

**MAD !** *widgets*

**ASP.NET Validation  
Package**

# Table of Contents

<b>1</b>	<b>INTRODUCTION.....</b>	<b>3</b>
<b>2</b>	<b>INSTALLATION.....</b>	<b>4</b>
2.1	Installing the assembly .....	4
2.1.1	Installing the assembly on the website directly.....	4
2.1.2	Installing the assembly in the GAC.....	4
2.2	Installing the license .....	5
2.3	Copying secondary files .....	6
2.4	Setting up the web.config file.....	7
<b>3</b>	<b>GETTING STARTED.....</b>	<b>9</b>
3.1	Creating the WebForm .....	9
3.2	Adding Mad! Widgets validation .....	12
3.3	Validation styles and highlight elements.....	15
3.4	Placing validation in your Code Behind class .....	21
3.5	Using validation on non-server form fields .....	25
<b>4</b>	<b>VALIDATION GROUPS.....</b>	<b>28</b>
4.1	Sub forms and form controls .....	28
4.2	Validation groups on a single control.....	31
4.3	Using validation groups with non-server fields.....	31
<b>5</b>	<b>INLINE MESSAGE BLOCKS .....</b>	<b>34</b>
<b>6</b>	<b>CREATING CUSTOM VALIDATORS.....</b>	<b>39</b>
6.1	Extending ValidatorEntrySingleObject.....	40
6.2	Extending ValidatorEntryRegex.....	44
6.3	Extending ValidatorEntryServerOnly .....	46
<b>7</b>	<b>CREATING CUSTOM VALIDATOR SUMMARIES.....</b>	<b>48</b>
7.1	A basic validation summary .....	48
7.2	Validation summaries for specific validation groups .....	50
<b>8</b>	<b>INLINE FUNCTION REFERENCE.....</b>	<b>52</b>

# 1 Introduction

Anyone who has spent any time developing form-based websites will know that a lot of time is spent on writing code that does nothing more than validate form fields. Invariably, such code is written on a need-to-have basis which leads to poor maintainability and a deficiency in sensible architecture.

With the introduction of .NET the situation improved slightly. Developers could make use of Microsoft's built in validators by including them as controls on a WebForm. The controls were bound to form field controls, and could validate both client-side and server-side with no more coding than the placement of the control on the page.

But Microsoft's built in controls have proven to be woefully inadequate for any more than the simplest of websites (using the .NET Framework 1.1 or below). They are not capable of validating multiple forms on a single page, they are not aware of relationships that various input fields may have, the client-side code produced by them is safe for use only on Microsoft browsers, error messages generated by invalid form input cannot include properties such as the form fields' values, and the list goes on.

The Mad! Widgets ASP.NET Validation Package overcomes all of these weaknesses by providing a robust solution to validation. The package is a well architected API that is easily extensible and very full featured. Unlike the .NET Framework's built in validators, some basic programming knowledge is required to use the features of the package. In most cases, form fields can be validated with nothing more than a short line of code which generates client-side validation code and enforces server-side validation in the same manner.

The Mad! Widgets ASP.NET Validation Package takes advantage of the underlying validation mechanism built into the Page control and thus, the code required to validate your forms can fit neatly into any existing page design. Validation code can be placed either on the page itself or in the Code Behind class.

## 2 Installation

Installation of the Mad! Widgets ASP.NET Validation Package is a four step process. Precisely what these steps are depends on the use of your web server and which license you purchased.

When installing the package you have the choice to install the primary DLL (the assembly) in the Global Assembly Cache (GAC) or on the website on which the package will be used. If you have purchased a Server License, a Multiple Server License or a Redistribution License and you are running several websites on each server, you may prefer to install the DLL in the GAC as it reduces the chance of having multiple copies of the same DLL on the server. If you have not purchased one of these licenses or you do not have many websites on each server, you may prefer to install the DLL on each website, as this is a simpler process.

### 2.1 *Installing the assembly*

#### 2.1.1 **Installing the assembly on the website directly**

Unpack the MadWidgets.Validation-[version].zip file into a temporary directory and copy the */bin/MadWidgets.Validation.dll* file into your site's */bin* directory. Once done, the assembly is installed on your website.

Alternatively, if you prefer to place the assembly into a sub-domain only, copy it to the sub-domain's */bin* directory.

#### 2.1.2 **Installing the assembly in the GAC**

There is more than one way to install the assembly into the GAC. First unpack the MadWidgets.Validation-[version].zip file into a temporary directory. Then do one of the following:

- Run *gacutil.exe /i [temporary-directory]\bin\MadWidgets.Validation.dll*. *gacutil.exe* is installed with the .NET Framework files and will be located in the directory *C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705*, or whatever the equivalent directory is on your server.

- Drag *[temporary-directory]\bin\MadWidgets.Validation.dll* and drop it into your local *C:\WINDOWS\assembly* directory or whatever your local equivalent is using Windows Explorer. This will automatically install the assembly into the GAC.

Once the assembly has been installed in the GAC, you will need to tell ASP.NET where to find it. You can do this by adding the following line to the *machine.config* XML file. This file is normally located in the directory *C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705\Config* or whatever your local equivalent is. This line is placed in the section */configuration/system.web/compilation/assemblies*.

```
<configuration>
  <system.web>
    <compilation>
      <assemblies>
        <add assembly="MadWidgets.Validation, Version=1.3.0.0,
          Culture=neutral, PublicKeyToken=f238745449bc9bec" />
      </assemblies>
    </compilation>
  </system.web>
</configuration>
```

Note that the Version value in this example is based on the version number of the assembly. The version number of your assembly can be found by right-clicking on the DLL file, selecting Properties, then the Version tab.

You should not remove any existing entries relating to the Mad! Widgets ASP.NET Validation Package unless you are certain that no sites on the server are using the earlier version of the control any longer. Similarly, you should not remove any earlier versions of the assembly from the GAC either for the same reason.

You may need to restart IIS before the new assembly will be picked up by the websites running on it.

## 2.2 Installing the license

The license file you received in your email will have the name *MadWidgets.Validation.Validator.lic*. Copy this file to one of the locations listed below.

- The */bin/licenses* directory (you may need to create this) on the website(s) that will use the package.
- The *[sub domain root]/bin/licenses* directory (you may need to create this) on the website(s) that will use the package where *[sub domain root]* is the root directory of the sub domain. E.g. if you have a site with the sub-domain *mysub.mydomain.com*, then "mysub" will typically be the root directory of the sub-domain and it is normally located directly under site's primary root directory. The sub-domain's bin directory will then be located at */mysub/bin*.
- The location where the *MadWidgets.Validation.dll* file is located in the GAC. The assembly will be located in a directory resembling *C:\WINDOWS\assembly\GAC\MadWidgets.Validation\1.3.0.0\_\_f238745449bc9bec\*. Windows Explorer will not allow you into this directory, however, you can

navigate to it through third-party file system management software or through a command prompt. Note that you should not put the license file here if you are referencing the assembly through VS.NET or WebMatrix.

- In the Windows SYSTEM32 directory.
- In the directory *C:\Program Files\Common Files\Mad! Widgets\Validator*, or whatever your local equivalent is. You may need to create this directory.

In most situations, you will copy the license file to */bin/licenses* on your website (the first option above).

Note that if you have placed the license in the correct directory and the Validator control is not finding it, you may not have sufficient permissions on the file for the control to read it. You should have read permissions on the file for the account that ASP.NET is running under (e.g. the local ASPNET account).

## 2.3 Copying secondary files

The ASP.NET Validation Package comes with the following files in addition to the main assembly file *MadWidgets.Validation.dll*:

File	Purpose
/javascript/Validator.js	The JavaScript file that manages the client-side validation.
/javascript/ValidatorUtils.js	A JavaScript file containing several helper functions that can be used even when not using the validation package.
/Validator.ascx	A sample Validation Summary control.
/css/Validator.css	The style sheet used with the Validation Summary. You will need to explicitly include this in your page for the sample Validation Summary to display correctly, e.g. in the <head> section.
/images/tick.gif	The "tick" image that appears in the top right of a textual field as that field passes validation. This image appears for a couple of seconds and appears only when the UseTicks option is enabled.
/images/cross.gif	The bright red "cross" image that appears in the top right of a textual field as that field fails validation. This image appears for a couple of seconds and appears only when the UseTicks option is enabled.
/images/crossdim.gif	The dim red "cross" image that remains in a textual field that has failed validation. This image appears only when the UseTicks option is enabled.

<b>File</b>	<b>Purpose</b>
/images/error.gif	The icon in the top left corner of the built-in Validation Summary. This is contained in the CSS (/css/Validation.css) file associated with the Validation Summary.
/LicenseAgreement.doc	The license agreement in MS Word format.
/LicenseAgreement.pdf	The license agreement in PDF format.

Each of the files above can be copied anywhere that you desire within your website. They do not need to remain in the directory structure found in the ZIP package. In step 4 of this installation procedure you will be shown how the Validator control learns where these files are located with entries in the web.config file.

Note that the license agreement files (*LicenseAgreement.doc* and *LicenseAgreement.pdf*) do not need to be copied to your website.

## **2.4 Setting up the web.config file**

In order for the Validator control to know where you have placed each of the files listed in the previous step, you need to add entries into the web.config file. You will need to add these entries in the appSettings section of the file.

The following entries may be added to the file:

<b>Property</b>	<b>Purpose</b>	<b>Required</b>
validatorJavascriptPath	The path of the Validator.js file relative to the root of the website.	Yes
validatorSummaryTitle	The title of the validation summary control that is shipped with the ASP.NET Validator. This property can be accessed through the SummaryTitle property of the Validator control.	No
validatorInitialHighlightClass	The name of the CSS class that is used on the highlight element when the page first loads. validatorHighlightClass used when it is not specified.	No
validatorHighlightClass	The name of the CSS class that is used on the highlight element when the respective field or fields fail validation.	No

Property	Purpose	Required
validatorDefaultHighlightElement	The default signature of the highlight element. Be sure to escape any < and > characters in the signature with &lt; and &gt;.	No
validatorUseTicks	When "true" this property instructs the Validator control to enable client-side ticks and cross in textual fields that are being validated. True by default.	No
validatorTickPath	The path of the tick.gif image relative to the root of the website or its customised equivalent.	Yes - when UseTicks enabled
validatorCrossPath	The path of the cross.gif image relative to the root of the website or its customised equivalent.	Yes - when UseTicks enabled
validatorDimCrossPath	The path of the crossdim.gif image relative to the root of the website or its customised equivalent.	Yes - when UseTicks enabled
validatorClientSideEnabled	When "true" instructs the Validator control to enable client-side code. True by default.	No

A typical appSettings section might look like the following:

```
<configuration>
  <appSettings>
    <add key="validatorSummaryTitle" value="The follows errors
      occurred on the page:" />
    <add key="validatorInitialHighlightClass"
      value="validatorInitial" />
    <add key="validatorHighlightClass" value="validatorError" />
    <add key="validatorDefaultHighlightElement"
      value="&lt;div[className=formInput]&gt;" />
    <add key="validatorUseTicks" value="true" />
    <add key="validatorJavascriptPath" value="/javascript/" />
    <add key="validatorTickPath" value="/images/tick.gif" />
    <add key="validatorCrossPath" value="/images/cross.gif" />
    <add key="validatorDimCrossPath" value="/images/crossdim.gif" />
    <add key="validatorClientSideEnabled" value="true" />
  </appSettings>
</configuration>
```

Once all four steps above are completed, the Mad! Widgets ASP.NET Validation Package should be ready to use. If you have not been successful in installing and setting up the control, please contact [support@madwidgets.com](mailto:support@madwidgets.com) for assistance.

## 3 Getting started

To begin integrating the Mad! Widgets ASP.NET Validators into your pages, you will need a page with a form in it (a WebForm). If you have existing .NET Framework validators in your WebForm, remove all of them so that you are left with a bare page or control with nothing more than form controls in it.

### 3.1 Creating the WebForm

Throughout this section we will be developing a simple WebForm containing a few form fields that are used for an electronic magazine subscription.

We begin with the bare WebForm. Example 3.1 shows the WebForm used for the electronic magazine subscription. Note that example 3.1 is a C# example. A VB.NET equivalent of this example is shipped with the package under the directory */samples*.

*Example 3.1. e-Magazine subscription.*

```
<%@ Page Language="C#" %>
<%@ Import namespace="MadWidgets.Validation.Samples" %>
<%@ Import namespace="System.Globalization" %>

<script runat="server">

void Submit_Clicked(object o, EventArgs ea)
{
    if (Page.IsValid)
    {
        try
        {
            // Subscribe the user to the magazine
            Subscription s = new Subscription();
            s.FirstName = FirstName.Text;
            s.LastName = LastName.Text;
            s.Email = Email.Text;
            s.DateOfBirth = DateTime.Parse(DateOfBirth.Text,
                CultureInfo.CurrentCulture);
            s.Type = EmailType.Items[EmailType.SelectedIndex].Value;
            s.WhereHeardOfUs = WhereHeardOfUs.Items[
                WhereHeardOfUs.SelectedIndex].Value;
            s.WhereHeardOfUsOther = WhereHeardOfUsOther.Text;
            s.Save();
        }
    }
}
```



```

        <asp:ListItem value="TEXT">Textual
            Format</asp:ListItem>
        <asp:ListItem value="HTML">HTML
            Format</asp:ListItem>
        <asp:ListItem value="RTF">Rich Text
            Format</asp:ListItem>
    </asp:DropDownList>
</td>
</tr>
<tr class="row2">
    <td class="label" valign="baseline">Where did you hear
        about e-Magazine?</td>
    <td class="field">
        <asp:DropDownList id="WhereHeardOfUs" runat="server">
            <asp:ListItem value="">-- Please select where you
                heard of e-Magazine --</asp:ListItem>
            <asp:ListItem value="FRIEND">Through a
                friend</asp:ListItem>
            <asp:ListItem value="ENGINE">Through a search
                engine</asp:ListItem>
            <asp:ListItem value="LINK">By clicking a
                hyperlink</asp:ListItem>
            <asp:ListItem value="PAPER">In a paper
                magazine</asp:ListItem>
            <asp:ListItem value="OTHER">Other (see
                below)</asp:ListItem>
        </asp:DropDownList><br />
        <asp:TextBox id="WhereHeardOfUsOther"
            width="180" runat="server" />
    </td>
</tr>
<tr>
    <td colspan="2" align="right"><br /><asp:Button
        id="Submit" runat="server"
        onclick="Submit_Clicked"
        text=" Submit " /></td>
</tr>
</table>
</form>
</asp:Panel>

<asp:Panel id="Page2" runat="server" visible="false">
    <p>Thank you for subscribing to e-Magazine!</p>
</asp:Panel>
</body>
</html>

```

This WebForm contains just 7 fields:

- First name.
- Last name.
- Email address.
- Date of birth.
- The type of email format desired.
- A survey question about where the user heard of e-Magazine.

It also contains a Submit handler called Submit\_Clicked which creates the subscription and stores it. Once stored successfully a second panel is made visible and the first form hidden. This second panel merely states “Thank you for subscribing to e-Magazine!”

Note that for the sake of this demonstration it is not necessary to discuss or understand the Subscription class used in the Submit handler.

## 3.2 Adding Mad! Widgets validation

The WebForm in example 3.1 contains no validation except for the try/catch block in the Submit handler. It's now time to add Mad! Widgets validation to the page.

Add the following two lines to the top of the page:

```
<%@ Register TagPrefix="madwidgets" TagName="Validator"
    Src="/Controls/Validation/Validator.ascx" %>
```

```
<%@ Import Namespace="MadWidgets.Validation" %>
```

The Register tag tells the CLR where to find Mad! Widgets Validation Summary control while the Import tag tells the CLR where to find the validator classes that come with the package. Note that the path to the Validation Summary control can be any path of your choosing. You will have decided where to place this control (Validator.ascx) when you installed the package on your web server (see [Setting up the web.config file](#)). You may have also created a custom Validation Summary control (see *Creating a custom Validation Summary*).

Add the following line in the location that you would like the Validation Summary to appear.

```
<madwidgets:Validator runat="server" id="TheValidator" />
```

This is the Validation Summary control. The Validation Summary control that comes with the package doubles as the Validator control because it hierarchically extends the Validator control class. In some circumstances you may wish to split these. More information is provided later in this document about doing this.

Note that the Validation Summary is a control that displays error message in relation to the form fields that failed validation. It serves no other purpose. On the other hand, the Mad! Widgets Validator control is the control that actually performs the validation in the background. The Validator control is located in the package MadWidgets.Validation.

With the Validator control now present on the WebForm, you can start adding validation to individual fields.

Add a Page\_Load method to the script section (the `<script runat="server">` section at the top of the page) as in example 3.2.

Example 3.2. The `Page_Load` method of the ASPX page.

```
void Page_Load(object o, EventArgs ea)
{
    // Setup validation
    Validator v = Validator.Current;

    v.Add(new ValidatorEntryRequired(FirstName,
        "You have not supplied your first name.));

    v.Add(new ValidatorEntryRequired(LastName,
        "You have not supplied your last name.));

    v.Add(new ValidatorEntryEmail(Email, "{switch(value, [' => 'You
        have not supplied', default => concat('', value, ' is not')]]}
        a valid email address.));

    v.Add(new ValidatorEntryDate(DateOfBirth, "{switch(value, [' => 'You
        have not supplied', default => concat('', value, ' is not')]]}
        a valid date of birth.",
        ValidatorEntryDate.Format.MMDDYYYY));

    v.Add(new ValidatorEntryRequired(EmailType, "You have not selected
        an email format type.));

    v.Add(new ValidatorEntryRequired(WhereHeardOfUs, "Please tell us
        where you heard of e-Magazine.));

    v.Add(new ValidatorEntryOr("Please tell us where you heard of
        e-Magazine by entering something in the 'Other' field.",
        new ValidatorEntryTextComparer(WhereHeardOfUs, "{OTHER}",
            ComparisonMethod.NotEqualTo),
        new ValidatorEntryRequired(WhereHeardOfUsOther)));
}
```

If you do not have `AutoEventWireup="true"` in your `@Page` tag, you will need to add it as follows:

```
<%@ Page Language="C#" AutoEventWireup="true" %>
```

This tells the CLR to call the local script's `Page_Load` method upon load.



**Note:** if you are placing the validation code in your Code Behind class, *you should not* add this attribute to the tag unless you have a `Page_Load` method in the page script. Similarly, if `AutoEventWireup` is true by default on your site (as determined by your `web.config` file), you will need to explicitly set it to false in the `@Page` tag if you do not have a `Page_Load` method on the page script. This prevents the Code Behind's `Page_Load` from getting called twice.

Take another look at the `Page_Load` that we've just added to the script section of the page. You will see seven validators. The last validator, `ValidatorEntryOr`, is a more complex one and we'll discuss that shortly. Firstly, let's go through each validator to find out what it's doing.

1. The first validator is a `ValidatorEntryRequired` validator.

```
v.Add(new ValidatorEntryRequired(FirstName,
    "You have not supplied your first name.));
```

This will produce the message “You have not supplied your first name” if nothing is entered in the `FirstName` field.

2. The second validator is also a `ValidatorEntryRequired` validator.

```
v.Add(new ValidatorEntryRequired(LastName,
    "You have not supplied your last name.));
```

This produces the message “You have not supplied your last name” if nothing is entered in the `LastName` field.

3. The third validator is a `ValidatorEntryEmail` validator.

```
v.Add(new ValidatorEntryEmail(Email, "{switch(value, ['' => 'You
    have not supplied', default => concat(''', value, ''
    is not')])} a valid email address.));
```

This produces one of two messages. If the user has not entered anything in the `Email` field, the message will be “You have not supplied a valid email address.” If the user has entered something in the field but it is not a valid email address (e.g. `support@domain`), the message will be “‘support@domain’ is not a valid email address.”

As you can see, the message contains the inline message function *switch*. You will learn more about inline message functions later in this document (see [Inline message blocks](#)).

4. The fourth validator is a `ValidatorEntryDate` validator.

```
v.Add(new ValidatorEntryDate(DateOfBirth, "{switch(value,
    ['' => 'You have not supplied', default => concat(''',
    value, '' is not')])} a valid date of birth.",
    ValidatorEntryDate.Format.MMDDYYYY));
```

Like the third validator, this validator produces one of two messages. If the user has not entered anything in the `DateOfBirth` field, the message will be “You have not supplied a valid date of birth.” If the user has entered something in the field but it is not a valid date (e.g. `13/13/1968`), the message will be “‘13/13/1968’ is not a valid date of birth.”

5. The fifth validator is another a `ValidatorEntryRequired` validator.

```
v.Add(new ValidatorEntryRequired(EmailType, "You have not
    selected an email format type.));
```

This produces the message “You have not selected an email format” if nothing is selected from the EmailType drop-down list.

6. The sixth validator is yet another a ValidatorEntryRequired validator.

```
v.Add(new ValidatorEntryRequired(WhereHeardOfUs, "Please tell us where you heard of e-Magazine."));
```

This produces the message “Please tell us where you heard of e-Magazine” if nothing is selected from the WhereHeardOfUs drop-down list.

7. The final validator is a ValidatorEntryOr validator.

```
v.Add(new ValidatorEntryOr("Please tell us where you heard of e-Magazine by entering something in the 'Other' field.", new ValidatorEntryTextComparer(WhereHeardOfUs, "{OTHER}", ComparisonMethod.NotEqualTo), new ValidatorEntryRequired(WhereHeardOfUsOther)));
```

This more complex validator is used in this instance to ensure that if the user has selected ‘Other’ from the drop-down list, that they have entered something in the ‘Other’ text field as well. If the user does not select ‘Other’ from the drop-down list, then this validator will not produce an error message, regardless whether something is entered in the ‘Other’ text field or not. If the user selected ‘Other’ from the drop-down list and they did not enter something in the ‘Other’ text field, then this validator would produce the message “Please tell us where you heard of e-Magazine by entering something in the 'Other' field.”

The logic of this validator is equivalent to the C# statement:

```
WhereHeardOfUs.SelectedValue != "OTHER" ||  
WhereHeardOfUsOther.Text != ""
```

Or in VB.NET:

```
WhereHeardOfUs.SelectedValue <> "OTHER" Or  
WhereHeardOfUsOther.Text <> ""
```

As you can imagine the ValidatorEntryOr and the ValidatorEntryAnd validators are very powerful. They can be used to create, if desired, quite complex hierarchical validation logic. They can be nested and be used with any other type of validator class.

### **3.3 Validation styles and highlight elements**

If you have turned on ticks in the Mad! Widgets Validation Package on your website, you will see crosses and ticks appear in your form fields as they progress from an invalid to a valid state. This works by manipulating the CSS styles on the form fields directly behind the scenes as the user types or clicks in the form and changes the validation state of the form accordingly.

Ticks and crosses are not the only type of visual indication that a form field has passed or not passed validation. In addition to ticks and crosses, you can manipulate the style of the surrounding HTML elements in the form. Every form field can have a nominated 'highlight element.' A highlight element is an HTML element that the form field is associated with and that can be used to present a visual indication of the validation state of the field. Highlight elements are assigned to the form field when the page first loads and cannot be reassigned to the form field thereafter.

You can assign the highlight element to the form field in a number of ways. The most likely way to assign it is through the web.config file. Most sites on the internet use a common look and feel throughout the site, including all the forms on the site. Invariably the HTML structure of the each of the forms on the site is identical throughout. You can take advantage of this by adding an entry in the web.config that is used to locate the highlight element for every form field on your site without having to add information on a page level. The entry in the web.config file is 'validatorDefaultHighlightElement.'

In [Setting up the web.config file](#), you were shown how to set up this entry as well as all other entries associated with the Mad! Widgets Validation Package. This entry takes a 'highlight element signature.' The highlight element signature is a conditional statement that the client-side code uses to locate a form field's highlight element. The signature comes in the form:

```
<tagName[property=value]>
```

This condition states:

```
Starting from the form field element, navigate up the HTML hierarchy until an element with the tag name tagName is found that possesses a property called property whose value is equal to value.
```

When the client-side code finds the element that matches the condition, it is registered as the 'highlight element' for the form field.

The highlight element signature comes in a simpler form as follows:

```
<tagName>
```

In this case the condition is:

```
Starting from the form field element, navigate up the HTML hierarchy until an element with the tag name tagName is found.
```

The simpler form makes no check on the properties of the element that is found.



**Note:** more than one form field can have the same highlight element. For example, if the highlight element on our example form was <tr>, then the form fields WhereHeardOfUs and WhereHeardOfUsOther would share the same highlight element.



**Note:** the property noted in the highlight element signature refers to a property on the HTML DOM element – it is not an HTML attribute. For example, if you wanted to match a row with the CSS class equal to ‘row2,’ then property would need to be ‘className’ and not ‘class’ because ‘className’ is the DOM property for the HTML attribute ‘class.’ In other words the highlight element signature would look like this:

```
<tr[className=row2]>
```

So, what does the client-side code actually do with the highlight element?

When the page containing the form first loads on the browser and all highlight elements are found and recorded, the client-side code also remembers the CSS class names associated with the elements. If, upon page load, any of the validators are in an invalid state on the browser, the client-side code will swap the CSS class on the associated highlight element with CSS class registered with the Validator control as the ‘InitialHighlightClass’ or if this was not set or is null, with the CSS class registered with the Validator control as the HighlightClass. If any form field is not in an invalid state (i.e. it is valid), then the original CSS class on the highlight element remains untouched.

Once the page has loaded, the InitialHighlightClass is never used again. Thereafter, if a form field is invalid, the highlight element will be assigned the HighlightClass.

In both cases, once the validation passes for a given field or set of fields, the CSS class originally found on the highlight element is reassigned to the element.

It’s now time to add a highlight element signature to our form and to prepare the CSS classes.

Assuming you have set up the web.config file as per the examples provided in the section [Setting up the web.config file](#), add the class validatorError to the CSS style block on the page as follows:

```
<style type="text/css">
  body, td, input, select {font-family: Verdana,Helvetica,Arial;
                          font-size: 80%;}
  .row1 {background-color: #FFA189; padding: 3px;}
  .row2 {background-color: #FFD1C9; padding: 3px;}
  .validatorError {background-color: red; padding: 3px;}
</style>
```

Next modify the control tag as follows:

```
<madwidgets:Validator runat="server" id="TheValidator"
  defaulthighlightelement="&lt;tr&gt;" initialhighlightclass="" />
```

In this example, you will notice two changes, the addition of the attribute 'defaulthighlightelement' and the addition of the attribute 'initialhighlightclass.' The former of these is the highlight element signature and it states merely '<tr>.' Note that it is escaped. You should escape this value in order to maintain well-formed HTML. The second attribute, initialhighlightclass, blanks out the validatorInitialHighlightClass property set up in the web.config file, i.e.:

```
<add key="validatorInitialHighlightClass" value="validatorInitial" />
```

In our example, we do not want to alter the style on the highlight elements when the page loads, only when the page is being edited. Blanking out the Validator control property 'InitialHighlightClass' at the page level prevents the client-side code from swapping the styles on the highlight elements during page load.

Example 3.3 now shows the final WebForm complete with all validation logic and style information.

*Example 3.3. e-Magazine subscription complete with validation logic.*

```
<%@ Page Language="C#" AutoEventWireup="true" %>
<%@ Register TagPrefix="madwidgets" TagName="Validator"
    Src="/Controls/Validation/Validator.ascx" %>
<%@ Import Namespace="MadWidgets.Validation" %>
<%@ Import namespace="MadWidgets.Validation.Samples" %>
<%@ Import namespace="System.Globalization" %>

<script runat="server">

void Page_Load(object o, EventArgs ea)
{
    // Setup validation
    Validator v = Validator.Current;

    v.Add(new ValidatorEntryRequired(FirstName,
        "You have not supplied your first name.));

    v.Add(new ValidatorEntryRequired(LastName,
        "You have not supplied your last name.));

    v.Add(new ValidatorEntryEmail(Email, "{switch(value, ['' => 'You
        have not supplied', default => concat(''', value, '' is not')]]}
        a valid email address.));

    v.Add(new ValidatorEntryDate(DateOfBirth, "{switch(value, ['' => 'You
        have not supplied', default => concat(''', value, '' is not')]]}
        a valid date for the date of birth.",
        ValidatorEntryDate.Format.MMDDYYYY));

    v.Add(new ValidatorEntryRequired(EmailType, "You have not selected
        an email format type.));

    v.Add(new ValidatorEntryRequired(WhereHeardOfUs, "Please tell us
        where you heard of e-Magazine.));

    v.Add(new ValidatorEntryOr("Please tell us where you heard of
        e-Magazine by entering something in the 'Other' field.",
        new ValidatorEntryTextComparer(WhereHeardOfUs, "{OTHER}",
            ComparisonMethod.NotEqualTo),
        new ValidatorEntryRequired(WhereHeardOfUsOther));
```

```

}

void Submit_Clicked(object o, EventArgs ea)
{
    if (Page.IsValid)
    {
        try
        {
            // Subscribe the user to the magazine
            Subscription s = new Subscription();
            s.FirstName = FirstName.Text;
            s.LastName = LastName.Text;
            s.Email = Email.Text;
            s.DateOfBirth = DateTime.Parse(DateOfBirth.Text,
                CultureInfo.CurrentCulture);
            s.Type = EmailType.Items[EmailType.SelectedIndex].Value;
            s.WhereHeardOfUs = WhereHeardOfUs.Items[
                WhereHeardOfUs.SelectedIndex].Value;
            s.WhereHeardOfUsOther = WhereHeardOfUsOther.Text;
            s.Save();

            // Swap panels upon success
            Page1.Visible = false;
            Page2.Visible = true;
        }
        catch (Exception e)
        {
            Validator.Current.AddError("An error occurred in the form");
        }
    }
}

</script>

<html>
<head>
<title>Subscribe to e-Magazine</title>
</head>
<style type="text/css">
    body, td, input, select {font-family: Verdana,Helvetica,Arial;
        font-size: 80%;}
    .row1 {background-color: #FFA189; padding: 3px;}
    .row2 {background-color: #FFD1C9; padding: 3px;}
    .validatorError {background-color: red; padding: 3px;}
</style>
<body>
<h1>Subscribe to e-Magazine</h1>

<asp:Panel id="Page1" runat="server" visible="true">
<p>To receive the monthly e-Magazine FREE,
please fill in the form below.</p>

<p>Note that all fields are <b>required</b>.

<madwidgets:Validator runat="server" id="TheValidator"
    defaulthighlightelement="&lt;tr&gt;" initialhighlightclass="" />

<form name="TheForm" id="TheForm" method="POST" runat="server">
<table cellpadding="0" cellspacing="1" border="0">
<tr class="row1">
<td class="label" valign="baseline">First name</td>
<td class="field"><asp:TextBox id="FirstName"
    width="180" runat="server" /></td>
</tr>

```

```

<tr class="row2">
  <td class="label" valign="baseline">Last name</td>
  <td class="field"><asp:TextBox id="LastName"
    width="180" runat="server" /></td>
</tr>
<tr class="row1">
  <td class="label" valign="baseline">Email address</td>
  <td class="field"><asp:TextBox id="Email"
    width="180" runat="server" /></td>
</tr>
<tr class="row2">
  <td class="label" valign="baseline">Date of birth</td>
  <td class="field"><asp:TextBox id="DateOfBirth"
    width="180" runat="server" /></td>
</tr>
<tr class="row1">
  <td class="label" valign="baseline">Type of email</td>
  <td class="field">
    <asp:DropDownList id="EmailType" runat="server">
      <asp:ListItem value="">-- Please select a
        type --</asp:ListItem>
      <asp:ListItem value="TEXT">Textual
        Format</asp:ListItem>
      <asp:ListItem value="HTML">HTML
        Format</asp:ListItem>
      <asp:ListItem value="RTF">Rich Text
        Format</asp:ListItem>
    </asp:DropDownList>
  </td>
</tr>
<tr class="row2">
  <td class="label" valign="baseline">Where did you hear
    about e-Magazine?</td>
  <td class="field">
    <asp:DropDownList id="WhereHeardOfUs" runat="server">
      <asp:ListItem value="">-- Please select where you
        heard of e-Magazine --</asp:ListItem>
      <asp:ListItem value="FRIEND">Through a
        friend</asp:ListItem>
      <asp:ListItem value="ENGINE">Through a search
        engine</asp:ListItem>
      <asp:ListItem value="LINK">By clicking a
        hyperlink</asp:ListItem>
      <asp:ListItem value="PAPER">In a paper
        magazine</asp:ListItem>
      <asp:ListItem value="OTHER">Other (see
        below)</asp:ListItem>
    </asp:DropDownList><br />
    <asp:TextBox id="WhereHeardOfUsOther"
      width="180" runat="server" />
  </td>
</tr>
<tr>
  <td colspan="2" align="right"><br /><asp:Button
    id="Submit" runat="server"
    onclick="Submit_Clicked"
    text=" Submit " /></td>
</tr>
</table>
</form>
</asp:Panel>

<asp:Panel id="Page2" runat="server" visible="false">
  <p>Thank you for subscribing to e-Magazine!</p>

```

```

        </asp:Panel>
    </body>
</html>

```

### 3.4 Placing validation in your Code Behind class

In ASP.NET you are not restricted to referencing your form fields in the ASPX page itself, you can also reference the form field controls through a code behind class. Some people prefer to add validation code in the code behind class because it keeps the backend code (C#, VB.NET, etc.) separate from the display code (XML, HTML, etc.).

With the Mad! Widgets ASP.NET Validation Package, this is very easy to do and is little different from adding validation in the ASPX page itself. In the following example, we've created a copy of the e-Magazine subscription form, placing the Page\_Load and Submit\_Clicked methods in the code behind.

*Example 3.4. The code behind class GettingStarted\_CB. This can be found in the file GettingStarted\_CB.aspx.cs.*

```

using System;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Globalization;
using MadWidgets.Validation;

namespace MadWidgets.Validation.Samples
{
    public class GettingStarted_CB : Page
    {
        protected TextBox    FirstName;
        protected TextBox    LastName;
        protected TextBox    Email;
        protected TextBox    DateOfBirth;
        protected DropDownList EmailType;
        protected DropDownList WhereHeardOfUs;
        protected TextBox    WhereHeardOfUsOther;
        protected Panel      Page1;
        protected Panel      Page2;

        private void Page_Load(object o, EventArgs ea)
        {
            // Setup validation
            Validator v = Validator.Current;

            v.Add(new ValidatorEntryRequired(FirstName,
                "You have not supplied your first name.));

            v.Add(new ValidatorEntryRequired(LastName,
                "You have not supplied your last name.));

            v.Add(new ValidatorEntryEmail(Email, "{switch(value, ['' => 'You
                have not supplied', default => concat(''', value, ''
                is not')]]} a valid email address.));

            v.Add(new ValidatorEntryDate(DateOfBirth, "{switch(value, ['' =>
                'You have not supplied', default => concat(''', value, ''
                is not')]]} a valid date for the date of birth.",

```

```

        ValidatorEntryDate.Format.MMDDYYYY));

v.Add(new ValidatorEntryRequired(EmailType, "You have not selected
    an email format type.));

v.Add(new ValidatorEntryRequired(WhereHeardOfUs, "Please tell us
    where you heard of e-Magazine.));

v.Add(new ValidatorEntryOr("Please tell us where you heard of
    e-Magazine by entering something in the 'Other' field.",
    new ValidatorEntryTextComparer(WhereHeardOfUs, "{OTHER}",
        ComparisonMethod.NotEqualTo),
    new ValidatorEntryRequired(WhereHeardOfUsOther)));
}

protected void Submit_Clicked(object o, EventArgs ea)
{
    if (Page.IsValid)
    {
        try
        {
            // Subscribe the user to the magazine
            Subscription s = new Subscription();
            s.FirstName = FirstName.Text;
            s.LastName = LastName.Text;
            s.Email = Email.Text;
            s.DateOfBirth = DateTime.Parse(DateOfBirth.Text,
                CultureInfo.CurrentCulture);
            s.Type = EmailType.Items[EmailType.SelectedIndex].Value;
            s.WhereHeardOfUs = WhereHeardOfUs.Items[
                WhereHeardOfUs.SelectedIndex].Value;
            s.WhereHeardOfUsOther = WhereHeardOfUsOther.Text;
            s.Save();

            // Swap panels upon success
            Page1.Visible = false;
            Page2.Visible = true;
        }
        catch (FormatException e)
        {
            Response.Write("The date of birth is in an invalid format");
        }
        catch (Exception e)
        {
            Response.Write("An error occurred in the form");
        }
    }
}

#region Web Form Designer generated code
override protected void OnInit(EventArgs e)
{
    //
    // CODEGEN: This call is required by the ASP.NET Web
    // Form Designer.
    //
    InitializeComponent();
    base.OnInit(e);
}

/// <summary>
/// Required method for Designer support - do not modify

```

```

    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
        this.Load += new System.EventHandler(this.Page_Load);
    }
}
#endregion
}
}

```

*Example 3.5. The ASPX page. This can be found in the file GettingStarted\_CB.aspx.*

```

<%@ Page Language="c#" AutoEventWireup="false"
CodeBehind="GettingStarted_CB.aspx.cs"
Inherits="MadWidgets.Validation.Samples.GettingStarted_CB" %>
<%@ Register TagPrefix="madwidgets" TagName="Validator"
Src="/Controls/Validation/Validator.ascx" %>

<html>
  <head>
    <title>Subscribe to e-Magazine</title>
  </head>
  <style type="text/css">
    body, td, input, select {font-family: Verdana,Helvetica,Arial;
                             font-size: 80%;}
    .row1 {background-color: #FFA189; padding: 3px;}
    .row2 {background-color: #FFD1C9; padding: 3px;}
    .validatorError {background-color: red; padding: 3px;}
  </style>
  <body>
    <h1>Subscribe to e-Magazine</h1>

    <asp:Panel id="Page1" runat="server" visible="true">
      <p>To receive the monthly e-Magazine FREE,
      please fill in the form below.</p>

      <p>Note that all fields are <b>required</b>.</p>

      <madwidgets:Validator runat="server" id="TheValidator"
        defaulthighlightelement="&lt;tr&gt;" initialhighlightclass="" />

      <form name="TheForm" id="TheForm" method="POST" runat="server">
        <table cellpadding="0" cellspacing="1" border="0">
          <tr class="row1">
            <td class="label" valign="baseline">First name</td>
            <td class="field"><asp:TextBox id="FirstName"
              width="180" runat="server" /></td>
          </tr>
          <tr class="row2">
            <td class="label" valign="baseline">Last name</td>
            <td class="field"><asp:TextBox id="LastName"
              width="180" runat="server" /></td>
          </tr>
          <tr class="row1">
            <td class="label" valign="baseline">Email address</td>
            <td class="field"><asp:TextBox id="Email"
              width="180" runat="server" /></td>
          </tr>
          <tr class="row2">
            <td class="label" valign="baseline">Date of birth</td>
            <td class="field"><asp:TextBox id="DateOfBirth"
              width="180" runat="server" /></td>
          </tr>
        </table>
      </form>
    </asp:Panel>
  </body>
</html>

```

```

</tr>
<tr class="row1">
  <td class="label" valign="baseline">Type of email</td>
  <td class="field">
    <asp:DropDownList id="EmailType" runat="server">
      <asp:ListItem value="">-- Please select a
        type --</asp:ListItem>
      <asp:ListItem value="TEXT">Textual
        Format</asp:ListItem>
      <asp:ListItem value="HTML">HTML
        Format</asp:ListItem>
      <asp:ListItem value="RTF">Rich Text
        Format</asp:ListItem>
    </asp:DropDownList>
  </td>
</tr>
<tr class="row2">
  <td class="label" valign="baseline">Where did you hear
    about e-Magazine?</td>
  <td class="field">
    <asp:DropDownList id="WhereHeardOfUs" runat="server">
      <asp:ListItem value="">-- Please select where you
        heard of e-Magazine --</asp:ListItem>
      <asp:ListItem value="FRIEND">Through a
        friend</asp:ListItem>
      <asp:ListItem value="ENGINE">Through a search
        engine</asp:ListItem>
      <asp:ListItem value="LINK">By clicking a
        hyperlink</asp:ListItem>
      <asp:ListItem value="PAPER">In a paper
        magazine</asp:ListItem>
      <asp:ListItem value="OTHER">Other (see
        below)</asp:ListItem>
    </asp:DropDownList><br />
    <asp:TextBox id="WhereHeardOfUsOther"
      width="180" runat="server" />
  </td>
</tr>
<tr>
  <td colspan="2" align="right"><br /><asp:Button
    id="Submit" runat="server"
    onclick="Submit_Clicked"
    text=" Submit " /></td>
</tr>
</table>
</form>
</asp:Panel>

<asp:Panel id="Page2" runat="server" visible="false">
  <p>Thank you for subscribing to e-Magazine!</p>
</asp:Panel>
</body>
</html>

```

As you can see, all that we've done here is removed the `<script runat="server">` from the ASPX page and placed it in the code behind class. We've also removed the Import statements and placed them in the code behind file.

This example demonstrates how easy it is to support modern development patterns such as Model-View-Controller with the validation package.

### 3.5 Using validation on non-server form fields

The Mad! Widgets ASP.NET Validation Package supports both server forms and non-server forms. Server forms are forms containing fields that are instantiated as controls on the server. Non-server forms are the traditional kind of HTML forms that, as far as the server is concerned, is simply raw HTML that gets sent to the browser.

The Mad! Widgets ASP.NET Validation Package can support both types of forms because, behind the scenes, the two are represented in very much the same manner. There are a few differences, but these mainly centre around inline message blocks. And even then the differences are subtle.

Setting up validation on non-server form fields is almost identical to setting up validation on server form field controls. The primary difference is that instead of passing Control objects to the validation constructors, you instead pass the names of the form fields to the constructors.

In example 3.6 we've taken the Page\_Load method from the previous examples in this chapter and converted it to use non-server form field names instead.

*Example 3.6. Setting up validation for non-server form fields.*

```
private void Page_Load(object o, EventArgs ea)
{
    // Setup validation
    Validator v = Validator.Current;

    v.Add(new ValidatorEntryRequired("FirstName",
        "You have not supplied your first name.));

    v.Add(new ValidatorEntryRequired("LastName",
        "You have not supplied your last name.));

    v.Add(new ValidatorEntryEmail("Email", "{switch(value, [' => 'You
        have not supplied', default => concat('', value, ''
        is not')]]} a valid email address.));

    v.Add(new ValidatorEntryDate("DateOfBirth", "{switch(value, [' =>
        'You have not supplied', default => concat('', value, ''
        is not')]]} a valid date for the date of birth.",
        ValidatorEntryDate.Format.MMDDYYYY));

    v.Add(new ValidatorEntryRequired("EmailType", "You have not selected
        an email format type.));

    v.Add(new ValidatorEntryRequired("WhereHeardOfUs", "Please tell us
        where you heard of e-Magazine.));

    v.Add(new ValidatorEntryOr("Please tell us where you heard of
        e-Magazine by entering something in the 'Other' field.",
        new ValidatorEntryTextComparer("WhereHeardOfUs", "{OTHER}",
            ComparisonMethod.NotEqualTo),
        new ValidatorEntryRequired("WhereHeardOfUsOther")));

    // Force validation since there are no server forms
    // in the control tree
    if (Request.Form["Submit"] != null)
    {
```

```

        Page.Validate();
        Submit_Clicked(o, ea);
    }
}

```

*Example 3.7 The HTML form associated with example 3.6.*

```

<form name="TheForm" id="TheForm" method="POST">
  <table cellpadding="0" cellspacing="1" border="0">
    <tr class="row1">
      <td class="label" valign="baseline">First name</td>
      <td class="field"><input type="text" name="FirstName"
        width="180" /></td>
    </tr>
    <tr class="row2">
      <td class="label" valign="baseline">Last name</td>
      <td class="field"><input type="text" name="LastName"
        width="180" /></td>
    </tr>
    <tr class="row1">
      <td class="label" valign="baseline">Email address</td>
      <td class="field"><input type="text" name="Email"
        width="180" /></td>
    </tr>
    <tr class="row2">
      <td class="label" valign="baseline">Date of birth</td>
      <td class="field"><input type="text" name="DateOfBirth"
        width="180" /></td>
    </tr>
    <tr class="row1">
      <td class="label" valign="baseline">Type of email</td>
      <td class="field">
        <select name="EmailType">
          <option value="">-- Please select a
            type --</option>
          <option value="TEXT">Textual
            Format</option>
          <option value="HTML">HTML
            Format</option>
          <option value="RTF">Rich Text
            Format</option>
        </select>
      </td>
    </tr>
    <tr class="row2">
      <td class="label" valign="baseline">Where did you hear
        about e-Magazine?</td>
      <td class="field">
        <select name="WhereHeardOfUs">
          <option value="">-- Please select where you
            heard of e-Magazine --</option>
          <option value="FRIEND">Through a
            friend</option>
          <option value="ENGINE">Through a search
            engine</option>
          <option value="LINK">By clicking a
            hyperlink</option>
          <option value="PAPER">In a paper
            magazine</option>
          <option value="OTHER">Other (see
            below)</option>
        </select><br />
        <input type="text" name="WhereHeardOfUsOther"

```

```

        width="180" />
    </td>
</tr>
<tr>
    <td colspan="2" align="right"><br /><input
        name="Submit" type="submit"
        text=" Submit " /></td>
</tr>
</table>
</form>

```

There are a few things that you may have noticed between example 3.6 and 3.7. Firstly, the Control objects in the constructors of the ValidatorEntry classes in example 3.6 have been replaced with each of the names from the respective form field from example 3.7. Secondly, due to the lack of any server form control on the page, the CLR will no longer try to validate the page on the server, so we have to do this explicitly by calling *Page.Validate()*. Finally, you will notice that the server controls embedded in the HTML in the previous examples have been replaced with their traditional HTML counterparts, specifically, INPUT and SELECT tags. Note that the name takes over where the id was used on the server control.

## 4 Validation groups

ASP.NET WebForms (.ASPX pages that contain form fields) can contain only a single HTML form. This limitation in the development of HTML form rich pages is by design and has caused a good deal of grief within the ASP.NET development community. To exacerbate the situation, the ASP.NET built-in validation mechanism provides no means of allowing two or more sub forms (logical forms within the primary outer form) to be validated (versions 1.0 and 1.1 of the framework).

The Mad! Widgets ASP.NET Validation Package overcomes these limitations with the introduction of validation groups and support for multiple forms when using non-server form fields.

Validation groups are sets of logically grouped form fields that can be validated separately from one another on both the client and the server. For example, you might have a page that has a login box on the left, a search form in the header and a registration form within the main content area. When using server form fields, all of these sub forms are physically placed within a large outer HTML form. Each sub form contains a means of submitting the form, e.g. a button or submit image. When the form is submitted, all information contained in each of the form fields is sent back to the server, not just the data from the sub form that was submitted.

### 4.1 Sub forms and form controls

Most developers place these logical groups of HTML (whether they contain form fields or not) into separate and distinct Web Controls. This is good practise as it keeps your code compartmentalised. Let's assume that the login sub form mentioned above is contained in a Web Control called Login.ascx as per example 4.1.

*Example 4.1. The login sub form, Login.ascx.*

```
<%@ Control Language="c#" AutoEventWireup="false"
Codebehind="Login.ascx.cs"
Inherits="MadWidgets.Validation.Samples.Login" %>

<div style="padding-left: 5px;">
  Login<br />
  <input type="text" id="Username" size="20" runat="server" />
  <input type="password" id="Password" size="20" runat="server" />
```

```

        <input type="submit" value="Login" runat="server"
            onserverclick="Login_Clicked" />
    </div>

```

*Example 4.2. The Code Behind for the login sub form, Login.ascx.cs.*

```

using System;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Text.RegularExpressions;
using MadWidgets.Validation;

namespace MadWidgets.Validation.Samples
{
    public abstract class Login : UserControl
    {
        protected HtmlInputText Username;
        protected HtmlInputText Password;

        private void Page_Load(object sender, EventArgs e)
        {
            Validator v = Validator.Current;
            v.Add(new ValidatorEntryRegex(Username, "Please enter a
                valid username", "^[a-z][a-z0-9_]*$",
                RegexOptions.IgnoreCase));

            v.Add(new ValidatorEntryRegex(Password, "The password is
                invalid", "^[a-z][a-z0-9_]*$", RegexOptions.IgnoreCase));
        }

        protected void Login_Clicked(object sender, EventArgs e)
        {
            if (Page.IsValid)
            {
                // Log the user in
            }
        }

        #region Web Form Designer generated code
            // Auto generated code
        #endregion
    }
}

```

The page containing the login form, the search form and the registration form might look like example 4.3.

*Example 4.3. The page containing the three sub forms.*

```

<%@ Page language="c#" AutoEventWireup="false" %>
<%@ Register TagPrefix="madwidgets" TagName="Search"
    Src="Search.ascx" %>
<%@ Register TagPrefix="madwidgets" TagName="Login"
    Src="Login.ascx" %>
<%@ Register TagPrefix="madwidgets" TagName="Subscribe"
    Src="Subscribe.ascx" %>
<%@ Register TagPrefix="madwidgets" TagName="Validator"
    Src="/Controls/Validation/Validator.ascx" %>

```

```

<html>
  <head>
    <title>Validation Groups</title>
  </head>
  <style type="text/css">
    body, td, input, select {font-family: Verdana,Helvetica,Arial;
      font-size: 80%;}
    .row1 {background-color: #FFA189; padding: 3px;}
    .row2 {background-color: #FFD1C9; padding: 3px;}
    .validatorError {background-color: red; padding: 3px;}
  </style>
  <body style="margin: 0px;">
    <form runat="server">

      <!-- This is the header -->
      <div style="background-color: #ECE8E3; height: 80px;
        padding: 10px;">
        <h1>Validation Groups</h1>
        <madwidgets:Search runat="server" id="SearchForm"
          validationGroup="Search" />
      </div>

      <!-- This is the left column -->
      <div style="width: 130px; height: 400px; float: left;
        background-color: #FCF8F3;">
        <madwidgets:Login runat="server" id="LoginForm"
          validationGroup="Login" />
      </div>

      <!-- This is the content area -->
      <div style="background-color: #DEFEDE; height: 400px;"
        align="center">
        <br />
        <madwidgets:Validator runat="server" id="TheValidator"
          defaulthighlightelement="&lt;tr&gt;"
          initialhighlightclass="" />
        <h3>Please fill in the form below</h3>
        <madwidgets:Subscribe runat="server" id="SubscribeForm"
          validationGroup="Subscribe" />
      </div>
    </form>
  </body>
</html>

```

Example 4.3. shows how simple it is to assign the group to the sub form. Each of the Web Controls in the example (SearchForm, LoginForm and SubscribeForm) contain an attribute *validationGroup*. The presence of this attribute on the control tells the Mad! Widgets Validator control (TheValidator in this example) that all form elements contained within the control as well as all child controls adopt that group.

So, the validation group for the login control is ‘Login,’ the validation group for the search control is ‘Search,’ and the validation group for the registration form is ‘Subscribe.’

Whichever one of these sub forms is submitted is the one that is validated. No other sub form is validated.

## 4.2 Validation groups on a single control

Some users may have a need to split a form into validation groups while keeping the individual form fields together. This is easily done with the Mad! Widgets ASP.NET Validation Package.

In the previous section, we learned that existence of the *validationGroup* attribute on a control that is hierarchically higher than a form element forces the form element to belong to that group. The *validationGroup* attribute can also, however, be placed on the form element itself. By placing this attribute on the form element directly, you have much greater control over which element belongs to which group. Consequently, it is possible to have more than one validation group contained within a single page or Web Control. Example 4.4 shows how this might be done.

*Example 4.4. Multiple validation groups within a single control.*

```
<form id="TheForm" method="POST" runat="server">
  <div class="formLabel">Group 1 Field</div>
  <div class="formInput"><input type="text" id="Group1Field1"
    size="35" runat="server" validationGroup="Group1" /></div>

  <div class="formLabel">Group 1 Field</div>
  <div class="formInput"><input type="text" id="Group1Field2"
    size="35" runat="server" validationGroup="Group1" /></div>

  <div class="formLabel">Group 2 Field</div>
  <div class="formInput"><input type="text" id="Group2Field1"
    size="35" runat="server" validationGroup="Group2" /></div>

  <div class="formLabel">Group 2 Field</div>
  <div class="formInput"><input type="text" id="Group2Field2"
    size="35" runat="server" validationGroup="Group2" /></div>

  <div class="formButtonBar">
    <input type="submit" id="Submit" value="Submit Group 1"
      class="formButton" runat="server"
      onserverclick="SubmitGroup1_Clicked"
      validationGroup="Group1" />

    <input type="submit" id="Submit" value="Submit Group 2"
      class="formButton" runat="server"
      onserverclick="SubmitGroup2_Clicked"
      validationGroup="Group2" />
  </div>
</form>
```

Note that the *validationGroup* attribute must be also placed on the submit buttons so that the Validator control knows which group to validate.

## 4.3 Using validation groups with non-server fields

Adding validation groups to non-server forms is possible, although a fraction more tedious than their server control counterparts. Due to the need for the Validator control on the server to know which field belongs to which group, we must explicitly add each

of the form fields to their associated groups respectively. Let's take the e-Magazine subscription form in example 3.7 with its accompanying Page\_Load method from example 3.6. Near the end of the Page\_Load method, we might add each of the form fields to the 'Subscribe' group as in example 4.5.

*Example 4.5. Adding non-server form fields to the 'Subscribe' validation group.*

```
private void Page_Load(object o, EventArgs ea)
{
    // Setup validation
    Validator v = Validator.Current;

    v.Add(new ValidatorEntryRequired("FirstName",
        "You have not supplied your first name.));

    v.Add(new ValidatorEntryRequired("LastName",
        "You have not supplied your last name.));

    v.Add(new ValidatorEntryEmail("Email", "{switch(value, ['' => 'You
        have not supplied', default => concat('', value, ''
        is not')]]} a valid email address.));

    v.Add(new ValidatorEntryDate("DateOfBirth", "{switch(value, ['' =>
        'You have not supplied', default => concat('', value, ''
        is not')]]} a valid date for the date of birth.",
        ValidatorEntryDate.Format.MMDDYYYY));

    v.Add(new ValidatorEntryRequired("EmailType", "You have not selected
        an email format type.));

    v.Add(new ValidatorEntryRequired("WhereHeardOfUs", "Please tell us
        where you heard of e-Magazine.));

    v.Add(new ValidatorEntryOr("Please tell us where you heard of
        e-Magazine by entering something in the 'Other' field.",
        new ValidatorEntryTextComparer("WhereHeardOfUs", "{OTHER}",
            ComparisonMethod.NotEqualTo),
        new ValidatorEntryRequired("WhereHeardOfUsOther"));

    // Register fields into validation group
    v.RegisterGroupForObject("FirstName", "Subscribe");
    v.RegisterGroupForObject("LastName", "Subscribe");
    v.RegisterGroupForObject("Email", "Subscribe");
    v.RegisterGroupForObject("DateOfBirth", "Subscribe");
    v.RegisterGroupForObject("EmailType", "Subscribe");
    v.RegisterGroupForObject("WhereHeardOfUs", "Subscribe");
    v.RegisterGroupForObject("WhereHeardOfUsOther", "Subscribe");

    // Force validation since there are no server forms
    // in the control tree
    if (Request.Form["Submit"] != null)
    {
        Page.Validate();
        Submit_Clicked(o, ea);
    }
}
```



**Note:** if you have more than one logical form on a page and you are using non-server controls, you must use validation groups, regardless whether you are containing them within a single outer HTML form or in multiple HTML forms.

## 5 Inline message blocks

In [Getting Started](#), you learned how to add Mad! Widgets validation to your WebForms. Some of the example validators introduced in that chapter used messages that contained embedded blocks of code. These blocks of code are *inline message blocks*. They are used to incorporate dynamic content into your error messages, allowing them to become more descriptive and helpful to the end user than their static counterparts.

Inline message blocks allow you to be more creative with your error messages. You can use the value entered in a text field, the value or set of values selected from a drop-down list, a radio button value, checkbox values and more as a source for the dynamic content that you use within the messages.

Say, for example, you have a date field on which there is an age restriction of 18 years old. If the date entered indicates that the user is 16 years old, you can produce a message saying, “The birth date you entered is 5 March 93, you are too young to register here.” Example 3 demonstrates how this can be done using the `ValidatorEntryDateRange` validator.

*Example 5.1. Inline message block using `ValidatorEntryDateRange`.*

```
v.Add(new ValidatorEntryDateRange(DateOfBirth, "The birth  
date you entered is {value}, you are too young to register  
here.", ValidatorEntryDate.Format.MMDDYYYY, DateTime.MinValue,  
DateTime.Now.AddYears(-18)));
```

This validator is tied to a form control called `DateOfBirth`. It states that if the date entered is less than today’s date minus 18 years, then all is OK. However, if the date entered is greater than today’s date minus 18 years, then produce the error message.

In our example, we have assumed that the user literally entered ‘5 May 93,’ but the user might just as well have entered 3/5/93. The problem with this type of date format is that it is ambiguous. Did the user mean 3 May 93 or 5 March 93? This is where the enumeration `ValidatorEntryDate.Format` comes in handy. In our example, we are using the enumeration constant `ValidatorEntryDate.Format.MMDDYYYY`. This constant tells the validator that if a date is entered in a format that might be considered ambiguous, that the first set of digits should be considered the month value, the second the day of month value and the third the year value. So in our example, the value 3/5/93 would be interpreted as 5 March 1993.

The next thing we might want to do to our dynamic value is ensure, regardless of the format that the date is entered in, that it is displayed in the message in a fixed format. Modify the message as per example 5.2:

*Example 5.2. Inline message block using ValidatorEntryDateRange and the functions cdate and formatDate.*

```
v.Add(new ValidatorEntryDateRange(DateOfBirth, "The birth date you entered is {formatDate(cdate(value, format), 'd MMMM yyyy')}, you are too young to register here.", ValidatorEntryDate.Format.MMDDYYYY, DateTime.MinValue, DateTime.Now.AddYears(-18)));
```

The inline message block now performs the following actions on the value. First, using the function, *cdate*, it converts the string value entered to a date type (*Date* on the browser, *DateTime* on the server). Next it passes the date value into *formatDate* which formats the value into the format 'd MMM yyyy,' which in our example would look like '3 Mar 1993' regardless whether the user entered the value as 5 March 93 or 3/5/93. Note that when the conversion of the value entered is performed using the function *cdate*, it takes as its second parameter *format*. This property is used on the validator classes *ValidatorEntryDate*, *ValidatorEntryDateRange* and *ValidatorEntryDateComparer* and refers to the expected input format determined by the *ValidatorEntryDate.Format* constant. If this parameter is not supplied, *cdate* will make its best guess of the input value.

The question springs to mind, what happens if the user doesn't enter anything or if the value entered is not a date? In this example, *cdate* will pass through the invalid value to *formatDate* and *formatDate* will pass it through to the message. So whether the value is blank or an invalid date, then it will still be embedded in the message. You can prevent this behaviour with the *switch* function.

Once again modify the message as per example 5.3.

*Example 5.3. Getting fancy with the switch function.*

```
v.Add(new ValidatorEntryDateRange(DateOfBirth, "{switch(value, ['' => 'Please enter your birth date', default => switch(isdate(value, format), [true => concat('The birth date you entered is ', formatDate(cdate(value, format), 'd MMMM yyyy'), ', you are too young to register here'), default => 'The date of birth entered is invalid'])]}", ValidatorEntryDate.Format.MMDDYYYY, DateTime.MinValue, DateTime.Now.AddYears(-18)));
```

Let's break this elaborate concoction down:

```
switch(value,
[
    '' => 'Please enter your birth date',
    default => switch(isdate(value, 'MMDDYYYY'),
    [
        true => concat('The birth date you entered is ',
            formatDate(cdate(value, 'MMDDYYYY'), 'd MMMM yyyy'),
            ', you are too young to register here'),
        default => 'The date of birth entered is invalid'
```

```
    1)
1)
```

The inline message function, *switch*, is similar to a C# *switch* or a VB.NET *Select Case*. It takes a value and tries to match it against a key within an array. If it can match the value with a key, it takes the value of the key (the right side) and returns it. If it can't find a match, it looks for the keyword *default*, if it exists, and returns whatever the value of *default* is.

In example 5, we have two switches, one nested inside the other. The first switch checks the value entered, if it is blank (''), it returns the string, 'Please enter your birth date.' In every other case, i.e. if the user entered something, the second switch is called. This switch first calls the function, *isdate*, which checks whether the value entered is a valid date. If it is, it returns the concatenation of three strings, 'The birth date you entered is' plus the result of *formatDate* plus ', you are too young to register here.' If the value entered is not a valid date, it returns the default value, which is a string stating, 'The date of birth entered is invalid.'

In example 5.3, we demonstrated the use of an *associative array* within the inline message block. This is an array that contains a set of elements that are associated by *key* and *value*. Within the inline message block you can create associative arrays using the following format:

```
[key1 => value1, key2 => value2, ... keyn => valuen]
```

Some HTML form elements also form associative arrays. These are: radio button lists, select lists and checkbox lists. A dropdown list (*select list* in HTML speak) for example might look like the following .NET control:

```
<asp:DropDownList id="WhereHeardOfUs" runat="server">
  <asp:ListItem value="">-- Please select where you heard of
    e-Magazine --</asp:ListItem>
  <asp:ListItem value="FRIEND">Through a friend</asp:ListItem>
  <asp:ListItem value="ENGINE">Through a search engine</asp:ListItem>
  <asp:ListItem value="LINK">By clicking a hyperlink</asp:ListItem>
  <asp:ListItem value="PAPER">In a paper magazine</asp:ListItem>
  <asp:ListItem value="OTHER">Other (see below)</asp:ListItem>
</asp:DropDownList>
```

The associations in this example are:

```
" " => -- Please select where you heard of e-Magazine --
"FRIEND" => Through a friend
"ENGINE" => Through a search engine
"LINK" => By clicking a hyperlink
"PAPER" => In a paper magazine
"OTHER" => Other (see below)
```

So this dropdown list is equivalent to the inline message block associative array:

```
[
  '' => '-- Please select where you heard of e-Magazine --',
  'FRIEND' => 'Through a friend',
  'ENGINE' => 'Through a search engine',
  'LINK' => 'By clicking a hyperlink',
```

```

    'PAPER' => 'In a paper magazine',
    'OTHER' => 'Other (see below)'
]

```

Fortunately you don't have to reproduce the dropdown list array for use in the inline message blocks explicitly. You can access the dropdown list array through the property *allvalues*.

*Example 5.4. Using allvalues.*

```
switch(value, allvalues)
```

Example 5.4 shows how you might extract the text (display) value from a dropdown list. For example, if the item 'By clicking a hyperlink' were selected from the example above, the inline message property, *value*, would be equal to 'LINK.' The switch function in example 5.4 would use this value as a key to search through the associative array, *allvalues*, and return the text value selected.

There's a simpler way to obtain the text value selected from a dropdown list though, and that is by using the property *textvalue*. So example 5.4 is equivalent to the property *textvalue*.

Table 5.1 describes each of the global properties available in inline message blocks. These can be used with all validators. In addition to these, each validator typically has a small set of properties that are pertinent to its purpose.

*Table 5.1. Inline message properties.*

Property	Type	Form Field Types and Values
value	string	All field types except multiple select lists (i.e. DropDownList set to multiple) and checkbox list controls (CheckBoxList).
value	associative array	Multiple select lists and CheckBoxLists. This contains the elements that are currently selected in the list extracted from the <i>allvalues</i> array.
allvalues	associative array	For list-type fields, the array holds all of the items in the list in their key => value pairs.  For non list-type fields, this property is always null.
selectedvalues	associative array	For list-type fields, the array holds the items that are selected in the list in their key => value pairs. These items are extracted from the <i>allvalues</i> array.  For non list-type fields, this property is always null.

Property	Type	Form Field Types and Values
textvalue	string	<p>Multiple select lists and CheckBoxLists – a concatenation of all keys in the equivalent associative array delimited by commas.</p> <p>All other list types, the text value of the currently selected item.</p> <p>All other types, the value entered in the field.</p>



**Note:** allvalues and selectedvalues cannot be used with non-server controls. Instead, when using non-server controls, you will need to place explicit associative arrays in your inline message blocks if you require this functionality.

## 6 Creating custom validators

In most situations the validators that are shipped with the Mad! Widgets ASP.NET Validation Package will suffice. However, there may be situations where you need to extend the validator functionality perhaps due to a proprietary requirement in your company or you may simply want to make your code more reusable by overriding one of the ValidatorEntry classes instead of re-writing the same validator sequence over and over again on your site. Fortunately ValidatorEntry subclasses are extremely simple to write and require very little knowledge of how validation actually occurs on the client or the server.

The ValidatorEntry class is the super class for all validators in the package. By itself this class can't be used to validate form data. Instead it defines a set of methods that must be overridden in its subclasses.

ValidatorEntry subclasses come in two categories, server-side only validators, i.e. those that are derived from ValidatorEntryServerOnly and can hence only perform validation on the server, and mixed client-side and server-side validators, i.e. those that are derived from ValidatorEntryClientServer and can perform the same validation on the server as on the browser. In addition to these, there are three sub categories of the ValidatorEntryClientServer class: single field validation, multiple field validation and comparison validation. Finally it is common to subclass the regular expression validator, the ValidatorEntryRegex class, as this provides a means of code reusability and prevents the same regular expression from being used in multiple places on the site.

Table 6.1 offers a guide as to which class you are best to inherit from in a given situation.

*Table 6.1. Validator base classes included with the package.*

<b>Class</b>	<b>Situation</b>
ValidatorEntryClientServer	You want to provide special validation that would not be better suited to one of the subclasses of this class such as the ValidatorEntrySingleObject or the ValidatorEntryComparer. Note that this class is not normally inherited directly. Doing so requires knowledge of the internals of the ValidatorEntry. If you believe you should be directly inheriting from this class, please contact <a href="mailto:support@madwidgets.com">support@madwidgets.com</a> for further assistance.

Class	Situation
ValidatorEntrySingleObject	You want to validate a single object, for example a text field, and you want to support both client-side and server-side validation. Note that standard HTML sets (e.g. radio button lists, checkbox lists, dropdown lists, etc.) are normally considered single objects for the sake of validation.
ValidatorEntryServerOnly	You want to validate the input supplied by a user but you don't want to provide support for the validation on the client.
ValidatorEntryComparer	You want to compare two objects (e.g. two text fields) and validate the comparison between them. You also want to provide the same type of validation on the client. Note that this class is derived from the ValidatorEntryClientServer class.
ValidatorEntryMultipleEntry	You want to validate two or more ValidatorEntry objects against a certain condition that can be determined by the valid state of the set of ValidatorEntry objects as a group. You also want to provide the equivalent client-side validation.
ValidatorEntryRegex	You want to validate a single object against a regular expression but you don't want to have to write the regular expression in more than one spot in your code or you don't want the page designer to be aware of what the regular expression is. You also want the validation to take place on both the client and the server.

## 6.1 Extending ValidatorEntrySingleObject

When you are writing a validator that supports client-side code, you should always try to make the client-side validation match as close as possible to that of the server-side validation. If it's not possible to match them perfectly, the server-side validation should provide the best and most secure validation of the two.

You should understand the resources you have on the client and appreciate their weaknesses. Often client-side (e.g. JavaScript) functions are considerably more tolerant than their .NET counterparts, an example of which is the JavaScript *parseInt()* function which will return a valid number if the input string starts with a number even if the string has characters that could not be normally parsed numerically. On the other hand, the .NET equivalent, *Int32.Parse()* would throw a *FormatException* in this instance.



**Note:** The Mad! Widgets ASP.NET Validation Package is shipped with a number of useful, but currently undocumented, JavaScript functions and extensions to the JavaScript object model. You can use these within your client-side validation code or any other part of your site. These are packaged in the file *ValidatorUtils.js* and include the following functions and object extensions: *Date.addYears(i)*, *Date.addMonths(i)*, *Date.addDays(i)*, *Date.addHours(i)*, *Date.addMinutes(i)*, *Date.addSeconds(i)*, *Date.addMonths(i)*,

*Date.addPeriod(*period*), Date.format(format, culture), Number.format(format, culture), String.trim(), String.replaceSection(index, len, str), String.insertSection(index, count, str), String.splitUnescaped(delim, count), String.pad(minLen, c, where), parseNumber(str, culture), parseDate(str, culture, pref), arrayLength(a).* The extensions are compatible with most browsers.

Example 6.1 shows a validator class called `ValidatorEntryLicenseKey` which is designed validates a text field into which the user enters the license key of a piece of software they have purchased.

The validator uses the following algorithm to check the validity of the license key being entered.

- The first 5 digits are added together and the modulus 10 of the result must match the 6th digit.
- The 7th to the 11th digits are added together and the modulus 10 of the result must match the 12th digit.
- The 6th and 12th digits are added together and the modulus 10 of the result must match the 13th digit.

The example includes both client-side and server-side validation, although we should note that it would not be terribly advisable to disclose a license algorithm on the browser for an astute hacker to decode.

*Example 6.1. The `ValidatorEntryLicenseKey` class.*

```
using System;
using System.Text.RegularExpressions;
using MadWidgets.Validation;

namespace MadWidgets.Validation.Samples
{
    public class ValidatorEntryLicenseKey : ValidatorEntrySingleObject
    {
        public ValidatorEntryLicenseKey(object o, string sMessage)
            : this(o, sMessage, true)
        {
        }

        public ValidatorEntryLicenseKey(object o, string sMessage,
            bool bRequired) : base(o, sMessage, bRequired)
        {
        }

        protected override bool RunValidation()
        {
            // If the object is blank and blanks are allowed,
            // return true, otherwise continue
            if (base.RunValidation())
                return true;

            // Strip non-digits from the string
            string sText = Regex.Replace(ValidationValue, "\\D", "");
            if (sText.Length != 13)
                return false;
        }
    }
}
```

```

// Get the checksum values from the key
int iWork = 0, i;
int iCheck1 = sText[5] - '0';
int iCheck2 = sText[11] - '0';
int iCheck3 = sText[12] - '0';

// Work out the checksum for the first set of characters
for (i = 0; i < 5; i++)
    iWork += (sText[i] - '0');

// If the checksum matches, continue
if (iWork % 10 != iCheck1)
    return false;

// Work out the checksum for the next set of characters
iWork = 0;
for (i++; i < 11; i++)
    iWork += (sText[i] - '0');

// If the checksum matches, continue
if (iWork % 10 != iCheck2)
    return false;

// If the last checksum matches, continue
if ((iCheck1 + iCheck2) % 10 != iCheck3)
    return false;

return true;
}

protected override string ClientSideCode()
{
return "function validateLicenseKey(value)\r\n" +
    "{\r\n" +
    "    // Strip non-digits from the string\r\n" +
    "    var sText = value.replace(/\\D/g, '');\r\n" +
    "    if (sText.length != 13)\r\n" +
    "        return false;\r\n" +
    "\r\n" +
    "    // Get the checksum values from the key\r\n" +
    "    var iZero = \"0\".charCodeAt(0);\r\n" +
    "    var iWork = 0, i;\r\n" +
    "    var iCheck1 = sText.charCodeAt(5) - iZero;\r\n" +
    "    var iCheck2 = sText.charCodeAt(11) - iZero;\r\n" +
    "    var iCheck3 = sText.charCodeAt(12) - iZero;\r\n" +
    "\r\n" +
    "    // Work out the checksum for the first set of\r\n" +
    "    // characters\r\n" +
    "    for (i = 0; i < 5; i++)\r\n" +
    "        iWork += (sText.charCodeAt(i) - iZero);\r\n" +
    "\r\n" +
    "    // If the checksum matches, continue\r\n" +
    "    if (iWork % 10 != iCheck1)\r\n" +
    "        return false;\r\n" +
    "\r\n" +
    "    // Work out the checksum for the next set of\r\n" +
    "    // characters\r\n" +
    "    iWork = 0;\r\n" +
    "    for (i++; i < 11; i++)\r\n" +
    "        iWork += (sText.charCodeAt(i) - iZero);\r\n" +
    "    "
}

```

```

        "\r\n" +
        " // If the checksum matches, continue\r\n" +
        " if (iWork % 10 != iCheck2)\r\n" +
        "     return false;\r\n" +
        "\r\n" +
        " // If the last checksum matches, continue\r\n" +
        " if ((iCheck1 + iCheck2) % 10 != iCheck3)\r\n" +
        "     return false;\r\n" +
        "\r\n" +
        "return true;\r\n" +
        "};";
    }

    protected override string ClientSideValidateCode()
    {
        return "validateLicenseKey(value)";
    }
}

```

Let's analyse this example more closely. You will see that three methods have been overridden:

- RunValidation
- ClientSideCode
- ClientSideValidateCode

RunValidation is defined in the ValidatorEntry class while ClientSideCode and ClientSideValidateCode are defined in the ValidatorEntryClientSide class. It follows then RunValidation is a method that specifically deals with validation on the server while the other two methods deal with validation on the client.

As you can see from the example, RunValidation is performing the server-side validation for the validator. The result of this method must be a Boolean value. If the value being validated passes validation, true is returned from this method, otherwise false. If the validator fails validation, the IsValid state of the validator will be false (causing eventually Page.IsValid to return false) and the message associated with the validator, or its parent validator if it has one, will be displayed in the validation summary.

ClientSideCode returns a JavaScript function that contains, in our example, exactly the same validation logic as RunValidation. The code returned from ClientSideCode is written once to output page regardless of how many times this validator is used on the page, but only if it is used at least once on the page.

By itself, the client-side validation code provided in the package is unaware of any client-side code returned from the ClientSideCode method. This is where ClientSideValidateCode comes in. The JavaScript returned from this method actually performs the client-side validation. The result of the call, like the server-side validation, must be a Boolean value – true for success, false for failed. You can include any JavaScript code here that results in a Boolean value, even code that does not call the method (if any) returned from the ClientSideCode method.

## 6.2 Extending ValidatorEntryRegex

The ValidatorEntryRegex class can be used by itself when you need to validate a given field against a specific regular expression. However, it is often more useful and better design to extend the ValidatorEntryRegex class in order to either hide the regular expression from page developers or to simply keep the regular expression in a single central location.

Let's say that instead of writing a complex license key validator such as that in example 6.1, you want to write a very simple license key validator that did nothing more than validate against a regular expression, you can simply extend the ValidatorEntryRegex class and hard-code the regular expression into it. For the purposes of this example, the license key required must conform to the sequence:

*[five numeric digits][hyphen][five numeric digits][hyphen][3 numeric digits]*

e.g. 12345-57890-150 (by a remarkable coincidence this key will validate correctly against the algorithm in the example 6.1 above).

Example 6.2 now shows the new, simpler ValidatorEntryLicenseKey class.

*Example 6.2. A simple license key validator based on the ValidatorEntryRegex class.*

```
using System;
using System.Text.RegularExpressions;
using MadWidgets.Validation;

namespace MadWidgets.Validation.Samples
{
    public class ValidatorEntryLicenseKey : ValidatorEntryRegex
    {
        public ValidatorEntryLicenseKey(object o, string sMessage)
            : this(o, sMessage, true)
        {
        }

        public ValidatorEntryLicenseKey(object o, string sMessage,
            bool bRequired)
            : base(o, sMessage, "^\\d{5}-\\d{5}-\\d{3}$", RegexOptions.None,
                bRequired)
        {
        }
    }
}
```

Remember that ValidatorEntryRegex is inherited from ValidatorEntryClientServer, which means that the ValidatorEntryRegex subclass that we've created above will validate on both the client and the server provided that client-side validation is turned on.

As you can see from example 6.1, the simpler ValidatorEntryLicenseKey class is very simple, in fact it may be too simple. In reality you might want to make a compromise between example 6.1 and example 6.2. That is, you want to provide a simple mechanism

for validating the license key on the client, but on the server you want a more complex validation. This provides a means of hiding the underlying license key algorithm, while still prevent common typographical errors to pass through from the client.

Example 6.3 shows such a compromise.

*Example 6.3. Combined Regex client-side validation and complex server-side validation.*

```
using System;
using System.Text.RegularExpressions;
using MadWidgets.Validation;

namespace MadWidgets.Validation.Samples
{
    public class ValidatorEntryLicenseKey : ValidatorEntryRegex
    {
        public ValidatorEntryLicenseKey (object o, string sMessage)
            : this(o, sMessage, true)
        {
        }

        public ValidatorEntryLicenseKey (object o, string sMessage,
            bool bRequired)
            : base(o, sMessage, "^\\d{5}-\\d{5}-\\d{3}$", RegexOptions.None,
                bRequired)
        {
        }

        protected override bool RunValidation()
        {
            // If the object is blank and blanks are allowed,
            // return true, otherwise continue
            if (!Required && ValidationValue == "")
                return true;

            // If the regular expression fails, return false
            if (!base.RunValidation())
                return false;

            // Strip non-digits from the string
            string sText = Regex.Replace(ValidationValue, "\\D", "");
            if (sText.Length != 13)
                return false;

            // Get the checksum values from the key
            int iWork = 0, i;
            int iCheck1 = sText[5] - '0';
            int iCheck2 = sText[11] - '0';
            int iCheck3 = sText[12] - '0';

            // Work out the checksum for the first set of characters
            for (i = 0; i < 5; i++)
                iWork += (sText[i] - '0');

            // If the checksum matches, continue
            if (iWork % 10 != iCheck1)
                return false;
        }
    }
}
```

```

        // Work out the checksum for the next set of characters
        iWork = 0;
        for (i++; i < 11; i++)
            iWork += (sText[i] - '0');

        // If the checksum matches, continue
        if (iWork % 10 != iCheck2)
            return false;

        // If the last checksum matches, continue
        if ((iCheck1 + iCheck2) % 10 != iCheck3)
            return false;

        return true;
    }
}
}

```

Comparing the examples 6.1 and 6.3, you'll notice that the `ClientSideCode` and `ClientSideValidateCode` have not been implemented in the `ValidatorEntryLicenseKey` class in example 6.3. This does not mean that no client side validation takes place, but rather that the client-side validation that takes place is determined by the `ValidatorEntryRegex` class. In other words, the only validation that now occurs on the client is the regular expression validation, while on the server the regular expression is checked followed by the license key algorithm.

### 6.3 Extending `ValidatorEntryServerOnly`

All of the examples above provide a mechanism for validating something on both the browser (the client) and on the server. In some situations it may be undesirable to validate user input on the client, or it may be impossible to implement. For example, if you want to create a login validator, you cannot normally validate a user's credentials on the client (it would not be terribly wise at least to attempt such a thing). In this example, you can only validate the user's login credentials on the server. When you want to create a validator that requires this type of server-side only validation, you should extend from the `ValidatorEntryServerOnly` class.

The `ValidatorEntryServerOnly` class is almost identical to the `ValidatorEntryClientServer` class except that it has not methods that can be overridden that deal specifically with client-side code. We could create our `ValidatorEntryLicenseKey` class based on the `ValidatorEntryServerOnly` class if we decided that we did not want any validation to occur on the client. To do this we simply remove all client-side functionality from the class and extend `ValidatorEntryServerOnly` instead of `ValidatorEntrySingleObject` or `ValidatorEntryRegex` as in example 6.4.

*Example 6.4. The ValidatorEntryLicenseKey class based on the ValidatorEntryServerOnly class.*

```
namespace MadWidgets.Validation.Samples
{
    public class ValidatorEntryLicenseKey : ValidatorEntryServerOnly
    {
        public object LicenseKeyField;

        public ValidatorEntryLicenseKey (object o, string sMessage)
            : base(sMessage)
        {
            LicenseKeyField = o;
        }

        protected override bool RunValidation()
        {
            // Strip non-digits from the string
            string sText = Regex.Replace(
                ValidatorEntry.GetValidationValue(LicenseKeyField),
                "\\D", "");

            if (sText.Length != 13)
                return false;

            // Get the checksum values from the key
            int iWork = 0, i;
            int iCheck1 = sText[5] - '0';
            int iCheck2 = sText[11] - '0';
            int iCheck3 = sText[12] - '0';

            // Work out the checksum for the first set of characters
            for (i = 0; i < 5; i++)
                iWork += (sText[i] - '0');

            // If the checksum matches, continue
            if (iWork % 10 != iCheck1)
                return false;

            // Work out the checksum for the next set of characters
            iWork = 0;
            for (i++; i < 11; i++)
                iWork += (sText[i] - '0');

            // If the checksum matches, continue
            if (iWork % 10 != iCheck2)
                return false;

            // If the last checksum matches, continue
            if ((iCheck1 + iCheck2) % 10 != iCheck3)
                return false;

            return true;
        }
    }
}
```

## 7 Creating custom validator summaries

The validation summary that comes shipped with the Mad! Widgets ASP.NET Validation Package is a very simple ASPX file (a basic UserControl) that extends the Mad! Widgets Validator control. This file contains code that iterates over the entire collection of ValidatorEntry objects, checking for validity status and adding the text message of any that are not in a valid state to an HTML list.

It's a very simple piece of code and can consequently be easily duplicated or rewritten to suit any specific formatting or styling requirement.

The validation summary does not need to extend the Validator control contained in the package. It can be a stand alone UserControl that references the Validator control separately. It may be necessary to do this if more than one validation summary is required on a page.

If the validation summary does not extend the Validator control supplied with the package, the Validator control must be placed somewhere in the control hierarchy still – in a location where it is valid to place JavaScript code. Note that when the control writes JavaScript code to the page, it also writes out the `<script>` tags that surround it. So the Validator control can be placed anywhere within the body of the page or within the `<head>` section.

### 7.1 A basic validation summary

Example 7.1 shows a simple validation summary control that extends the Validator control (MadWidgets.Validation.Validator). The Validator control is an object that manages the ValidatorEntry objects and instructs them to perform validation. By extending the Validator control, the validation summary has access to all public and protected methods available in the control. Consequently, the validation summary does not need to use a reference to the Validator control to access its methods.

The example shown simply iterates over the collection of ValidatorEntry objects held by the Validator control and writes the error message defined respectively to the HTML output. It also provides a 'show me' link where appropriate.

The example uses the *Count* property and the indexer method *this[i]* to perform the iteration. *this[i]* returns the ValidatorEntry object at the index position *i*.

The JavaScript function `showValidationError(id)` sends the cursor to the field that identified by 'id.' This can be either the object id as returned by the `GetObjectId` method or the `ValidatorEntry` object instance id which is written to the client if the object is an instance of `ValidatorEntryClientServer`.

Note that in example 7.1, it is assumed that the CSS styles are defined elsewhere.

*Example 7.1. A basic validation summary.*

```
<%@ Control Language="c#" AutoEventWireup="false"
    Inherits="MadWidgets.Validation.Validator" %>
<%@ Import namespace="MadWidgets.Validation"%>

<%
    if (!IsValid && IsPostBack)
    {
%>

<table cellpadding="0" cellspacing="0" border="0" class="validatorTable">
    <tr>
        <td class="validatorTitle" colspan="2"><%=SummaryTitle%></td>
    </tr>
<%
    int iError = 0;
    for (int i = 0; i < Count; i++)
    {
        ValidatorEntry ve = this[i];
        if (!ve.IsValid)
        {
            string sErrorMessage = ve.ProcessMessage();
            sErrorMessage = sErrorMessage.Replace("\r\n", "<br /><br />");
            string sStatusMessage =
                Server.HtmlEncode(sErrorMessage.Substring(0,
                    sErrorMessage.Length > 85 ? 85 : sErrorMessage.Length));
            sStatusMessage = sStatusMessage.Replace("\"", "&#" +
                ((int)'\\"').ToString("000") + ";");
            sStatusMessage = sStatusMessage.Replace("'", "&#" +
                ((int)'\'').ToString("000") + ";");
%>
<tr class="validatorRow"<%= (iError % 2) + 1%>">
    <td class="validatorEntry" width="1%"><%= iError + 1%>.&nbsp;&nbsp;&nbsp;</td>
    <td class="validatorEntry" width="99%">
        <%= sErrorMessage%>
<%
        if (ClientSideCodeEnabled && (!(ve is
            ValidatorEntryClientServer) ||
            ((ValidatorEntryClientServer)ve).ClientSideCodeEnabled))
        {
            object oFirst = ve.FirstObject;
            if (oFirst != null)
            {
%>
                [ <a href="javascript:showValidationError('<%= ve is
                    ValidatorEntryClientServer ? ve.Id :
                    Validator.GetObjectId(oFirst)%>');" class="validatorLink"
                    onmouseover="window.status='Validation Failure'; return true;"
                    title="Validation Failure">show me</a> ]
<%
            }
        }
%>
    </td>
</tr>
</table>
%>
```

```

        </tr>
    <%
        iError++;
    }
}
%>
</table>
<br />
<br />
<%
}
%>

```

## 7.2 Validation summaries for specific validation groups

Example 7.1 is a sample validation summary that will display all error messages regardless of the validation group that they belong to. In some circumstances, you may not want this behaviour. For example, you may have three validation groups on your page – a Login group, a Search group and a Subscribe group – where the associated forms provide fields respectively for logging in, searching the site and subscribing to a service of the site. Rather than display errors from each of these groups in a single validation summary, you can assign a validation summary to each group and place it in a location that suits the group. You may want the error messages associated with the login form to appear directly under the login form and the errors associated with the subscribe form to appear directly above that form. Using the Mad! Widgets ASP.NET Validation Package, this is very easy to achieve.

*Example 7.2. A simple validator summary that handles only a specific validation group.*

```

<%@ Control Language="c#" AutoEventWireup="false" %>
<%@ Import namespace="MadWidgets.Validation"%>
<%@ Import namespace="System.Collections"%>

<%
    Validator v = Validator.Current;
    if (v.WasGroupPostedForObject(this) && !IsValid && IsPostBack)
    {
%>

<table cellpadding="0" cellspacing="0" border="0" class="validatorTable">
    <tr>
        <td class="validatorTitle" colspan="2"><%=SummaryTitle%></td>
    </tr>
<%
    int iError = 0;
    IList l = v.GetEntriesForPostedGroup();
    for (int i = 0; i < l.Count; i++)
    {
        ValidatorEntry ve = (ValidatorEntry)l[i];
        if (!ve.IsValid)
        {
            string sErrorMessage = ve.ProcessMessage();
            sErrorMessage = sErrorMessage.Replace("\r\n", "<br /><br />");
            string sStatusMessage =
                Server.HtmlEncode(sErrorMessage.Substring(0,
                    sErrorMessage.Length > 85 ? 85 : sErrorMessage.Length));

```

```

        sStatusMessage = sStatusMessage.Replace("\'", "&#" +
            ((int)'').ToString("000") + ";");
        sStatusMessage = sStatusMessage.Replace("'", "&#" +
            ((int)'\').ToString("000") + ";");
    %>
    <tr class="validatorRow"<%= (iError % 2) + 1 %>">
        <td class="validatorEntry" width="1%"><%= iError + 1 %>. &nbsp;</td>
        <td class="validatorEntry" width="99%">
            <%= sErrorMessage %>
    <%
        if (v.ClientSideCodeEnabled && (!(ve is
            ValidatorEntryClientServer) ||
            ((ValidatorEntryClientServer)ve).ClientSideCodeEnabled))
        {
            object oFirst = ve.FirstObject;
            if (oFirst != null)
            {
    %>
                [ <a href="javascript:showValidationError('<%= ve is
                    ValidatorEntryClientServer ? ve.Id :
                    Validator.GetObjectId(oFirst) %>');" class="validatorLink"
                    onmouseover="window.status='Validation Failure'; return true;"
                    title="Validation Failure">show me</a> ]
    <%
            }
        }
    %>
    </td>
</tr>
<%
    iError++;
}
}
%>
</table>
<br />
<br />
<%
}
%>

```

Example 7.2 shows the same validation summary as example 7.1 except that it has been restricted to a specific validation group and it no longer extends the Validator control. The primary difference in the two examples is the `WasGroupPostedForObject(this)` call and the `GetEntriesForPostedGroup()` call, which returns true only if the form was posted back for the group associated with the validation summary control. Example 7.3 shows how the validation summary might be included in the page and associated with the group “Login”.

*Example 7.3. Assigning a validation group to a validation summary.*

```

<mycompany:MyValidationSummary runat="server" id="valsum"
    validationGroup="Login" />

```

## 8 Inline function reference

Inline message blocks are placed into messages using curly braces as in the following example:

```
"The birth date you entered is {value}, you are too young  
to register here."
```

Inline message blocks can contain any valid combination of properties and functions. Inline message functions follow the format:

```
functionName([param 1[, param 2[, ... param n]])
```

The result of the function is embedded into the message at the location of the inline message block.

The following reference describes all functions available within the inline message blocks. For a complete explanation of the use of inline message functions, see [Inline message blocks](#) earlier in this document.

```
cdate(value[, format])
```

Converts the value into a date if possible using the format if supplied. If the value cannot be converted to a date, the original value is returned.

**Return value:** string

**Parameters:**

value	A value capable of being parsed as a date.
format	An optional format specifier. If this is supplied it must be one of the values, 'DDMMYYYY', 'MMDDYYYY', 'YYYYMMDD', 'YYYYDDMM'. These values indicate which date format should be adopted if the format of the date being passed into the function seems ambiguous. If the format selected is 'MMDDYYYY' then the first digit(s) in the date represent the month value, the second the day value and the third the year value. The format specifier has no affect if the date being passed in as the value is not ambiguous. An example of an ambiguous date is 03/05/04.

**Example:**

```
cdate(value, 'DDMMYYYY')
```

```
cdbl(value)
```

Converts the value into a double (floating point number) if it is not already one. If the value cannot be converted, the original value is returned.

**Return value:** double

**Parameters:**

value	Any value capable of being parsed as a double..
-------	---

```
cint(value)
```

Converts the value into an integer if it is not already one. If the value cannot be converted, the original value is returned.

**Return value:** integer

**Parameters:**

value	Any value capable of being parsed as an integer.
-------	--

## concat

1.3

```
concat(value1[, value2[, ... value n]])
```

Returns the string concatenation of all values passed in.

**Return value:** string

**Parameters:**

`value1-n` Any value returned from a function or a property.

## cstr

1.3

```
cstr(value)
```

Converts the value into a string if it is not already one. If the value cannot be converted, the original value is returned.

**Return value:** string

**Parameters:**

`value` Any value returned from a function or a property.

## formatDate

1.3

```
formatDate(date, format[, default])
```

Returns a string representation of the date as specified by the format string. The format string follows the same rules as [Microsoft's custom date time string formatting](#). If the date value passed in cannot be formatted, it is returned as is unless *default* has been provided, in which case the default value is returned instead. Note that the value passed in as the date must be a Date type (DateTime on the server).

Unlike Microsoft's format specifiers, single character format strings do not represent fixed system specific format specifiers on the browser. However, single character format strings are treated this way on the server, so you should avoid using these strings altogether.

This function is fully culture sensitive and uses the culture specified in the Validator control or the web.config file (see [Setting up the web.config file](#)). If the culture cannot be determined from these methods, the culture of the website is used.

The following table describes the format specifiers and the output they produce.

Format Specifier	Description
D	The day of the month represented by the numbers 1 through 31.

<b>Format Specifier</b>	<b>Description</b>
Dd	The day of the month preceded, if the day is a single digit, by a 0.
Ddd	The abbreviated name of the day of the week.
Dddd	The full name of the day of the week.
M	The numeric month represented by the numbers 1 through 12.
MM	The numeric month preceded, if the month value is a single digit, by a 0.
MMM	The abbreviated name of the month.
MMMM	The full name of the month.
Y	The year in the century represented by the numbers 0 through 99.
Yy	The year in the century preceded, if the result is a single digit, by a 0.
Yyyy	The 4 digit year.
H	The hour in the 12 hour clock represented by the numbers 1 through 12.
Hh	The hour in the 12 hour clock preceded, if the hour is a single digit, by a 0.
H	The hour in the 24 hour clock represented by the numbers 0 through 23.
HH	The hour in the 24 hour clock preceded, if the hour is a single digit, by a 0.
M	The minute represented by the numbers 0 through 59.
Mm	The minute preceded, if the minute is a single digit, by a 0.
S	The second represented by the numbers 0 through 59.
Ss	The second preceded, if the second is a single digit, by a 0.
F	The seconds fractions returned as a single decimal place.
Ff	The seconds fractions returned as 2 decimal places.
Fff	The seconds fractions returned as 3 decimal places.
Ffff	The seconds fractions returned as 4 decimal places.
Fffff	The seconds fractions returned as 5 decimal places.
Ffffff	The seconds fractions returned as 6 decimal places.
Fffffff	The seconds fractions returned as 7 decimal places.
T	The first character of the AM/PM designator.
Tt	The full AM/PM designator.
Z	The time zone offset from UTC of the server in whole hours preceded by a '+' or a '-' character depending on whether the offset is positive or negative.
Zz	The time zone offset from UTC of the server in whole hours preceded by a '+' or a '-' character depending on whether the offset is positive or negative. If the offset is a single digit it is preceded with a 0.
Zzz	The time zone offset from UTC of the server in whole hours and minutes preceded by a '+' or a '-' character depending on whether the offset is positive or negative, e.g. +10:30. The separator used is culture specific and is determined by the time separator of the culture.
:	The culture specific time separator.

Format Specifier	Description
/	The culture specific date separator.

**Return value:** string

**Parameters:**

date                    A date object.  
format                    A format string. This can be any combination of the format specifiers in the table above.

**Example:**

```
formatDate(cdate(value, 'DDMMYYYY'), 'd MMM yyyy HH:mm:ss',
'Invalid date entered')
```

## formatNumber

1.3

```
formatNumber(value, format[, default])
```

Returns a string representation of the number as specified by the format string. The format string follows the same rules as [Microsoft's custom number string formatting](#). If the value passed in cannot be formatted, it is returned as is unless *default* has been provided, in which case the default value is returned instead.

This function is fully culture sensitive and uses the culture specified in the Validator control or the web.config file (see [Setting up the web.config file](#)). If the culture cannot be determined from these methods, the culture of the website is used.

The following table describes the format specifiers and the output they produce.

Format Specifier	Meaning	Description
0	Digit/zero placeholder	If the value being formatted has a digit in the position where the '0' appears in the format string, then that digit is copied to the result string. The position of the leftmost '0' before the decimal point and the rightmost '0' after the decimal point determines the range of digits that are always present in the result string. The '00' specifier causes the value to be rounded to the nearest digit preceding the decimal, where rounding away from zero is always used. For example, formatting 34.5 with '00' would result in the value 35.
#	Digit placeholder	If the value being formatted has a digit in the position where the '#' appears in the format string, then that digit is copied to the result string. Otherwise, nothing is stored in that position in the result string.

Format Specifier	Meaning	Description
		<p>Note that this specifier never displays the '0' character if it is not a significant digit, even if '0' is the only digit in the string. It will display the '0' character if it is a significant digit in the number being displayed. The '###' format string causes the value to be rounded to the nearest digit preceding the decimal, where rounding away from zero is always used. For example, formatting 34.5 with '###' would result in the value 35.</p>
.	Decimal point placeholder	<p>The first '.' character in the format string determines the location of the decimal separator in the formatted value; any additional '.' characters are ignored. The actual character used as the decimal point separator is culture specific and is determined by the NumberDecimalSeparator property of the NumberFormatInfo being used on the server.</p>
,	Thousands separator and number scaling	<p>The ',' character serves two purposes. First, if the format string contains a ',' character between two digit placeholders (0 or #) and to the left of the decimal point if one is present, then the output will have thousand separators inserted between each group of three digits to the left of the decimal separator. The actual character used as the thousands separator is culture specific and is determined by the NumberGroupSeparator property of the NumberFormatInfo being used on the server.</p> <p>Second, if the format string contains one or more ',' characters immediately to the left of the decimal point, then the number will be divided by the number of ',' characters multiplied by 1000 before it is formatted. For example, the format string "0,," will represent 100 million as simply 100. Use of the ',' character to indicate scaling does not include thousand separators in the formatted number. Thus, to scale a number by 1 million and insert thousand separators you would use the format string '#,##0,,'.</p>

Format Specifier	Meaning	Description
%	Percentage placeholder	The presence of a '%' character in a format string causes a number to be multiplied by 100 before it is formatted. The appropriate symbol is inserted in the number itself at the location where the '%' appears in the format string.
E0 E+0 E-0 e0 e+0 e-0	Scientific notation	If any of the strings 'E', 'E+', 'E-', 'e', 'e+', or 'e-' are present in the format string and are followed immediately by at least one '0' character, then the number is formatted using scientific notation with an 'E' or 'e' inserted between the number and the exponent. The number of '0' characters following the scientific notation indicator determines the minimum number of digits to output for the exponent. The 'E+' and 'e+' formats indicate that a sign character (plus or minus) should always precede the exponent. The 'E', 'E-', 'e', or 'e-' formats indicate that a sign character should only precede negative exponents.
\	Escape character	Escapes the character following the backslash as a literal character.
;	Section delimiter	The ';' character is used to separate sections for positive, negative, and zero numbers in the format string.
All other characters		All other characters are copied to the result string as literal characters in the positions they appear.

Note that for fixed-point format strings (strings not containing an 'E', 'E+', 'E-', 'e', 'e+', or 'e-'), numbers are rounded to as many decimal places as there are digit placeholders to the right of the decimal point. If the format string does not contain a decimal point, the number is rounded to the nearest integer. If the number has more digits than there are digit placeholders to the left of the decimal point, the extra digits are copied to the result string immediately before the first digit placeholder.

Different formatting can be applied to a string based on whether the value is positive, negative, or zero. To produce this behavior, a custom format string can contain up to three sections separated by semicolons:

*One section:*

The format string applies to all values.

*Two sections:*

The first section applies to positive values and zeros, and the second section applies to negative values.

*Three sections:*

The first section applies to positive values, the second section applies to negative values, and the third section applies to zeros. The second section might be left empty (by having nothing between the semicolons), in which case the first section applies to all nonzero values.

This type of formatting ignores any pre-existing formatting associated with a number when the final value is formatted. For example, negative values are always displayed without a minus sign when section separators are used. If you want the final formatted value to have a minus sign, you should explicitly include the minus sign as part of the custom format specifier. The following example illustrates how section separators can be used to produce formatted strings.

**Return value:** string

**Parameters:**

value	A numeric value.
format	A format string as described in the table above.

**Example:**

```
formatNumber(value, '$#,##0.00;($#,##0.00);Zero', 'Invalid number')
```

---

## isbool

1.3

isbool(value)

Returns true if the value passed in is a boolean value. If the value is a string, then it is treated as a boolean value if it equals 'true' or 'false' in any case.

**Return value:** boolean

**Parameters:**

value	Any value of any type.
-------	------------------------

---

## isdbl

1.3

isdbl(value)

Returns true if the value passed in is a number.

**Return value:** boolean

**Parameters:**

value	Any value of any type.
-------	------------------------

```
isint(value)
```

Returns true if the value passed in is an integer. If the value is a number with a value right of the decimal place, it is not treated as an integer and this function therefore will return false.

**Return value:** boolean

**Parameters:**

value	Any value of any type.
-------	------------------------

```
isdate(value[, format])
```

Returns true if the value can be converted to a date using the same conditions as [cdate](#).

**Return value:** boolean

**Parameters:**

value	A value capable of being parsed as a date.
format	An optional format specifier. If this is supplied it must be one of the values, 'DDMMYYYY', 'MMDDYYYY', 'YYYYMMDD', 'YYYYDDMM'. These values indicate which date format should be adopted if the format of the date being passed into the function seems ambiguous. If the format selected is 'MMDDYYYY' then the first digit(s) in the date represent the month value, the second the day value and the third the year value. The format specifier has no affect if the date being passed in as the value is not ambiguous. An example of an ambiguous date is 03/05/04.

```
join(array, delimiter1[, delimiter2[, template]])
```

Joins the keys and/or values of an associative array using one or more delimiters and a template. If `delimiter2` and `template` are not supplied, the associative array's values are concatenated together using `delimiter1` as the delimiter. If `delimiter2` is supplied, it is used as the final delimiter, e.g.

```
join(allvalues, ', ', ' and ')
```

This might produce the result “value1, value2, value3 and value4.”

If the template is supplied, it must contain ‘{value}’ and/or ‘{key}’ or any combination of these. It can additionally contain any combination of characters outside the curly braces. For each pair of elements in the associative array, the template is evaluated and ‘{value}’ and ‘{key}’ replaced with the respective value and key from the array.

**Return value:** string

**Parameters:**

<code>array</code>	An associative array.
<code>delimiter1</code>	The primary delimiter.
<code>delimiter2</code>	The final delimiter. This is used only once at the end of the join.
<code>template</code>	A string containing a combination of ‘{value}’ and/or ‘{key}’.

**Example:**

```
join(allvalues, ', ', ' and ', '{value} ( {key} )')
```

```
lc(str)
```

Returns a lowercase copy of the string passed in.

**Return value:** string

**Parameters:**

<code>str</code>	A string value.
------------------	-----------------

## length

1.3

---

`length(value)`

Returns the length of the string or array passed in.

**Return value:** integer

**Parameters:**

<code>value</code>	If the value is an array, the length of the array (the number of elements) is returned. If the value is a string, the length of the string is returned.
--------------------	---

## slice

1.3

---

`slice(array, beginIndex[, endIndex])`

Returns a new associative array using the elements the source array starting from `beginIndex` and ending at `endIndex` or the end of the source array if no `endIndex` is provided.

**Return value:** array

**Parameters:**

<code>array</code>	The source array.
<code>beginIndex</code>	The index position to begin the slice at.
<code>endIndex</code>	The index position to end the slice at.

## uc

1.3

---

`uc(str)`

Returns an uppercase copy of the string passed in.

**Return value:** string

**Parameters:**

<code>str</code>	A string value.
------------------	-----------------