperty("PropertyName")
```

For details on how DES uses the ViewState and the **ViewStateMgr** property, see "The ViewState and Preserving Properties for PostBack" in the **General Features Guide**.

Example

Suppose that you want to preserve the **AutoPostBack** property on TextBox1:

```
TextBox1.ViewStateMgr.TrackProperty("AutoPostBack")
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Validation and the PostBackUrl Property

Submit controls provide the **PostBackUrl** property. This property can make validation very difficult to communicate to the user and is not recommended when the page has validators and the submit control is set up to validate (**CausesValidation**=`true`).

Here's the problem. When you submit the page, server side validation occurs. (You should *never* assume client-side validation will be protecting you as users can turn off javascript or have the wrong browser.) If it finds a problem, it should prevent you from taking a further action, like saving the data and going to another page. Instead, the current page should be redrawn with error messages shown.

When you use **PostBackUrl**, it always jumps to the page specified by **PostBackUrl**, regardless of the validity of the page. (*This only occurs when client-side validation has validated the page.*) While validation will run on the original page (**PostBackUrl** still runs through the page cycle and redirects before OnPreRender), the user will always be placed on the next page. That page does not have the validator controls of the original page. So you have to handle this case. It's not easy:

- You look at the web controls defined on the **Page.PreviousPage** and test each validator's **IsValid** property for `false`.
- You provide a user interface on this second page that tells the user there were errors
- You provide a way to go back to the original page
- The original page needs to be validated against that data so error messages now appear on the page so the user has a context for the error messages.

It is so much easier just to leave **PostBackUrl** blank and use the validation of the original page. If it passes validation (**PeterBlum.DES.Globals.WebFormDirector.IsValid** is `true`) in your post back event handler, call `Server.Transfer()` to show the second page. Otherwise, return to the current page with all validators showing their error messages. The **Page.PreviousPage** property is set up by `Server.Transfer()`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Invoking Validation With Client-Side Scripts

Programmers can easily force a Validator to validate by calling its `Validate()` method in server side code. To do the same on the client-side, call the `DES_FieldChanged()` function. `DES_FieldChanged()` takes one parameter, the ClientID of the field whose value has changed. It returns nothing. `DES_FieldChanged()` runs all Validators attached to the field that was changed. It ignores the **Group** property but respects **Enabler** and **Enabled** properties.

```
DES_FieldChanged("TextBox1");
```

When run, the Validator will show or hide its ErrorFormatter, update ValidationSummary and CombinedErrorMessages controls that support it, and use the features of "Drawing The User's Attention To The Error" associated with onchange events.

To see if the field is valid or not after calling `DES_FieldChanged()`, call `DES_IsValid()`. It takes one parameter, the ClientID of the field that was validated. It returns true for `valid`, `false` for invalid, and `null` if there are no validators attached.

```
DES_FieldChanged("TextBox1");
if (DES_IsValid("TextBox1") == true)
   // code to run when its valid
else
   // code to run when its not valid
```

Example: Changing The Value Of A TextBox

Suppose you have a RequiredTextValidator associated with a TextBox and have written a JavaScript function to change the text within the TextBox. You want to validate the Validator based on this change.

Here is the ASP.NET text:

```
<asp:TextBox id=TextBox1 runat="server"></asp:TextBox>
<des:RequiredTextValidator id=RequiredTextValidator1 runat="server"
   ErrorMessage="Required" ControlIDToEvaluate="TextBox1"
   SupportClientSideLookupByID="True" />
```

Set up the script on the page as a function. This function takes the ClientID to the TextBox and the new text value. The `DES_GetById()` function used here is a browser independent way to convert the ID into the element object.

```
<script type="text/javascript" language="javascript">
function ChangeTextBox(pTextBoxID, pNewText)
{
   var vTBFld = DES_GetById(pTextBoxID);
   vTBFld.value = pNewText;   // change the text
   DES_FieldChanged(pTextBoxID);    // signal validators to update
}
</script>
```

Evaluating all Validators for a Validation Group

If you want to evaluate all validators for a Validation Group, call the `DES_ValidateGroup()` function. `DES_ValidateGroup()` takes the validation group name and returns `true` if all validators are valid and `false` if not.

```
var vIsValid = DES_ValidateGroup("group1");
```

You can have DES install this script into your code on the server side using the PeterBlum.DES.Globals.WebFormDirector.ValidationManager.GetValidateGroupScript Method.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Running a Script when validation occurs

DES provides several hooks where you can add JavaScript code into the validation process. The following properties are defined in the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager** object and on the PageManager control.

Run a script before page-level validation

To running a script before page-leve validation, use the **CustomSubmitFunctionName** property. Define your function according to the property description. Define the **SubmitOrder** property to define that your script runs before validation occurs.

Note: Page-level validation is associated with submitting the page. This feature is not available after a field change causes validation.

Example

Suppose you have a textbox called UserName that gets the user's new user name. It needs to do a test to confirm that user name isn't taken. The site already has a JavaScript function called CheckUserNameCallback() that does a syncronous AJAX call and returns `true` if the name is valid.

```
<script type="text/javascript">
function CheckUserName(group)
{
   var vUserName = DES_GetById("<% = UserName.ClientID %>");
   return CheckUserNameCallback(vUserName.value);
}
</script>
```

```
<des:PageManager id="PageManager1" runat="server"
   CustomSubmitFunctionName="CheckUserName"
   SubmitOrder="CustomConfirmValidate" />
```

Run a script after page-level validation

To run a script after page-level validation occurs, use the **PostPageValidationFunctionName** property. Define your function according to the property description. This function is called whether validation succeeds or fails and is passed a flag indicating what happened.

Example

Suppose you want to dispo.

```
<script type="text/javascript">
function AssistOnError(group, isvalid)
{
   if (!isvalid)
      alert('Please review the list of errors shown here.');
}
</script>
```

```
<des:PageManager id="PageManager1" runat="server"
   PostPageValidationFunctionName="AssistOnError" />
```

Run a script after any validation occurs

Since validation can show or hide elements on the page, you may need to run a script to reposition elements after validation occurs. Use the **PostValidationUpdateScript** property to run your code. It is called after field changes and page submission validation. It does not provide a flag indicating what happened during validation.

```
<des:PageManager id="PageManager1" runat="server"
   PostValidationUpdateScript="YourScript" />
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Changing The Properties of a Validator With Client-Side Scripts

Programmers can easily change any of the server side properties in the `Page_Load()` method. What about doing the same on the client side? Here are the steps.

1. Set the **SupportClientSideLookupByID** property to `true` on the Validator control whose values you want to evaluate or modify on the client side. This will assign the control's **ClientID** to the client-side representation of the Validator so that you can look it up.
2. Call the client-side function `DES_FindAOById()` in your JavaScript code. It takes one parameter, the **ClientID** of the Validator control. It returns an object representing the Validator, called the "Action Object". If it does not find the **ClientID**, it returns `null`. See "Adding Your JavaScript to the Page".

   ```
   var vAO = DES_FindAOById("ClientID");
   ```

3. If `DES_FindAOById()` returns an Action Object, evaluate or modify its properties. To learn about each property, see the "Developers Guide.pdf". If you want to modify the Enabled state, DES provides the `DES_SetEnabled()` function on the client-side for you to call. It takes two parameters: the Action Object (returned from `DES_FindAOById`) and the enabled state (as a Boolean).

   ```
   DES_SetEnabled(vAO, false);   // disables the control
   ```

Example Setting the Enabled State

Suppose you have a RequiredTextValidator associated with a TextBox and two radio buttons which enable and disable the RequiredTextValidator.

Here is the ASP.NET text:

```
<asp:TextBox id="TextBox1" runat="server"></asp:TextBox>
<des:RequiredTextValidator id="RequiredTextValidator1" runat="server"
   ErrorMessage="Required" ControlIDToEvaluate="TextBox1"
   SupportClientSideLookupByID="True" />
<br />
<asp:RadioButton id="RadioButton1" runat="server" Text="Disabled"
   GroupName="Group1" Checked="True"></asp:RadioButton>
<asp:RadioButton id="RadioButton2" runat="server" Text="Enabled"
   GroupName="Group1"></asp:RadioButton>
```

Set up the script on the page as a function. It takes the **ClientID** to RequiredTextValidator1 and the enabled flag.

```
<script type="text/javascript" language="javascript">
function ChangeEnabled(pClientID, pEnabled)
{
   var vAO = DES_FindAOById(pClientID);
   if (vAO != null)
   {
      DES_SetEnabled(vAO, pEnabled);
   }
}
</script>
```

Attach onclick events to the radio buttons to run `ChangeEnabled()`. Do this programmatically in `Page_Load()`:

[C#]

```
RadioButton1.Attributes.Add("onclick", "ChangeEnabled('" +
   RequiredTextValidator1.ClientID + "', false);");
RadioButton2.Attributes.Add("onclick", "ChangeEnabled('" +
   RequiredTextValidator1.ClientID + "', true);");
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

[VB]

```
RadioButton1.Attributes.Add("onclick", "ChangeEnabled('" + _
   RequiredTextValidator1.ClientID + "', false);")
RadioButton2.Attributes.Add("onclick", "ChangeEnabled('" + _
   RequiredTextValidator1.ClientID + "', true);")
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Adding Client-Side Validation Scripts Within Your Server Side Code

Click on any of these topics to jump to them:

- PeterBlum.DES.Globals.WebFormDirector.SubmitManager.GetPostBackEventReference method
- PeterBlum.DES.Globals.WebFormDirector.SubmitManager.GetPostBackClientHyperlink method
- PeterBlum.DES.Globals.WebFormDirector.ValidationManager.GetValidateGroupScript Method

PeterBlum.DES.Globals.WebFormDirector.SubmitManager.GetPostBackEventReference method

Use `PeterBlum.DES.Globals.WebFormDirector.SubmitManager.GetPostBackEventReference()` with controls that submit the page when you want to validate first. `GetPostBackEventReference()` returns a string containing the JavaScript to establish the client-side code that validates and submits the page. It is the same concept as `System.Web.UI.Page.GetPostBackEventReference()`.

All native controls that submit the page can be registered for client-side code using the methods described in "Using Native Button Controls to Submit the Page".

All DES supplied controls that submit the page are automatically registered and require no additional work on your part. (See "DES's Button, LinkButton, and ImageButton Controls".)

All other controls need to have their onclick event handler call the client-side validation code and the ASP.NET framework's `__doPostBack()` function, which submits the page. Use this method for these controls.

This method will not set up server-side validation. For that, see "Validating With All Other PostBack Event Handlers".

The string returned will include two parts: DES validation for the group specified and a call to "__doPostBack". It sets up the call to `__doPostBack()` by using `System.Web.UI.Page.GetPostBackEventReference()`.

Note: This string does not contain the lead text "javascript:" which is required for hyperlinks. Use PeterBlum.DES.Globals.WebFormDirector.SubmitManager.GetPostBackClientHyperlink() to establish a string with that lead text.

[C#]

```
public string GetPostBackEventReference(Control pControl,
   string pGroup);

public string GetPostBackEventReference(Control pControl,
   string pArgument, string pGroup);

public string GetPostBackEventReference(Control pControl,
   string pArgument, string pGroup, bool pCausesValidation);

public string GetPostBackEventReference(Control pControl,
   string pArgument, string pGroup, bool pCausesValidation,
   string pConfirmMessage);

public string GetPostBackEventReference(PostBackOptions pOptions);
```

[VB]

```
Public Function GetPostBackEventReference(ByVal pControl As Control, _
   ByVal pGroup As String) As String

Public Function GetPostBackEventReference(ByVal pControl As Control, _
   ByVal pArgument As String, ByVal pGroup As String) As String

Public Function GetPostBackEventReference(ByVal pControl As Control, _
   ByVal pArgument As String, ByVal pGroup As String, _
   ByVal pCausesValidation As Boolean) As String
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

```
Public Function GetPostBackEventReference(ByVal pControl As Control, _
   ByVal pArgument As String, ByVal pGroup As String, _
   ByVal pCausesValidation As Boolean,
   ByVal pConfirmMessage As String) As String

Public Function GetPostBackEventReference(
   ByVal pOptions As PostBackOptions) As String
```

Parameters

pControl

The submit control. It will not be modified.

pArgument

The parameter passed back to the server control. It is the second parameter in the System.Web.UI.Page.GetPostBackEventReference method.

pGroup

The group name for Validators on the page that are run when this code is invoked. It can be "".

pCausesValidation

When `true`, set up validation. When `false`, prevent client-side validation. Usually you will pass `false` when setting up AutoPostBack features that should not validate, such as a checkbox that must post back to draw other controls.

pConfirmMessage

Optional confirmation message. When supplied, a JavaScript `confirm()` messagebox appears prior to submitting the page. If the user clicks Cancel, the page is not submitted.

pOptions

The `System.Web.UI.PostBackOptions` class gives you a number of additional features for post back.

`GetPostBackEventReference()` uses these properties and passes the rest to `Page.ClientScript.GetPostBackEventReference()`:

- **RequiresJavaScriptProtocol**
- **ValidationGroup**
- **PerformValidation** – when false, do not allow client-side validation

Return value

A string containing JavaScript to validate and submit the page.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

PeterBlum.DES.Globals.WebFormDirector.SubmitManager.GetPostBackClientHyperlink method

`PeterBlum.DES.Globals.WebFormDirector.SubmitManager.GetPostBackClientHyperlink()` returns a string containing the JavaScript to establish the client-side code that validates and submits the page. Use it with hyperlink href= attributes. It really is just a call to `GetPostBackEventReference()` with the lead text "`JavaScript:`".

This method will not set up server-side validation. For that, see "Validating With All Other PostBack Event Handlers".

[C#]

```
public string GetPostBackClientHyperlink(Control pControl,
   string pArgument, string pGroup);
```

[VB]

```
Public Function GetPostBackClientHyperlink(ByVal pControl As Control, _
   ByVal pArgument As String, ByVal pGroup As String) As String
```

Parameters

pControl

The submit control. It will not be modified.

pArgument

The parameter passed back to the server control. It is used by the Command event handler on the server side control.

pGroup

The group name for Validators on the page that are run when this code is invoked. It can be "".

Return value

A string containing JavaScript to validate and submit the page. The string starts with the text "`javascript:`".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

PeterBlum.DES.Globals.WebFormDirector.ValidationManager.GetValidateGroupScript Method

If you have a command that you want to validate the page without posting back, use the `PeterBlum.DES.Globals.WebFormDirector.ValidationManager.GetValidateGroupScript()` method. Use it to return a string that contains the JavaScript that runs validation on all Validator controls of a particular group. You will insert the string into the JavaScript that you attach to your command's onclick or another client-side event.

[C#]

```
public string GetValidateGroupScript(string pGroupName);
```

[VB]

```
Public Function GetValidateGroupScript(ByVal pGroupName As String) As String
```

Parameters

pGroupName

The menu or toolbar control.

Return value

A string containing the JavaScript that updates all Validators for the given group on the client side. The string is actually a JavaScript function that itself returns a Boolean result: `true` if the page is valid and `false` if not. Your code can test the result to determine if any JavaScript code that follows should be executed.

Example

Suppose that you have added a JavaScript function called `ClearFields()` to the page and want it used from Button1. Here is how you would use `GetValidateGroupScript()`:

[C#]

```
Button1.Attributes.Add("onclick", "ClearFields();" +
   PeterBlum.DES.Globals.WebFormDirector.ValidationManager.GetValidateGroupScript("" ) +
   "return false;";
```

[VB]

```
Button1.Attributes.Add("onclick", "ClearFields();" + _
   PeterBlum.DES.Globals.WebFormDirector.ValidationManager.GetValidateGroupScript("" ) +  _
   "return false;"
```

Note: Use 'return false;' to prevent the button from submitting the page.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Support for the Native Validation Framework

Peter's Data Entry Suite is designed to replace all aspects of the Native Validation Framework. While you can keep some of the native Validators on a page, it often makes sense to fully convert to DES. Use the Web Application Updater program to convert an entire application or a selected list of pages. See the **Installation Guide**.

Here are the guidelines if you want to keep some of Microsoft's Validators.

- If you are starting a new page, simply use DES's controls.
- If you have an existing page that doesn't need any of DES's capabilities, you can leave it alone. DES doesn't affect pages that lack its controls.
- You can mix the Validators of both frameworks on the same page with these warnings:
    - The ValidationSummary controls of both frameworks will only show error messages of their own framework.
    - All features introduced by DES are only available to DES controls. This includes the validation group capability.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Using Validator controls to evaluate data not from Controls

The Validator controls are designed to evaluate data in other controls found on the page. However, the concept of validation can be applied to other holders of data. For example, your business logic may want to compare two integer variables. That would be a case for the CompareTwoFieldsValidator, if only you could avoid using controls to deliver those two integers.

DES Validators allow you to deliver data from other sources. If the Validator has a **ControlIDToEvaluate** property, it also has a **ValueToEvaluate** property. When **ValueToEvaluate** is assigned the data, you no longer assign **ControlIDToEvaluate**. Similarly, the **SecondControlIDToEvaluate** property can be ignored if you assign values to **SecondValueToEvaluate**.

Similarly, Condition objects offer **ValueToEvaluate**, and **SecondValueToEvaluate** properties. They also can be used in a very similar way, as shown below.

Advantages of using a Validator object

The Validator control can manage the error message for you. Its error message capabilities include using the String Lookup System and tokens like "{MINIMUM}" and "{COUNT}". See "Defining the Error Message and Associated Labels".

The **Enabler** property is available to provide the additional logic of enabling the validator based on other data.

Advantages of using a Condition object

The Condition object is much lighter weight. It has fewer internal objects and certainly less properties, hosting *only* properties involved in your primary action: evaluating the data.

Example: Validator object

Using the CompareTwoFieldsValidator as an example, where it compares two integers, requiring the first to be greater or equal to the second.

[C#]

```
using PeterBlum.DES.Web.WebControls;
…
CompareTwoFieldsValidator vValidator = new CompareTwoFieldsValidator();
vValidator.ValueToEvaluate = 1;
vValidator.SecondValueToEvaluate = 10;
vValidator.Operator = PeterBlum.DES.ConditionOperator.GreaterThanEquals;
vValidator.ErrorMessage = "First must be greater or equal to the second";
if (!vValidator.Validate())
   throw new Exception(vValidator.GetErrorMessage(true));
```

[VB]

```
Imports PeterBlum.DES.Web.WebControls
…
Dim vValidator As CompareTwoFieldsValidator = New CompareTwoFieldsValidator()
vValidator.ValueToEvaluate = 1
vValidator.SecondValueToEvaluate = 10
vValidator.Operator = PeterBlum.DES.ConditionOperator.GreaterThanEquals
vValidator.ErrorMessage = "First must be greater or equal to the second"
If Not vValidator.Validate()
   Throw New Exception(vValidator.GetErrorMessage(True))
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Example: Condition object

Using the CompareTwoFieldsCondition as an example, where it compares two integers, requiring the first to be greater or equal to the second.

[C#]

```
using PeterBlum.DES.Web.WebControls;
…
CompareTwoFieldsCondition vCondition = new CompareTwoFieldsCondition();
vCondition.ValueToEvaluate = 1;
vCondition.SecondValueToEvaluate = 10;
vCondition.Operator = PeterBlum.DES.ConditionOperator.GreaterThanEquals;
if (!vCondition.Evaluate())
   throw new Exception("First must be greater or equal to the second");
```

[VB]

```
Imports PeterBlum.DES.Web.WebControls
…
Dim vCondition As CompareTwoFieldsCondition = New CompareTwoFieldsCondition()
vCondition.ValueToEvaluate = 1
vCondition.SecondValueToEvaluate = 10
vCondition.Operator = PeterBlum.DES.ConditionOperator.GreaterThanEquals
If Not vCondition.Evaluate()
   Throw New Exception("First must be greater or equal to the second")
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Guidelines for using these objects

- **ValueToEvaluate** and **SecondValueToEvaluate** are not strongly typed. They accept any type of data. Yet the Validator or Condition class itself may require a specific type. For example, SelectedIndexCondition expects an integer (to represent the selected index) and CheckStateCondition expects a boolean.
- Do not assign the **DataType** property. It is intended for conversion from string to a native type. See below.
- To evaluate Conditions, call its `Evaluate()` method. It takes no parameters and returns a boolean. When it returns `true`, the evaluation matched the rules. When it returns `false`, it did not match.
- To evaluate Validators, call its `Validate()` method. It takes no parameters and returns a boolean. When it returns `true`, the evaluation matched the rules. When it returns `false`, it did not match.
- Validators have many properties that play no role in evaluation such as **HiliteFields** and **ShowRequiredFieldMarker**. They should not be assigned. However, you can use the ErrorMessage property to define the error. Use its tokens feature to insert elements, including runtime values, into the error message. If validation fails, call GetErrorMessage(true) to get back the formatted error message.

Converting a string to a native type

There are many mechanisms to convert a string to a native type. .net provides the Parse() method on many of its type classes, like Int32.Parse(value), and Double.Parse(value). If you happen to have a string that was formatted by one of DES's data type textboxes, it may have a format that won't work with Parse(). Instead, use the global method DESTypeConverterManager.ConvertStringToNative():

[C#]

```
static object ConvertStringToNative(string pValue, string pDataTypeName)
```

[VB]

```
Shared Function ConvertStringToNative(ByVal pValue As String,
  ByVal pDataTypeName As String) As Object
```

Example: Converting a string to a native type

[C#]

```
using PeterBlum.DES.Web.WebControls;
…
CompareTwoFieldsCondition vCondition = new CompareTwoFieldsCondition();
vCondition.ValueToEvaluate =
  DESTypeConverterManager.ConvertStringToNative("1", "Integer");
vCondition.SecondValueToEvaluate =
  DESTypeConverterManager.ConvertStringToNative("10", "Integer");
vCondition.Operator = PeterBlum.DES.ConditionOperator.GreaterThanEquals;
if (!vCondition.Evaluate())
  throw new Exception("First must be greater or equal to the second");
```

[VB]

```
Imports PeterBlum.DES.Web.WebControls
…
Dim vCondition As CompareTwoFieldsCondition = New CompareTwoFieldsCondition()
vCondition.ValueToEvaluate =
  DESTypeConverterManager.ConvertStringToNative("1", "Integer")
vCondition.SecondValueToEvaluate =
  DESTypeConverterManager.ConvertStringToNative("10", "Integer")
vCondition.Operator = PeterBlum.DES.ConditionOperator.GreaterThanEquals
If Not vCondition.Evaluate()
  Throw New Exception("First must be greater or equal to the second")
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Attaching Validators to controls inside of a User Control

Let's suppose that you have a User Control with a textbox and you want to attach a validator to that textbox, but the validator is outside of the user control. Here's how to set it up.

1. Add public properties to the user control that exposes the controls you want to validate.

   [C#]

   ```
   public class MyUserControl : System.Web.UI.UserControl
   {
      public System.Web.UI.WebControls.TextBox txtFirstName
      {
         get { return txtFirstName; }
      }
   }
   ```

   [VB]

   ```
   Public Class MyUserControl Inherits System.Web.UI.UserControl

      Public Property txtFirstName As System.Web.UI.WebControls.TextBox
         Get
            Return txtFirstName
         End Get
      End Property
   End Class
   ```

2. Assign the properties to the validator. How you do this depends on if you are working in the ASP.NET markup or in code.

   ### ASP.NET Markup

   The Validator's **ControlIDToEvaluate** and **SecondControlIDToEvaluate** properties accept strings with a special syntax. Normally you just enter the ID of the desired control, but when that ID is inside of another naming container (and a user control is a naming container), you must use that syntax.

   The syntax is:

   [ID of the UserControl].[Name of the property]

   ```
   <des:RequiredTextValidator id="RequiredTextValidator1" runat="server"
      ControlIDToEvaluate="UserControl1.txtFirstName" />
   ```

   ### Code

   Programmatically assign the value of the first property to the validator's **ControlToEvaluate** (not ControlIDToEvaluate) property. If you have a second data entry controls to assign to the validator, use the **SecondControlToEvaluate** property.

   - Write your code `Page_Load()` unless one of the next two items applies.
   - If you create the User Control programmatically, do it anytime after you've attached the User Control object to the Controls collection of another control.
   - If the User Control is in a row of a GridView, ListView, FormView, etc, write you code in the ItemCreated or RowCreated event handler.

   [C#]

   ```
   RequiredTextValidator1.ControlToEvaluate =
      ((MyUserControl) UserControl1).txtFirstName;
   ```

   [VB]

   ```
   RequiredTextValidator1.ControlToEvaluate = _
      CType(UserControl1, MyUserControl).txtFirstName
   ```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

ValidationSummary Control

The ValidationSummary control displays all errors on the page when the page is submitted. It is hidden until the user submits the page and errors are found. It consolidates the errors into one area. (The Validator controls still report their errors unless you elect to hide them.)

Click on any of these topics to jump to them:

- Features
- Using ValidationSummary Control
  - When the Control Is Shown and Hidden
  - Setting the Appearance and Behavior
  - The Related Control Feature
  - Adding ErrorMessages At Runtime
  - Using the ValidationSummary as a Label for "There are errors"
- Adding a ValidationSummary Control
- Properties of the ValidationSummary Control
- HTML Structure of the ValidationSummary Control
- Online examples

You can have several ValidationSummary controls on the page to show the same error information above and below the form fields. Or you may have different groups of fields, each with its own ValidationSummary control.

By default, the ValidationSummary control shows the same error message as shown on the Validator control reporting the error. However, you can specify a second error message on each Validator control, in the **SummaryErrorMessage** property to provide a different error description in the ValidationSummary control. For example, when your Validator's error message is "This field requires an entry", that text is ambiguous when shown in a list of errors. Provide the error "The *FieldName* field requires an entry" in the **SummaryErrorMessage**. See "Defining the Error Message and Associated Labels".

This control provides extensive formatting properties including a header, footer, and different formats for the list of error messages. There are separate style sheets for each of these elements. Here are several formats for the same list of errors.

This shows the default formatting:

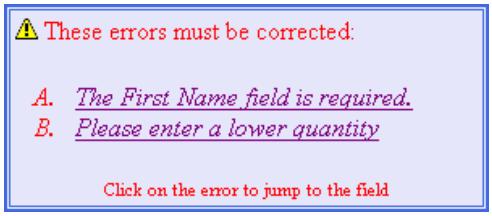
- The First Name field is required.
- Please enter a lower quantity

This includes a header and a different list format:

These errors must be corrected:

+ The First Name field is required.
+ Please enter a lower quantity

Here is a heavily formatted ValidationSummary with a footer and hyperlinks on each of the error messages.



With some programming, you can develop your own appearance for the ValidationSummary Control. See the Developer's Kit.

The class `PeterBlum.DES.Web.WebControls.ValidationSummary` is subclassed from System.Web.UI.WebControls.WebControl. It inherits all of the properties, methods, and events of WebControl.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Features

Like the native ValidationSummary control supplied with the .Net framework, it appears when the user submits and errors were found. It shows the error messages found throughout the page in one place. It has much more features than the native ValidationSummary.

- You can define a header and footer text string and control the horizontal alignment. (Microsoft's only has a left-justified header.)
- The header supports an optional image and includes formatting rules where to position the image against the text.
- The error messages can be shown with the following styles using the **DisplayMode** property:
    - List – Each message is on its own line. You can define text that precedes each message such as a dash ("-").
    - BulletedList – A list of messages that uses the HTML tags: `<ul>` or `<ol>`. You determine what symbol precedes each message: circle, disc, square, alphabetic, numeric, and roman.
    - SingleParagraph – Each message is consolidated into one paragraph. You can define the text that separates each message such as a semicolon or space.
    - Table – A list style that permits changing the styles of alternating rows or assigning borders on each table row.
- By setting **HyperLinkToFields**, each error message can support a hyperlink. When clicked, the focus is moved to the first field associated with the error message.
- There are separate style classes for the overall validation summary and the error messages. It's all placed into a `<div>` control to allow you to develop nice borders and backgrounds.
- Since the ValidationSummary may appear on a very tall web form, users may not always see it. The ValidationSummary can be connected to another field on the form that is shown and hidden simultaneously with the ValidationSummary. For example, define a Label with the text "There are errors on the page" on the opposite side of the form. Or use this feature to add other elements around the ValidationSummary that are not built into the control, such as an enclosing table.
- If you have different validation groups, you may want multiple ValidationSummary controls. A ValidationSummary can show any specific group or all groups at once.
- Use the **AutoUpdate** property to interactively change the ValidationSummary as the user corrects the fields. Once the ValidationSummary is shown, each field change validation will also redraw the ValidationSummary. This way, users can see their progress in correcting the errors.
- Use the **ScrollIntoView** property to scroll the ValidationSummary control into view when the user submits and an error is displayed in the control.
- The ValidationSummary supports localization.
- The RelatedControl feature allows you to have the ValidationSummary show and hide another control on the page. Its used for:
    - To show a message elsewhere on the page telling the user to fix errors
    - To show a message when errors were shown but now cleaned up
    - To show a message when there are no errors on the page
    - To augment the ValidationSummary control with additional HTML parts outside of the ValidationSummary itself.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Using ValidationSummary Control

To add the ValidationSummary control, see "Adding a ValidationSummary Control".

Click on any of these topics to jump to them:

- When the Control Is Shown and Hidden
- Setting the Appearance and Behavior
  - Overall Appearance
  - Header
  - Error Messages
  - Footer
- The Related Control Feature
- Adding ErrorMessages At Runtime
- Using the ValidationSummary as a Label for "There are errors"
- Online examples

When the Control Is Shown and Hidden

This control is displayed when the user attempts to submit the page and validation errors are found. The control that submits the page has a validation group name in its **Group** property. Set the ValidationSummary's **Group** property to match the button's **Group** property value. If you want the ValidationSummary to display the errors from different groups, use a pipe (|) delimited list of group names or "*" for all groups.

Normally the ValidationSummary control will change its contents to reflect the current list of errors each time the user attempts to submit the page. You can override this behavior by setting **AutoUpdate** to `true`. It will first show errors based on the **AutoUpdateFirstShows** property. Then it will update each time a validator shows or hides its error message. **AutoUpdateFirstShows** defaults to showing when the user first attempts to submit the page. You can make the ValidationSummary appear on the first error message or after at least 2-5 errors are shown.

With the **ScrollIntoView** property, the ValidationSummary control will scroll into view each time the user attempts to submit the page, so they can review the errors.

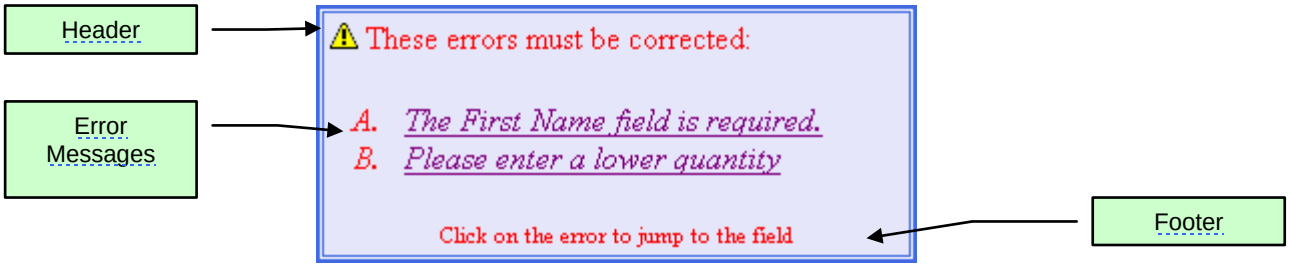
See "When to Update Properties".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Setting the Appearance and Behavior

Here are the elements to the ValidationSummary control:



Click on any of these topics to jump to them:

- Overall Appearance
- Header
- Error Messages
- Footer
- Online example

Overall Appearance

The outer frame of the control, which contains the remaining parts. You often establish a border and background color on this part. Edit these characteristics with the style sheet class `DESValSummary`, which is declared in the **\DES\Appearance\Validation\Validation.css** file as follows:

```
.DESValSummary
{
   color: red;
}
```

By default, the ValidationSummary's width is 100% of the containing control or Page. It's a good idea to set the **Width** property to a preferred size.

See "Overall Appearance Properties".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Header

The header provides text and an optional image before the error messages. Most of the time it appears above the error messages. But you can place it to the left of the messages when you use the "paragraph style" for the error messages.

To add text to the header, set it in the **HeaderText** property. It accepts HTML, so you can create additional formatting. It also allows the "{COUNT}" token to show the number of errors.

If you want an image to appear in the header, set its URL in the **HeaderImageUrl** property. By default, the image appears to the left of the header text in separate columns. You can customize the relationship between text and image with the **HeaderElementPositions** property.

See "Header Properties".

The header uses the style sheet class `DESValSummaryHeader` to adjust its appearance. It is declared in the **\DES\Appearance\Validation\Validation.css** file as follows:

```
.DESVALSummaryHeader
{
   color: red;
}
.DESVALSummaryHeader TR
{
   color: red;
}
```

If you want to show a single message without the individual error messages, set up the HeaderText to that message. Then set **DisplayMode** to `None`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Error Messages

The body of the ValidationSummary shows a list of the error messages from validators reporting errors. The text of these messages comes from the Validators, either in the **SummaryErrorMessage** or **ErrorMessage** property, whichever is defined. See "Defining the Error Message and Associated Labels".

Make it easier for the user to locate the field with an error by setting **HyperLinkToFields** to `true`. Each error message will become a hyperlink that when clicked, moves the focus to the control with the error.

There are several ways to present the error messages: a list, a "bulleted" list using the `<ul>` or `<ol>` tags, table, and, paragraph, where messages are back-to-back without line breaks. They are set using the **DisplayMode** property.

Online example

Here are examples of each **DisplayMode** where 3 error messages are displayed.

DisplayMode: List

Each error message is shown on a separate line, optionally preceded by an image in the **ListLeadImageURL** property and text in the **ListLeadText** property. You can add any HTML between the items with the **ListRowSeparator**.

The Start Date must be assigned.
The End Date must be assigned.
The FullName must be assigned.

DisplayMode: BulletList

The error messages are shown using the `<ul>` or `<ol>` tags for formatting. The **BulletListType** property determines the **type** attribute for the `<ul>` or `<ol>` tags.

- The Start Date must be assigned.
- The End Date must be assigned.
- The FullName must be assigned.

DisplayMode: SingleParagraph

Error messages are concatenated, with the **SingleParagraphSeparator** property providing a separator character. They word-wrap at the overall width of the control.

The Start Date must be assigned. The End Date must be assigned. The FullName must be assigned.

DisplayMode: Table

The error messages are formatted in a one column table, where each error message gets its own row. Initially, this doesn't appear to be much different from the `List` style (which uses `<br />` tags to separate error messages). However, with table cells, you can apply a different style sheet to even and odd rows. This allows for different backgrounds on alternating rows. You can also use the styles to include a border, such as a top border on each row or to change the justification of error messages (for example, centering them). Normally each row uses the style sheet class of `DESValSummaryErrors`. The alternating row uses the style sheet class of `DESVALSummaryAltRows`. Both are declared in the **\DES\Appearance\Validation\Validation.css** file.

The Start Date must be assigned.
The End Date must be assigned.
The FullName must be assigned.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Style Sheet Classes

The `DESValSummaryErrors` class is associated with the error messages. Generally you edit it to change the font of the errors. If you enable hyperlinks on error messages (with the **HyperLinkToFields** property), they can have a different style by declaring new styles of the same name like this "`DESValSummaryErrors A`", "`DESValSummaryErrors A:hover`", "`DESValSummaryErrors A:active`", and "`DESValSummaryErrors A:link`". See an HTML style sheet guide for more on setting up classes for particular tags.

These classes are declared in the **\DES\Appearance\Validation\Validation.css** file:

```
.DESValSummaryErrors
{
   color: red;
   margin-top:6px;
}

.DESVALSummaryErrors A
{
   color: red;
   margin-top:6px;
}
```

If you want to apply font styles to any of the tokens embedded in the **SummaryErrorMessage** property, see "Applying Styles To Tokens".

If you want certain validators to use alternative style sheets for their error messages, set their **OtherStyles** property to `Alternative`. The ValidationSummary control will use these style sheet classes instead of those above:

```
.DESValSummaryErrorsAlt
{
   color: #ff8c00; /* dark orange */
   margin-top:6px;
}

.DESVALSummaryErrorsAlt A
{
   color: #ff8c00; /* dark orange */
   margin-top:6px;
}
```

See "Error Message Formatting Properties".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Footer

The footer provides text and HTML that appears after the error messages.

To add text to the footer, set it in the **FooterText** property. It accepts HTML, so you can create additional formatting. It also allows the "{COUNT}" token to show the number of errors.

See "Footer Properties".

The footer uses the style sheet class `DESValSummaryFooter` to adjust its appearance. It is declared in the **\DES\Appearance\Validation\Validation.css** file as follows:

```
.DESVALSummaryFooter
{
   color: red;
}
.DESVALSummaryFooter TR
{
   color: red;
}
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

The Related Control Feature

The ValidationSummary control may not be the only element on the page that you want to appear while errors are shown. Perhaps you want a label near the top of the page to say "Fix the errors", while the ValidationSummary appears near to the Submit button. Use the **RelatedControlID** property to let the ValidationSummary control the visibility of another control.

The RelatedControl can show when the ValidationSummary is hidden or visible depending on the **RelatedControlDisplayMode** property. This gives you several ways to communicate errors:

- Show a message elsewhere on the page telling the user to fix errors. This is the default.
- Show a message when errors were shown but now cleaned up. Set **RelatedControlDisplayMode** to `AllErrorsFixed`.
- Show a message when there are no errors on the page, even before any data entry. Set **RelatedControlDisplayMode** to `NoErrorsShown`.

Online example

See "Related Control Properties"

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Adding ErrorMessages At Runtime

By design, the ValidationSummary only shows error messages from Validators on the page. Those validators must be enabled and have their **IsValid** property set to `false`. A common situation is when a business object returns a list of error messages that you want to show in the ValidationSummary. Use the `AddMessage()` method on the ValidationSummary control to add those messages.

The `AddMessage()` method lets you add messages with optional HTML. You can associate them with a specific data entry control so that the **HyperLinkToFields** and **AutoUpdate** features work with the message.

Note: Since the ValidationSummary only shows on a postback, this method respects that and does not show when IsPostBack = false. You can override this behavior by setting PeterBlum.DES.Globals.WebFormDirector.IsPostBack to true when you call AddMessage().

Online example

There are two forms of the `AddMessage()` method:

[C#]

```
public void AddMessage(string pMessage);

public void AddMessage(string pMessage,
   Control pDataEntryControl, bool pHideWhenChanged);
```

[VB]

```
Public Sub AddMessage(ByVal pMessage As String)

Public Sub AddMessage(ByVal pMessage As String, _
   ByVal pDataEntryControl As Control, _
   ByVal pHideWhenChanged As Boolean)
```

Parameters

pMessage

The error message. It can contain HTML tags.

pDataEntryControl

When assigned to a data entry control object, the error message will be connected to that control. If the **HyperLinkToFields** property is `true`, the message will include a hyperlink to move the focus to that control.

If you pass `null`/`Nothing`, the error message will remain visible after the user submits. In all other situations, the error message is removed when the user clicks the submit button, even if there are other validation errors.

pHideWhenChanged

When *pDataEntryControl* is assigned, use this to determine if an edit to that data entry control will immediately hide the message in the ValidationSummary. This requires the **AutoUpdate** property to be `true`. If this or the **AutoUpdate** property is `false`, the error message will be removed when the user clicks the submit button, even if there are other validation errors.

When you use the `AddMessage(message)` method, the error message will be removed when the user clicks the submit button and when **AutoUpdate** refreshes the control, even if there are other validation errors.

Use the second form of `AddMessage()` to have more control over when there are validation errors.

The `AddMessage(message)` method always positions its errors at the end of the ValidationSummary's list of errors. The second form of `AddMessage()` positions them first, if the ValidationSummary is above validators and last if the ValidationSummary is below validators.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Example

Adds two messages. The first will be a message without a data entry control. The second will connect to TextBox1.

[C#]

```
ValidationSummary1.AddMessage("Enter a smaller quantity");
ValidationSummary1.AddMessage("The password cannot contain your name",
   TextBox1, true);
```

[VB]

```
ValidationSummary1.AddMessage("Enter a smaller quantity")
ValidationSummary1.AddMessage("The password cannot contain your name", _
   TextBox1, True)
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Using the ValidationSummary as a Label for "There are errors"

Sometimes you want your page to show a label saying "There are errors" or perhaps a graphic above the entry fields. If you are already using the ValidationSummary, you can add a label and hook it up with the RelatedControl feature. See "The Related Control Feature".

If there is no ValidationSummary on the page, here are two approaches.

Show when the user attempts to submit the page

Add a ValidationSummary control where the label should go. Then set these properties:

- **HeaderText** property – Set to the text of the label, such as "`There are errors`"
- **HeaderImageUrl** property – If you want an image, assign its URL here.
- **DisplayMode** property – Set to `None`.
- **Group** property – Set to the validation group name or "*" if you want to use it for all groups.

Show as edits are made

If you want a label to appear as edits are made, follow these steps:

1. Add a Label control with the desired text and style or add an Image control with the desired image.
2. Add this line of code in `Page_Load()`:

```
PeterBlum.DES.Globals.WebFormDirector.ValidationManager.PostValidationUpdateS
cript =
   "DES_GetByID('" + Control.ClientID + "').style.display =
   gDES_PageIsValid ? 'none' : 'inline';"
```

   Replace *Control* with the ID of the Label or Image control.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Adding a ValidationSummary 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)

These steps anticipate that the `<%@ Register TagPrefix="des" %>` tag and the `<link>` to the stylesheet file have been established. If not, see "Preparing a page for DES controls" in the **General Features Guide**. It covers issues like style sheets, AJAX, and localization.

1. Select the location on the page where the ValidationSummary will appear. The ValidationSummary creates a `<div>` tag when written to the page. As a result, any HTML to put immediately after this control will appear on a new row of the browser. So if you want the ValidationSummary to appear side-by-side with other HTML, use a `<table>` where the ValidationSummary is within one cell and the other HTML is in another.
2. Add a ValidationSummary control to the page.

Visual Studio and Visual Web Developers Design Mode Users

Drag the ValidationSummary control from the Toolbox onto your web form.

Text Entry Users

Add the control to the `<form>` area:

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

Programmatically creating the Validator control

- Identify the control which you will add the ValidationSummary control to its **Controls** collection. Like all ASP.NET controls, the ValidationSummary 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 ValidationSummary control class. The constructor takes no parameters.
- Assign the **ID** property.
- Add the ValidationSummary control to the **Controls** collection.

In this example, the ValidationSummary is created with an **ID** of "ValidationSummary1". It is added to PlaceHolder1.

[C#]

```
PeterBlum.DES.Web.WebControls.ValidationSummary vVS = new
PeterBlum.DES.Web.WebControls.ValidationSummary();
vVS.ID = "ValidationSummary1";
PlaceHolder1.Controls.Add(vVS);
```

*Note: The namespace for these controls is PeterBlum.DES. If you prefer, add a **using** clause to that namespace on your form.*

[VB]

```
Dim vVS As PeterBlum.DES.Web.WebControls.ValidationSummary = _
   New PeterBlum.DES.Web.WebControls.ValidationSummary()
vVS.ID = "ValidationSummary1"
PlaceHolder1.Controls.Add(vVS)
```

*Note: The namespace for these controls is PeterBlum.DES. If you prefer, add an **Imports** clause to that namespace on your form.*

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

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 ValidationSummary. Here are some recommended properties to set:
   - When you are using validation groups, the ValidationSummary will appear with error messages associated with the submit button's group. Use the **Group** property to control the groups.
   - The ValidationSummary is contained within a `<div>` tag. The default `<div>` is the width of the page or containing HTML element (like a table cell). If you don't enclose it in something that controls its width, set the **Width** property to the width that you'd like.
   - Set the style of the error messages section with the **DisplayMode** property. All options have a number of supporting properties to customize the format further. See "Error Messages".
   - If you want some text to precede the error messages, use the **HeaderText** property.
   - Add hyperlinks to the error messages that set focus to the field with the error using **HyperLinkToFields**.
   - Make the ValidationSummary update as errors are fixed by using the **AutoUpdate** property.
4. 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.

For a demo, see http://www.peterblum.com/DES/DemoValSum.aspx.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Properties of the ValidationSummary Control

The ValidationSummary control is subclassed from System.Web.UI.WebControls.WebControl. It contains additional properties and methods not described here.

Click on any of these topics to jump to them:

- When to Update Properties
- Error Message Formatting Properties
- Overall Appearance Properties
- Header Properties
- Footer Properties
- Behavior Properties
- Related Control Properties

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

When to Update Properties

For an overview, see "When the Control Is Shown and Hidden".

The Properties Editor lists these properties in the "When To Update" category.

- **Group** (string) – The ValidationSummary control shows the error messages of Validators whose **Group** property matches the **Group** property on the submit control. Set this property to match the group name of a submit control. It will not show error messages when a submit control's **Group** differs from this property. For example, if you have a submit control with "Group1", this Validator will only show Group1 Validator errors when this property is also set to "Group1".

  The ValidationSummary can support multiple groups. This allows you to have one ValidationSummary on the page that shows the error messages specific to the submit control's group that activated it. To support multiple groups, enter each group name separated by a pipe character. For example: "Group1|Group2" (case insensitive).

  If you want to match to any submit control's group, enter "`*`".

  When **Group** = "", it matches a submit control whose **Group** property is also "".

  It defaults to "".

- **AutoUpdate** (Boolean) – Once the ValidationSummary control is shown, it can maintain the initial list of error messages or remove them as the user corrects them. When `true`, the errors are removed as they are corrected without posting back to the server. When `false`, the initial list of error messages remains until the user submits again.

  When `true`, once the ValidationSummary is shown, any change to the Validators will be reflected, including new errors. When all errors are fixed, the ValidationSummary control is removed from the page and will not show up again unless the user clicks the submit control again.

  By default, the ValidationSummary first shows when the user attempts to submit the page. From there, it updates until all errors are gone. You can change this rule with the **AutoUpdateFirstShows** property, below.

  It defaults to `false`.

- **AutoUpdateFirstShows** (enum PeterBlum.DES.AutoUpdateFirstShows) – When using **AutoUpdate**, this determines the first time the ValidationSummary appears. It usually appears after the user attempts to submit. Change it to appear after a certain number of errors are on the page.

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

  - `Submit` - User attempts to submit.
  - `OneError` - At least one error is shown.
  - `TwoErrors` - At least two errors are shown.
  - `ThreeErrors` - At least three errors are shown.
  - `FourErrors` - At least four errors are shown.
  - `FiveErrors` - At least five errors are shown.

  It defaults to `AutoUpdateFirstShows.Submit`.

- **ScrollIntoView** (enum PeterBlum.DES.ScrollIntoViewStates) – When the page is submitted and the ValidationSummary is shown, this scrolls the page on the client-side to show the ValidationSummary control. It has no effect on server side generation of this control.

  This feature requires that the browser support the DOM/DHTML method 'scrollIntoView' on `<div>` tags. For browsers that do not, this property is ignored. Browsers that support this property are Internet Explorer for Windows, FireFox, Netscape 6+, and Safari.

  The values of the enumerated type `PeterBlum.DES.ScrollIntoViewStates` are:

  - `None` – The feature is disabled. This is the default.
  - `Top` – Assure that the top of the ValidationSummary is shown. This is usually best when the ValidationSummary is above the data entry area.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- `Bottom` – Assure that the bottom of the ValidationSummary is shown. This is usually best when the ValidationSummary is below the data entry area.

The actual scrolling is handled by the browser using the DOM/DHTML `scollIntoView()` method. DES asks the browser to assure that the top or bottom of the ValidationSummary is in the view. As a result, the browser may not fully expose the ValidationSummary and the user may still need to do some scrolling.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Error Message Formatting Properties

For an overview, see "Error Messages".

The Properties Editor lists these properties in the "Error Message Formatting" category.

- **DisplayMode** (enum PeterBlum.DES.Web.WebControls.ValidationSummaryDisplayMode) – Determines how error messages are displayed. See "Error Messages" for pictures of each style.

  The enumerated type `PeterBlum.DES.Web.WebControls.ValidationSummaryDisplayMode` has these values:

  - `None` – Do not display the error messages. This is useful when you want to show a single piece of text instead of the error messages. That text should be assigned to the HeaderText property.
  - `List` - Each error message is shown on a separate line, optionally preceded by an image in the **ListLeadImageURL** property and text in the **ListLeadText** property. You can add any HTML between the items with the **ListRowSeparator**.
  - `BulletList` – The error messages are shown using the `<ul>` or `<ol>` tags for formatting. The **BulletListType** property determines the **type** attribute for the `<ul>` or `<ol>` tags.
  - `SingleParagraph` – Error messages are concatenated, with the **SingleParagraphSeparator** property providing a separator character. They word-wrap at the overall width of the control.
  - `Table` – The error messages are formatted in a one column table, where each error message gets its own row. This allows for different backgrounds on alternating rows.

    Odd number rows use the style sheet class of `DESValSummaryErrors`. The alternating rows use the style sheet class of `DESVALSummaryAltRows`. Both are declared in the **\DES\Appearance\Validation\Validation.css** file.

  It defaults to `ValidationSummaryDisplayMode.BulletList`.

- **ErrorMessagesCssClass** – Provides the style sheet class name used for all error messages.

  It defaults to "`DESValSummaryErrors`".

  These styles are declared in **\DES\Appearance\Validation\Validation.css**:

```
.DESValSummaryErrors
{
   color: red;
   margin-top:6px;
}

.DESVALSummaryErrors A
{
   color: red;
   margin-top:6px;
}
```

  If **DisplayMode** = `Table` and **TableAltRowCssClass** is associated with another style sheet class name, this will appear on odd rows including the first row.

- **ErrorMessagesCssClass_Alt** – Provides the style sheet class name used for error messages of validators whose **OtherStyles** property is set to `Alternative`. It only affects the error message itself (plus the hyperlink), not the overall row, which is managed by **ErrorMessagesCssClass** and **TableAltRowCssClass**.

  It defaults to "`DESValSummaryErrorsAlt`".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

These styles are declared in **\DES\Appearance\Validation\Validation.css**:

```
.DESValSummaryErrorsAlt
{
   color: #ff8c00; /* dark orange */
   margin-top:6px;
}

.DESVALSummaryErrorsAlt A
{
   color: #ff8c00; /* dark orange */
   margin-top:6px;
}
```

- **HyperLinkToFields** (Boolean) – When `true`, each error message becomes a hyperlink. When clicked, the focus will jump to the first field associated with the error. When `false`, the text is not shown as a hyperlink. It defaults to `false`.

  The control assigned to the Validator's **ControlIDToEvaluate** property is where focus jumps when the user clicks on the hyperlink. When using a CustomValidator or IgnoreConditionValidator, be sure to assign **ControlIDToEvaluate** if you want it to offer a hyperlink. For MultiConditionValidators and others that have a list of controls, the first control identified in the list of controls is used.

- **HyperLinksToolTip** (string) – When **HyperLinkToFields** is `true`, the error messages section can include a tooltip to describe what happens if they click. When this is "", no tooltip is shown. It defaults to "`Click to select the field`".

- **HyperLinksToolTipLookupID** (string) – Gets the value for **HyperLinksToolTip** 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 ValidateMisc. If no match is found OR this is blank, **HyperLinksToolTip** 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 "".

- **ListLeadImageURL** (string) – Used when **DisplayMode** = `List`. Its value precedes each error message. Use it to add an image that differs from what you can do with `<ol>` and `<ul>` tags. It defaults to "".

  When **ListLeadImageURL** and **ListLeadText** are both assigned, the image appears to the left of the text. This allows you to use **ListLeadText** to insert some space (like '` `') between the image and the error message.

  You can put a tilde ("~") as the first character. The tilde will be replaced by the path to the web application. This allows you to relocate the web application without updating the property. For example, if your web app starts at http://localhost/MyApp and the image is in /MyApp/Images/Image.gif, set this to "~/Images/Image.gif". Later when the site relocates to a real domain where http://www.mydomain.com/images/image.gif references the file, it will continue to work.

- **ListLeadText** (string) – Used when **DisplayMode** = `List`. Its value precedes each error message. Use it to add formatting that differs from what you can do with `<ol>` and `<ul>` tags, such as using "*" and "-". It defaults to "".

- **ListRowSeparator** (string) – Used when **DisplayMode** = `List`. Normally, error messages are preceded by the **ListLeadText** and followed by a `<br />` tag. The **ListRowSeparator** allows you to add HTML that appears after the `<br>` to separate one error message from another. Typically, this will be used with the `<hr />` tag although one could set up an image to go between them.

  When assigned, this text is added after every error message's `<br />` except for the last one:

  `[ListLeadText]`*`errormessage`*`<br />`

  `[ListRowSeparator]`

  `[ListLeadText]`*`errormessage`*`<br />`

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

`[ListRowSeparator]`

`[ListLeadText]`*`errormessage`*`<br />`

It does not add a `<br />` tag when defined. That's because an `<hr />` tag forms its own line. So the user will have to add `<br />` for tags that do not form its own line (like an `<img>` tag).

Reminder: If your page follows the XHTML standard, make sure your tags conform to XHTML.

- **BulletListType** (enum PeterBlum.DES.BulletListType) - Used when **DisplayMode** = `BulletList`. It determines the style of the bullets, using either `<ul>` or `<ol>` tags with a specific **type** attribute. The enumerated type `PeterBlum.DES.BulletListType` has these values:
  - `Circle - <ul type='circle'>`
  - `Disc - <ul type='disc'>`
  - `Square - <ul type='square'>`
  - `AlphaCharsUppercase - <ol type='A'>`
  - `AlphaCharsLowercase - <ol type='a'>`
  - `RomanNumeralsUppercase - <ol type='I'>`
  - `RomanNumeralsLowercase - <ol type='i'>`
  - `Numbers - <ol type='1'>`

  It defaults to `BulletListType.Disc`.

- **SingleParagraphSeparator** (string) – Used when **DisplayMode** = `SingleParagraph`. It is a string inserted between each error message. It defaults to " " (a space).

- **TableAltRowCssClass** (string) – Used when **DisplayMode** = `Table`. Provides the style sheet class name used for even rows. When "", the **ErrorMessageCssClass** value is used for all rows.

  It defaults to "`DESValSummaryAltRows`".

  This style is declared in **\DES\Appearance\Validation\Validation.css**:

```
.DESValSummaryAltRows
{
   color: red;
   background-color: #add8e6; /* lightblue */
}

.DESValSummaryAltRows A
{
   color: red;
   background-color: #add8e6; /* lightblue */
}
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Overall Appearance Properties

For an overview, see "Overall Appearance".

The Properties Editor lists these properties in the "Overall Appearance" category.

- **CssClass** (string) – The style sheet class that is applied to the overall control. Use it to establish the standard font, borders, size (or use the Width property), and background.

  It defaults to "`DESVALSummary`".

  The style is declared in **DES\Appearance\Validation\Validation.css**:

  ```
  .DESVALSummary
  {
     color: Red;
  }
  ```

- **Width** (System.Web.UI.WebControls.Unit) – The width of the control. By default, the width is 100% of the containing block-type tag or the page if there is no containing tag.

- **BackColor**, **BorderColor**, **BorderStyle**, **BorderWidth**, **Columns**, **Font**, **ForeColor**, **Style**, and **Height** – These properties are described in System.Web.UI.WebControls.WebControl Members.

  Recommendation: Create a style sheet class and assign it to the **CssClass** property.

- **InvisiblePreservesSpace** (Boolean) – Determines if the ValidationSummary takes up space on the page when it is invisible. When `true`, the style is set to `visibility:hidden` with no change to the `display` style. This makes it preserve space. When `false`, the style is `visibility:hidden;display:none`. When the control becomes visible once again, the `display` style is restored to its original value. It defaults to `false`.

- **MakeInlineBlockStyle** (Boolean) – Determines if the enclosing div tag is normal or uses an inline-block style.

  When `false`, the div tag is not modified.

  When `true`, the div tag uses the `style=display:inline-block` when its made visible.

  It defaults to `false`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Header Properties

For an overview, see "Header".

The Properties Editor lists these properties in the "Header" category.

- **HeaderText** (string) – The text shown in the header. When "", no header is shown unless **HeaderImageURL** is assigned. It defaults to "".

  You can use these tokens in the text:

  {COUNT} – The number of error messages listed.

  {COUNT:singular:plural} – Helps build sentences where singular and plural forms are needed when you use the {COUNT} token. For example, "There is 1 error." and "There are 2 errors.". You replace the term "singular" with the singular form of the word. You replace the term "plural" with the plural form of the word. For example, "{COUNT:error:errors}".

  It defaults to "".

- **HeaderTextLookupID** (string) – Gets the value for **HeaderText** 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 ValidateMisc. If no match is found OR this is blank, **HeaderText** 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 "".

- **HeaderImageURL** (string) – A URL to an image file that is shown in the header. It defaults to "".

  #### 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.

- **HeaderElementPositions** (enum PeterBlum.DES.Web.WebControls.ValidationSummaryHeaderElementPositions) – When both **HeaderText** and **HeaderImageURL** have text, this determines how the text and image are placed relative to each other. Several of these styles create a `<table>`. The enumerated type `PeterBlum.DES.HeaderElementPositions` has these values:

  - `ImageLeft` – A two-column table with the image on the left and **HeaderText** on the right. This is the default. The **HeaderImageColumnWidth** determines the size of the image column.

⚠	These errors must be corrected. Click on each to go to the field.

  - `ImageRight` – A two-column table with the **HeaderText** on the left and image on the right. The **HeaderImageColumnWidth** determines the size of the image column.

These errors must be corrected. Click on each to go to the field.	⚠

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

- `ImageTop` – A one column, two-row table with the image in the first row and text in the second row.

  These errors must be corrected.
  Click on each to go to the field.

- `ImageBottom` – A one column, two-row table with the text in the first row and image in the second row.

  These errors must be corrected.
  Click on each to go to the field.

- `ImageBeforeText` – No table is created. These two items are back-to-back

  These errors must be corrected. Click on each to go to the field.

- `ImageAfterText` – No table is created. These two items are back-to-back

  These errors must be corrected. Click on each to go to the field.

- **HeaderTextHorizonalAlign** (enum System.Web.UI.WebControl.HorizontalAlign) – When changed to `Left`, `Center`, `Right` or `Justify`, a table encloses the **HeaderText**, and it is aligned using this setting. When set to `NotSet`, the text is left justified and no table is created. It defaults to `HorizontalAlign.NotSet`.
- **HeaderImageHorizontalAlign** (enum System.Web.UI.WebControl.HorizontalAlign) – The horizontal position of the image in the header. It is only used when **HeaderImageURL** is assigned and **HeaderElementPositions** is not `ImageBeforeText` or `ImageAfterText`. It defaults to `HorizontalAlign.NotSet`.
- **HeaderTableVerticalAlign** (enum System.Web.UI.WebControl.VerticalAlign) – The vertical position of the **HeaderText** and **HeaderImageURL** within table cells. It is only used when both **HeaderText** and **HeaderImageURL** are assigned and **HeaderElementPositions** forms a table for these elements. It defaults to `VerticalAlign.NotSet`.
- **HeaderImageColumnWidth** (string) – The minimum width of a header column containing an image. It is used when **HeaderElementPositions** is set to `ImageLeft` or `ImageRight`. Images larger than the width always cause the browser to widen the column. Use values showing either pixels or percentages. It defaults to "`10%`".
- **HeaderCssClass** – Provides the style sheet class name used for the header.

  It defaults to "`DESVALSummaryHeader`".

  The style is declared in **DES\Appearance\Validation\Validation.css**:

```
.DESVALSummaryHeader
{
   color: Red;
}
.DESVALSummaryHeader TR
{
   color: Red;
}
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Footer Properties

For an overview, see "Footer".

The Properties Editor lists these properties in the "Footer" category.

- **FooterText** (string) – The text shown in the footer. When "", no footer is shown.

  You can use these tokens in the text:

  {COUNT} – The number of error messages listed.

  {COUNT:singular:plural} – Helps build sentences where singular and plural forms are needed when you use the {COUNT} token. For example, "There is 1 error." and "There are 2 errors.". You replace the term "singular" with the singular form of the word. You replace the term "plural" with the plural form of the word. For example, "{COUNT:error:errors}".

  It defaults to "".

- **FooterTextLookupID** (string) – Gets the value for **FooterText** 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 ValidateMisc. If no match is found OR this is blank, **FooterText** 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 "".

- **FooterHorizonalAlign** (enum System.Web.UI.WebControl.HorizontalAlign) – When changed to `Left`, `Center`, `Right` or `Justify`, a table encloses the **FooterText**, and it is aligned using this setting. When set to `NotSet`, the text is left justified and no table is created. It defaults to `HorizontalAlign.NotSet`.

- **FooterCssClass** – Provides the style sheet class name used for the footer.

  It defaults to "`DESVALSummaryFooter`".

  The style is declared in **DES\Appearance\Validation\Validation.css**:

```
.DESVALSummaryFooter
{
   color: Red;
}
.DESVALSummaryFooter TR
{
   color: Red;
}
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Behavior Properties

The Properties Editor lists 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`.
- **EnableClientScript** (Boolean) – Determines if the ValidationSummary updates itself using client-side scripting. When `false`, it updates on the server-side only. When `true`, it will use client-side scripting unless the browser doesn't support the client-side scripting. It defaults to `true`.

  Note: You can set all controls to a server-side only behavior when you set PeterBlum.DES.Globals.JavaScriptEnabled to false in Page_Load().
- **Enabled** (Boolean) – Determines if the ValidationSummary is available or not. When `false`, the control is turned off and performs no function, regardless of Validators that are reporting errors. It defaults to `true`.

  Note: Traditionally, the Visible property of a control turns it off. Since ValidationSummary shows and hides itself independently of Visible, it makes more sense to "enable" and "disable" the control.
- **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 for PostBack" 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").

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Related Control Properties

For an overview, see "The Related Control Feature".

The Properties Editor lists these properties in the "Related Control" category.

- **RelatedControlID** (string) – Allows you to set up another control that is shown and hidden when the ValidationSummary is.

  Here are some typical usages:

  - A label that also appears on the page, saying "Fix the errors".
  - An enclosing box with additional formatting that wasn't built into this Validator, such as images along the sides.
  - Show a message when the errors are all resolved, using **RelatedControlDisplayMode** = `AllErrorsFixed`.
  - Show a message so long as the ValidationSummary is hidden using **RelatedControlDisplayMode** = `NoErrorsShown`.

  Assign the ID property of the other control. It must be in the same or an ancestor naming container as the ValidationSummary control. When it is "", no related control is set up.

  Alternatively, the user can assign the **RelatedControl** programmatically.

  It defaults to "".

- **RelatedControl** (System.Web.UI.Control) – A reference to another control. It serves the same purpose as **RelatedControlID**. The **RelatedControlID** must specify a control in the same or ancestor naming container as the ValidationSummary control. This property allows the control to be in any naming container. Assignment must be done programmatically.

- **RelatedControl_InvisiblePreservesSpace** (Boolean) – Determines if the control specified by either **RelatedIDControl** or **RelatedControl** takes up space on the page when it is invisible. When `true`, the style is set to `visibility:hidden` with no change to the `display` style. This makes it preserve space. When `false`, the style is `visibility:hidden;display:none`. When the control becomes visible once again, the `display` style is set to `inline`. It defaults to `true`.

  Note: Since it sets display attribute to "inline", it works well with tags that are inline oriented but poorly with those that are block oriented. Block oriented tags include <table>, <p>, and .

- **RelatedControlDisplayMode** (enum PeterBlum.DES.RelatedControlDisplayMode) – Determines when to show and hide the Related Control, based on the state of the ValidationSummary control.

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

  - `ErrorsShown` – Show the Related Control any time the ValidationSummary is shown. For example, show a message elsewhere on the page telling the user to fix errors.
  - `AllErrorsFixed` – Show the Related Control after the ValidationSummary is removed because errors were fixed. For example, show a message when errors were shown but now cleaned up.
  - `NoErrorsShown` – Show the Related Control any time the ValidationSummary is hidden. For example, show a message when there are no errors on the page, even before any data entry.

  It defaults to `RelatedControlDisplayMode.ErrorsShown`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

HTML Structure of the ValidationSummary Control

This control has a complex HTML structure. Its outermost tag is a `<div>`. It contains three optional sections: header, error messages, and footer. The header and footer can be embedded in a `<table>` as determined by the **HeaderElementPositions**, **HeaderTextHorizontalAlign** and **FooterHorizontalAlign** properties.

The default ValidationSummary lacks a header and footer. It shows the error messages using a bullet list style (<ul type=disc>). Here is a sample of the default control's HTML:

```
<div id="ValidationSummary1" class="DESValSummary">
   <ul type="disc" class="DESValSummaryErrors" >
      <li>error message 1</li>
      <li>error message 2</li>
   </ul>
</div>
```

Here is a sample with the header and footer when the properties do not put them into a table.

```
<div id="ValidationSummary1" class="DESValSummary">
   <span class="DESValSummaryHeader">Here are the errors</span><br />
   <ul type="disc" class="DESValSummaryErrors" >
      <li>error message 1</li>
      <li>error message 2</li>
   </ul>
   <span class="DESValSummaryFooter">Please fix them.</span>
</div>
```

Here is a sample where the header and footer are placed into a table because the horizontal alignment specifies to center them.

```
<div id="ValidationSummary1" class="DESValSummary">
   <table width="100%" class="DESValSummaryHeader">
      <tr><td>Here are the errors</td></tr>
   </table><br />
   <ul type="disc" class="DESValSummaryErrors" >
      <li>error message 1</li>
      <li>error message 2</li>
   </ul>
   <table width="100%" class="DESValSummaryFooter">
      <tr><td>Please fix them.</td></tr>
   </table>
</div>
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

CombinedErrorMessages Control

ALERT: Requires a license covering Peter's More Validators.

The CombinedErrorMessages control lets you combine error messages from several Validators using one ErrorFormatter. It has several uses:

- When you are using an ErrorFormatter with an image, like ⚠, and there are several Validators that may appear simultaneously, you will see several images, back-to-back. That looks strange. The CombinedErrorMessages control will show just one image and display all of the relevant error messages.
- When you want to show all error messages in one place, add the CombinedErrorMessages control where you want to show the error. The idea is similar to a ValidationSummary control except it appears as fields are changed and its formatting matches other Validators.

Click on any of these topics to jump to them:

- Using the CombinedErrorMessages Control
- Adding the CombinedErrorMessages Control
- Properties for the CombinedErrorMessages Control
- Online example

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Using the CombinedErrorMessages Control

This control is assigned a list of Validators whose error messages it shows. Those Validators supply the validation logic and error messages. Add them to the **Validators** property, either as control ID or an object reference.

When assigned to a CombinedErrorMessages control, the Validator no longer contributes any HTML to the page. So it cannot show its error message where it's placed on the page. Instead, the CombinedErrorMessages control shows its error message along with any other assigned Validator's error message in one ErrorFormatter. The ValidationSummary still reports errors associated with these Validators.

You can select any of the ErrorFormatters offered in DES within the **ErrorFormatter** property. By using one ErrorFormatter, the CombinedErrorMessages control looks like a single Validator, only with a dynamically updated error message based on the Validators to which it's assigned.

The CombinedErrorMessages Control also provides an option to show the Required Field Marker, just like Validators do, with the **ShowRequiredFieldMarker** property. It does not override the Validator's Required Field Marker, nor does it affect the formatting rules of the HiliteFields feature.

Online example

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Adding the CombinedErrorMessages 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)

Here are the steps for using this control.

1. Add the desired Validators to the page. When used with the CombinedErrorMessages, its **ErrorFormatter** property is ignored. So leave the default ErrorFormatter settings.

   Note: Since the Validators will no longer show any HTML in their location, you can place them anywhere on the page. However, it may still be easier to leave them next to the fields that they validate.

   *Note: Validators will still support their **ShowRequiredFieldsMarker** and **HiliteFields** properties.*

   If the CombinedErrorMessages control is not located nearby the field that is being validated, you may want to include the name of the field in the error message. You can use the **Label** property and {LABEL} token in the **ErrorMessage** property, if desired.

2. Add the CombinedErrorMessages control in the location where you want to see the error.

   ### Visual Studio and Visual Web Developers Design Mode Users

   Drag the CombinedErrorMessages control from the Toolbox onto your web form.

   ### Text Entry Users

   Add the control to the `<form>` area:

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

   ### Programmatically Creating the Control

   - Identify the control which you will add the CombinedErrorMessages control to its **Controls** collection. Like all ASP.NET controls, the CombinedErrorMessages 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 at the desired location and use the PlaceHolder.
   - Create an instance of the CombinedErrorMessages control class. The constructor takes no parameters.
   - Assign the **ID** property.
   - Add the CombinedErrorMessages control to the **Controls** collection.

   In this example, the CombinedErrorMessages control is created with an **ID** of "CombinedErrorMessages1". It is added to PlaceHolder1.

   [C#]

   ```
   PeterBlum.DES.Web.WebControls.CombinedErrorMessages vCEM =
      new PeterBlum.DES.Web.WebControls.CombinedErrorMessages();
   vCEM.ID = "CombinedErrorMessages1";
   PlaceHolder1.Controls.Add(vCEM);
   ```

   *Note: The namespace for these controls is PeterBlum.DES. If you prefer, add a **using** clause to that namespace on your form.*

   [VB]

   ```
   Dim vCEM As PeterBlum.DES.Web.WebControls.CombinedErrorMessages = _
      New PeterBlum.DES.Web.WebControls.CombinedErrorMessages()
   vCEM.ID = "CombinedErrorMessages1"
   PlaceHolder1.Controls.Add(vCEM)
   ```

   *Note: The namespace for these controls is PeterBlum.DES. If you prefer, add an **Imports** clause to that namespace on your form.*

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

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. Assign each Validator to the **Validators** property. For example:

```
<des:CombinedErrorMessages id="CombinedErrorMessages1" runat="server">
   <Validators>
      <des:ValidatorControlConnection ControlID="RequiredTextValidator1" />
      <des:ValidatorControlConnection ControlID="DataTypeCheckValidator1" />
   </Validators>
</des:CombinedErrorMessages>
```

4. Multiple error messages can be separated by line breaks ("list style") or text ("paragraph style"). Use the **ErrorMessagesFormat** property to select the format and the **ListLeadText** and **ParagraphSeparator** properties to customize the formatting.
5. If any of the Validators have their **EnableClientScripts** property set to `false`, also set the **EnableClientScripts** property on the CombinedErrorMessages to `false`.
6. If you want to show the Required Field Marker, set the **ShowRequiredFieldMarker** property to `true`.
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 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.

Online example

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Properties for the CombinedErrorMessages Control

The class `PeterBlum.DES.Web.WebControls.CombinedErrorMessages` is subclassed from `System.Web.UI.WebControls.WebControl`. It inherits all of the properties, methods, and events of WebControl. However, it ignores (and in the Properties Editor, hides) all formatting properties because the **ErrorFormatter** property defines formatting.

Click on any of these topics to jump to them:

- Setting the Validators Properties
- Appearance Properties
- Behavior Properties

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Setting the Validators Properties

The Properties Editor lists these properties in the "Validators" category.

- **Validators** (PeterBlum.DES.ValidatorControlConnections) – Identifies each Validator control that will contribute its **ErrorMessage** to this control.

  This property is a collection of `PeterBlum.DES.Web.WebControl.ValidatorControlConnection` objects. You can assign the control's ID to the ValidatorControlConnection**.ControlID** property or a reference to the control in the ValidatorControlConnection**.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**.

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

Adding to Validators with ASP.NET Markup

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

```
<des:CombinedErrorMessages id="CombinedErrorMessages1" runat="server">
   <Validators>
      <des:ValidatorControlConnection ControlID="RequiredTextValidator1" />
      <des:ValidatorControlConnection ControlID="DataTypeCheckValidator1" />
   </Validators>
</des:CombinedErrorMessages>
```

Adding to Validators Programmatically

Use the `Validators.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.

This example shows how to update an existing `PeterBlum.DES.Web.WebControls.ValidatorControlConnection` and add a new one. Suppose the ASP.NET code looks like the text above and the RequiredTextValidator1 control is not in the same or ancestor naming container. Also suppose the control referenced in the property DataTypeCheckValidator2 control must be added.

[C#]

```
uses PeterBlum.DES.Web.WebControls;
...
ValidatorControlConnection vConnection =
   (ValidatorControlConnection) CEM1.Validators[1];
vConnection.ControlInstance = RequiredTextValidator1;
// add DataTypeCheckValidator2. It can be either a control reference or its ID
CEM1.Validators.Add(DataTypeCheckValidator2);
```

[VB]

```
Imports PeterBlum.DES.Web.WebControls
...
Dim vConnection As ValidatorControlConnection = _
   CType(CEM1.Validators(1), ValidatorControlConnection)
vConnection.ControlInstance = RequiredTextValidator1
' add DataTypeCheckValidator2. It can be either a control reference or its ID
CEM1.Validators.Add(DataTypeCheckValidator2)
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Appearance Properties

The Properties Editor lists these properties in the "Appearance" category.

- **ErrorFormatter** (PeterBlum.DES.IBaseErrorFormatter) – Describes the appearance of the error message on the page. See "ErrorFormatters: Customizing the Appearance of the Error Message".

  This property supersedes the **ErrorFormatter** property on each Validator that is identified in the Validators property. In fact, the Validators will act like their **ErrorFormatter** property has been turned off.

- **ShowRequiredFieldMarker** (Boolean) – When `true`, the RequiredFieldMarker control will be inserted to the left of the CombinedErrorMessages control automatically. See "Other Validator Properties that Customize the Appearance". It defaults to `false`.

- **NoErrorFormatter** – Displays text and/or an image when there is no error. See "Other Validator Properties that Customize the Appearance".

- **ErrorMessagesFormat** (enum PeterBlum.DES.CEM_ErrorMessagesFormat) – Determines how to format multiple error messages.

  The enumerated type has these values:

  - `Hidden` – Do not show the error messages. Effectively the same as using **ErrorFormatter.Display** = `None`. Use this when there is a ValidationSummary control to show the error messages.
  - `List` – Multiple error messages will be shown in list style, with line breaks between each error message. List style uses the **ListLeadText** property to establish additional formatting.

    Note: Error messages shown in tooltips, such as in the Image with Tooltip ErrorFormatter (ToolTipImageErrorFormatter class) will only use line breaks on Internet Explorer. Other browsers do not support line breaks in tooltips. They will switch to paragraph style.
  - `Paragraph` – Multiple error messages will be shown in paragraph style, with the text of the **ParagraphSeparator** property between each message. A "paragraph" is constructed with each error message representing a "sentence". This is the default.

- **ListLeadText** (string) – When using list style formatting (**ErrorMessagesFormat** = `List`), this text precedes each error message when there are two or more error messages shown. It defaults to "".

  You can use HTML in the TextErrorFormatter and PopupErrorFormatter. If using the TooltipImageErrorFormatter, HTML is supported when using the Enhanced ToolTip feature. (See the **Interactive Pages User's Guide** for details.)

- **ParagraphSeparator** (string) – When using paragraph style formatting (**ErrorMessagesFormat** = `Paragraph`), this text separates each error message. It defaults to " " (a space).

  You can use HTML in the TextErrorFormatter and PopupErrorFormatter. If using the TooltipImageErrorFormatter, HTML is supported when using the Enhanced ToolTip feature. (See the **Interactive Pages User's Guide** for details.)

- **ShowFirstMsgOnly** (Boolean) – While the CombinedErrorMessages control is designed to show several validators error messages simultaneously, some developers believe it's easier for their users to see just one error message at a time. When you set this property to `true`, only the first Validator that is invalid will show its error message. As that one is fixed, the CombinedErrorMessages control will show the next Validator that is invalid. It uses the order of Validators in the **Validators** property to determine first, second, etc.

  It defaults to `false`.

- **EnableClientScript** (Boolean) – Determines if client-side functionality is set up. When `true`, it sets up client-side support.

- **HiliteFields** (PeterBlum.DES.Web.WebControls.ControlConnectionCollection) – Identifies other controls on the page whose style will change when any of these Validators detect an error. It works in conjunction with "Change the Style of Other Fields Nearby the Error".

  Validators show their error message in the ErrorFormatter and optional ValidationSummary control. Since the goal is to attract the user to the error, this property can identify other controls on the page. These controls will have their style sheet changed when there is an error and return to the default when the error is fixed.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Choose this to change an enclosing control like a `<div>` or `<table>` to a different background color. Also consider changing the color of the field label.

By default, **HiliteFields** is not used. You must enable it by setting **HiliteFieldsNearbyError** to `true` in the PageManager control or in **PeterBlum.DES.Globals.WebFormDirector.ValidationManager** in `Page_Load()`. See "Change the Style of Other Fields Nearby the Error".

When you add textual controls like a Label, TextBox or anything that is formed with a tag, the following style sheet class is used:

```
.DESVALTextHiliteFields
{
   font-weight: bold;
}
```

The rest use this style sheet class:

```
.DESVALNonTextHiliteFields
{
   background-color: #ffebcd; /* blanchedalmond */
}
```

Note that any **Label** property assigned to a control is automatically included in this list at runtime. So you don't need to set up the labels twice.

Do not include the actual control being validated here because it is handled automatically elsewhere. See "Change the Style of the Field With the Error".

Elements of the **HiliteFields** collection must be `PeterBlum.DES.Web.WebControls.ControlConnection` objects. They can reference a control by its ID or an object reference.

Adding ControlConnections with ASP.NET Markup

```
<des:CombinedErrorMessages id="[id]" runat="server" >
  <HiliteFields>
     <des:ControlConnection ControlID="[controlid1]" />
     <des:ControlConnection ControlID="[controlid2]" />
  </HiliteFields>
</des:CombinedErrorMessages>
```

Adding ControlConnections Programmatically

The **HiliteFields** property will create `PeterBlum.DES.Web.WebControls.ControlConnections` for you by calling its `Add()` method. `HiliteFields.Add()` takes a reference to the control object or its ID. (The reference is recommended for performance.)

[C#]

```
CombinedErrorMessages1.HiliteFields.Add(Label1);      // reference to control
CombinedErrorMessages1.HiliteFields.Add("Table1");    // ID of a control
```

[VB]

```
CombinedErrorMessages1.HiliteFields.Add(Label1)       ' reference to control
CombinedErrorMessages1.HiliteFields.Add("Table1")     ' ID of a control
```

- **Label** (PeterBlum.DES.Web.WebControls.LabelText) – Provides text for the "{LABEL}" token that can appear in **ErrorMessage** and **SummaryErrorMessage** properties. Use it to separate the label from your standardized and localized error messages. This lets you use the same error message for differently named fields.

  See "Properties of the PeterBlum.DES.Web.WebControls.LabelText Class".

  The **Label** property can also cause the Label control to change its style when an error is shown. Simply set either the **LabelControlID** or **LabelControl** properties and set up the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager.HiliteFieldsNearbyError** property.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

When defining properties of any Label property into ASP.NET, embed each within the beginning tag using the style "LabelName-propertyname". For example:

```
<des:CombinedErrorMessages id="CombinedErrorMessages1" runat="server"
   Label-Text="Start Date" SecondLabel-LabelControlID="EndDateLabel">
   <Validators>
      <des:ValidatorControlConnection ControlID="RequiredTextValidator1" />
      <des:ValidatorControlConnection ControlID="DataTypeCheckValidator1" />
   </Validators>
</des:CombinedErrorMessages>
```

- **SecondLabel** (PeterBlum.DES.Web.WebControls.LabelText) – Provides text for the "{LABEL2}" token that can appear in **ErrorMessage** and **SummaryErrorMessage** properties. This property appears on controls with a **SecondControlIDToEvaluate** property and should be used as a label for the control assigned to **SecondControlIDToEvaluate**.

  See "Properties of the PeterBlum.DES.Web.WebControls.LabelText Class".

  The **SecondLabel** property can also cause the Label control to change its style when an error is shown. Simply set either the **LabelControlID** or **LabelControl** properties and set up the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager.HiliteFieldsNearbyError** property.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Behavior Properties

The Properties Editor lists 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 entirely unused. It defaults to `true`.
- scripting for this control. When `false`, no client-side code is written.

  You should always set this to false when any of the attached Validators do not have client-side scripting enabled. Unpredictable client-side behavior will occur if to you do not. This includes:

  - When the Validator.**EnableClientScript** property is `false`
  - On a CustomValidator without a client-side evaluation function.
  - On an IgnoreConditionValidator.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Required Field Marker Controls

Many web sites provide a special symbol near each field that requires an entry. DES defines this as the **Required Field Marker**, a textual or graphical symbol to indicate that a field requires an entry. In addition, those pages often have a description on the form that helps the user understand that special symbol. DES makes it easy to support these concepts.

Click on any of these topics to jump to them:

- RequiredFieldMarker Control
- RequiredFieldsDescription Control
- Setting up the Global Defaults
- Adding a RequiredFieldMarker Control
- Properties of the RequiredFieldMarker Control
- Adding a RequiredFieldsDescription Control
- Properties of the RequiredFieldsDescription Control

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

RequiredFieldMarker Control

For a demo, see http://www.peterblum.com/DES/DemoRFM.aspx.

The RequiredFieldMarker control shows the Required Field Marker. Just add it in the desired location, and it will use your global defaults for the Required Field Marker, whether it is textual or an image. It defaults to the image shown to the right of the textbox below.

To add the control, see "Adding a RequiredFieldMarker Control".

Global defaults avoid editing the symbol as you add each control, and you can edit the appearance of this control globally. For example, you build your web site with the RequiredFieldMarker set to an asterisk ("*"). Before release, your graphic artist provides an image file to replace the asterisk. You only need to change the globals. See "Setting up the Global Defaults".

In case you need to change the formatting on a page, the RequiredFieldMarker has properties to override the global defaults. See "Properties of the RequiredFieldMarker Control".

Each Validator control includes the **ShowRequiredFieldMarker** property. When set to `true`, it automatically adds the RequiredFieldMarker control to the left of the Validator control.

*Note: If your Validator uses the **Enabler** property, the RequiredFieldMarker remains visible even if the Validator is disabled. Similarly, if the data entry control has its visibility or enabled state changed, the RequiredFieldMarker remains visible. To avoid this, set up a FieldStateController that shows and hides the RequiredFieldMarker control as needed.*

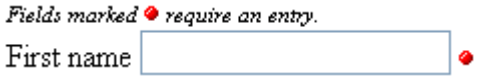
Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

RequiredFieldsDescription Control

For a demo, see http://www.peterblum.com/DES/DemoRFM.aspx.

The RequiredFieldsDescription control describes the purpose of the Required Field Marker to the user. It will also use global defaults for the description, making it consistent throughout your site. It is often placed above the fields on the form, as shown here.



To add the control, see "Adding a RequiredFieldsDescription Control".

The RequiredFieldsDescription control provides a textual description of the required field marker. It can embed the marker symbol, whether it is textual or graphical. You should set the description up globally. See "Setting up the Global Defaults". You can override them on a case-by-case basis. See "Properties of the RequiredFieldsDescription Control".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Setting up the Global Defaults

Use the "RequiredFieldMarker and RequiredFieldDescription Defaults" topic of the **Global Settings Editor** to establish your global defaults. The following properties are the global defaults for the RequiredFieldMarker and RequiredFieldDescription:

- **DefaultRequiredFieldMarkerText** (string) – Used by the RequiredFieldMarker control to show text or any HTML as the marker. When blank, the image is shown using **DefaultRequiredFieldImageURL**. When assigned, if it has the token "{IMAGE}", an image is set up with **DefaultRequiredFieldImageURL** to replace the token. It defaults to "` {IMAGE}`".

- **DefaultRequiredFieldMarkerTextLookupID** (string) – Gets the value for **DefaultRequiredFieldMarkerText** 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 ValidateMisc. If no match is found OR this is blank, **DefaultRequiredFieldMarkerText** 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 "".

- **DefaultRequiredFieldsDescriptionText** (string) – Used by the RequiredFieldsDescription control as its default text when its Text property contains "{DEFAULT}". It can include the token "{IMAGE}" that will be replaced by the HTML for an image using **DefaultRequiredFieldImageURL**. It can include the token "{MARKER}" that will be replaced by the Required Field Marker, which takes into account both textual and image parts of the marker. The default is "`Fields marked{MARKER} require an entry.`". Note that the default Required Field Marker starts with "` `".

- **DefaultRequiredFieldsDescriptionTextLookupID** (string) – Gets the value for **DefaultRequiredFieldsDescriptionText** 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 ValidateMisc. If no match is found OR this is blank, **DefaultRequiredFieldsDescriptionText** 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 "".

- **DefaultRequiredFieldImageURL** (string) – The URL to an image file that is used as the required field marker. It is used within both the RequiredFieldMarker and RequiredFieldsDescription controls. On the RequiredFieldMarker, the **DefaultRequiredFieldMarkerText** must be blank or contain the token "{IMAGE}" to show this URL.

  The default is "`{APPEARANCE}/Validation/requireddot.gif`". Here is the default image:

  ### 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.

- **DefaultRequiredFieldImageAltText** (string) – The alternative text used by the `<img>` tag when the image (**DefaultRequiredFieldImageUrl**) cannot be loaded. It defaults to "*".

- **DefaultRequiredFieldToolTip** (string) – Used when the RequiredFieldMarker.**ToolTip** property is "", it supplies the ToolTip. It defaults to "". A common value for this property is "Required".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

- **DefaultRequiredFieldToolTipLookupID** (string) – Gets the value for **DefaultRequiredFieldToolTip** 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 ValidateMisc. If no match is found OR this is blank, **DefaultRequiredFieldToolTip** 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 "".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Adding a RequiredFieldMarker Control

*Note: Alternatively, you can set the **ShowRequiredFieldsMarker** property on any Validator control positioned where you want this control.*

Visual Studio and Visual Web Developers Design Mode Users

Drag the RequiredFieldMarker control from the Toolbox onto your web form.

Text Entry Users

Add the control:

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

Programmatically Creating the Control

- Identify the control which you will add the RequiredFieldMarker control to its **Controls** collection. Like all ASP.NET controls, the RequiredFieldMarker 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 RequiredFieldMarker control class. The constructor takes no parameters.
- Assign the **ID** property.
- Add the RequiredFieldMarker control to the **Controls** collection.

In this example, the RequiredFieldMarker is created with an **ID** of "RequiredFieldMarker1". It is added to PlaceHolder1.

[C#]

```
PeterBlum.DES.Web.WebControls.RequiredFieldMarker vRFM =
   new PeterBlum.DES.Web.WebControls.RequiredFieldMarker();
vRFM.ID = "RequiredFieldMarker1";
PlaceHolder1.Controls.Add(vRFM);
```

[VB]

```
Dim vRFM As PeterBlum.DES.Web.WebControls.RequiredFieldMarker = _
   New PeterBlum.DES.Web.WebControls.RequiredFieldMarker()
vRFM.ID = "RequiredFieldMarker1"
PlaceHolder1.Controls.Add(vRFM)
```

*Note: If your Validator uses the **Enabler** property, the RequiredFieldMarker remains visible even if the Validator is disabled. Similarly, if the data entry control has its visibility or enabled state changed, the RequiredFieldMarker remains visible. To avoid this, set up a FieldStateController that shows and hides the RequiredFieldMarker control as needed.*

For a demo, see http://www.peterblum.com/DES/DemoRFM.aspx.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Properties of the RequiredFieldMarker Control

The class `PeterBlum.DES.Web.WebControls.RequiredFieldMarker` is subclassed from `PeterBlum.DES.Web.WebControls.LocalizableLabel` which itself is subclassed from `System.Web.UI.WebControls.Label`. It inherits all of the properties, methods, and events of Label.

See also "Setting up the Global Defaults".

- **CssClass** (string) – The style sheet class name for this control.

  It defaults to "`DESVALRequiredFieldMarker`".

  This style is declared in **\DES\Appearance\Validation\Validation.css**:

  ```
  .DESVALRequiredFieldMarker
  {
  }
  ```

  You can change it in the class above or create a new class and assign the name to the **CssClass** property.

- **Text** (string) – The text for the marker. When it contains "{DEFAULT}" it uses global **DefaultRequiredFieldMarkerText** property that you set in the **Global Settings Editor**.

  It defaults to "{DEFAULT}".

  Change it if you want to override the globally defined text. When blank, the image is shown using **RequiredFieldImageURL**. When assigned, if it has the token "{IMAGE}", an image is set up with **RequiredFieldImageURL** to replace the token.

- **TextLookupID** (string) – Gets the value for **Text** 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 ValidateMisc. If no match is found OR this is blank, **Text** 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 "".

- **RequiredFieldImageURL** (string) – The URL to an image file that is used as the required field marker. When it contains "{DEFAULT}" it uses the global **DefaultRequiredFieldImageURL** property that you set in the **Global Settings Editor**. See "Setting up the Global Defaults".

  It defaults to "{DEFAULT}".

  Change it if you want to override the globally defined image.

  #### 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.

- **ToolTip** (string) – A tooltip can help answer "what is this?" as the user points to it. When "", it uses the global **DefaultRequiredFieldToolTip** property that you set in the **Global Settings Editor**. See "Setting up the Global Defaults".

  It defaults to "".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Adding a RequiredFieldsDescription Control

Visual Studio and Visual Web Developers Design Mode Users

Drag the RequiredFieldsDescription control from the Toolbox onto your web form.

Text Entry Users

Add the control:

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

Programmatically Creating the Control

- Identify the control which you will add the RequiredFieldsDescription control to its **Controls** collection. Like all ASP.NET controls, the RequiredFieldsDescription 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 RequiredFieldsDescription control class. The constructor takes no parameters.
- Assign the **ID** property.
- Add the RequiredFieldsDescription control to the **Controls** collection.

In this example, the RequiredFieldsDescription is created with an **ID** of "RequiredFieldsDescription1". It is added to PlaceHolder1.

[C#]

```
PeterBlum.DES.Web.WebControls.RequiredFieldsDescription vRFD =
   new PeterBlum.DES.Web.WebControls.RequiredFieldsDescription();
vRFD.ID = "RequiredFieldsDescription1";
PlaceHolder1.Controls.Add(vRFD);
```

[VB]

```
Dim vRFD As PeterBlum.DES.Web.WebControls.RequiredFieldsDescription = _
   New PeterBlum.DES.Web.WebControls.RequiredFieldsDescription()
vRFD.ID = "RequiredFieldsDescription1"
PlaceHolder1.Controls.Add(vRFD)
```

For a demo, see http://www.peterblum.com/DES/DemoRFM.aspx.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Properties of the RequiredFieldsDescription Control

The class `PeterBlum.DES.Web.WebControls.RequiredFieldsDescription` is subclassed from `PeterBlum.DES.Web.WebControls.LocalizableLabel` which itself is subclassed from `System.Web.UI.WebControls.Label`. It inherits all of the properties, methods, and events of Label.

See also "Setting up the Global Defaults".

- **CssClass** (string) – The style sheet class name for this control.

  It defaults to "`DESVALRequiredFieldsDescription`".

  These styles are declared in **\DES\Appearance\Validation\Validation.css**:

  ```
  .DESVALRequiredFieldsDescription
  {
     font-size: 8pt;
     font-style: italic;
  }
  ```

  You can change it in the class above or create a new class and assign the name to the **CssClass** property.

- **Text** (string) – The text for the marker. When it contains "{DEFAULT}", it uses the global **DefaultRequiredFieldsDescriptionText** property that you set in the **Global Settings Editor**. See "Setting up the Global Defaults".

  It defaults to "{DEFAULT}".

  Change it if you want to override the globally defined text. If it has the token "{MARKER}", it generates the Required Field Marker from the globals for text and image. If it has the token "{IMAGE}", an image is set up with **RequiredFieldImageURL** to replace the token.

- **TextLookupID** (string) – Gets the value for **Text** 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 ValidateMisc. If no match is found OR this is blank, **Text** 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 "".

- **RequiredFieldImageURL** (string) – The URL to an image file that is used as the Required Field Marker. When it contains "{DEFAULT}", it uses the global **DefaultRequiredFieldImageURL** property that you set in the **Global Settings Editor**. See "Setting up the Global Defaults".

  It defaults to "{DEFAULT}".

  Change it if you want to override the globally defined image.

  #### 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.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Page Level Properties and Methods

Most of the behavior and formatting of the Validation Framework come from controls and their properties. DES also provides properties that are not found on any control. They are on the **PeterBlum.DES.Globals.WebFormDirector** object, which you have seen throughout this guide. **PeterBlum.DES.Globals.WebFormDirector** offers "page-level" settings, values shared between all controls on a page. Validators use many of them to provide formatting, such as "Drawing The User's Attention To The Error" and "Applying Styles To Tokens". Buttons use them to configure the behavior when submitting a page. See "Page-Level Properties used by Client-Side Validation".

PeterBlum.DES.Globals.WebFormDirector has several public methods. Submit controls use many of its methods as described in "Using Native Button Controls to Submit the Page" and "Using the Menu Control to Validate the Page".

The `Validate()` method and its companion property **IsValid** are on **PeterBlum.DES.Globals.WebFormDirector**. See "Properties and Methods to Validate the Page".

This section covers other important properties and methods on **PeterBlum.DES.Globals.WebFormDirector**.

Click on any of these topics to jump to them:

- Properties on PeterBlum.DES.Globals.WebFormDirector
  - PeterBlum.DES.Globals.WebFormDirector.ValidationManager properties
  - Set Focus to the Control
  - Show an Alert
  - Change the Style of the Field With the Error
  - Change the Style of Other Fields Nearby the Error
  - Blinking the ErrorFormatter
  - Warning Mode on Validators
  - Applying Styles To Tokens
  - Properties and Methods to Validate the Page
  - Page-Level Properties used by Client-Side Validation
  - Supporting a Reset or Clear Button
- PeterBlum.DES.Globals.WebFormDirector.ValidationManager methods
- Debugging PeterBlum.DES.Globals.WebFormDirector Properties

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.

Many of its properties can be set by using the PageManager control. See the **General Features Guide**.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

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.

- **ValidationManager** (PeterBlum.DES.Web.WebControls.ValidationManager) – Hosts numerous properties and several methods that support validation. See "PeterBlum.DES.Globals.WebFormDirector.ValidationManager 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. For example, the Date data type uses this to get the DateTimeFormatInfo class, which defines the short date pattern (ex: MM/dd/yyyy) and date separator. The Currency data types use the NumberFormatInfo class to get the currency symbol, decimal symbol, and number of decimal places.

  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.asax**. 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 = "-";
  vDTFI.ShortTimePattern = "HH:mm";
  vDTFI.LongTimePattern = "HH:mm:ss";

  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 = "-"
  vDTFI.ShortTimePattern = "HH:mm"
  vDTFI.LongTimePattern = "HH:mm:ss"

  Dim vNFI As System.Globalization.NumberFormatInfo = _
     PeterBlum.DES.Globals.CultureInfo.NumberFormat
  vNFI.DecimalSeparator = "."
  vNFI.CurrencySymbol = "€"
  ```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- **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, the **TrueBrowser.SupportsClientSideValidators** property is `false` and it automatically sets all Validator controls **EnableClientScript** property to `false` at runtime. 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. The server side will handle these controls gracefully on post back.

  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**.

- **IsPostBack** (Boolean) - DES behaves differently depending on the state of the Page.**IsPostBack** property. Sometimes users need to control this behavior. This property reflects the current state of Page.**IsPostBack**. However, the user can change its value, either to `false` (not postback) or to `true` (postback). Here are features that use this property:

  - The ValidationSummary control only shows up when this is `true`.
  - The Validator's **NoErrorFormatter.Mode** property.
  - **PeterBlum.DES.Globals.WebFormDirector.ValidationManager.ShowAlertOnSubmit** uses this to build an alert that appears after postback.
  - **PeterBlum.DES.Globals.WebFormDirector.ValidationManager.FocusOnSubmit** can sets the focus after post back.
  - The `<form onreset=>` code varies based on this.

- **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`.

- **SubmitPageManager** – Provides methods to programmatically attach native submit controls to DES. See these topics:

  - Using Native Button Controls to Submit the Page
  - Using the Menu Control to Validate the Page
  - Using the BulletedList Control

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- Using AJAX Callback Controls

- **InitialFocusControl** (System.Web.UI.Control) – Sets the focus on the page to this control when the page is first loaded. Assign this property to a reference to the control that should get the initial focus. If the control is hidden or disabled, focus will not be set because browsers do not permit it.

  Typically this is set within `Page_Load()` or a post back event handler.

  When `null/nothing`, no field gets initial focus. It defaults to `null/nothing`.

  Example

  Set focus to a textbox associated with TextBox1:

  [C#]

  ```
  PeterBlum.DES.Globals.WebFormDirector.InitialFocusControl = TextBox1;
  ```

  [VB]

  ```
  PeterBlum.DES.Globals.WebFormDirector.InitialFocusControl = TextBox1
  ```

- **SetFocusFunctionName** (string) – The name of a client-side function that is called whenever DES attempts to set focus to a field with an error. Use this function when the field is invisible and you want to show it. For example, if your fields are in a tabbed user interface, a field with the error is invisible when on a tabbed panel that is not visible. Your function can entirely replace the built in focus setting code. For example, if the field is a RichTextBox and it has a specialized JavaScript function to select the entire text when setting focus, use this.

  The function must take one parameter, the element that is getting the focus. It is an object, defined in DOM/DHTML, such as the `<input>` tag for a textbox. One key property of every element is its 'id'. Use that to determine the element so you know what you are attempting to act on.

  The function must return a Boolean value where `true` means set focus and `false` means do not set focus. When you replace the focus setting code with your own, return `false`.

  It defaults to "".

  Your function should appear on the page that is generated. See "Adding Your JavaScript to the Page".

  In this example, the parent of element id 'TextBox1' may be invisible. This JavaScript code makes it visible and returns true to indicate that it can set focus. **PeterBlum.DES.Globals.WebFormDirector.SetFocusFunctionName** is set to "ShowTextBox1".

  ```
  function ShowTextBox1(pFld)
  {
     if (pFld.id == 'TextBox1')
        pFld.parentNode.style.visibility = 'inherit';
     return true;
  }
  ```

- **BeforeValidation** (event) – This event is fired first when the `Globals.WebFormDirector.Validate()` methods are called. Use it to do any preparation for server side validation. It uses the `System.EventHandler` delegate definition, where *pSender* is the WebFormDirector object.

  Example

  [C#]

  ```
  public void Page_Load(object pSender, EventArgs pE)
  {
     PeterBlum.DES.Globals.WebFormDirector.BeforeValidation +=
        new EventHandler(WebFormDirector_BeforeValidation);
  }

  public void WebFormDirector_BeforeValidation(object pSender, EventArgs pE)
  {
     // your code goes here
  }
  ```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

[VB]

```
Public Sub Page_Load(ByVal pSender As Object, ByVal pE As EventArgs)
   AddHandler PeterBlum.DES.Globals.WebFormDirector.BeforeValidation _
      AddressOf WebFormDirector_BeforeValidation
End Sub

Public Sub WebFormDirector_BeforeValidation(ByVal pSender As Object, _
   ByVal pE As EventArgs)
   ' your code goes here
End Sub
```

- **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.`".
- **UseOnResetEventHandler** – See "Supporting a Reset or Clear Button".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

PeterBlum.DES.Globals.WebFormDirector.ValidationManager properties

The **PeterBlum.DES.Globals.WebFormDirector.ValidationManager** object hosts all page-level features related to validation. There are numerous properties and a few methods. Many of these properties are on the PageManager control.

- **IsValid, HasValidated** – See "Properties and Methods to Validate the Page".
- **Validators** (PeterBlum.DES.ValidatorCollection) – A collection of the Validators on this page. Available for page developers to iterate through as they need to review and modify settings. All validator objects returned are subclassed from PeterBlum.DES.BaseAnyValidator.

  Example

  [C#]

  ```
  foreach (PeterBlum.DES.IValidatorControl vValidator in
     PeterBlum.DES.Globals.WebFormDirector.ValidationManager.Validators)
  {
     // do something with vValidator
     // remember to typecast to the actual validator class
     // to edit properties specific to that class
  }
  ```

  [VB]

  ```
  Dim vValidator As PeterBlum.DES.IValidatorControl
  For Each vValidator In
  PeterBlum.DES.Globals.WebFormDirector.ValidationManager.Validators

     ' do something with vValidator
     ' remember to typecast to the actual validator class
     ' to edit properties specific to that class
  Next
  ```

- **AutoDisableValidators** (Boolean) – Validators normally evaluate controls that are hidden or disabled, unless you use their **Enabler** property with the VisibleCondition and/or EnabledCondition. This is excessive work for something that is pretty common: not validating hidden/disabled controls.

  AutoDisableValidators makes validators detect if any of the controls they evaluate are hidden or disabled and turns the validator off.

  When `true`, validators will not attempt to evaluate controls that are hidden or disabled.

  When `false`, they will unless you establish the **Enabler** property with a VisibleCondition and EnabledCondition.

  When not edited on your page, it uses the global setting **DefaultAutoDisableValidators**, which defaults to `true`.

- **AutoHideRequiredFieldMarkers** (Boolean) – When a validator's **ShowRequiredFieldMarker** property is `true` and it has an Enabler, set **AutoHideRequiredFieldMarkers** to `true` to hide the RequiredFieldMarker any time the validator is disabled.

  If you feel that the RequiredFieldMarker does not belong when the validator is disabled, use this feature. Not every client-side action that modifies what the **Enabler** is evaluating will trigger the RFM to be hidden. FieldStateControllers and user edits to those controls will properly hide them. For any other scripts you use to modify those controls, call this function:

  ```
  DES_UpdateRFM();
  ```

  When not assigned, it uses the global setting **DefaultAutoHideRequiredFieldMarkers**, which defaults to `false`.

- **FocusOnChange**, **FocusOnSubmit**, **FocusAfterAlert** – See "Set Focus to the Control".
- **ShowAlertOnChange**, **ShowAlertOnSubmit**, **ShowAlertAfterPostBackError**, **AlertTemplate**, **AlertTemplateLookupID**, **AlertTemplateLeadText**, **AlertTemplateListStyle**, **AlertShowsSummaryMessage**, **DelayAlertOnSubmitCount** – See "Show an Alert".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

- **ChangeStyleOnControlsWithError**, **ControlErrorCssClass**, **ListErrorCssClass**, **CheckBoxErrorCssClass, CheckBoxECCMode, ControlErrorCssClass_Alt**, **ListErrorCssClass_Alt**, **CeckBoxErrorCssClass_Alt, ChangeStyleOnControlsWithErrorIncludesExtraControls** – See "Change the Style of the Field With the Error".
- **HiliteFieldsNearbyError**, **TextHiliteFieldCssClass**, **NonTextHiliteFieldCssClass** – See "Change the Style of Other Fields Nearby the Error".
- **BlinkOnChange, BlinkOnSubmit, BlinkTime** – See "Blinking the ErrorFormatter".
- **WarningsConfirmOnSubmit**, **WarningsConfirmTemplate**, **WarningsConfirmTemplateID**, **WarningsConfirmLeadText** – See "Warning Mode".
- **LabelTokenCssClass, PropertyTokenCssClass, RuntimeTokenCssClass, SummaryLabelTokenCssClass, SummaryPropertyTokenCssClass, SummaryRuntimeTokenCssClass** – See "Applying Styles To Tokens".
- **ConfirmMessage**, **ConfirmMessageLookupID**, **ConfirmMessageGroup**, **CustomSubmitFunctionName**, **SubmitOrder**, **PostValidationUpdateScript**, **PostPageValidationFunctionName**, **DefaultGroup** – See "Page-Level Properties used by Client-Side Validation".

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

PeterBlum.DES.Globals.WebFormDirector.ValidationManager methods

Click on any of these topics to jump to them:

- PeterBlum.DES.Globals.WebFormDirector.Validate() method
- ValidationManager.EnableValidators() method
- ValidationManager.EnableClientScriptValidators() method

ValidationManager.EnableValidators() method

PeterBlum.DES.Globals.WebFormDirector.ValidationManager provides the `EnableValidators()` method to set the **Enabled** property on a group of Validators. EnableValidates() iterates through all of the Validators which match the *pGroup* parameter and set their **Enabled** property to the value of the *pEnabled* parameter. It does not return a result.

[C#]

```
public void EnableValidators(string pGroup, bool pEnabled)
```

[VB]

```
Public Sub EnableValidators(ByVal pGroup As String, ByVal pEnabled As Boolean)
```

In this example, Validators are disabled if their group name is "".

[C#]

```
PeterBlum.DES.Globals.WebFormDirector.ValidationManager.EnableValidators("",
false);
```

[VB]

```
PeterBlum.DES.Globals.WebFormDirector.ValidationManager.EnableValidators("", False)
```

ValidationManager.EnableClientScriptValidators() method

PeterBlum.DES.Globals.WebFormDirector.ValidationManager provides the `EnableClientScriptValidators()` method to set the **EnableClientScript** property on a group of Validators. `EnableClientScriptValidates()` iterates through all of the Validators which match the *pGroup* parameter and set their **EnableClientScript** property to the value of the *pEnabled* parameter. It does not return a result.

[C#]

```
public void EnableClientScriptValidators(string pGroup, bool pEnabled)
```

[VB]

```
Public Sub EnableClientScriptValidators(ByVal pGroup As String,
  ByVal pEnabled As Boolean)
```

In this example, client-side validation for Validators are disabled if their group name is "".

[C#]

```
PeterBlum.DES.Globals.WebFormDirector.ValidationManager.EnableClientScriptValidators("
", false);
```

[VB]

```
PeterBlum.DES.Globals.WebFormDirector.ValidationManager.EnableClientScriptValidators("
", False)
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Debugging PeterBlum.DES.Globals.WebFormDirector Properties

If you are uncertain that the values set in **PeterBlum.DES.Globals.WebFormDirector** are correct, use the `PeterBlum.DES.Globals.WebFormDirector.DescribeProperties()` method. It returns a string that you can assign to a Label or LiteralControl control to show on the page or you can set up `<@ Page Trace="true" >` and output them using `Page.Trace.Write()`.

`PeterBlum.DES.Globals.WebFormDirector.DescribeProperties(showHTML)`

showHTML

Pass `true` to format the text in HTML format (as a table) and `false` to format it as a carriage return delimited set of lines useful to output to a file or other system that cannot use HTML.

Call it after you have set any properties on **PeterBlum.DES.Globals.WebFormDirector**, like this.

```
DebugLabel1.Text =
PeterBlum.DES.Globals.WebFormDirector.DescribeProperties(true)
```

```
Page.Trace.Write(PeterBlum.DES.Globals.WebFormDirector.DescribeProperties(fal
se))
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

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
- Validation Functions

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
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

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
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

function DES_Trunc(pValue)

Returns the integer part of the decimal value. Converts the type from float to integer.

Parameters

pValue

The initial decimal value.

Return value

Returns the integer part of the 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 `<span>` tag with an ID. The inner HTML of that tag will be updated. A `System.Web.UI.WebControls.Label` creates such a `<span>` 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');
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Retrieving Values From Data Entry Controls

function DES_GetTextValue(pID, pTrim, pEvalFunction)

Retrieves the textual value of a control.

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".

pTrim (Boolean)

When true, strip off leading and trailing spaces.

pEvalFunction (function)

Optional parameter. Can be omitted or pass null. When evaluating a client-side condition object, this provides a custom function for a control to retrieve its current value. When not assigned, the value comes from the value= attribute on the HTML tag that uses pID, such as <input type='text' id='textbox1' value='current text' />.

Pass the condition object's GetText property when writing a custom evaluation function. See the example below.

Return value

A string which is the current value of the control.

Example 1

```
var vVal = DES_GetTextValue("TextBox1", true);
```

Example 2: Using a Condition object

```
var vVal = DES_GetTextValue(cond.IDToEval, cond.Trim, cond.GetText);
```

function DES_GetSelIdx(pID, pEvalFunction)

Retrieves the index of a list or DropDownList control.

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".

pEvalFunction (function)

Optional parameter. Can be omitted or pass null. When evaluating a client-side condition object, this provides a custom function for a control to retrieve its current value. When not assigned, the value comes from the value= attribute on the HTML tag that uses pID, such as <select id='listbox1' value='current text' />.

Pass the condition object's GetSelIdx property when writing a custom evaluation function. See the example below.

Return value

It returns an integer with one of these values: -1 for no selection, 0 for the first element selected, etc.

Example 1

```
var vVal = DES_GetSelIdx("ListBox1");
```

Example 2: Using a Condition object

```
var vVal = DES_GetSelIdx(cond.IDToEval, cond.GetSelIdx);
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

function DES_GetDTTBValue(pID)

Supports Date, Time, Duration, Integer, Decimal, Currency and Percent TextBoxes.

Retrieve the value of the numeric, date or time textbox. *The "DTTB" stands for "DataTypeTextBox".*

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

IntegerTextBox	Integer or `null` for invalid entry.
DecimalTextBox, CurrencyTextBox, PercentTextBox	Floating point or `null` for invalid entry.
DateTextBox, AnniversaryTextBox, MonthYearTextBox	JavaScript Date object or `null` for invalid entry.
TimeOfDayTextBox, DurationTextBox	Integer representing a number of seconds since 0:0:00 or `null` for invalid entry.

Example

```
var vNumber = DES_GetDTTBValue('IntegerTextBox1');
if (vNumber != null)
   // do something with vNumber
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Validation Functions

function DES_FieldChanged(pID)

Runs client-side validation on the control using the DES Validation Framework. It runs all Validators attached to the control.

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_FieldChanged('TextBox1');
```

function DES_IsValid(pID)

Determines if the data entry control (textbox, list, etc) is valid using the DES Validation Framework. It does NOT run validation. It merely returns the state from the most recent validation. It does NOT evaluate a single validator either. It is looking at the validation state of the data entry control.

Parameters

pID

The **ClientID** property value from the server side data entry control that has validators assigned to it. It is the value written into the `id=` attribute of the HTML element. See "Embedding the ClientID into your Script".

Return value

Returns `true` if all validators on the data entry control are valid.

Returns `false` if at least one validator on the data entry control is invalid.

Returns `null` if the data entry control has no client-side validators attached or they are all disabled.

Example

```
if (DES_IsValid('<% =TextBox1.ClientID %>') == false)
   // take an action when the control is invalid
```

function DES_ValidateGroup(pGroup)

Runs client-side validation on the control using the DES Validation Framework. It runs all Validators attached to the control.

Parameters

pGroup

The Validation Group name. Can be "". Pass "*" to validate every enabled validator regardless of the group.

Return value

`True` when all validators are valid. `False` otherwise.

Example

```
DES_ValidateGroup('');
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

function DES_FindAOById(pID)

Returns the "Action object" for a validator. The Action object represents the validator on the client-side. Use it to modify the state of the validator usually by following these directions: "Changing The Properties of a Validator With Client-Side Scripts".

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

The Action object (a javascript object) for the validator, or `null` if the ID cannot be found.

Example

```
var vAO = DES_FindAOById("<% = RequiredTextValidator1.ClientID %>");
if (vAO)
   vAO.Enabled = false;
```

function DES_SetEnabled(pAO, pEnabled)

Changes the enabled state of a validator. If the validator is disabled, any associated error message is hidden.

Recommendation: Use the validator's **Enabler** property if possible.

Parameters

pAO

The "Action Object" for the validator. See "Changing The Properties of a Validator With Client-Side Scripts".

pEnabled

When `true`, enable the validator. When `false`, disable the validator.

Example

```
var vAO = DES_FindAOById("<% = RequiredTextValidator1.ClientID %>");
DES_SetEnabled(vAO, false);
```

function DES_HideVal(pAO)

Hides the validator (effectively sets it to being valid). It does not modify or use any of the "Drawing The User's Attention To The Error" features. If you need them to be changed, use `DES_UpdVal()`.

Parameters

pAO

The "Action Object" for the validator. See "Changing The Properties of a Validator With Client-Side Scripts".

Example

```
var vAO = DES_FindAOById("<% = RequiredTextValidator1.ClientID %>");
DES_HideVal(vAO);
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

function DES_UpdVal(pID, pCondResult, pErrMsg, pSumErrMsg)

Modifies the state of the validator. Adjusts the Condition Result (which is how to specify valid or invalid) and both error messages. You can pass null into the last 3 parameters to avoid modifying them.

Parameters

pID

The **ClientID** property value from the server side Validation control. It is the value written into the `id=` attribute of the HTML element. See "Embedding the ClientID into your Script".

pCondResult (integer)

Change the Condition Result. Here are the valid values:

- 1 – Valid
- 0 – Invalid
- -1 – Cannot evaluate
- `null` – ignore this parameter

pErrMsg

A new error message to show in the ErrorFormatter. It can contain runtime tokens. Pass `null` to avoid changing the error message.

pSumErrMsg

A new error message to show in the ValidationSummary control. It can contain runtime tokens. Pass `null` to avoid changing the error message.

Example 1

Makes the RequiredFieldValidator1 invalid with new messages.

```
DES_UpdVal("<% = RequiredTextValidator1.ClientID %>", 0,
     "New error message", "New summary message");
```

Example 2

Makes the RequiredFieldValidator1 invalid without changing messages.

```
DES_UpdVal("<% = RequiredTextValidator1.ClientID %>", 0, null, null);
```

function DES_VALReset(pPostBack)

Restores validators to their original visibility settings from when the HTML was loaded. If the page is new, they will all be hidden. If there is a postback, you can show their values from immediately after the postback or hide them all, depending on the *pPostBack* property.

Note: It only adjusts the IsValid property and control's visibility. It will not restore any other client-side property.

Parameters

pPostBack

When `true`, and there has been a postback or callback, restore the validator state to the value set during the callback or postback. When `false` or if the page is new, hide the validator.

Example

```
DES_VALReset(false);
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

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' >
<!--
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' >\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' >")
   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' 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 + "');"
```

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

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

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

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 **DES Validation Framework**. 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".

A great tool is this report: Analyzing the Page's Validation configuration.

Runtime Problems

Also see the "Troubleshooting" section of the **General Features Guide.**

When you submit a page, validation doesn't run on the server side

A great tool is this report: Analyzing the Page's Validation configuration.

1. The method `PeterBlum.DES.Globals.WebFormDirector.Validate()` is not called. It is called automatically from inside a DES button's **Click** event if **CausesValidation**=`true`. It won't happen on a non-DES button unless you attach the NativeControlExtender to the button or use the `PeterBlum.DES.Globals.WebFormDirector.SubmitPageManager.RegisterSubmitControl()` method. It won't happen when **CausesValidation**=`false`.

   When it does not happen automatically, call it within the postback event handler method before saving or using the data.

2. You are not testing **PeterBlum.DES.Globals.WebFormDirector.IsValid**=`true` inside your post back event handler where you attempt to save the data. You must test **IsValid** either on the `PeterBlum.DES.Globals.WebFormDirector` or individual Validator objects to determine if the page is valid. Do this in your postback event handler method before saving or using the data.

3. The validators are disabled, either through the **Enabled** or **Enabler** properties.

4. The validation group name in the **Group** property does not match between the button and the validators. See "Validation Groups".

For an overview, see "Submitting the Page: Server-Side Validation".

When you submit a page, validation doesn't run on the client side

A great tool is this report: Analyzing the Page's Validation configuration.

Any control that submits the page must be set up to run DES's client-side code that validates upon submission. If you are using DES's submit buttons, this is built in. See "DES's Button, LinkButton, and ImageButton Controls". Otherwise, please consider these types of controls that submit the page:

- Native buttons. See "Using Native Button Controls to Submit the Page".
- Buttons in a GridView or DataGrid. See "DES's Submit Controls For The GridView and DetailsView" or "DES's Submit Controls For The DataGrid".
- The **AutoPostBack** property on a WebControl. See "Using Validation with AutoPostBack".

Additionally, here are several reasons why client-side validation doesn't occur:

- The validation group assigned to the submit control does not match any validators.
- The Validator has its **EnableClientScript** property set to `false`.
- The Validator has its **EventsThatValidate** property set to `OnChange`.
- The browser doesn't support client-side validation.
- The user has disabled JavaScript on the browser.
- There is a JavaScript error on the page. Please see the "Handling JavaScript Errors" topic in the "Troubleshooting" section of the **General Features Guide**.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

The button's Click or Command event doesn't run

DES's Button, LinkButton, and ImageButton will call their **Click** and **Command** event when they are the control that submits the page. You can override this behavior by setting their **SkipPostBackEventsWhenInvalid** property to `TrueFalseDefault.True`. In that case, when the button detects a validation error, it will skip the **Click** and **Command** event. Change the **SkipPostBackEventsWhenInvalid** property to `TrueFalseDefault.False` to always call these events.

The Validators and FieldStateControllers don't run on the client-side

There are a number of settings that can disable the Validator controls. See "Changing When the Validator is Evaluated" for properties on the Validator controls, and "Browser Support" in the **General Features Guide** for browsers affecting the client-side.

Additionally, the scripting may not be properly set up. Please see the "Handling JavaScript Errors" topic in the "Troubleshooting" section of the **General Features Guide**.

Finally, if the page is called from `Server.Transfer()`, you must add some code to the original and destination pages. See "Using Server.Transfer" in the **General Features Guide**.

I am using Page.Validate() or Page.IsValid. These don't work

You are using methods from the Native Validation Framework. The DES Validation Framework does not use the **Page.IsValid** property or `Page.Validate()` method. DES supplies matching properties and methods: **PeterBlum.DES.Globals.WebFormDirector.IsValid** and `PeterBlum.DES.Globals.WebFormDirector.Validate()`. See "Submitting the Page: Server-Side Validation".

Validator controls do not have the correct styles

By default, Validator controls should have a red font. This comes from a style sheet that you must link to each page. Most likely, you forgot to establish the style sheet. See "Using Style Sheets" in the **General Features Guide**.

Validator error messages do not show in the ValidationSummary

Consider the following:

- You are using a Native Validation Framework Validator control instead of a DES Validation Framework Validator control. Native controls have an <asp:*classname* > tag and the namespace System.Web.UI.WebControls.
- The ValidationSummary from the Native Validation Framework.

To quickly convert to DES Validation Framework controls, see run the **Web Application Updater** program with the option **Convert native controls to their DES equivalents**. If you have a Validator control that doesn't come with the ASP.NET controls from Microsoft, you will need to create an equivalent control using DES's classes.

After selecting an item from an AutoComplete popup in IE, it doesn't validate

Internet Explorer popups up little windows called "AutoComplete" on textboxes. When you select from the autocomplete, Internet Explorer does not fire the textbox's onchange event that DES uses to validate when the user changes a field. *This was tested in IE for Windows v6.0.28.*

When the user submits the page, it will still detect and report a validation error.

Solution: Use one the TextBox classes supplied with DES. They have a feature to avoid this problem. See the **TextBoxes User's Guide**.

Validation causes the page to move elements. However, absolutely positioned elements don't move and appear out of place

Client-side validation adds and removes elements on the page. This causes the browser to move other elements on the page to fit the validation error message or ValidationSummary. If you have absolutely positioned elements on the page that are carefully positioned next to non-absolutely positioned elements, the non-absolutely position element may move. But the browser will not move the absolutely position element. That's the design of absolutely positioned elements.

To fix this, use the **PeterBlum.DES.Globals.WebFormDirector.ValidationManager.PostValidationUpdateScript** property to call JavaScript that updates the positions of absolutely positioned elements on the page.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

If you are using a third party control, contact the developer for any available JavaScript code they have to handle this case.

The user needs to click twice on the button to submit the page

This happens when focus is in a data entry control that was just edited to correct a problem and the user clicks on the button.

The problem is that the button control moves between the moment you click the mouse on it and when it's ready to run its DHTML onclick event. The reason is that before the button's onclick event runs, a validator runs its validation due to the DHTML onchange event of the associated data entry control. When it does this, the screen has been rearranged to hide its error message, causing the button to move. The browser will not run the button's DHTML onclick event if the mouse is pulled off the button and due to it moving, that's exactly what's happening.

Here are the solutions:

- If you are using DES's Button, LinkButton, or ImageButton control, set the **MayMoveOnClick** property to `true`. If you are using the Microsoft Button or ImageButton controls, switch them to DES's and use that property.
- Otherwise, make sure your button is not able to relocate in this situation. It may take repositioning it or restructuring something about your page so it doesn't cause a repositioning. For example, use the validator.ErrorFormatter.Display = Static value (which gives a fixed size for the error message) or move the ValidationSummary the button.

When several error messages are overlayed by using the Display=Dynamic setting, they do not align

Suppose you have two Validators to the right of a textbox, a RequiredTextValidator and CompareToValueValidator. Each is set to Display=Dynamic. Only one will show at a time. When one appears, it does not start at the exact same left coordinate as the other. Here is a visual representation using an image to better show the problem.

There are two possibilities.

1. If you are using the GapBefore, only use it on the leftmost Validator. The GapBefore is always shown on the page, even when the error message is hidden.
2. You have a carriage return between the two Validators in the HTML. The browser adds a few pixels where it finds a carriage return.

   Change this:

```
<des:validator [parameters]>
   [innerparameters]
</des:validator>
<des:validator [parameters]>
   [innerparameters]
</des:validator>
```

   To this:

```
<des:validator [parameters]>
   [innerparameters]
</des:validator><des:validator [parameters]>
   [innerparameters]
</des:validator>
```

ValidationSummary does not list validators in the right order

Normally the Validators appear in the order they are in the page's Control tree. Here are two cases that break that rule:

1. You have used absolute positioning to make the validators visually in a different order than they are on the page. Move their order on the page to be the desired order.
2. If you have an **Enabler** property set up, make sure its **EvaluateOnClickOrChange** property is `false`.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

TextBoxes use a yellow background color in IE for Windows

The user has Google Toolbar installed. It modifies the background color of textboxes that don't have their own style sheet class assigned.

Enabler, ErrorFormatter, or HiliteFields properties are ignored

Does the validator start with <despval:? If so, the ASP.NET Markup does not support these properties.

When you convert native validators to DES, it creates validators with the same class name and properties as the native validators, but underlying it is the power of a DES validator. <despval: indicates one of these converted validators.

Since any <despval: validator does not allow **Enabler**, **ErrorFormatter**, or **HiliteFields** properties in the ASP.NET Markup, you have these options:

1. Assign the property programmatically, in `Page_Load()` or if in a control that creates them during databinding, its ItemCreated or RowCreated event. Be sure to remove the nested property from the ASP.NET Markup.
2. Replace the control with the actual DES validator. Start fresh, with no properties assigned. It may be smart to add the new validator side-by-side, then apply properties to their new names using the Properties Editor.

   `<despval:RequiredFieldValidator>` → `<des:RequiredTextValidator>`

   `<despval:CompareValidator Operator=DataTypeCheck>` → `<des:DataTypeCheck>`

   `<despval:CompareValidator ValueToCompare=`*`value`*`>` → `<des:CompareToValueValidator>`

   `<despval:CompareValidator ControlToCompare=`*`controlid`*`>` → `<des:CompareTwoFieldsValidator>`

   `<despval:RangeValidator>` → `<des:RangeValidator>`

   `<despval:RegularExpressionValidator>` → `<des:RegexValidator>`

   `<despval:CustomValidator>` → `<des:CustomValidator>`

   Property mappings:

   ControlToValidate → ControlIDToEvaluate

   Operator=`DataTypeCheck` - remove it

   Type → DataType

   ErrorMessage stays the same unless you also have a **Text** property. In that case, **Text** becomes **ErrorMessage** and **ErrorMessage** becomes **SummaryErrorMessage**

   ControlToCompare → SecondControlIDToEvaluate

   ValidationExpression → Expression

   For the CustomValidator, the client and server side functions are different. You will need to recreate them. Yet it may be possible to create the same logic with other DES validators.

DropDownLists reporting a JavaScript error in Google Chrome 1

When a DropDownList has **AutoPostBack** set to `true` and you assign a DES validator to that DropDownList, Google Chrome v1 will generate a JavaScript error. Use the NativeControlExtender to avoid this. Add it to any DropDownList whose **AutoPostBack** property is set to `true` and has validators.

Internally the NativeControlExtender swaps the original AutoPostBack scripts with alternatives that work with all browsers.

Javascript expressions don't consistently work

When using the RegexValidator, there are two regular expression parsers at work. During client-side validation, the Javascript RegExp class engine is used. During server-side validation, the .net Regex class engine is used. .net's engine has more features. When writing regular expressions, if you intend them to work on the client side, always design your expression to work with the JavaScript RegExp class engine.

JavaScript RegExp class guide

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

JavaScript regular expression testing tool – Set **Select Regex Engine** to "Client-side Engine" and **Engine** (at the bottom of the Regex Input box) to "JavaScript".

Internet Explorer has a few reported bugs in its engine that you should be aware of. They are described in this post:

"A JScript/VBScript Regex Lookahead Bug" by Steven Levithan

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Design Mode Problems

Also see the "Troubleshooting" section of the **General Features Guide.**

Validator controls have the wrong or missing properties in the Properties Editor

You are probably looking at one of Microsoft's Validator Controls or a Validator subclassed from System.Web.UI.WebControls.BaseValidator. You need to convert it to a DES control.

If the **ErrorFormatter** property is missing but others are present, you are looking at a "Microsoft-style" DES Validator created by the **Web Application Updater** program when it converted native controls to their DES equivalents. Unless you need the **ErrorFormatter** property, you can continue to use it, as it is fully DES compatible. If you need the **ErrorFormatter** property, you will need to replace the control with the desired DES Validator control.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

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

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

- 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.

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Table of Contents

GETTING STARTED 3

Evaluating data: The Validator controls and their Condition objects 4
Evaluating Textboxes and other controls with textual values 5
Evaluating the Listbox, DropDropList, RadioButtonList, and CheckBoxList 8
Evaluating CheckBoxes, RadioButtons, and other controls 9
Evaluating non-data value states of controls 10
Evaluating all other cases 11

Determining When Validators Fire 12

Communicating Errors 13

Other Controls in the DES Validation Framework 15
Submitting The Page 15
Showing the "Required Field" Marker 16
Supported data entry controls 16
Working within Grid Controls 16

ADDING VALIDATION TO A WEBFORM 17

Big Picture 17

Step-by-Step 17

VALIDATOR CONTROLS 22

Evaluation rules for Validators: Condition objects 23
Available Validator controls 23
Finding the Validator by the Data Entry Control 23
Properties Common to most Conditions 24
Quick jump to the Condition properties of individual Validator controls 26

RequiredTextValidator 27
Using This Condition 27
Condition Properties for RequiredTextValidator 28

RequiredListValidator 29
Using This Condition 29
Condition Properties for RequiredListValidator 30

DataTypeCheckValidator 31
Using This Condition 31
Condition Properties for DataTypeCheckValidator 32
Range Properties on the DataTypeCheckCondition Class 32
ControlToNative() method 33

CompareToValueValidator 35
Using This Condition 35
Condition Properties for CompareToValueCondition 37

CompareTwoFieldsValidator 39
Using This Condition 39
Condition Properties for CompareTwoFieldsCondition 40

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

RangeValidator **42**
Using This Condition 42
Condition Properties for RangeValidator 43

RegexValidator **45**
Learning About Regular Expressions 46
Using This Condition 47
Condition Properties for RegexValidator 48

CheckStateValidator **51**
Using This Condition 51
Condition Properties for CheckStateValidator 52

SelectedIndexValidator **53**
Using This Condition 53
Condition Properties for SelectedIndexValidator 54

TextLengthValidator **55**
Using This Condition 56
Condition Properties for TextLengthValidator 57

EmailAddressValidator **58**
Using This Condition 59
Omitting Certain Domains 59
Condition Properties for EmailAddressValidator 60
Using aspNetMX 62
Changing the Default Email Address Pattern 63

MultipleRequiredControlsValidator **64**
Using This Condition 65
Adding Textual Controls to ControlsToEvaluate 65
Adding List Controls to ControlsToEvaluate 66
Adding CheckBox/RadioButton Controls to ControlsToEvaluate 67
Determining the number of controls that are required 68
Condition Properties for MultipleRequiredControlsValidator 69

RequiredSelectionValidator **71**
Using This Condition 72
Supporting Other Controls 73
Adding your class to the ControlAdapterFactory 73
Setting up Client Side Validation 74
Condition Properties for RequiredSelectionValidator 75

CharacterValidator **76**
Using This Condition 76
Condition Properties for CharacterValidator 77

CompareToStringsValidator **80**
Using This Condition 81
Condition Properties for CompareToStringsValidator 82

WordCountValidator **85**
Using This Condition 85
Condition Properties for WordCountValidator 86

DifferenceValidator **87**
Using This Condition 88
Condition Properties for DifferenceValidator 89

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

CountSelectionsValidator **91**
Using This Condition 91
Condition Properties for CountSelectionsValidator 92

DuplicateEntryValidator **93**
Using This Condition 94
Condition Properties for DuplicateEntryValidator 95

UnwantedWordsValidator **98**
Using This Condition 99
Condition Properties for UnwantedWordsValidator 100

SelectedIndexRangesValidator **103**
Using This Condition 103
Condition Properties for SelectedIndexRangesValidator 104

ListSizeValidator **106**
Using This Condition 106
Condition Properties for ListSizeValidator 108

CreditCardNumberValidator **109**
Using This Condition 110
Condition Properties for CreditCardNumberValidator 111
Editing Credit Card Brands 112

ABARoutingNumberValidator **114**
Using This Condition 114
Condition Properties for ABARoutingNumberValidator 115

CountTrueConditionsValidator **116**
Using This Condition 117
Condition Properties for CountTrueConditionsValidator 118

MultiConditionValidator **120**
Using This Condition 121
Condition Properties for MultiConditionValidator 122

CustomValidator **124**
Using This Condition 126
Steps to Defining Your CustomValidator 126
Server-Side Condition 127
Properties of PeterBlum.DES.Web.WebControls.ConditionTwoFieldEventArgs 128
Server-Side Condition Example 129
Changing The Error Message On The Server Side 131
Client-Side Condition 132
The Client-Side Evaluation Function 132
Accessing Data On the Client-Side Condition object 133
Client-Side Condition Example 134
Changing The Error Message On The Client Side 135
Passing around data into your evaluation function 136
Defining your data 137
Attaching the data to your CustomValidator 138
Using your data inside the server-side evaluation method 139
Using your data inside the client-side evaluation function 139
Condition Properties for CustomValidator 140

IgnoreConditionValidator **142**
Using This Validator 143

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Condition Properties for IgnoreConditionValidator 145

Non-Data Entry Conditions 147
Non-Data Entry Condition Classes 148
Using Non-Data Entry Conditions 149
Properties for VisibleCondition 150
Properties for EnabledCondition 151
Properties for ClassNameCondition 152
Properties for ReadOnlyCondition 153
Properties for CompareToValueAttributeCondition 154

Extending Existing Validators and Conditions 156
Extending the Server Side 157
Extending The Client-Side 159

Changing When the Validator is Evaluated 160
Validation Groups 161
Using Validation Groups 161
Special Characters In Group Names 161
The Enabler Property 162
Other Properties That Disable Validation 165

Defining the Error Message and Associated Labels 167
Properties for Error Messages and Associated Labels 168
Properties of the PeterBlum.DES.Web.WebControls.LabelText Class 171
Tokens in Error Messages 172
Applying Styles To Tokens 176

ErrorFormatters: Customizing the Appearance of the Error Message 178
Using ErrorFormatters 179
Setting the Appearance 180
Editing the ErrorFormatter in Design Mode 181
ASP.NET Markup for the ErrorFormatter Property 182
Creating an ErrorFormatter Programmatically 183
ErrorFormatter Skins 184
Using ErrorFormatter Skins 184
Defining a Skin in Design Mode 186
Defining a Skin in a Text Editor 187
Defining a Skin Programmatically 188
Properties Common To All ErrorFormatter Classes 189
Text 191
Using The TextErrorFormatter 191
Properties for TextErrorFormatter (“Text”) 191
Hyperlink with Alert 194
Using This ErrorFormatter 194
Properties for HyperLinkErrorFormatter (“HyperLink with Alert”) 194
Image with Tooltip 196
Using This ErrorFormatter 196
Properties for TooltipImageErrorFormatter (“Image with ToolTip”) 196
Image with Alert 197
Using This ErrorFormatter 197
Properties for AlertImageErrorFormatter (“Image with Alert”) 197
PopupView 199
Using This ErrorFormatter 199
Properties for PopupErrorFormatter (“PopupView”) 200
Working With PopupViews 204
PopupView Features 204
Using the Global Settings Editor to define PopupViews 205

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

View an existing definition 206
Edit a definition 208
Add a definition 208
Rename a definition 208
Delete a definition 208
Creating your own Callouts 208
Adding your own Callouts to the PopupView Definition 209

Properties for the PeterBlum.DES.Web.WebControls.ErrorFormatterPopupView Class 210
Overall Appearance Properties 211
Header Properties 212
Body Properties 215
Footer Properties 217
Callout Properties 219
Positioning Properties 221
Other Properties 222

Other Validator Properties that Customize the Appearance 225
PeterBlum.DES.Web.WebControls.NoErrorFormatter Class 227

Other Properties 229

DRAWING THE USER'S ATTENTION TO THE ERROR 230

Set Focus to the Control 231
Page-Level Properties 232

Show an Alert 233
Page-Level Properties 234

Change the Style of the Field With the Error 236
Alternative Styles 237
Using This Feature 238
Page-Level Properties 239

Change the Style of Other Fields Nearby the Error 243
Using This Feature 244
Page-Level Properties 246

Blinking the ErrorFormatter 248
Page-Level Properties 249
Properties on the Validator 250

Warning Mode on Validators 251
Page-Level Properties 251

Submitting the Page: Server-Side Validation 253
Setting Up Server-Side Validation 254
DES Submit Control PostBack Event Handlers when CausesValidation=true 254
PostBack Event Handlers when CausesValidation=false 255
Validating With All Other PostBack Event Handlers 256
Preventing Validation Or Evaluating Individual Validators 257
Validating a group of validators 257
Properties and Methods to Evaluate a Validator Control 258
DES's Button, LinkButton, and ImageButton Controls 259
Using DES's Button, LinkButton, and ImageButton Controls 260
Properties on DES's Button, LinkButton, and ImageButton Controls 262

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

DES's Submit Controls For The GridView and DetailsView 264
Using DES's CommandField Control 264
Using DES's ButtonField Control 265
DES's Submit Controls For The DataGrid 266
Using the EditCommandColumn Control 266
Using the ButtonColumn Control 267
Using Native Button Controls to Submit the Page 268
The RegisterSubmitControl method 269
The PeterBlum.DES.SubmitBehavior Class 270
The RegisterChildSubmitControls method 272
The PeterBlum.DES.ChildSubmitBehavior Class 274
Using the Menu Control to Validate the Page 277
The PrepareMicrosoftMenuControl() method 278
Using the BulletedList Control 279
Steps to set up the BulletedList Control 279
The RegisterBulletedListControl() method 280
Using AJAX Callback Controls 281
Update panels 281
Buttons 281
Other controls 281
The RegisterCallbackControl() method 281
Using Validation with AutoPostBack 283
How to Add DES Validation to AutoPostBack 283
AutoPostBack with Peter's Date Package TextBoxes 285
Overcoming a bug in Google Chrome 1 285

Submitting the Page: Client-Side Validation 286
Page-Level Properties used by Client-Side Validation 287

ADDITIONAL VALIDATION TOPICS 290

Validation Best Practices 291

Using the DES Validation Framework with each ASP.NET web control 292
AdRotator 293
BulletedList 293
Button 293
Calendar 293
ChangePassword 293
Example 294
CheckBox 296
CheckBoxList 296
CompareValidator 296
CreateUserWizard 297
Example 297
CustomValidator 301
DataGrid 301
DataList 301
DetailsView 301
DropDownList 302
Overcoming a bug in Google Chrome 1 302
FileUpload 303
Checking the FilePath 303
Server Side Validation 303
Example 304
FormView 306
GridView 306
HiddenField 306

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

HyperLink 306
Image 306
ImageButton 306
ImageMap 307
Label 307
LinkButton 307
ListBox 307
Literal 307
Localize 307
Login 307
Example 309
LoginName 310
LoginStatus 310
LoginView 310
Menu 310
MultiView 310
Panel 310
PasswordRecovery 310
Example 311
PlaceHolder 312
RadioButton 312
RadioButtonList 312
RangeValidator 312
RegularExpressionValidator 312
RequiredFieldValidator 312
Repeater 313
SiteMapPath 313
Substitution 313
Table 313
TextBox 313
TreeView 313
ValidationSummary 314
WebZone and WebParts 314
Wizard 315
XML 315

Using Validators with Third Party Controls 316

Analyzing the Page's Validation configuration 317
Querystring parameter 317
Add a control to the page 317

Supporting a Reset or Clear Button 318

The ViewState and Preserving Properties for PostBack 319

Validation and the PostBackUrl Property 320

Invoking Validation With Client-Side Scripts 321
Evaluating all Validators for a Validation Group 321

Running a Script when validation occurs 322
Run a script before page-level validation 322
Run a script after page-level validation 322
Run a script after any validation occurs 322

Changing The Properties of a Validator With Client-Side Scripts 323

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

Adding Client-Side Validation Scripts Within Your Server Side Code **325**
PeterBlum.DES.Globals.WebFormDirector.SubmitManager.GetPostBackEventReference method 325
PeterBlum.DES.Globals.WebFormDirector.SubmitManager.GetPostBackClientHyperlink method 327
PeterBlum.DES.Globals.WebFormDirector.ValidationManager.GetValidateGroupScript Method 328

Support for the Native Validation Framework **329**

Using Validator controls to evaluate data not from Controls **330**
Advantages of using a Validator object 330
Advantages of using a Condition object 330
Guidelines for using these objects 332
Converting a string to a native type 332

Attaching Validators to controls inside of a User Control **333**

VALIDATIONSUMMARY CONTROL **334**

Features **335**

Using ValidationSummary Control **336**
When the Control Is Shown and Hidden 336
Setting the Appearance and Behavior 337
Overall Appearance 337
Header 338
Error Messages 339
Footer 341
The Related Control Feature 342
Adding ErrorMessages At Runtime 343
Using the ValidationSummary as a Label for “There are errors” 345
Show when the user attempts to submit the page 345
Show as edits are made 345

Adding a ValidationSummary Control **346**

Properties of the ValidationSummary Control **348**
When to Update Properties 349
Error Message Formatting Properties 351
Overall Appearance Properties 354
Header Properties 355
Footer Properties 357
Behavior Properties 358
Related Control Properties 359

HTML Structure of the ValidationSummary Control **360**

COMBINEDERRORMESSAGES CONTROL **361**

Using the CombinedErrorMessages Control **362**

Adding the CombinedErrorMessages Control **363**

Properties for the CombinedErrorMessages Control **365**
Setting the Validators Properties 366
Appearance Properties 367
Behavior Properties 370

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved

Technical support and other assistance: http://www.peterblum.com/des/support.aspx

REQUIRED FIELD MARKER CONTROLS 371

RequiredFieldMarker Control 372

RequiredFieldsDescription Control 373

Setting up the Global Defaults 374

Adding a RequiredFieldMarker Control 376

Properties of the RequiredFieldMarker Control 377

Adding a RequiredFieldsDescription Control 378

Properties of the RequiredFieldsDescription Control 379

PAGE LEVEL PROPERTIES AND METHODS 380

Properties on PeterBlum.DES.Globals.WebFormDirector 381

PeterBlum.DES.Globals.WebFormDirector.ValidationManager properties 385

PeterBlum.DES.Globals.WebFormDirector.ValidationManager methods 387
ValidationManager.EnableValidators() method 387
ValidationManager.EnableClientScriptValidators() method 387

Debugging PeterBlum.DES.Globals.WebFormDirector Properties 388

JAVASCRIPT SUPPORT FUNCTIONS 389

General Utilities 389

Retrieving Values From Data Entry Controls 392

Validation Functions 394

ADDING YOUR JAVASCRIPT TO THE PAGE 397

Embedding the ClientID into your Script 397

Debugging Your JavaScript 398

TROUBLESHOOTING 399

Runtime Problems 399
Enabler, ErrorFormatter, or HiliteFields properties are ignored 402

Design Mode Problems 404

TECHNICAL SUPPORT AND OTHER ASSISTANCE 405
Troubleshooting Section of this Guide 405
Developer's Kit 405
PeterBlum.Com Forums 405
Getting Product Updates 405
Technical Support 405

Copyright © 2002-2012 Peter L. Blum. All Rights Reserved