

## 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;
        }
    }
}
```