jTAC: JavaScript Types and Conditions v0.5
User's Guide

Click on any of these topics to jump to them:

Overview

jTAC is a JavaScript library for enhancing data entry user interfaces. It expands *jquery-validate*, adding new rules and features. It also includes several *jquery-ui* widgets, including the DataTypeEditor, DateTextBox, and Calculator.

Underlying validation are three tools that you will want to use in other situations: TypeManagers, Conditions, and Connections.

- **TypeManagers** understand data types, like integers, dates, currencies, and email addresses. They provide parsers to convert strings into the native type and report errors when conversion fails. As a result, validation delegates significant work to TypeManagers. See "TypeManager classes".
- **Conditions** are the actual validation rules, like "compare to value", "range", and "text length". See "Condition classes".
- **Connections** are used to access data from elements on the page or other sources. They work with HTML form elements, custom widgets that are composed of several form elements (like a checkbox list), or API calls to get a value, and even your custom calculations that return a value. Conditions use Connections for all their data retrieval. See "Connection classes".

While *jQuery* is obviously supported, the jTAC core does not use *jQuery*, making its classes useable in other JavaScript frameworks. The *jQuery* tools are merely fully realized uses of these classes. It's these classes that will contribute most to your application. It all starts with validation…

- Issues with jTAC solves for jquery-validate
- Improvements to jquery-validate
- Understanding TypeManagers, Conditions, and Connections
- Overview of the jquery-ui widgets
- More benefits of using jTAC
- Platform support

Issues with jTAC solves for jquery-validate

Let's start with the issues jTAC solves for *jquery-validate*. Its rules are limited to handling single cases. Take the Range rule that comes with *jquery-validate*.

```
range: function( value, element, param ) {
   return this.optional(element) || ( value >= param[0] && value <= param[1] );
}
```

It evaluates a number against a minimum and maximum value. If you want the range to work with dates, you must create a new rule.

```
dateRange: function( value, element, param ) {
   var min = convertTextToDate(param[0]); // expected: "yyyy-MM-dd"
   var max = convertTextToDate(param[1]);
   var val = convertTextToDate(value);
   return this.optional(element) || ( val >= min && val <= min );
}
```

That `convertTextToDate()` function is a big deal, as it's a parser that you have to write. You can write a simple date parser, but if you are dealing with localization or real-world user input, parsers can get pretty fancy.

Even if you create the new rule, it may not work with all ui-widgets. For example, your new DateRange will work with a textbox, but not a calendar widget, which requires an API call to access its value.

```
calendarDateRange: function( value, element, param ) {
   var min = convertTextToDate(param[0]); // expected: "yyyy-MM-dd"
   var max = convertTextToDate(param[1]);
   var val = element.calendar("getDate"); // value param is ignored
   return this.optional(element) || ( val >= min && val <= min );
}
```

jTAC builds all of this into a single rule (shown here using unobtrusive setup):

```
<input type='text' id='textbox1' name='textbox1'
   data-val="true"
   data-jtac-datatype="date"
   data-val-datatypecheck="Enter a valid date"
   data-val-advrange="Value must be between {MINIMUM} and {MAXIMUM}"
   data-val-advrange-json="{'minimum': '2000-01-01', 'maximum'='2009-12-31'}" />
```

Note: All data-val properties are defined according to jquery-validate. Any of jTAC's rules are the Condition name, except "required" and "range" which conflict with the native rules of the same name. Any properties to set on the Condition are assigned using JSON to the data-val-rulename-json attribute, as shown above.

You can also see here a second aspect of the enhanced *jquery-validate* framework: clear tokens in the error messages. Each validation rule defined by jTAC can define a default error message and convert any tokens you like.

Improvements to jquery-validate

- New rules including one that lets you built other rules into a single Boolean expression
- Rules can operate on multiple data types. Includes support for: integer, Float, Currency, percent, date, time of day, duration, month+year, and day+month. You can expand and customize this.
- Rules have properties to let you customize their behavior. For example, the rule for regular expression parsing includes properties to indicate “ignore case sensitive”, “global”, and “multiline” (the 3 options passed into the regular expression parser.)
- Support for dependencies while in unobtrusive validation.
- Build powerful dependencies without writing your own functions. Define any Condition object on the **depends** property of the **param** property for the validation rule.
- Validation can use data from multiple fields. While the Compare Two Elements rule is obvious, also consider the simple Required rule. jTAC’s version allows you to define a list of elements to require, and then require one from the list, all, one or all, etc.
- Validation can use data from fields that need to be accessed by API. Take a calendar widget that holds a selected date value. It may not have an <input> to host that value. Instead, it offers an API function like `$("#calendar").getDate()`.
- Powerful parsers to convert text into a native type. They support localized formats of dates, times, and numbers.
- You don’t have to create a new validation rule for each string-based data type, like Phone Number and Email Address. There are TypeManagers defined to handle data types. So reuse the “DataTypeCheck” validation rule, assigning its **datatype** property to the appropriate TypeManager.
- Can define the form’s validation options when using unobtrusive validation.
- Error messages can support multiple localized values. In fact, all strings shown to the user can be localized.
- Expands how tokens work within error messages. They use clear words like {COUNT} and {MINIMUM} instead of {1} and {2}.
- Tokens can be updated at runtime with information calculated during validation, such as {COUNT} can show the current number of characters typed.
- The {LABEL} token can display the field’s label. Validators automatically know that `<label for="textbox">` tags are the source of error messages. In addition, you can assign the **data-msglabel** attribute to the textbox with the appropriate label.
- Highlight fields feature supports validation on multiple elements. For example, when the Compare Two Elements rule reports an error, both textboxes have their style sheet class changed.
- Highlight fields feature supports changing the style sheet on the field label or a containing element without you writing code. Labels work automatically. Containing elements only need the **data-jtac-valinputs** attribute.
- When using the highlight fields feature, individual inputs, labels, and containing elements can override the default style sheet classes on a case-by-case basis. Just define the alternative style sheet class in the **data-jtac-errorclass** attribute of the element.

Understanding TypeManagers, Conditions, and Connections

The validation framework uses “conditions” to evaluate some data and return a result of “valid” or “invalid”. Conditions often have simple logic, such as a testing the input value is within a range. Yet in real world applications, conditions can be developed from more complex Boolean logic, such as *if checkbox1 is checked, confirm textbox1 has a date before 1/1/2000. Otherwise, confirm textbox1 has a date starting at 1/1/2000.*

The challenge is to develop these conditions. *jquery-validate* implements conditions in what it calls rules. However, its existing rules are not flexible. Each is data type specific (you need separate rules for dates, integers, etc) and only knows how to interact with pure HTML widgets (many widgets require an API call to get their value).

jTAC recognizes that there are three essential elements to a validation rule: Conditions, TypeManagers, and Connections.

Click on any of these topics to jump to them:

TypeManager classes

A **TypeManager** class understands data types, like integer, currency, date, time of day, and email address. Unique TypeManager classes are defined for each data type.

Each TypeManager class can convert between string and native forms of their data type. Converting from string requires a parser. The supplied TypeManagers provide some flexible parsers to allow the user to enter data in their own way. They also have formatters to convert from native type to string. You can replace the parsers and formatters as needed.

For dates, times, and numbers, conversion from and to strings is localizable. jTAC includes support for the *jquery-globalize* library to handle localization. You can substitute other localization systems too.

You will attach a TypeManager to any textbox that takes a specific data type, by assigning the data type name to the **data-jtac-datatype** attribute. *jquery-validate* will validate it by using the DataTypeCheck rule. Turn it into a powerful editor with the DataTypeEditor *jquery-ui* widget.

Here are the TypeManagers supplied with jTAC. You can build your own or subclass these to customize them.

TypeManager	Purpose
Boolean	Typically used with the CompareToValue condition to compare the state of a checkbox or radio button.
CreditCardNumber	Use with credit card numbers. Validation uses Luhns algorithm.
Currency	Optionally formats with the localized currency symbol and thousands separators.
Date	Handles localized short, abbreviated and long date formats.
DateTime	Date and time of day in the same string or Date object.
DayMonth	A date that omits the year. Good for anniversary dates.
Duration	Time value that can go up to 9999 hours.
EmailAddress	Handles strings that reflect an email address. Supports multiple addresses.
Float	Can specify a maximum number of decimal places and have it fill in trailing zeros. Supports various rounding modes.
Integer	
MonthYear	A date that omits the day of month. Often used for credit card expiry dates.
PhoneNumber	Handles strings that reflect a phone number. Supports different countries.
PostalCode	Handles strings that reflect a postal code. Supports different countries.
String	Evaluates strings, either case sensitive or insensitive. Often used for case insensitive comparisons.
TimeOfDay	Localized. Can optionally show or hide the number of seconds.
Url	Handles strings that reflect a URL.

For details, see “The TypeManager classes”.

Condition classes

The **Condition** is the primary object used by validation. Its job is to evaluate something and return "success", "failed" or "cannot evaluate". jTAC extends *jquery-validate* with new rules that utilize all of the Condition classes.

You can use the Conditions in your own validation libraries too. They can even be used when you are writing JavaScript that needs to evaluate an element on the page like in an IF statement.

```
if (jTAC.isValid({jtacClass: "CompareToValue",
   elementId: "TextBox1", operator: "<>", valueToCompare:1}) {
   // do something
}
```

Much of the power of Conditions comes from their use of two other classes: TypeManagers and Connections. The TypeManager converts the string entered by the user into the native type. For example, in the above code, if TextBox1 uses TypeManagers.Integer, the value from TextBox1 and the **valueToCompare** property are each run through the TypeManager to create two integers. Then the Condition's evaluation logic is merely a comparison operator.

The following is pseudo code.

```
var tm = getTypeManager();
var inputVal = tm.toValue(document.getElementId("TextBox1").value);
var valueToCompare = tm.Value(this.getValueToCompare());
return (inputVal <> valueToCompare) ? 1 : 0;     // success or failed
```

The Connection class interacts with the element on the page, providing a layer between the Condition and your UI. This allows using elements that require an API to access their values, such as a *jquery-ui* widget might. Let's modify the above to use a Connection.

```
var tm = getTypeManager();
var conn = jTAC.resolveConnection.create("TextBox1");
var inputVal = tm.toValueFromConnection(conn);
var valueToCompare = tm.Value(this.getValueToCompare());
return (inputVal <> valueToCompare) ? 1 : 0;     // success or failed
```

Here are the Conditions supplied with jTAC. You can build your own or subclass these to customize them.

Condition	Purpose	Replaces Rule
BooleanLogic	Defines an AND or OR expression with other Conditions. For example, Required on TextBox1 OR CompareToValue on DecimalTextBox1.	New!
CharacterCount	Determines if the number of characters in a string is within a range. Includes powerful tokens that assist the user in error messages, such as {COUNT} and {DIFF}.	minLength, maxLength, rangeLength
CompareToValue	Compares the value from a widget to a fixed value. It allows for all of the usual comparisons: equal, less than, etc. Handles multiple data types.	min, max
CompareTwoElements	Compares two widgets to each other. It allows for all of the usual comparisons: equal, less than, etc. Handles multiple data types.	equalTo
CountSelections	Evaluates list-style elements that support multiple selections. It counts the number of selections and compares the value to a range.	New!
DataTypeCheck	Parses text to ensure it converts to a native data type. Handles multiple data types. This is likely to be the most used item. It handles all of the types defined as TypeManagers in the next topic.	date, dateISO, number, digits, url, email
Difference	Evaluates if the difference between the values of two widget. Compares the difference to another value. For example, a date range must be at least 10 days apart. Handles multiple data types.	New!
DuplicateEntry	Ensures that the values from a list of widgets are all different.	New!
Range	Compares the value from a widget to a range. Handles multiple data types.	range

RegExp	Evaluates the value of a widget against a regular expression.	New!
Required	Evaluates one or more widgets to determine if they are empty. With multiple widgets, use rules like “All”, “One”, “OneOrAll”, “OneOrMore”, and “Range”.	Required
SelectedIndex	Compares an index (or list of indices) with the actual selected index of list style controls. Supports multiselection lists too.	New!
WordCount	Counts the number of words.	New!

For details, see “The Condition classes”.

Connection classes

The **Connection** class gets and sets the value of an element on the page, either by a string representing its value, or by the native value itself. Connections also know about setting style sheet classes, attaching event handlers, and accessing data attributes.

They are effectively a layer between any element and a Condition so that when you introduce a new type of element to the page, you don't have to rewrite your Condition's logic to handle it. Instead, a new Connection class is created and registered with the page. From there, jTAC knows to create the Connection each time the particular element is used.

Connections can get data from other resources. Suppose you have a calculated value to use in a Condition, such as combining the text of First name and Last name textboxes. You create a Connection, overriding a few methods. Then plug it into the Condition's **connection** property.

A Connection can supply a TypeManager if its element is type specific. For example, when using the HTML5 <input type="date" /> tag and the *jquery-ui* datepicker widget, the connection knows to create a TypeManagers.Date object.

Connection	Purpose
FormElement	Native HTML form elements including input, select, and textarea. Supports HTML 5 input types.
UserFunction	You write a function that supplies the Connection's value to the caller. Typically used with calculated fields.
Value	Instead of accessing a widget, it holds a constant value that any consumer can use.
InnerHtml	Works with HTML tags that hold HTML in their body, such as div and span. This is mostly used with calculations.
jquiDatePicker	Supports the *jquery-ui* datepicker.

For details, see "The Connection classes".

Overview of the jquery-ui widgets

The TypeManager, Condition, and Connection classes all can be used outside of validation. jTAC's *jquery-ui* widgets all are built to use them. Let's look at them.

- **DataTypeEditor** – Turns an ordinary `<input type='text' />` into a powerful editor of a specific data type. Its primary tool is the TypeManager class. It uses any TypeManager to parse the text input and reformat it as the user leaves the field. It also filters out invalid keystrokes. Finally, it has a command processor so you can hook up special keys to change the value. For example, the TypeManagers.Date class uses the up and down arrows to increment and decrement the date.

```
<input type='text' id='integer1' name='integer1'
   data-jtac-datatype="Currency" data-jtac-datatypeeditor="" />
```

  See "Using the DataTypeEditor widget".

- **DateTextBox** – A merger of the DataTypeEditor with the *jquery-ui* datepicker widget. Effectively replaces the parser of the datepicker with the parser from the TypeManagers supplied with jTAC. With the TypeManagers.PowerDateParser, the user gets a more fluid data entry field. It supports keyboard shortcut commands.

```
<input type='text' id='integer1' name='integer1'
   data-jtac-datetextbox=
      "{datepicker: {buttonImage: '/Images/Button.gif'} }" />
```

  See "Using the DateTextBox widget".

- **Calculator** – Create powerful numeric calculations that interact with input elements on the page and show the result. Often used in shopping carts and grid UIs. This widget uses the Connection and TypeManager classes to quickly connect to an input widget and convert its value to a number.

```
<input type='text' id='r1c1' name='r1c1' data-jtac-datatype="integer"
   data-jtac-datatypeeditor="" />
<input type='text' id='r1c2' name='r1c2' data-jtac-datatype="integer"
   data-jtac-datatypeeditor="" />
<span id="Row1Result"></span>
<input type="hidden" id="r1Total" name="r1Total"
   data-jtac-calculator='{ expression: &#39;"r1c1" + "r1c2"&#39;,
       displayElementId: "Row1Result", useKeyEvt: true }' />
```

  Note: &#39 is the HTML character representation of a single quote. This allows nested single quotes.

  The example has two textboxes that accept integers (due to the DataTypeEditor feature). The hidden input hosts the calculation, which adds the values of both textboxes and displays the result in the *Row1Result* span tag.

  In addition to providing interactive calculations, validation often needs the result of a calculation. jTAC's validators and Conditions accept the id of the hidden input and will ensure the calculation is up-to-date before using the value.

  See "Using the Calculator widget".

More benefits of using jTAC

- Localization supported for dates, times, and numbers. In addition to the built-in localization rules, it allows plugging in third party localization libraries. *jquery-globalize* support is already included as a plug in. See "Localizing dates, times, and numbers".
- Multilingual support of text shown to the user, such as validation error messages. See "jTAC.Translations system: Replacing strings shown to the user".
- TypeManagers convert strings to native types using a parser. Parsers are often very specialized. So jTAC provides plug-in parsers and includes two very powerful powers for date and times. See "Alternative parsers for TypeManagers".
- The entire framework is built for expansion by using classes. You can develop your own classes using inheritance. Extensive documentation is both in this PDF and inline. See "Adding your own classes to jTAC". You may actually appreciate using jTAC's class building framework for your own classes that are not used with jTAC. See "jTAC class members".
- Debugging is made easier by using the browser's Console view where jTAC writes its error messages.
- jTAC is open source and community driven. Submit your ideas and modifications at https://github.com/plblum/jtac.
- The author, Peter Blum, is also the author of a successful commercial ASP.NET control suite, "Peter's Data Entry Suite". That product is similar to jTAC; it handles client-side validation and extends the features of textboxes. He has rolled in much of the knowledge learned from customers over a 10+ year period. Special cases are abound, whether is an option for parsing or dealing with an idiosyncrasy of a browser.

Platform support

HTML versions

jTAC was built to use modern JavaScript and HTML features (modern means "as of 2012"). It does not require HTML 5, but provides support for HTML 5's <input> types and utilizes attributes starting with "data" (like `<input data-jtac-name="value" />`. It does not require using the `<!DOCTYPE>` for HTML5 either. If you are concerned about valid markup in pre-HTML5 platforms, do not use the unobtrusive solutions offered in jTAC. It has ways to set up attributes programmatically.

JavaScript version

jTAC takes advantage of JavaScript 1.8.5 to support properties (`Object.defineProperty`). However, that is for convience of the user, and not required. Although testing is needed to prove it, the minimum version of JavaScript should be 1.6 (requires support of `Array.indexOf()`).

Browsers

As of jTAC version 0.5 (Oct 2012), jTAC has been tested on these browsers: IE 9, FireFox 16 on Windows, Chrome 22 on Windows, Safari 5 on Windows, Opera 12 on Windows. All were tested with this `<!DOCTYPE>` tag:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

Please help out by testing it on other browsers and offering bug fixes where needed. Use the jsunit tests included within the Development Site folder. Run the individual test page or **\jsunit tests\TestRunner.htm**.

Other libraries

jTAC does not require any other library to support its TypeManagers, Conditions, Connections, or CalcItem classes.

Its support of *jquery-validate* was tested with *jquery-validate* 1.8.0 and *jQuery* 1.5.1.

Its support of *jquery-ui* was tested with *jquery-ui* 1.8.11 and *jQuery* 1.5.1

Its support of *jquery-globalize* was tested with v0.1.0a2-47.

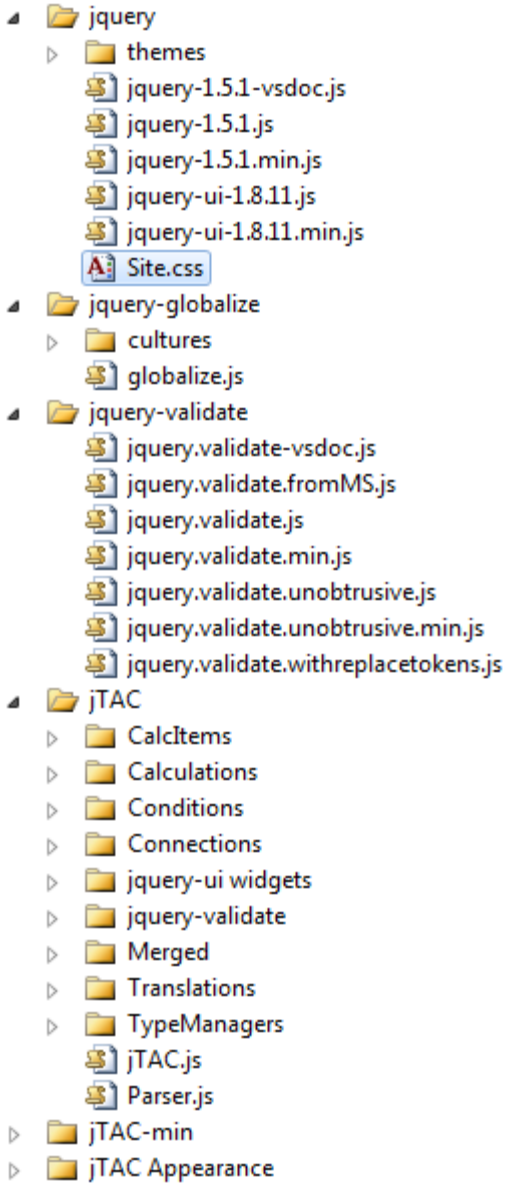
Server side

jTAC does not care about how the server delivered it. It should work with any web server.

Adding jTAC to your application

When first retrieved, jTAC creates a folder with these elements.

- Development site – Hosts an application used to develop jTAC. It includes numerous example files and jsunit tests.
- Docs – This file.
- Preparing Scripts folder – Tools to merge and minify the jTAC library. Windows batch files that do most of the work. The yaocompressor application from yahoo that is used for compression.
- Scripts – Hosts the script files that you will add into your application.

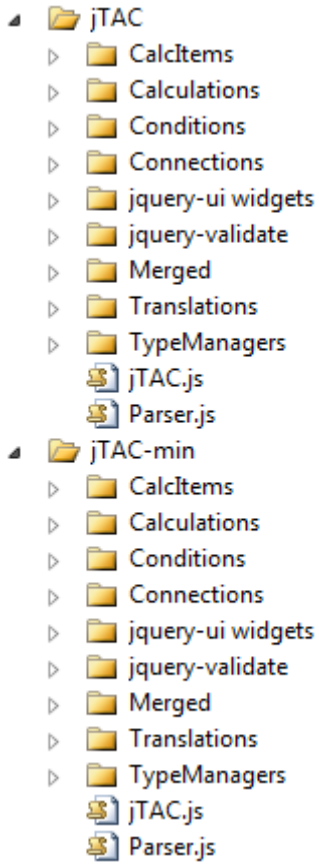


Instructions

1. Copy the contents **of \jTAC\Scripts** into the root of your application. This gives you two folders, one with source files and the other with minified files.
2. Copy the **\jTAC\jTAC Appearance** folder into the root of your application.
3. When using *jQuery*, know that jTAC was developed with *jQuery* 1.5.1 and *jquery-ui* 1.8.11. It has not been tested on earlier frameworks.
4. Continue by adding scripts specific to the jTAC tools you need for a page.

   - Understanding the available scripts files
   - Working with jquery-validate
   - Using the DataTypeEditor widget
   - Using the DateTextBox widget
   - Using the Calculator widget
   - Localizing dates, times, and numbers

Understanding the available scripts files



jTAC defines each class in its own script file and there are many! Since classes inherit from others, you will not only need to add the actual class's script file, but also the files of all ancestor classes. jTAC will notify you in the browser's console view when a script file is not loaded.

Hint: Always use the browser's console view when developing with jTAC. jTAC directs errors into it.

It provides script files in several formats:

- Individual classes for development and debugging – The individual classes fully commented. These files are large due to the comment and lack of minification. You must add each class explicitly, including ancestor classes.

  Use this folder: **\jTAC**

  Always include **jTAC.js**. The remaining files depend on the classes you intend to use.

- Minified individual classes for deployment – The individual classes, minified. Appropriate for testing, staging, and production. These files are much smaller.

  Use this folder: **\jTAC-min**

  If you want to switch between source and minified, simply change the URL in the <script> tag from "**\jTAC**" to "**\jTAC-min**".

- Merged for development and debugging – Many related classes have been merged into files to provide quicker setup and fewer requests from the browser. They ensure all ancestor classes are included when you require a specific class.

  Use this folder: **\jTAC\Merged**

  Always include the **\jTAC\Merged\core.js** file. Then add the appropriate files for the features you need.

  You can create other merged files as needed. Just edit the **[jTAC product folder]/Preparing Script files/Make Merged files.bat** file to include an XCOPY line that puts together other files into one. Your new file will be included in the minification process.

- Minified merged for deployment – Same as Merged, except all files have been minified. Appropriate for testing, staging, and production.

  Use this folder: **\jTAC-min\Merged**

  If you want to switch between source and minified, simply change the URL in the <script> tag from "**\jTAC**" to "**\jTAC-min**".

If you want to add all scripts in a single <script> tag, two files are available: with and without jquery support.

```
<script src="/jTAC-min/Merged/jquery extensions/jtac-all.js" type="text/javascript"></script>
```

or

```
<script src="/jTAC-min/Merged/jtac-all.js" type="text/javascript"></script>
```

Otherwise, the following links provide suggestions on which files in the Merged folders to use for each situation.

- Loading scripts supporting jquery-validate
- Loading scripts for the DataTypeEditor widget
- Loading scripts for the DateTextBox widget
- Loading script files for the Calculator widget
- Loading scripts for Calculations without jquery-ui support
- Loading scripts when using Conditions without jquery-validate

Working with jquery-validate

Click on any of these topics to jump to them:

Working with inline code (no unobtrusive validation)

1. Start by understanding how to use *jquery-validate* here.
2. Load the script files. See "Loading scripts supporting jquery-validate".

```
<script src="/jquery/jquery-#.#.#.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/core.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/jquery extensions/validation-all.js"
    type="text/javascript"></script>
<script src="/jTAC-min/Merged/typemanagers-all.js" type="text/javascript"></script>
```

   Note: The validation-all and typemanagers-all script files are suggestions. These have merged all available scripts. If you are looking for smaller scripts, see Loading scripts supporting jquery-validate.

3. If an `<input type="text" >` tag is data type specific, add the **data-jtac-datatype** attribute to it. Its value can be any TypeManager name including a number of custom names that create TypeManagers with specific properties already set. See "Using TypeManagers in datatype properties".

```
<input type="text" id="textbox1" name="textbox1" data-jtac-datatype="Integer" />
```

   Note: This attribute's value is case sensitive.

   *Note: This attribute is only validated within HTML 5. If you are using something else and demand validated markup, you can define the same value in the **datatype** property of Conditions that need it.*

   Note: jquery-validate requires both the id and name properties are defined.

4. Add the rules to each element on the page that needs validation.

   In `$(document).ready()`, add a rule using this syntax.

```
$("#ElementID").rules("add", {
   rulename: {
      param: {propertyname: value, propertyname2: value}
   },
   rulename2: {
      param: {propertyname: value, propertyname2: value}
   }
});
```

   You can use rules already defined by *jquery-validate* in addition to those from jTAC.

   Here are the rules introduced by jTAC: (* needs a TypeManager specified)

Rule name	Condition class
advrange	Conditions.Range *
advrequired	Conditions.Required
booleanlogic	Conditions.BooleanLogic
charactercount	Conditions.CharacterCount
comparetovalue	Conditions.CompareToValue *
comparetwoelements	Conditions.CompareTwoElements *
countselections	Conditions.CountSelections
datatypecheck	Conditions.DataTypeCheck *
difference	Conditions.Difference *
duplicateentry	Conditions.DuplicateEntry
regexp	Conditions.RegExp
requiredindex	Conditions.RequiredIndex
selectedindex	Conditions.SelectedIndex
wordcount	Conditions.WordCount

When using a Condition, the **param** property always hosts a JavaScript object (including in JSON). Use it to define properties to override on the Condition object that was created. You never need to assign the **elementId** property to the element's ID.

If you have no properties to override, the **param** property can be omitted.

```
$("#TextBox1").rules("add", {
   datatypecheck: {
   },
   advrange: {
      param: {minimum: 1, maximum: 10}
   }
});
```

5. Add dependencies if needed. A dependency is a way to enable and disable the rule based on the state of other inputs on the page.

   Dependencies can be defined in two places.

   - The Condition object has a **depends** property.

```
$("#TextBox1").rules("add", {
   advrange: {
      param: {minimum: 1, maximum: 10, depends: rule goes here }
   }
});
```

   - The Rule object also has a **depends** property. This is the standard for *jquery-validate*, but it requires extra work to support conditions. Therefore, the Condition's property is preferred.

```
$("#TextBox1").rules("add", {
   advrange: {
      param: {minimum : 1, maximum : 10 },
       depends: rule goes here
   }
});
```

   A dependency can be defined in several ways:

   - As a string representing a *jQuery* selector. The *jQuery* selector is passed to $(selector). If it returns one or more results, the rule is enabled. This example requires checkbox1 to be checked.

```
depends: "#CheckBox1:checked"
```

   - As a function that is passed the element to validate and returns true or false. (This is not available in unobtrusive validation.)

```
depends: function(element) {
    return $("CheckBox1")[0].get().checked;
}
```

   - As a Condition object. If the Condition evaluates as "success", validation runs. The object can be defined either as an instance of the Condition class or as a JSON object, with the property "**jtacClass**" assigned to a string which is the name of the class. The remaining properties populate the Condition's properties. Always include the **elementId** property to specify which element is evaluated.

     When using this, you must define a **depends** property inside the Condition object that validates.

```
depends: {'jtacClass': 'Conditions.CompareToValue',
    elementId : 'CheckBox1', operator: '=', valueToCompare: true}
```

     or:

```
depends: jTAC.create("Conditions.CompareToValue",
    {elementId: 'CheckBox1', operator: '=', valueToCompare: true})
```

6. Override the default error message if desired. There are two ways.

   Note: The default error messages use tokens to allow runtime substitution of values. These tokens may be enough to avoid replacing an error message. In addition, you can use the tokens in your own messages. See the documentation for each Condition to learn about its tokens.

   ### Use the messages property on a rule

   Specify the **messages** property within the rule definition. Within it, create an object where each property is a rule name to have an error message and the value is the error message.

```
$("#ElementID").rules("add", {
   rulename: {
      param: {propertyname: value, propertyname2: value}
   },
   rulename2: {
      param: {propertyname: value, propertyname2: value}
   },
    messages: {
        rulename: "error message",
        rulename2: "error message"
    }
});
```

   ### Define error messages globally and make them localizable

   Take advantage of jTAC's tools to replace default strings globally and localize them.

   jTAC provides its *jTAC.Translations system* (included in **jTAC.js**) to register key and value pairs. All jTAC code that needs to show the user a string will ask the jTAC.Translations for a string that you defined, and only if not found will it use a hard-coded default.

   See "jTAC.Translations system: Replacing strings shown to the user" to setup and customize strings.

7. All default error messages include the {LABEL} token. This will be replaced with values taken from one of these resources.

   - If there is a `<label for="elementid">` tag, its text will be used. Note: This is overridden by the items below. If used, it will be cleaned up, by removing HTML tags and non-textual lead and trailing characters.
   - Add the **data-msglabel** attribute to the element's HTML tag with the value to show.

```
<input type="text" id="textbox1" name="textbox1"
   data-jtac-datatype="Integer" data-msglabel="label text" />
```

   - If localizing, define a new key in the **\jTAC\Translations** scripts with the desired label. Then add the **data-msglabel-lookupkey** attribute assigned to your new key into the HTML tag.

```
<input type="text" id="textbox1" name="textbox1"
   data-jtac-datatype="Integer" data-msglabel-lookupkey="newkeyname" />
```

   - If none of the above, {LABEL} is replaced by "Field".

8. Establish the *jquery-validate* options on the <form> tag that contains the inputs.

   In `$(document).ready()`, use this syntax. Place it prior to the rule definitions.

```
$("#Form1").validate({optionname: value, optionname2: value}});
```

9. Specify the location of the error message with a or similar container that identifies the input that supplies errors with the **data-valmsg-for** attribute.

```
<input type="text" id="textbox1" name="textbox1" data-jtac-datatype="Integer" />
<span data-valmsg-for="textbox1" ></span>
```

   Alternatively, you can define a function on the **errorPlacement** property of validation options.

10. Supply style sheet classes. They are found in the **\jTAC Appearance\jquery.validate.css** file. Either add the following tag or merge these styles into your own files.

```
<link href="/jTAC%20Appearance/jquery.validate.css" rel="stylesheet" type="text/css" />
```

WARNING: Modify the URL shown based on the location of the web page under the root folder.

You can use different class names by specifying the new name in the correct property of the options you setup in step 8.

Original class name	Property in options	Description
input-validation-error	inputErrorClass	The class assigned to the input elements showing an error.
input-validation-valid	inputValidClass	The class assigned to the input elements that are valid.
message-validation-error	messageErrorClass	Assigned to the container of the error message when there is an error.
message-validation-valid	messageValidClass	Assigned to the container of the error message when there is no error.
message-label	messageLabel	Replacement for the "{ERRORLABEL}" token within error messages.
label-validation-error	labelErrorClass	Assigned to the <label for="inputelement"> element associated with the error message.
container-validation-error	containerErrorClass	Assigned to elements with the **data-jtac-valinputs** attribute.

11. Your input fields will automatically change their style sheets when they reflect an error. This feature may need some setup.
    - The input fields use the class “input-validation-error”.
    - If you want to override the default class on a specific input, add the **data-jtac-errorclass** attribute to the input, and specify its value as the desired style sheet class.

```
<input type="text" id="textbox1" name="textbox1"  data-jtac-errorclass="customClass" />
```

12. Labels and container elements can also have their style sheet changed when their input(s) show an error.
    - Labels using the **for=** attribute will automatically work when **for=** specifies the input being validated. They are assigned the class “label-validation-error”.

```
<label for="textbox1" >First Name</label>
<input type="text" id="textbox1" name="textbox1"/>
```

    - For anything else, add the **data-jtac-valinputs** attribute to identify that its class will be changed to “container-validation-error”. If **data-jtac-valinputs** is assigned to "", it will change for any input that it contains. Otherwise assign a pipe delimited list of IDs to inputs that it responds to.

      Using a container that knows about all inputs.

```
<div data-jtac-valinputs="" >
   <input type="text" id="textbox1" name="textbox1"/>
</div>
```

      Using a container that does not contain the inputs.

```
<span data-jtac-valinputs="textbox1" >First Name</span>
<input type="text" id="textbox1" name="textbox1"/>
```

    - If you want to override the default class, add the **data-jtac-errorclass** attribute to the element, and specify its value as the desired style sheet class.

```
<label for="textbox1" data-jtac-errorclass="customClass" >First Name</label>
<input type="text" id="textbox1" name="textbox1" />
```

13. Optionally add a call to `jTAC.jqueryValidate.attachMultiConnections(selector)` after the rules code. This helps Conditions that support multiple inputs to report errors when the user edits any of those inputs.

    The function's parameter is a *jQuery* selector that identifies the form element to modify.

```
$("#Form1").validate({});
$("#TextBox1").rules("add", {
      datatypecheck: { },
   comparetwoelements: {
      param: {operator: "<",  elementId2: "TextBox2"}
   }
});
$("#TextBox2").rules("add", {
   datatypecheck: { }
});
jTAC.jqueryValidate.attachMultiConnections("#Form1");
```

Working with unobtrusive validation

1. Start by understanding how to use *jquery-validate* here.
2. Load the script files. See "Loading scripts supporting jquery-validate".

```
<script src="/jquery/jquery-#.#.#.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.unobtrusive.js"
   type="text/javascript"></script>

<script src="/jTAC-min/Merged/core.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/jquery extensions/validation-all.js"
   type="text/javascript"></script>
<script src="/jTAC-min/Merged/typemanagers-all.js" type="text/javascript"></script>
```

   Note: The validation-all and typemanagers-all script files are suggestions. These have merged all available scripts. If you are looking for smaller scripts, see Loading scripts supporting jquery-validate.

3. If an <input type="text" > tag is data type specific, add the **data-jtac-datatype** attribute to it. Its value can be any TypeManager name including a number of custom names that create TypeManagers with specific properties already set. See "Using TypeManagers in datatype properties".

```
<input type="text" id="textbox1" name="textbox1" data-jtac-datatype="Integer" />
```

   Note: This attribute's value is case sensitive.

   *Note: This attribute is only validated within HTML 5. If you are using something else and demand validated markup, you can define the same value in the **datatype** property of Conditions that need it.*

   Note: jquery-validate requires both the id and name properties are defined.

4. Add the rules to each element on the page that needs validation.

   Always define the `data-val="true"` property on the HTML input element. This tells unobtrusive validation to locate validation rules in the tag.

   For each rule, specify `data-val-rulename=""`. This enables the rule.

   You can use rule names already defined by *jquery-validate* and those introduced by jTAC.

   Here are the rules introduced by jTAC: (* needs a TypeManager specified)

Rule name	Condition class
advrange	Conditions.Range *
advrequired	Conditions.Required
booleanlogic	Conditions.BooleanLogic
charactercount	Conditions.CharacterCount
comparetovalue	Conditions.CompareToValue *
comparetwoelements	Conditions.CompareTwoElements *
countselections	Conditions.CountSelections
datatypecheck	Conditions.DataTypeCheck *
difference	Conditions.Difference *
duplicateentry	Conditions.DuplicateEntry
regexp	Conditions.RegExp
requiredindex	Conditions.RequiredIndex
selectedindex	Conditions.SelectedIndex
wordcount	Conditions.WordCount

Here are examples:

```
<input type="text" id="textbox1" name="textbox1"
    data-jtac-datatype="Integer" data-val="true" data-val-advrequired="" />

<input type="text" id="textbox2" name="textbox1"
    data-jtac-datatype="Integer" data-val="true" data-val-advrequired="" />
```

Most conditions need you to assign property values. For example, the Range Condition needs **minimum** and **maximum** assigned. Use the **data-val-*rulename*-json** attribute to hold a JavaScript object in JSON format that identifies each property and its value.

```
<input type="text" id="textbox1" name="textbox1"
    data-jtac-datatype="Integer" data-val="true"
    data-val-advrange="" data-val-advrange-json="{'minimum': 1, 'maximum': 10}" />
```

Properties that take a data type specific value can be assigned as strings so long as you use the culture neutral format for that data type. This is most important for dates and times. (Integers and floating point values can be entered as numbers.)

Here are the culture neutral formats:

Type	Pattern	Examples
Integers	[-]digits	"1", "10000", "-1"
Float, Currency, Percent	[-]digits.digits	"1.0", "10000.0", "-1.0"
Date	yyyy-MM-dd	"2000-05-02"
Time of Day, Duration	H:mm:ss	"0:00:00", "16:30:21"
Date and Time	yyyy-MM-dd H:mm:ss	"2000-05-02 16:30:21"
Day Month	MM-dd	"05-02"
Month Year	yyyy-MM	"2000-05"

```
<input type="text" id="textbox1" name="textbox1"
    data-jtac-datatype="Date" data-val="true"
    data-val-advrange=""
    data-val-advrange-json="{'minimum': '2000-01-01', 'maximum': '2009-12-31'}" />
```

5. Add dependencies if needed. A dependency is a way to enable and disable the rule based on the state of other inputs on the page.

   Each Condition object has a **depends** property where you define the dependency rule.

```
<input type="text" id="textbox1" name="textbox1"
    data-jtac-datatype="Integer" data-val="true"
    data-val-advrange=""
    data-val-advrange-json="{'minimum': 1, 'maximum': 10}, depends:'rule goes here'" />
```

   The **depends** property can be defined in several ways:

   - As a string representing a *jQuery* selector. The *jQuery* selector is passed to $(selector). If it returns one or more results, the rule is enabled. This example requires checkbox1 to be checked.

```
depends:'#CheckBox1:checked'
```

   - As a Condition object. If the Condition evaluates as "success", validation runs.

     The object must be defined in JSON format, with the property "**jtacClass**" assigned to a string which is the name of the class. The remaining properties populate the Condition's properties. Always include the **elementId** property to specify which element is evaluated.

```
depends: {'jtacClass': 'Conditions.CompareToValue',
    'elementId': 'CheckBox1', 'operator': '=', 'valueToCompare': 'true'}
```

6. Override the default error message if desired. There are two ways.

Note: The default error messages use tokens to allow runtime substitution of values. These tokens may be enough to avoid replacing an error message. In addition, you can use the tokens in your own messages. See the documentation for each Condition to learn about its tokens. See the next step to learn how to replace the {LABEL} token.

Specify the message directly with rule

When you setup the rule in the HTML tag, the attribute data-val-*rulename* was assigned to an empty string. Replace that empty string with the desired error message.

```
<input type="text" id="textbox2" name="textbox1"
    data-jtac-datatype="Integer" data-val="true" data-val-advrequired="error message" />
```

Define error messages globally and make them localizable

Take advantage of jTAC's tools to replace default strings globally and localize them.

jTAC provides its *jTAC.Translations system* (included in **jTAC.js**) to register key and value pairs. All jTAC code that needs to show the user a string will ask the jTAC.Translations for a string that you defined, and only if not found will it use a hard-coded default.

See "jTAC.Translations system: Replacing strings shown to the user" to setup and customize strings.

7. All default error messages include the {LABEL} token. This will be replaced with values taken from one of these resources.

   - If there is a `<label for="elementid">` tag, its text will be used. Note: This is overridden by the items below. If used, it will be cleaned up, by removing HTML tags and non-textual lead and trailing characters.
   - Add the **data-msglabel** attribute to the element's HTML tag with the value to show.

```
<input type="text" id="textbox1" name="textbox1"
    data-jtac-datatype="Integer" data-msglabel="label text" />
```

   - If localizing, define a new key in the **\jTAC\Translations** scripts with the desired label. Then add the **data-msglabel-lookupkey** attribute assigned to your new key into the HTML tag.

```
<input type="text" id="textbox1" name="textbox1"
    data-jtac-datatype="Integer" data-msglabel-lookupkey="newkeyname" />
```

   - If none of the above, {LABEL} is replaced by "Field".

8. Establish the *jquery-validate* options on the <form> tag that contains the inputs.

   Use the **data-val-options** attribute with its value defining the options in JSON format.

```
<form method="post" id="Form1" data-val-options="{'onkeyup': false, 'debug': true}" >
```

   Several options take functions, which you cannot define in JSON. Instead, define those functions globally (in the window object) and pass their name as a string for the option. They will be resolved to the function. Remember that function names are case sensitive.

```
function mySubmitHandler(form) {
    // do something
}

<form method="post" id="Form1" data-val-options="{'submitHandler': 'mySubmitHandler'}" >
```

9. Specify the location of the error message on the page with a or similar container that identifies the input that supplies errors with the **data-valmsg-for** attribute.

```
<input type="text" id="textbox1" name="textbox1"
    data-jtac-datatype="Integer" data-val="true"
    data-val-advrange="" data-val-advrange-json="{'minimum': 1, 'maximum': 10}" />

<span data-valmsg-for="textbox1" ></span>
```

   Alternatively, you can define a function on the **errorPlacement** property of validation options.

10. Supply style sheet classes. They are found in the **\jTAC Appearance\jquery.validate.css** file. Either add the following tag or merge these styles into your own files.

```
<link href="/jTAC%20Appearance/jquery.validate.css" rel="stylesheet" type="text/css" />
```

WARNING: Modify the URL shown based on the location of the web page under the root folder.

You can use different class names by specifying the new name in the correct property of the options you setup in step 8.

Original class name	Property in options	Description
input-validation-error	inputErrorClass	The class assigned to the input elements showing an error.
input-validation-valid	inputValidClass	The class assigned to the input elements that are valid.
message-validation-error	messageErrorClass	Assigned to the container of the error message when there is an error.
message-validation-valid	messageValidClass	Assigned to the container of the error message when there is no error.
message-label	messageLabel	Replacement for the "{ERRORLABEL}" token within error messages.
label-validation-error	labelErrorClass	Assigned to the <label for="inputelement"> element associated with the error message.
container-validation-error	containerErrorClass	Assigned to elements with the data-jtac-valinputs attribute.

11. Your input fields will automatically change their style sheets when they reflect an error. This feature may need some setup.
    - The input fields use the class “input-validation-error”.
    - If you want to override the default class on a specific input, add the **data-jtac-errorclass** attribute to the input, and specify its value as the desired style sheet class.

```
<input type="text" id="textbox1" name="textbox1"     data-jtac-errorclass="customClass" />
```

12. Labels and container elements can also have their style sheet changed when their input(s) show an error.
    - Labels using the **for=** attribute will automatically work when **for=** specifies the input being validated. They are assigned the class “label-validation-error”.

```
<label for="textbox1" >First Name</label>
<input type="text" id="textbox1" name="textbox1"/>
```

    - For anything else, add the **data-jtac-valinputs** attribute to identify that its class will be changed to “container-validation-error”. If **data-jtac-valinputs** is assigned to "", it will change for any input that it contains. Otherwise assign a pipe delimited list of IDs to inputs that it responds to.

      Using a container that knows about all inputs.

```
<div data-jtac-valinputs="" >
    <input type="text" id="textbox1" name="textbox1"/>
</div>
```

      Using a container that does not contain the inputs.

```
<span data-jtac-valinputs="textbox1" >First Name</span>
<input type="text" id="textbox1" name="textbox1"/>
```

    - If you want to override the default class, add the **data-jtac-errorclass** attribute to the element, and specify its value as the desired style sheet class.

```
<label for="textbox1" data-jtac-errorclass="customClass" >First Name</label>
<input type="text" id="textbox1" name="textbox1" />
```

Loading scripts supporting jquery-validate

jTAC offers several ways to add scripts. This section shows using the merged format. If you want minified files, specify the **jTAC-min** folder. Otherwise specify the **jTAC** folder. For more, see "Understanding the available scripts files".

1. Add support for *jQuery* and *jquery-validate*. (The unobtrusive validation feature is optional.)

```
<script src="/jquery/jquery-#.#.#.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.unobtrusive.js" type="text/javascript"></script>
```

2. If you want to load a single script file with all of jTAC, use this:

```
<script src="/jTAC-min/Merged/jquery extensions/jtac-all.js" type="text/javascript"></script>
```

   Then skip to step 6. Otherwise continue with the next step.

3. Always load **core.js** before any other jTAC script file.

```
<script src="/jTAC-min/Merged/core.js" type="text/javascript"></script>
```

4. Include one of these files defining the desired Condition objects:

```
<script src="/jTAC-min/Merged/jquery extensions/validation-basic.js"
    type="text/javascript"></script>
```

   or

```
<script src="/jTAC-min/Merged/jquery extensions/validation-typical.js"
    type="text/javascript"></script>
```

   or

```
<script src="/jTAC-min/Merged/jquery extensions/validation-all.js"
    type="text/javascript"></script>
```

Condition class	basic	typical	all
Conditions.BooleanLogic			x
Conditions.CharacterCount			x
Conditions.CompareToValue *		x	x
Conditions.CompareTwoElements *		x	x
Conditions.CountSelections			x
Conditions.DataTypeCheck *	x	x	x
Conditions.Difference *			x
Conditions.DuplicateEntry			x
Conditions.Range *	x	x	x
Conditions.RegExp		x	x
Conditions.Required	x	x	x
Conditions.RequiredIndex			x
Conditions.SelectedIndex			x
Conditions.WordCount			x

   ** requires a TypeManager*

5. If using a Condition that requires a TypeManager, load one or more of these files defining the desired TypeManager objects:

```
<script src="/jTAC-min/Merged/typemanagers-numbers.js" type="text/javascript"></script>
```

   or

```
<script src="/jTAC-min/Merged/typemanagers-date-time.js" type="text/javascript"></script>
```

   or

```
<script src="/jTAC-min/Merged/typemanagers-date-time-all.js" type="text/javascript"></script>
```
or
```
<script src="/jTAC-min/Merged/typemanagers-strings-common.js" type="text/javascript"></script>
```
or
```
<script src="/jTAC-min/Merged/typemanagers-strings-common-all.js"
   type="text/javascript"></script>
```
or
```
<script src="/jTAC-min/Merged/typemanagers-all.js" type="text/javascript"></script>
```

TypeManager class	numbers	date-time	d-t-all	strings common	strings all	all
TypeManagers.Boolean						x
TypeManagers.CreditCardNumber					x	x
TypeManagers.Currency	x					x
TypeManagers.DateTime		x	x			x
TypeManagers.Date		x	x			x
TypeManagers.DayMonth			x			x
TypeManagers.Duration			x			x
TypeManagers.EmailAddress				x	x	x
TypeManagers.Float	x					x
TypeManagers.Integer	x	x	x			x
TypeManagers.MonthYear			x			x
TypeManagers.Percent	x					x
TypeManagers.PhoneNumber				x	x	x
TypeManagers.PostalCode				x	x	x
TypeManagers.String			x	x	x	x
TypeManagers.TimeOfDay		x	x			x
TypeManagers.Url					x	x

Note: If you choose to load individual condition script files, add the Merged/jquery extensions/jquery-validate-extensions.js file to install those conditions into jquery-validate. Do not use it when using the above validation-basic/typical/all.js files.

6. If you are need localization of number, date, or time TypeManagers, add the appropriate scripts. See "Localizing dates, times, and numbers".

   Here is an example using *jquery-globalize*. Add its script file first. Add any culture specific files from *jquery-globalize*. Then add the **TypeManagers\Culture engine for jquery-globalize.js** file. All of these files can be located below the TypeManager scripts.

```
<script src="/jquery-globalize/globalize.js" type="text/javascript"></script>
<script src="/jquery-globalize/cultures/globalize.culture.fr-FR.js"
   type="text/javascript"></script>
<script src="/jTAC-min/TypeManagers/Culture engine for jquery-globalize.js"
   type="text/javascript"></script>
```

7. If you are using dates and times, consider switching parsers from the default to the TypeManagers.PowerDateParser or TypeManagers.PowerTimeParser. If you do, here are the links for those classes.

```
<script src="/jTAC-min/TypeManagers/PowerDateParser.js" type="text/javascript"></script>
<script src="/jTAC-min/TypeManagers/PowerTimeParser.js" type="text/javascript"></script>
```

Example

```
<script src="/jquery/jquery-#.#.#.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.unobtrusive.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/core.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/jquery extensions/validation-typical.js"
   type="text/javascript"></script>
<script src="/jTAC-min/Merged/typemanagers-numbers.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/typemanagers-date-time.js" type="text/javascript"></script>
<script src="/jquery-globalize/globalize.js" type="text/javascript"></script>
<script src="/jquery-globalize/cultures/globalize.culture.fr-FR.js"
   type="text/javascript"></script>
<script src="/jTAC-min/TypeManagers/Culture engine for jquery-globalize.js"
   type="text/javascript"></script>
<script src="/jTAC-min/TypeManagers/PowerDateParser.js" type="text/javascript"></script>
```

The TypeManager classes

A **TypeManager** class understands data types, like integer, currency, date, time of day, and email address. Unique TypeManager classes are defined for each data type.

TypeManagers do a lot of things specific to each data type:

- Convert between string and native type with the `toValue()` and `toString()` methods. If you have a Connection object, its `toValueFromConnection()` simplifies retrieving the value from the connection before conversion.
- Check for validity of a string with the `isValid()` method.
- Test a character to see if its supported by the type (something used by the DataTypeEditor widget) with the `isValidChar()` method.
- Handles both culture specific and culture neutral formats. Its `toValueNeutral()` and `toStringNeutral()` are used to get and set the value of the hidden field used by the DataTypeEditor widget.
- Compares two values with its `compare()` function, converting each value from a string if needed.

Many Condition classes use TypeManagers. If their evaluation function requires one, its `_evaluateRule()` method gets the TypeManager in use from the Condition's **typeManager** property.

While all TypeManagers shipped with jTAC focus on numbers, dates, and times, you can use TypeManagers on string-oriented data types, like phone numbers, credit card numbers, and other types and have a strong pattern. If you develop a TypeManager like this, you will not need to create a separate Condition or extend support for validation. Instead, just use the Conditions.DataTypeCheck object, with its data type set to the name of your TypeManager.

In addition, when you create a TypeManager for strings, the DataTypeEditor widget will automatically support it, providing keystroke filtering and reformatting (string cleanup) when focus is lost.

Click on any of these topics to jump to them:

Locate a TypeManager

Here are the TypeManager classes supplied by jTAC. Click on their name to learn more about them. Each of these classes is defined in files of the **jTAC\TypeManagers**\ folder.

Class name	Purpose
TypeManagers.Boolean	Typically used with Conditions.CompareToValue to compare the state of a checkbox or radio button.
TypeManagers.CreditCardNumber	Use with credit card numbers. Validation uses Luhns algorithm.
TypeManagers.Currency	Optionally formats with the localized currency symbol and thousands separators.
TypeManagers.Date	Handles localized short, abbreviated and long date formats.
TypeManagers.DateTime	Date and time of day in the same string or Date object.
TypeManagers.DayMonth	A date that omits the year. Good for anniversary dates.
TypeManagers.Duration	Time value that can go up to 9999 hours.
TypeManagers.EmailAddress	Handles strings that reflect an email address. Supports multiple addresses.
TypeManagers.Float	Can specify a maximum number of decimal places and have it fill in trailing zeros. Supports various rounding modes.
TypeManagers.Integer	Handles integers.
TypeManagers.MonthYear	A date that omits the day of month. Often used for credit card expiry dates.
TypeManagers.Percent	Handles both integer and decimal forms of percents. Optionally includes the localized percent symbol.
TypeManagers.PhoneNumber	Handles strings that reflect a phone number. Supports different countries.
TypeManagers.PostalCode	Handles strings that reflect a postal code. Supports different countries.
TypeManagers.String	For strings when no other string oriented TypeManager is available.
TypeManagers.TimeOfDay	Localized. Can optionally show or hide the number of seconds.
TypeManagers.Url	Handles strings that reflect a URL.

Here are some base classes that you may use to create your own subclasses.

Class name	Description
TypeManagers.Base	The top-most ancestor of all TypeManagers.
TypeManagers.BaseCulture	For building TypeManagers that need localization data when parsing and formatting, such as dates, times, and numbers.
TypeManagers.BaseDatesAndTimes	Supports the JavaScript Date object to handle dates, times and durations. It also can use an integer with times and durations.
TypeManagers.BaseNumber	Supports numbers.
TypeManagers.BaseString	Supports string types.
TypeManagers.BaseStrongPatternString	Supports string types that can use a regular expression to validate them.
TypeManagers.BaseRegionString	Supports string types that have different rules based on the region (country).

Using TypeManagers in datatype properties

Throughout jTAC, you can specify the name of a TypeManager in a **datatype** property. For example, in the **data-jtac-datatype** attribute that you add to a textbox.

```
<input type="text" id="TextBox1" name="TextBox1" data-jtac-datatype="Currency" />
```

These properties take a case sensitive string. While you can assign the full class name of the TypeManager, jTAC defines aliases for these names. Aliases offer shorter names. They also setup the TypeManager instance with specific properties already set.

The following chart identifies all TypeManager class names and aliases that are included with jTAC.

"TypeManagers.Boolean" - default.

"Boolean" - default.

"TypeManagers.CreditCardNumber" - default

"CreditCardNumber" – default
"CreditCardNumber.AllBrands" – **brands** = null (does not check brand data)

"TypeManagers.Currency" - default

"Currency" - default
"Currency.Positive" - **allowNegatives** = false

"TypeManagers.Date" - default (Short date format, **dateFormat** = 0)

"Date" - default (**dateFormat** = 0)
"Date.Abbrev" - Abbreviated date format without day of week (**dateFormat** = 10)
"Date.Long" - Long date format without day of week (**dateFormat** = 20)

"TypeManagers.DateTime" - default (Short date format, **dateFormat** = 0)

"DateTime" - default (**dateFormat** = 0)
"DateTime.Abbrev" - Abbreviated date format without day of week (**dateFormat** = 10)
"DateTime.Long" - Long date format without day of week (**dateFormat** = 20)

"TypeManagers.DayMonth" - default (Short date format, **dateFormat** = 0)

"DayMonth" - default (**dateFormat** = 0)
"DayMonth.Abbrev" - Abbreviated date format without day of week (**dateFormat** = 10)
"DayMonth.Long" - Long date format without day of week (**dateFormat** = 20)

"TypeManagers.Duration" - default (always includes seconds, **timeFormat** = 10)

"Duration" - default (**timeFormat** = 10). Value is a Date object.
"Duration.NoSeconds" - Exclude seconds. (**timeFormat** = 11). Value is a Date object.
"Duration.NoZeroSeconds" - Omit seconds when zero (**timeFormat** = 12). Value is a Date object.
"Duration.InSeconds" - Value is a number where 1 is one second. Always shows seconds. (**timeFormat** = 10).
"Duration.InHours" - Value is a number where 1 is one hour. Always shows seconds. (**timeFormat** = 10).
"Duration.InSeconds.NoSeconds" - Value is a number where 1 is one second. Exclude seconds. (**timeFormat** = 11).
"Duration.InHours.NoSeconds" - Value is a number where 1 is one hour. Exclude seconds. (**timeFormat** = 11).
"Duration.InSeconds.NoZeroSeconds" - Value is a number where 1 is one second. Omit seconds when zero (**timeFormat** = 12).
"Duration.InHours.NoZeroSeconds" - Value is a number where 1 is one hour. Omit seconds when zero (**timeFormat** = 12).

"TypeManagers.EmailAddress" - default

"EmailAddress" - default
"EmailAddress.Multiple" - **multiple**= true

"TypeManagers.Float" - default

"Float" - default
"Float.Positive" - **allowNegatives** = false

"TypeManagers.Integer" - default

"Integer" - default
"Integer.Positive" - **allowNegatives** = false

"TypeManagers.MonthYear" - default (Short date format, **dateFormat** = 0)

"MonthYear" - default (Short date format, **dateFormat** = 0)
"MonthYear.Abbrev" - Abbreviated date format without day of week (**dateFormat** = 10)
"MonthYear.Long" - Long date format without day of week (**dateFormat** = 20)

"TypeManagers.Percent" - default

"Percent" - default
"Percent.Positive" - **allowNegatives** = false
"Percent.Integer" - Works with integer values (**maxDecimalPlaces** = 0)
"Percent.IntegerPositive" - Works with positive integer values (**maxDecimalPlaces** = 0, **allowNegatives** = false)

"TypeManagers.PhoneNumber" - default

"PhoneNumber" - default
"PhoneNumber.Canada" - **region** = "Canada"
"PhoneNumber.China" - **region** = "China"
"PhoneNumber.France" - **region** = "France"
"PhoneNumber.Germany" - **region** = "Germany"
"PhoneNumber.Japan" - **region** = "Japan"
"PhoneNumber.NorthAmerica" - **region** = "NorthAmerica"
"PhoneNumber.NorthAmerica.CountryCode" - **region** = "NorthAmerica.CountryCode"
"PhoneNumber.UnitedStates" - **region** = "UnitedStates"
"PhoneNumber.UnitedKingdom" - **region** = "UnitedKingdom"

"TypeManagers.PostalCode" - default

"PostalCode" - default
"PostalCode.Canada" - **region** = "Canada"
"PostalCode.China" - **region** = "China"
"PostalCode.France" - **region** = "France"
"PostalCode.Germany" - **region** = "Germany"
"PostalCode.Japan" - **region** = "Japan"
"PostalCode.NorthAmerica" - **region** = "NorthAmerica"
"PostalCode.UnitedStates" - **region** = "UnitedStates"
"PostalCode.UnitedKingdom" - **region** = "UnitedKingdom"

"TypeManagers.String" - default (case sensitive by default)

"String" - default
"String.caseins" – **caseIns** = `true`

"TypeManagers.TimeOfDay" - default (always includes seconds)

"TimeOfDay" - default (always includes seconds)
"TimeOfDay.NoSeconds" - Exclude seconds. (**timeFormat** = 1)
"TimeOfDay.NoZeroSeconds" - Omit seconds when zero (**timeFormat** = 2)
"TimeOfDay.InSeconds" - Value is a number where 1 is one second. Always shows seconds. (**timeFormat** = 10).
"TimeOfDay.InHours" - Value is a number where 1 is one hour. Always shows seconds. (**timeFormat** = 10).
"TimeOfDay.InSeconds.NoSeconds" - Value is a number where 1 is one second. Exclude seconds. (**timeFormat** = 11).

"TimeOfDay.InHours.NoSeconds" - Value is a number where 1 is one hour. Exclude seconds. (**timeFormat** = 11).

"TimeOfDay.InSeconds.NoZeroSeconds" - Value is a number where 1 is one second. Omit seconds when zero (**timeFormat** = 12).

"TimeOfDay.InHours.NoZeroSeconds" - Value is a number where 1 is one hour. Omit seconds when zero (**timeFormat** = 12).

"Url" - default

"Url" - default

"Url.FTP" - **uriScheme** = “ftp”

TypeManagers.Boolean

Data types supported:	**Boolean**
Alias names (case insensitive):	"**Boolean**"
Inherits from:	**TypeManagers.Base**
Source file:	**\jTAC\TypeManagers\Boolean.js**

Handles Boolean types. Often used in the Conditions.CompareToValue to compare a Boolean value to the value of a checkbox or radio button.

Set up

Unless you need to customize its properties, you don't need to do anything. When you specify the id of a checkbox or radio button in a Condition's **elementId**, it knows to use the TypeManagers.Boolean.

This is an example of using this TypeManager with the Conditions.CompareToValue when the checkbox's state determines if a textbox is required.

```
<input type="checkbox" id="CheckBox1" name="CheckBox1" />
<input type="text" id="TextBox1" name="TextBox1" data-val="true"
   data-val-advrequired="" data-val-advrequired-json=
   "{'depends': {'jtacClass': 'CompareToValue', 'valueToCompare': true, 'operator': '='} }" />
```

If you need to specify properties in unobtrusive validation, use either the element's **data-jtac-typemanager** attribute or the typeManager property on the Condition like this:

```
<input type="text" id="TextBox1" name="TextBox1" data-val="true"
   data-val-advrequired=""
   data-val-advrequired-json="{'depends': {'jtacClass': 'CompareToValue',
      'valueToCompare': true, 'operator': '=',
      'typeManager': {'jtacClass': 'boolean', 'numFalse': [0, -1] } } }" />
```

However, this is a rare case, where your input is not a checkbox or radio button and needs strings or non-standard numeric values converted to true and false.

TypeManagers.Boolean Properties

- **reFalse** (string or regex) - Regular expression used to convert a string into false. Must match valid strings representing "false".

  Can pass either a RegExp object or a string that is a valid regular expression pattern.

  It defaults to "`^(false)|(0)$`".

- **reTrue** (string or regex) - Regular expression used to convert a string into true. Must match valid strings representing "true".

  Can pass either a RegExp object or a string that is a valid regular expression pattern.

  It defaults to "`^(true)|(1)$`".

- **numFalse** (array of integers) - Array of numbers representing `false`. If `null` or an empty array, no numbers match.

  It defaults to [0]. (An array with the number zero.)

- **numTrue** (array of integers or true) - Array of numbers representing `true` or the Boolean value of `true`. If `null` or an empty array, no numbers match. If `true`, then all numbers not defined in **numFalse** match.

  It defaults to [1]. (An array with the number one.)

- **falseStr** (string) - The string representing "false". It defaults to "false".

- **trueStr** (string) - The string representing "true". It defaults to "true".

- **emptyStrFalse** (boolean) - When `true`, the empty string represents `false` in the `toValue()` function. It defaults to `true`.

TypeManagers.CreditCardNumber

Data types supported:	**string**
Alias names (case sensitive):	"**CreditCardNumber**", "**CreditCardNumber.AllBrands**"
Inherits from:	**TypeManagers.BaseNumber**
Source file:	**\jTAC\TypeManagers\CreditCardNumber.js**

Validates a credit card number against Luhn's algorithm.

In addition, it can limit the brand of the credit card supported and allow optional spaces.

Set up

By default, it limits the valid cards to specific brands identified by the **brands** property. This property is an object that identifies the prefix digits that make it recognizable as the brand, and the length of digits required. You can modify the existing **brands** object or replace it.

If you want to allow all brands, set **brands** to `null`.

Review the remaining properties to determine how you prefer to parse and format values.

If you want to allow the user to enter spaces, set **allowSeps** to a single space character.

Example: On the element using data-jtac-datatype

```
<input type="text" id="TextBox1" name="TextBox1" data-jtac-datatype="CreditCardNumber" />
```

Example: On the element using data-jtac-typemanager

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-typemanager="{'jtacClass': 'CreditCardNumber', 'brands': null}" />
```

or

```
<input type="text" id="TextBox1" name="TextBox1" data-jtac-datatype="CreditCardNumber"
   data-jtac-typemanager="{'brands': null}" />
```

Example: Creating a Condition without data-jtac-datatype

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.datatype = "CreditCardNumber";
```

Example: Creating a Condition without data-jtac-typemanager

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.typeManager = jTAC.create("CreditCardNumber");
cond.typeManager.brands = null;
```

TypeManagers.CreditCardNumber Properties

- **brands** (array) - Defines the signatures of various brands of credit cards. These are checked if defined. To disable this feature, assign brands to null.

  This list is prepopulated with populate cards. Add, edit, and delete as needed.

  Each item of the array is a JavaScript object with these properties:

  - **len** (integer) - If defined, it is the exact number of digits allowed.
  - **prefix** (string) - A pipe delimited list of digits that start the credit card number and are used to identify the brand.

  Here is the default:

```
[{len: 16, prefix: '51|52|53|54|55'}, // Mastercard
 {len: 13, prefix: '4'}, // Visa-13char
 {len: 16, prefix: '4'}, // Visa-16char
```

```
{len: 15, prefix: '34|37'}, // American Express
{len: 14, prefix: '300|301|302|303|305|36|38'}, // Diners Club/Carte Blanche
{len: 16, prefix: '6011'}] // Discover
```

- **allowSeps** (char) - Assign to a single character that is a legal value for the user to use while typing, such as space or minus. It defaults to "" (which means not allowed).

TypeManagers.Currency

Data types supported:	**number**
Alias names (case sensitive):	"**Currency**", "**Currency.Positive**"
Inherits from:	**TypeManagers.BaseFloat**
Source file:	**\jTAC\TypeManagers\Currency.js**

Supports float as the native type. Strings are formatted as currencies. There are several formatting and parsing options to provide flexibility. Supports localization.

Set up

Always consider if localization is needed. See "Localizing dates, times, and numbers". If it matters, set the culture's name in the **cultureName** property.

Review the remaining properties to determine how you prefer to parse and format values.

Example: On the element using data-jtac-datatype

```
<input type="text" id="TextBox1" name="TextBox1" data-jtac-datatype="Currency.Positive" />
```

Example: On the element using data-jtac-typemanager

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-typemanager="{'jtacClass': 'Currency', 'allowNegatives': false}" />
```

or

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="Currency" data-jtac-typemanager="{'allowNegatives': false}" />
```

Example: Creating a Condition without data-jtac-datatype

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.datatype = "Currency.Positive";
```

Example: Creating a Condition without data-jtac-typemanager

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.typeManager = jTAC.create("Currency");
cond.typeManager.allowNegatives = false;
```

TypeManagers.Currency Properties

- **cultureName** (string) - The globalization name for the culture used by *jquery-globalize*. For example, "en-US", "fr-CA".
- **showCurrencySymbol** (boolean) - Determines if converting from native to string shows the culture specific currency symbol. It defaults to `true`.
- **allowCurrencySymbol** (boolean) - Determines if converting from string to native type reports an error if the currency symbol is found. It defaults to `true`.
- **useDecimalDigits** (boolean) - Determines if the number of decimal digits is constrained to the culture (culture.numberFormat.currency.decimals) or the user can add more digits. It defaults to `true`.
- **hideDecimalWhenZero** (boolean) - Determines a number that has only zeros in the decimal section shows the decimal part. For example, when the currency is 12.00, "12" is shown while 12.1 shows "12.10".
- **allowNegatives** (boolean) - Property determines if a negative number is valid. When it is not, `isValidChar()` and `isValid()` functions both report errors when they find a negative char or number. It defaults to `true`.
- **showGroupSep** (boolean) - Determines if the `toString()` function includes group separator character (aka "thousands separator"). It defaults to `true`.

TypeManagers.Date

Data types supported:	**Date object**
Alias names (case sensitive):	**“Date”, “Date.Short”, “Date.Abbrev”, “Date.Long”**
Inherits from:	**TypeManagers.BaseDatesAndTimes**
Source file:	**\jTAC\TypeManagers\Date.js**

Its native type is the JavaScript Date object, working with its date component, but not its time of day. Supports localization.

Its parser is limited to the short date pattern and is fairly strict. To provide more flexibility, you can switch to the TypeManagers.PowerDateParser or one of your own.

Set up

Always consider if localization is needed. See “Localizing dates, times, and numbers”. If it matters, set the culture’s name in the **cultureName** property.

There are a number of date patterns you can use to parse and reformat the input, as determined by the **dateFormat** property. The typical pattern is called “Short”, which uses digits and date separator characters, such as “MM/dd/yyyy” and “d M yyyy”. This is used by default.

The **dateFormat** property lets you work with abbreviated and long date patterns. Both include the month name, in abbreviated and full name formats. However, the built-in parser does not support them. If you want to use them, switch to the TypeManagers.PowerDateParser.

Here are values supported by **dateFormat**:

0 - Short date pattern with all digits. Ex: dd/MM/yyyy

1 - Short date pattern with abbreviated month name. Ex: dd/MMM/yyyy

2 - Short date pattern with abbreviated month name. Ex: dd/MMM/yyyy

Month name is shown in uppercase only when reformatting. (The parser supports either case.)

10 - Abbreviated date pattern. Ex: MMM dd, yyyy

20 - Long date pattern. Ex: MMMM dd, yyyy

100 - Culture neutral and sortable short format. Always uses "yyyy-MM-dd".

Example: On the element using data-jtac-datatype

```
<input type="text" id="TextBox1" name="TextBox1" data-jtac-datatype="Date" />
```

Example: On the element using data-jtac-typemanager

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-typemanager="{'jtacClass': 'Date', 'dateFormat': 10}" />
```

or

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="Date" data-jtac-typemanager="{'dateFormat': 10}" />
```

Example: Creating a Condition without data-jtac-datatype

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.datatype = "Date";
```

Example: Creating a Condition without data-jtac-typemanager

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.typeManager = jTAC.create("Date");
cond.typeManager.dateFormat = 10;
```

TypeManagers.Date Properties

- **dateFormat** (integer) - Determines how to parse and reformat the value.

  Here are its values:

  - 0 - Short date pattern with all digits. Ex: dd/MM/yyyy
  - 1 - Short date pattern with abbreviated month name. Ex: dd/MMM/yyyy
  - 2 - Short date pattern with abbreviated month name. Month name is shown in uppercase only when reformatting. (The parser supports either case.) Ex: dd/MMM/yyyy
  - 10 - Abbreviated date pattern. Ex: MMM dd, yyyy
  - 100 - Culture neutral and sortable short format. Always uses "yyyy-MM-dd".

- **useUTC** (boolean) - When `true`, the Date object is in UTC format. You should only pass UTC formatted Date objects to `toString()` and expect UTC format back from `toValue()`. When `false`, it is in local format.

  It defaults to `false`.

- **twoDigitYear** (boolean) - When `true`, two digit years are supported and converted to 4 digit years. When `false`, two digit years are an error. It defaults to `true`.

  For example, “05/31/65” is treated as 5/31/1965 and “05/31/10” is treated as 5/31/2010.

- **nativeParser** (boolean) – *Only when using a plug-in parser.* When `false`, use the plug-in parser. When `true`, use the native parser. It defaults to `false`.

- **parserOptions** (object) – *Only when using a plug-in parser*. Options specific to the plug-in parser. For TypeManagers.PowerDateParser, see “parserOptions Properties”.

TypeManagers.DateTime

Data types supported:	**Date object**
Alias names (case sensitive):	**"DateTime", "DateTime.Short", "DateTime.Abbrev", "DateTime.Long"**
Inherits from:	**TypeManagers.BaseDatesAndTimes**
Source file:	**\jTAC\TypeManagers\DateTime.js**
Requires:	TypeManagers.Date **and TypeManagers.TimeOfDay**

Its native type is the JavaScript Date object, working with both date and time of day components. Supports localization.

This class uses the TypeManagers.Date and TypeManager.TimeOfDay classes to do most of the work. It primarily splits a string into date and time parts for use by the other classes, and combines the strings returned by those classes when formatting.

TypeManagers.Date and TypeManagers.TimeOfDay use a fairly strict parser. If you want to support abbreviated or long date formats, or some other ways to let the user enter values, add the TypeManagers.PowerDateParser and TypeManagers.PowerTimeParser.

Set up

Always consider if localization is needed. See "Localizing dates, times, and numbers". If it matters, set the culture's name in the **cultureName** property.

All date parsing and formatting rules are available on a TypeManagers.Date object assigned to the **dateOptions** property. Modify its properties, including **dateFormat**. See "TypeManagers.Date Properties".

All time parsing and formatting rules are available on a TypeManagers.TimeOfDay object assigned to the **timeOptions** property. Modify its properties, including **timeFormat**. See "TypeManagers.TimeOfDay Properties".

If the user is allowed to omit the time of day, set **timeRequired** to `false`.

Example: On the element using data-jtac-datatype

```
<input type="text" id="TextBox1" name="TextBox1" data-jtac-datatype="DateTime" />
```

Example: On the element using data-jtac-typemanager

```
<input type="text" id="TextBox1" name="TextBox1" data-jtac-typemanager=
  "{'jtacClass': 'DateTime', 'dateOptions': {'dateFormat': 10}}, 'timeOptions': {'timeFormat:2'}"
/>
```

or

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="DateTime"
   data-jtac-typemanager="{'dateOptions': {'dateFormat': 10}, 'timeOptions': {'timeFormat:2'}}" />
```

Example: Creating a Condition without data-jtac-datatype

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.datatype = "DateTime";
```

Example: Creating a Condition without data-jtac-typemanager

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.typeManager = jTAC.create("DateTime");
cond.typeManager.dateOptions.dateFormat = 10;
cond.typeManager.timeOptions.timeFormat = 10;
```

TypeManagers.DateTime Properties

- **dateOptions** (TypeManagers.Date object) – The TypeManagers.Date object used to handle the work for the date part. Modify its properties, including **dateFormat**. See "TypeManagers.Date Properties".

  You can set this property to an object hosting the same-named properties as on the TypeManagers.Date object without actually creating a TypeManagers.Date object. It will copy those property values onto the existing TypeManagers.Date object. This technique is used here:

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="DateTime" data-jtac-typemanager="{'dateOptions': {'dateFormat': 10}}"
/>
```

  and here:

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.typeManager = jTAC.create("DateTime");
cond.typeManager.dateOptions = {dateFormat: 10};
```

- **timeOptions** (TypeManagers.TimeOfDay object) – The TypeManagers.TimeOfDay object used to handle the work for the time of day part. Modify its properties, including **timeFormat**. See "TypeManagers.TimeOfDay Properties".

  You can set this property to an object hosting the same-named properties as on the TypeManagers.TimeOfDay object without actually creating a TypeManagers.TimeOfDay object. It will copy those property values onto the existing TypeManagers.TimeOfDay object. This technique is used here:

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="DateTime" data-jtac-typemanager="{'timeOptions': {'timeFormat:2'}}" />
```

  and here:

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.typeManager = jTAC.create("DateTime");
cond.typeManager.timeOptions.timeFormat = 10;
```

- **timeRequired** (boolean) - Determines if only the date is supplied, is the value valid, where the time is 0:0:0. When `true`, the time is required and an error is reported. When `false`, time is considered 0:0:0.

  It defaults to `true`.

- **useUTC** (boolean) - When `true`, the Date object is in UTC format. You should only pass UTC formatted Date objects to `toString()` and expect UTC format back from `toValue()`. When `false`, it is in local format.

  It defaults to `false`.

TypeManagers.DayMonth

Data types supported:	**Date object**
Alias names (case sensitive):	**“DayMonth”, “DayMonth.Short”, “DayMonth.Abbrev”, “DayMonth.Long”**
Inherits from:	**TypeManagers.BaseDatesAndTimes**
Source file:	**\jTAC\TypeManagers\DayMonth.js**

Its native type is the JavaScript Date object, working with its date component, but not its time of day. It ignores the year and will create a Date object using the current year when converting string to native value. Supports localization.

Its parser is limited to the short date pattern and is fairly strict. To provide more flexibility, you can switch to the TypeManagers.PowerDateParser or one of your own.

Set up

Always consider if localization is needed. See “Localizing dates, times, and numbers”. If it matters, set the culture’s name in the **cultureName** property.

There are a number of date patterns you can use to parse and reformat the input, as determined by the **dateFormat** property. The typical pattern is called “Short”, which uses digits and date separator characters, such as “MM/dd” and “d M”. This is used by default.

The **dateFormat** property lets you work with abbreviated and long date patterns. Both include the month name, in abbreviated and full name formats. However, they are not supported by the native parser. To use them, switch to the TypeManagers.PowerDateParser.

Example: On the element using data-jtac-datatype

```
<input type="text" id="TextBox1" name="TextBox1" data-jtac-datatype="DayMonth" />
```

Example: On the element using data-jtac-typemanager

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-typemanager="{'jtacClass': 'DayMonth', 'dateFormat': 10}" />
```

or

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="DayMonth" data-jtac-typemanager="{'dateFormat': 10}" />
```

Example: Creating a Condition without data-jtac-datatype

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.datatype = "DayMonth";
```

Example: Creating a Condition without data-jtac-typemanager

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.typeManager = jTAC.create("DayMonth");
cond.typeManager.dateFormat = 10;
```

TypeManagers.DayMonth Properties

- **dateFormat** (integer) - Determines how to parse and reformat the value.

  Here are its values:

  - 0 - Short date pattern with all digits. Ex: dd/MM
  - 1 - Short date pattern with abbreviated month name. Ex: dd/MMM
  - 2 - Short date pattern with abbreviated month name. Ex: dd/MMM

    Month name is shown in uppercase only when reformatting. (The parser supports either case.)
  - 10 - Abbreviated date pattern. Ex: MMM dd, yyyy
  - 20 - Long date pattern. Ex: MMMM dd, yyyy

- **useUTC** (boolean) - When `true`, the Date object is in UTC format. You should only pass UTC formatted Date objects to `toString()` and expect UTC format back from `toValue()`. When `false`, it is in local format.

  It defaults to `false`.

- **nativeParser** (boolean) – *Only when using a plug-in parser.* When `false`, use the plug-in parser. When `true`, use the native parser. It defaults to `false`.

- **parserOptions** (object) – *Only when using a plug-in parser*. Options specific to the plug-in parser. For TypeManagers.PowerDateParser, see "parserOptions Properties".

TypeManagers.Duration

Data types supported:	**Date object and integer**
Alias names (case sensitive):	**“Duration”, “Duration.InSeconds”, “Duration.InHours”, “Duration.NoSeconds”, “Duration.NoZeroSeconds”, “Duration.InSeconds.NoSeconds”, “Duration.InSeconds.NoZeroSeconds”, “Duration.InHours.NoSeconds”, “Duration.InHours.NoZeroSeconds”**
Inherits from:	**TypeManagers.BaseDatesAndTimes**
Source file:	**\jTAC\TypeManagers\Duration.js**

Its native type is the JavaScript Date object, working with its time of day component but not date. Because a Date object is limited to 24 hours, if you want a duration with a larger limit, you can use a number type to hold the number of seconds or hours starting at 0:00:00. When using the number to hold hours, the value is a decimal where 1.0 = 1 hour, 1.5 = 1:30:00, etc. Supports localization.

Its parser is fairly strict. To provide more flexibility, you can switch to the TypeManagers.PowerTimeParser or one of your own.

Set up

Always consider if localization is needed. See “Localizing dates, times, and numbers”. If it matters, set the culture’s name in the **cultureName** property.

If you want to use a number to hold the duration, consider using these alias names:

“Duration.InSeconds” – an integer where 1 = 1 second

```
var tm = jTAC.create("Duration.InSeconds");
```

“Duration.InHours” – a float where 1 = 1 hour

```
var tm = jTAC.create("Duration.InHours");
```

Alternatively, set the **valueAsNumber** property to true and **timeOneEqualsSeconds** to 1 (for seconds) or 3600 (for hours).

There are several time format patterns that you can defined in the **timeFormat** property:

10 - Includes seconds. Ex: HH:mm:ss

11 - Omits seconds. Ex: HH:mm

12 - Same as 0 except it omits seconds when they are 0 (while formatting a string).

Example: On the element using data-jtac-datatype

```
<input type="text" id="TextBox1" name="TextBox1" data-jtac-datatype="Duration" />
```

Example: On the element using data-jtac-typemanager

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-typemanager="{'jtacClass': 'Duration', 'timeFormat': 12}" />
```

or

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="Duration" data-jtac-typemanager="{'timeFormat': 12}" />
```

Example: Creating a Condition without data-jtac-datatype

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.datatype = "Duration";
```

Example: Creating a Condition without data-jtac-typemanager

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.typeManager = jTAC.create("Duration");
cond.typeManager.timeFormat = 12;
```

TypeManagers.Duration Properties

- **timeFormat** (integer) - Determines how to setup the time pattern. It is also used to format a string.

  Here are its values:

  - 10 - Includes seconds. Ex: HH:mm:ss. This is the default.
  - 11 - Omits seconds. Ex: HH:mm
  - 12 - Same as 0 except it omits seconds when they are 0 (while formatting a string).
  - 100 - Culture neutral and sortable format with seconds. Always uses "HHHH:mm:ss".
  - 101 – Culture neutral and sortable format without seconds. Always uses “HHHH:mm”.

- **maxHours** (integer) - The upper limit for number of hours allowed. It defaults to 9999.
- **timeOneEqualsSeconds** (integer) - Used when the value is a number representing only time. Determines how many seconds is represented by a value of 1.

  For 1 hour, use 3600. For 1 minute, use 60. For 1 second, use 1.

- **valueAsNumber** (boolean) - When `true`, it works with number types. When `false`, it works with Date objects.

  It defaults to `false`.

- **parseStrict** (boolean) - When this is `false`, the native parser lets you omit or incorrectly position the AM/PM designator. When `true`, it requires the AM/PM designator if its part of the time pattern, and it must be in the same location as that pattern.

  It defaults to `false`.

- **parseTimeRequires** (string) - When the time pattern has seconds, set this to "`s`" to require seconds be entered. Valid: “1:00:00” but not “1:00” or “1”.

  When "`m`", minutes are required but not seconds. Valid: “1:00” and “1:00:00” but not “1”.

  When "`h`", only hours are required. Valid: “1”, “1:00”, “1:00:00”.

  It defaults to "`h`".

- **useUTC** (boolean) - When `true`, the Date object is in UTC format. You should only pass UTC formatted Date objects to `toString()` and expect UTC format back from `toValue()`. When `false`, it is in local format.

  It defaults to `false`.

- **nativeParser** (boolean) – *Only when using a plug-in parser*. When `false`, use the plug-in parser. When `true`, use the native parser. It defaults to `false`.
- **parserOptions** (object) – *Only when using a plug-in parser*. Options specific to the plug-in parser. For TypeManagers.PowerTimeParser, see “parserOptions Properties”.

TypeManagers.EmailAddress

Data types supported:	**string**
Alias names (case sensitive):	"**EmailAddress**", "**EmailAddress.Multiple**"
Inherits from:	TypeManagers.BaseStrongPatternString
Source file:	**\jTAC\TypeManagers\EmailAddress.js**

Validates the pattern of an Email using a regular expression. You can optionally support multiple email addresses, defining the desired delimiters between them.

Set up

If you want to allow multiple email addresses, set **multiple** to `true`. By default, it only allows semicolon and optional space between emails. You can change this separator by specifying a regular expression pattern in **delimiterRE**.

Example: On the element using data-jtac-datatype

```
<input type="text" id="TextBox1" name="TextBox1" data-jtac-datatype="EmailAddress " />
```

Example: On the element using data-jtac-typemanager

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-typemanager="{'jtacClass': 'EmailAddress', 'multiple': true}" />
```

or

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="EmailAddress" data-jtac-typemanager="{'multiple': true}" />
```

Example: Creating a Condition without data-jtac-datatype

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.datatype = "EmailAddress.Multiple";
```

Example: Creating a Condition without data-jtac-typemanager

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.typeManager = jTAC.create("EmailAddress");
cond.typeManager.multiple = true;
```

TypeManagers.EmailAddress Properties

- **multiple** (boolean) -Determines if multiple email addresses are permitted. When `true`, they are. It defaults to `false`.
- **delimiterRE** (string) - A regular expression pattern that determines the delimiter between multiple email addresses. As a regular expression, be sure to escape any character that must be a kept as is.

  It defaults to "`;[ ]?`". It permits an optional space.

TypeManagers.Float

Data types supported:	**number**
Alias names (case sensitive):	"**Float**", "**Float.Positive**"
Inherits from:	**TypeManagers.BaseNumber**
Source file:	**\jTAC\TypeManagers\Float.js**

Supports float as the native type. Strings are formatted as decimal numbers. There are formatting and parsing options to provide flexibility. Supports localization.

Set up

Always consider if localization is needed. See "Localizing dates, times, and numbers". If it matters, set the culture's name in the **cultureName** property.

It is common to require only positive numbers. You can either set the **allowNegatives** property to `false` or specify "Float.Positive" in the **datatype** and **jtacClass** properties used throughout jTAC.

Review the remaining properties to determine how you prefer to parse and format values.

Example: On the element using data-jtac-datatype

```
<input type="text" id="TextBox1" name="TextBox1" data-jtac-datatype="Float.Positive" />
```

Example: On the element using data-jtac-typemanager

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-typemanager="{'jtacClass': 'Float', 'allowNegatives': false}" />
```

or

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="Float" data-jtac-typemanager="{'allowNegatives': false}" />
```

Example: Creating a Condition without data-jtac-datatype

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.datatype = "Float.Positive";
```

Example: Creating a Condition without data-jtac-typemanager

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.typeManager = jTAC.create("Float");
cond.typeManager.allowNegatives = false;
```

TypeManagers.Float Properties

- **cultureName** (string) - The globalization name for the culture used by *jquery-globalize*. For example, "en-US", "fr-CA".
- **trailingZeroDecimalPlaces** (integer) - Determines how many trailing decimal places should appear when reformatting a float value to a string. It defaults to 1.
- **maxDecimalPlaces** (integer) - Determines the maximum number of decimal digits that are legal. If there are more, it is either a validation error or rounded, depending on **roundMode**.

  Set to 0 or `null` to ignore this property. It defaults to `null`.
- **roundMode** (integer) - When **maxDecimalPlaces** is set and it's value is exceeded, this determines how to round.
  - 0 - Point5: round to the next number if .5 or higher; round down otherwise
  - 1 - Currency: round to the nearest even number.
  - 2 - Truncate: drop any decimals after **maxDecimalPlaces**; largest integer less than or equal to a number.
  - 3 - Ceiling: round to the nearest even number.

  - 4 - NextWhole: Like ceiling, but negative numbers are rounded lower instead of higher
  - `null` – Error: Throw an exception to report an error. Conditions will intercept this exception in their `evaluate()` function, returning either 0 or -1 depending on their rules. This is the default.

- **allowNegatives** (boolean) - Property determines if a negative number is valid. When it is not, `isValidChar()` and `isValid()` functions both report errors when they find a negative char or number. It defaults to `true`.
- **showGroupSep** (boolean) - Determines if the `toString()` function includes group separator character (aka "thousands separator"). It defaults to `true`.

TypeManagers.Integer

Data types supported:	**integer**
Alias names (case sensitive):	"**Integer**", "**Integer.Positive**"
Inherits from:	**TypeManagers.BaseNumber**
Source file:	**\jTAC\TypeManagers\Integer.js**

Supports an integer number as the native type. (If you pass a decimal number to its `toString()` function, it will report an error.) There are several formatting and parsing options to provide flexibility.

Set up

Always consider if localization is needed. See "Localizing dates, times, and numbers". If it matters, set the culture's name in the **cultureName** property.

It is common to require only positive numbers. You can either set the **allowNegatives** property to `false` or specify "Integer.Positive" in the **datatype** and **jtacClass** properties used throughout jTAC.

Review the remaining properties to determine how you prefer to parse and format values.

Example: On the element using data-jtac-datatype

```
<input type="text" id="TextBox1" name="TextBox1" data-jtac-datatype="Integer.Positive" />
```

Example: On the element using data-jtac-typemanager

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-typemanager="{'jtacClass': 'Integer', 'allowNegatives': false}" />
```

or

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="Integer" data-jtac-typemanager="{'allowNegatives': false}" />
```

Example: Creating a Condition without data-jtac-datatype

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.datatype = "Integer.Positive";
```

Example: Creating a Condition without data-jtac-typemanager

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.typeManager = jTAC.create("Integer");
cond.typeManager.allowNegatives = true;
```

TypeManagers.Integer Properties

- **cultureName** (string) - The globalization name for the culture used by *jquery-globalize*. For example, "en-US", "fr-CA".
- **allowNegatives** (boolean) - Property determines if a negative number is valid. When it is not, `isValidChar()` and `isValid()` functions both report errors when they find a negative char or number. It defaults to `true`.
- **showGroupSep** (boolean) - Determines if the `toString()` function includes group separator character (aka "thousands separator"). It defaults to `true`.
- **fillLeadZeros** (integer) - Provides additional formatting when converting an integer to text by adding lead zeros. It adds enough lead zeroes to make the length of the text match the value of this property. When 0, it is not used. It defaults to 0.

TypeManagers.MonthYear

Data types supported:	**Date object**
Alias names (case sensitive):	**"MonthYear", "MonthYear.Short", "MonthYear.Abbrev", "MonthYear.Long"**
Inherits from:	**TypeManagers.BaseDatesAndTimes**
Source file:	**\jTAC\TypeManagers\MonthYear.js**

Its native type is the JavaScript Date object, working with its date component, but not its time of day. It omits the day of the month, assigning a value of 1 for day of month when converting from string to Date object.

Its parser is limited to the short date pattern and is fairly strict. To provide more flexibility, you can switch to the TypeManagers.PowerDateParser or one of your own.

Set up

Always consider if localization is needed. See "Localizing dates, times, and numbers". If it matters, set the culture's name in the **cultureName** property.

There are a number of date patterns you can use to parse and reformat the input, as determined by the **dateFormat** property. The typical pattern is called "Short", which uses digits and date separator characters, such as "MM/yyyy" and "M yyyy". This is used by default.

The **dateFormat** property lets you work with abbreviated and long date patterns. Both include the month name, in abbreviated and full name formats. However, this is not supported with the native parser. To use it, switch to the TypeManagers.PowerDateParser.

```
<input type="text" id="TextBox1" name="TextBox1" data-jtac-datatype="MonthYear" />
```

Example: On the element using data-jtac-typemanager

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-typemanager="{'jtacClass': 'MonthYear', 'dateFormat': 10}" />
```

or

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="MonthYear" data-jtac-typemanager="{'dateFormat': 10}" />
```

Example: Creating a Condition without data-jtac-datatype

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.datatype = "MonthYear";
```

Example: Creating a Condition without data-jtac-typemanager

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.typeManager = jTAC.create("MonthYear");
cond.typeManager.dateFormat = 10;
```

TypeManagers.MonthYear Properties

- **dateFormat** (integer) - Determines how to parse and reformat the value.

  Here are its values:

  - 0 - Short date pattern with all digits. Ex: MM/yyyy
  - 1 - Short date pattern with abbreviated month name. Ex: MMM/yyyy
  - 2 - Short date pattern with abbreviated month name. Ex: MMM/yyyy
  - Month name is shown in uppercase only when reformatting. (The parser supports either case.)
  - 10 - Abbreviated date pattern. Ex: MMM, yyyy
  - 20 - Long date pattern. Ex: MMMM, yyyy

- **useUTC** (boolean) - When `true`, the Date object is in UTC format. You should only pass UTC formatted Date objects to `toString()` and expect UTC format back from `toValue()`. When `false`, it is in local format.

  It defaults to `false`.

- **twoDigitYear** (boolean) - When `true`, two digit years are supported and converted to 4 digit years. When `false`, two digit years are an error. It defaults to `true`.

  For example, “05/31/65” is treated as 5/31/1965 and “05/31/10” is treated as 5/31/2010.

- **nativeParser** (boolean) – *Only when using a plug-in parser*. When `false`, use the plug-in parser. When `true`, use the native parser. It defaults to `false`.

- **parserOptions** (object) – *Only when using a plug-in parser*. Options specific to the plug-in parser. For TypeManagers.PowerDateParser, see “parserOptions Properties”.

TypeManagers.Percent

Data types supported:	**number**
Alias names (case sensitive):	"**Percent**", "**Percent.Positive**", "**Percent.Integer**", "**Percent.Integer.Positive**"
Inherits from:	**TypeManagers.BaseNumber**
Source file:	**\jTAC\TypeManagers\Percent.js**

Supports number as the native type. It can represent values either as an integer or decimal number. Strings are formatted as a percentage. There are several formatting and parsing options to provide flexibility. Supports localization.

Set up

Always consider if localization is needed. See "Localizing dates, times, and numbers". If it matters, set the culture's name in the **cultureName** property.

Use **maxDecimalPlaces** set to 0 to keep percents as integers. Use the alias name "Percent.Integer" to quickly define this.

If using floating point numbers, leave **maxDecimalPlaces** set to null to show all available decimal places, or to a specific value to impose a limit.

A native value of 1 can mean 100% or 1% based on the usage. Use the **oneEqualsOneHundred** to determine which.

It is common to require only positive numbers. You can either set the **allowNegatives** property to `false` or specify "Percent.Positive" or "Percent.Integer.Positive" in the **datatype** and **jtacClass** properties used throughout jTAC.

Review the remaining properties to determine how you prefer to parse and format values.

Example: On the element using data-jtac-datatype

```
<input type="text" id="TextBox1" name="TextBox1" data-jtac-datatype="Percent.Positive" />
```

Example: On the element using data-jtac-typemanager

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-typemanager="{'jtacClass': 'Percent', 'allowNegatives': false}" />
```

or

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="Percent" data-jtac-typemanager="{'allowNegatives': false}" />
```

Example: Creating a Condition without data-jtac-datatype

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.datatype = "Percent.Positive";
```

Example: Creating a Condition without data-jtac-typemanager

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.typeManager = jTAC.create("Percent");
cond.typeManager.allowNegatives = true;
```

TypeManagers.Percent Properties

- **cultureName** (string) - The globalization name for the culture used by *jquery-globalize*. For example, "en-US", "fr-CA".
- **showPercentSymbol** (boolean) - Determines if converting from native to string shows the culture specific percent symbol. It defaults to `true`.
- **allowPercentSymbol** (boolean) - Determines if converting from string to native type reports an error if the percent symbol is found. It defaults to `true`.
- **allowNegatives** (boolean) - Property determines if a negative number is valid. When it is not, `isValidChar()` and `isValid()` functions both report errors when they find a negative char or number. It defaults to `true`.

- **showGroupSep** (boolean) - Determines if the `toString()` function includes group separator character (aka "thousands separator"). It defaults to `true`.
- **trailingZeroDecimalPlaces** (integer) - Determines how many trailing decimal places should appear when reformatting a float value to a string. It defaults to 1.
- **maxDecimalPlaces** (integer) - Determines the maximum number of decimal digits that are legal. If there are more, it is either a validation error or rounded, depending on **roundMode**.

  Set to 0 or `null` to ignore this property. It defaults to `null`.
- **roundMode** (integer) - When **maxDecimalPlaces** is set and it's value is exceeded, this determines how to round.
  - 0 - Point5: round to the next number if .5 or higher; round down otherwise
  - 1 - Currency: round to the nearest even number.
  - 2 - Truncate: drop any decimals after **maxDecimalPlaces**; largest integer less than or equal to a number.
  - 3 - Ceiling: round to the nearest even number.
  - 4 - NextWhole: Like ceiling, but negative numbers are rounded lower instead of higher
  - `null` – Error: Throw an exception to report an error. Conditions will intercept this exception in their `evaluate()` function, returning either 0 or -1 depending on their rules. This is the default.
- **oneEqualsOneHundred** (boolean) - Determines if the numeric value of 1.0 is shown as 1% or 100%, but only when using floating point values. If **maxDecimalPlaces** = 0, it is not used.

  When `true`, a value of 1 is shown as 100%. When false, a value of 1 is shown as 1%.

  It defaults to `true`.

TypeManagers.PhoneNumber

Data types supported:	**string**
Alias names (case sensitive):	"**PhoneNumber**" and region specific values in this pattern: "**PhoneNumber.*regionname***"
Inherits from:	**TypeManagers.BaseRegionString**
Source file:	**\jTAC\TypeManagers\PhoneNumber.js**

Supports the data type for phone numbers, which are strings with a specific pattern. The pattern differs based on the region (country usually).

Validates the pattern of a phone number using a region specific regular expression. Also supports region specific character validation for the DataTypeEditor widget. It powers the DataTypeEditor widget in several ways: key filtering and reformatting. Reformatting is demonstrated in the United States phone number pattern as when the user enters "12345678901", it reformats to "1(234)5678901.

Regional patterns of phone numbers are defined in `window.jTAC_PhoneNumberRegionData` object. This object is fully customizable. You can replace its regular expression for validation or even provide a function that validates. You can code how to reformat the string too. See the TypeManagers.BaseRegionString for more.

To add, you can use this:

```
window.jTAC_PhoneNumberRegionData["regionname"] = {pattern: value, more properties};
```

To replace the **pattern** property:

```
window.jTAC_PhoneNumberRegionData["regionname"].pattern = "pattern";
```

You can also create your own full RegionData object and merge its properties with the existing one like this:

```
jTAC.extend(your object, window.jTAC_PhoneNumberRegionData);
```

Set up

Set the **region** property to the name of the desired region, or to "" to select the name defined by the **window.jTAC_PhoneNumberRegionData.defaultName** property.

This property allows multiple regions. Create a pipe delimited list of region names. Example of multiple regions: "NorthAmerica|CountryCode".

If you have a way to change the region in the UI, you will want that change to apply to any region-sensitive TypeManager. To do that, pass the new region name to the `jTAC.setGlobalRegionName(region)` function. (You may have to convert from the strings you use in your Country DropDownList to the regions specified in `jTAC_PhoneNumberRegionData`. One way to do this is to use the alias feature of the region property.) Once `jTAC.setGlobalRegionName()` has been called, set the **region** property to "GLOBAL".

By default, phone number extensions are not considered valid. To allow them, set **supportsExt** to `true`.

Example: On the element using data-jtac-datatype

```
<input type="text" id="TextBox1" name="TextBox1" data-jtac-datatype="PhoneNumber.UnitedKingdom"/>
```

Example: On the element using data-jtac-typemanager

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-typemanager="{'jtacClass': 'PhoneNumber', 'region': 'UnitedKingdom'}" />
```

or

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="PhoneNumber" data-jtac-typemanager="{'region': 'UnitedKingdom'}" />
```

Example: Creating a Condition without data-jtac-datatype

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.datatype = "PhoneNumber.UnitedKingdom";
```

Example: Creating a Condition without data-jtac-typemanager

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.typeManager = jTAC.create("PhoneNumber");
cond.typeManager.region = "UnitedKingdom";
```

TypeManagers.PhoneNumber Properties

- **region** (string) – One of the region names defined in patterns. If "", it uses **window.jTAC_PhoneNumberRegionData.defaultName** value as the region name. You can include a mixture of regions by specifying a pipe delimited list of region names.

  If "GLOBAL", it uses the value set by calling `jTAC.setGlobalRegionName()`.

  It defaults to "".

- **supportsExt** (boolean) - When `true`, the string can contain the extension. It defaults to `false`.

  Extensions insert their RegExp pattern into the main pattern before "$)".There is a default pattern for most extensions, but a region can override it with the **extensionRE** property in its region object within TypeManagers.PhoneNumber. That will contain the regular expression pattern for the extension.

TypeManagers.PostalCode

Data types supported:	**string**
Alias names (case sensitive):	"**PostalCode**" and region specific values in this pattern: "**PostalCode.*regionname***"
Inherits from:	**TypeManagers.BaseRegionString**
Source file:	**\jTAC\TypeManagers\PostalCode.js**

Supports the data type for postal codes, which are strings with a specific pattern. The pattern differs based on the region (country usually).

Validates the pattern of a postal code using a region-specific regular expression. Also supports region specific character validation for the DataTypeEditor widget. It powers the DataTypeEditor widget in several ways: key filtering and reformatting. (While no region has been setup with reformatting by default, you can add it.)

Regional patterns of phone numbers are defined in `window.jTAC_PostalCodeRegionData` object. This object is fully customizable. You can replace its regular expression for validation or even provide a function that validates. You can code how to reformat the string too. See the TypeManagers.BaseRegionString for more.

To add, you can use this:

```
window.jTAC_PostalCodeRegionData["regionname"] = {pattern: value, more properties};
```

To replace the **pattern** property:

```
window.jTAC_PostalCodeRegionData["regionname"].pattern = "pattern";
```

You can also create your own full RegionData object and merge its properties with the existing one like this:

```
jTAC.extend(your object, window.jTAC_PostalCodeRegionData);
```

Set up

Set the **region** property to the name of the desired region, or to "" to select the name defined by the **window.jTAC_PostalCodeRegionData.defaultName** property.

This property allows multiple regions. Create a pipe delimited list of region names. Example of multiple regions: "NorthAmerica|CountryCode".

If you have a way to change the region in the UI, you will want that change to apply to any region-sensitive TypeManager. To do that, pass the new region name to the `jTAC.setGlobalRegionName(region)` function. (You may have to convert from the strings you use in your Country DropDownList to the regions specified in `jTAC_PostalCodeRegionData`. One way to do this is to use the alias feature of the region property.) Once `jTAC.setGlobalRegionName()` has been called, set the **region** property to "GLOBAL".

Example: On the element using data-jtac-datatype

```
<input type="text" id="TextBox1" name="TextBox1" data-jtac-datatype="PostalCode.UnitedKingdom" />
```

Example: On the element using data-jtac-typemanager

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-typemanager="{'jtacClass': 'PostalCode', 'region': 'UnitedKingdom'}" />
```

or

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="PostalCode" data-jtac-typemanager="{'region': 'UnitedKingdom'}" />
```

Example: Creating a Condition without data-jtac-datatype

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.datatype = "PostalCode.UnitedKingdom";
```

Example: Creating a Condition without data-jtac-typemanager

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.typeManager = jTAC.create("PostalCode");
cond.typeManager.region = "UnitedKingdom";
```

TypeManagers.PostalCode Properties

- **region** (string) – One of the region names defined in patterns. If "", it uses **window.jTAC_PostalCodeRegionData.defaultName** value as the region name. You can include a mixture of regions by specifying a pipe delimited list of region names.

  If "GLOBAL", it uses the value set by calling `jTAC.setGlobalRegionName()`.

  It defaults to "".

TypeManagers.String

Data types supported:	**string**
Alias names (case sensitive):	"**String**", "**String.caseins**"
Inherits from:	**TypeManagers.BaseString**
Source file:	**\jTAC\TypeManagers\String.js**

For string-based types that are not covered by other TypeManager

Set up

By default, its `compare()` function uses a case sensitive match. Set **caseIns** to `true` for a case insensitive match.

Example: On the element using data-jtac-datatype

```
<input type="text" id="TextBox1" name="TextBox1" data-jtac-datatype="String.caseins" />
```

Example: On the element using data-jtac-typemanager

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-typemanager="{'jtacClass': 'String', 'caseIns': true}" />
```

or

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="String" data-jtac-typemanager="{'caseIns': true}" />
```

Example: Creating a Condition without data-jtac-datatype

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.datatype = "String.caseins";
```

Example: Creating a Condition without data-jtac-typemanager

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.typeManager = jTAC.create("String");
cond.typeManager.caseIns = true;
```

String Properties

- **caseIns** (Boolean) - When `true`, use case insensitive comparison in the `compare()` method. It defaults to `false`.

TypeManagers.TimeOfDay

Data types supported:	**Date object and integer**
Alias names (case sensitive):	**"TimeOfDay", "TimeOfDay.InSeconds", "TimeOfDay.InHours", "TimeOfDay.NoSeconds", "TimeOfDay.InSeconds.NoSeconds", "TimeOfDay.InHours.NoSeconds", "TimeOfDay.NoZeroSeconds", "TimeOfDay.InSeconds.NoZeroSeconds", "TimeOfDay.InHours.NoZeroSeconds"**
Inherits from:	**TypeManagers.BaseDatesAndTimes**
Source file:	**\jTAC\TypeManagers\Date and time.js**

Its native type is the JavaScript Date object, working with its time of day component but not date. You can also use a number to hold the number of seconds or hours starting at 0:00:00 (12 AM). When using the number to hold hours, the value is a decimal where 1.0 = 1 hour, 1.5 = 1:30:00, etc. Supports localization.

Its parser is fairly strict. To provide more flexibility, you can switch to the TypeManagers.PowerTimeParser or one of your own.

Set up

Always consider if localization is needed. See "Localizing dates, times, and numbers". If it matters, set the culture's name in the **cultureName** property.

If you want to use a number to hold the time of day, consider using these alias names:

"TimeOfDay.InSeconds" – an integer where 1 = 1 second

```
var tm = jTAC.create("TimeOfDay.InSeconds");
```

"TimeOfDay.InHours" – a float where 1 = 1 hour

```
var tm = jTAC.create("TimeOfDay.InHours");
```

Alternatively, set the **valueAsNumber** property to true and **timeOneEqualsSeconds** to 1 (for seconds) or 3600 (for hours).

There are several time format patterns that you can defined in the **timeFormat** property:

0 - Includes seconds. Uses culture's choice of 12/24 hr format. Ex: hh:mm:ss tt; HH:mm:ss

1 - Omits seconds. Uses culture's choice of 12/24 hr format. Ex: hh:mm tt; HH:mm

2 - Same as 0 except it omits seconds when they are 0 (while formatting a string).

Example: On the element using data-jtac-datatype

```
<input type="text" id="TextBox1" name="TextBox1" data-jtac-datatype="TimeOfDay" />
```

Example: On the element using data-jtac-typemanager

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-typemanager="{'jtacClass': 'TimeOfDay', 'timeFormat': 2}" />
```

or

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="TimeOfDay" data-jtac-typemanager="{'timeFormat': 2}" />
```

Example: Creating a Condition without data-jtac-datatype

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.datatype = "TimeOfDay";
```

Example: Creating a Condition without data-jtac-typemanager

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.typeManager = jTAC.create("TimeOfDay");
cond.typeManager.timeFormat = 2;
```

TypeManagers.TimeOfDay Properties

- **timeFormat** (integer) - Determines how to setup the time pattern. It is also used to format a string.

  Here are its values:

  - 0 - Includes seconds. Uses culture's choice of 12/24 hr format. Ex: hh:mm:ss tt; HH:mm:ss. This is the default.
  - 1 - Omits seconds. Uses culture's choice of 12/24 hr format. Ex: hh:mm tt; HH:mm
  - 2 - Same as 0 except it omits seconds when they are 0 (while formatting a string).
  - 100 - Culture neutral and sortable format with seconds. Always uses "HH:mm:ss".
  - 101 – Culture neutral and sortable format without seconds. Always uses “HH:mm”.

- **timeOneEqualsSeconds** (integer) - Used when the value is a number representing only time. Determines how many seconds is represented by a value of 1.

  For 1 hour, use 3600. For 1 minute, use 60. For 1 second, use 1.

- **valueAsNumber** (boolean) - When `true`, it works with number types. When `false`, it works with Date objects.

  It defaults to `false`.

- **parseStrict** (boolean) - When this is `false`, the native parser lets you omit or incorrectly position the AM/PM designator. When `true`, it requires the AM/PM designator if its part of the time pattern, and it must be in the same location as that pattern.

  It defaults to `false`.

- **parseTimeRequires** (string) - When the time pattern has seconds, set this to "`s`" to require seconds be entered. Valid: “1:00:00” but not “1:00” or “1”.

  When "`m`", minutes are required but not seconds. Valid: “1:00” and “1:00:00” but not “1”.

  When "`h`", only hours are required. Valid: “1”, “1:00”, “1:00:00”.

  It defaults to "`h`".

- **useUTC** (boolean) - When `true`, the Date object is in UTC format. You should only pass UTC formatted Date objects to `toString()` and expect UTC format back from `toValue()`. When `false`, it is in local format.

  It defaults to `false`.

- **nativeParser** (boolean) – *Only when using a plug-in parser*. When `false`, use the plug-in parser. When `true`, use the native parser. It defaults to `false`.

- **parserOptions** (object) – *Only when using a plug-in parser*. Options specific to the plug-in parser. For TypeManagers.PowerTimeParser, see “parserOptions Properties”.

TypeManagers.Url

Data types supported:	**string**
Alias names (case sensitive):	"**Url**", "**Url.FTP**"
Inherits from:	**TypeManagers.BaseStrongPatternString**
Source file:	**\jTAC\TypeManagers\Url.js**

This class validates the pattern of a URL using a regular expression. It offers numerous options to select the parts of a URL that you want to allow, such as uri schemes and domain name extensions.

Set up

By default, it supports these Uri Schemes: "http" and "https". You can modify this list with the **uriScheme** property. If the Uri Scheme is optional, set **requireUriScheme** to `false`.

It checks the domain name extension (like "com" and "edu") with a predefined list which you can modify in the **domainExt** property.

It allows the user to enter a path under the domain name. Turn this off by setting **supportsPath** to `false`.

You can enable support of IP addresses and port numbers with the **supportsIP** and **supportsPort** properties.

Example: On the element using data-jtac-datatype

```
<input type="text" id="TextBox1" name="TextBox1" data-jtac-datatype="Url" />
```

Example: On the element using data-jtac-typemanager

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-typemanager="{'jtacClass': 'Url', 'supportsPath': false}" />
```

or

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="Url" data-jtac-typemanager="{'supportsPath': false}" />
```

Example: Creating a Condition without data-jtac-datatype

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.datatype = "Url";
```

Example: Creating a Condition without data-jtac-typemanager

```
var cond = jTAC.create("Conditions.DataTypeCheck");
cond.elementId = "TextBox1";
cond.typeManager = jTAC.create("Url");
cond.typeManager.supportsPath = false;
```

TypeManagers.Url Properties

- **uriScheme** (string) - A pipe delimited list of valid Uri Schemes. These are "http", "https", "ftp", etc. It defaults to "`http|https`".
- **domainExt** (string) - A pipe delimited list of domain extensions, ignoring the country part. For example: "com|co|edu". Do not use "co.uk" as it contains a specific country part. If you want to allow any extension, set this to "".

  It defaults to "aero|biz|com|coop|edu|gov|info|int|mil|museum|name|net|org|travel|jobs|mobi|pro|co".
- **supportsIP** (boolean) - When `true`, allow the domain name part to be an IP address instead of a domain name. It defaults to `false`.
- **supportsPort** (boolean) - When `true`, allow the port number to be specified. It defaults to `false`.
- **supportsPath** (boolean) - When `true`, allow a path after the domain name. It defaults to `true`.
- **requireUriScheme** (boolean) - When `true`, there must be a Uri Scheme. It defaults to `true`.

TypeManagers.Base

Inherits from:	**jTAC's universal base class**
Source file:	**\jTAC\TypeManagers\Base.js**

This is the base class for all TypeManager classes. It defines the essential methods, `toString()`, `toValue()`, `compare()`, etc. Many are abstract methods that need subclasses to implement.

You may be better off inheriting from one of its subclasses, due to the extended functionality they offer:

Condition class name	Purpose
TypeManagers.BaseCulture	Abstract base class for building TypeManagers that use localization.
TypeManagers.BaseDatesAndTimes	Abstract base class for building TypeManagers that handle dates and times from a JavaScript Date object.
TypeManagers.BaseNumber	Abstract base class for building TypeManagers that handle numbers.

TypeManagers.Base Public Methods

See "Using TypeManagers in your own JavaScript".

TypeManagers.Base Properties

- **cultureName** (string) - The culture name. Its value depends on the localization framework such as *jquery-globalize*. However, it is typically a standardized format like "en-US" and "fr-CA".

  When left unassigned, it uses a default culture.

- **friendlyName** (string) - Provides a string that can be shown to the user representing this name. If you don't assign a value to it, it uses the **friendlyLookupKey** with the jTAC.Translations system to locate a string. If that is not supplied, it uses the value from `dataTypeName()`.

- **friendlyLookupKey** (string) – Provides a key into the string translations tables of jTAC.Translations for localization and replacing the default value (which comes from `dataTypeName()`).

Subclassing TypeManagers.Base methods

Getting started:

- All TypeManagers inherit from `jTACClassBase` in **\jTAC\jTAC.js**. It has some useful utilities for your class development. Learn more in the **\jTAC\jTAC.js** file.
- See "Adding your own classes to jTAC" to use best practices in creating your subclasses' structure.
- For an example, see TypeManagers.Boolean in **\jTAC\TypeManagers\Boolean.js**.

This topic will identify methods that you are likely to override or call in TypeManagers.Base. It will not describe parameters. Instead, please use the class definition in **\jTAC\TypeManagers\Base.js**.

	Element Name	When
✓	`dataTypeName()`	Always.
✓	`storageTypeName()`	Always
✓	`_nativeToString()`	Always Used by the `toString()` method to handle actual conversion between a native value and string. It can apply formatting rules from properties introduced by the subclass.
✓	`_stringToNative()`	Used by `toValue()` method to do the actual conversion from a string to the native type, applying globalization rules for the culture defined in the **cultureName** property.
✓	`_isNull()`	Used by `toValue()` to determine if the value represents null. This class tests for null or the empty string. Subclass when other values represent null.
✓	`_reviewValue()`	Evaluates the value. Looks for illegal cases. Throws exceptions when found. It may change the value too, if needed, such as to round it.
✓	`toValueNeutral()`	Always
✓	`toStringNeutral()`	Always
✓	`compare()`	When the native values are not primitive types that compare nicely with the =, >, or < operators.
✓	`isValidChar()`	When your TypeManager supports character filtering.
✓	`toNumber()`	When the native value can be represented by a number.

Key: ✓ always override; ✓ sometimes override; ✓ rarely override

TypeManagers.BaseCulture

Inherits from:	**TypeManagers.Base**
Source file:	**\jTAC\TypeManagers\BaseCulture.js**

Extends TypeManagers.Base to support localization. Most supplied TypeManagers subclass from this to access its localization data.

It introduces the methods that access culture specific localization data. If you want to customize how localization works, see "Localizing dates, times, and numbers".

Subclassing TypeManagers.BaseCulture methods

Getting started:

- All TypeManagers inherit from `jTACClassBase` in **\jTAC\jTAC.js**. It has some useful utilities for your class development. Learn more in the **\jTAC\jTAC.js** file.
- See "Adding your own classes to jTAC" to use best practices in creating your subclasses' structure.
- For an example, see TypeManagers.BaseNumber in **\jTAC\TypeManagers\BaseNumber.js**.

This topic will identify methods that you are likely to override or call in TypeManagers.BaseCulture. It will not describe parameters. Instead, please use the class definition in **\jTAC\TypeManagers\Base.js**.

	Element Name	When
✓	`dataTypeName()`	Always.
✓	`storageTypeName()`	Always
✓	`_nativeToString()`	Always Used by the `toString()` method to handle actual conversion between a native value and string. It can apply formatting rules from properties introduced by the subclass.
✓	`_stringToNative()`	Used by `toValue()` method to do the actual conversion from a string to the native type, applying globalization rules for the culture defined in the **cultureName** property.
✓	`_isNull()`	Used by `toValue()` to determine if the value represents null. This class tests for null or the empty string. Subclass when other values represent null.
✓	`_reviewValue()`	Evaluates the value. Looks for illegal cases. Throws exceptions when found. It may change the value too, if needed, such as to round it.
✓	`toValueNeutral()`	Always
✓	`toStringNeutral()`	Always
✓	`compare()`	When the native values are not primitive types that compare nicely with the =, >, or < operators.
✓	`isValidChar()`	When your TypeManager supports character filtering. However, it has added code that you can use by setting up supporting fields and methods. Override if you cannot use a regular expression to test for valid characters.
✓	`toNumber()`	When the native value can be represented by a number.
✓	`_valCharRE()`	When using `isValidChar()` as implemented, this must return a RegExp object that will test a single character to determine if it is valid.

Key: ✓ always override; ✓ sometimes override; ✓ rarely override

TypeManagers.BaseDatesAndTimes

Inherits from:	**TypeManagers.BaseCulture**
Source file:	**\jTAC\TypeManagers\BaseDatesAndTimes.js**

Abstract base class for supporting date and time values. It works with Date objects and number (when using only time or duration values). It has extensive support for parsing and formatting that is used by subclasses. It also implements the **useUTC** property.

Subclassing TypeManagers.BaseDatesAndTimes methods

Getting started:

- All TypeManagers inherit from `jTACClassBase` in **\jTAC\jTAC.js**. It has some useful utilities for your class development. Learn more in the **\jTAC\jTAC.js** file.
- See "Adding your own classes to jTAC" to use best practices in creating your subclasses' structure.
- For an example, see TypeManagers.DateTime in **\jTAC\TypeManagers\DateTime.js**.

This topic will identify methods that you are likely to override or call in TypeManagers.BaseDatesAndTimes. It will not describe parameters. Instead, please use the class definition in **\jTAC\TypeManagers\Base.js**.

	Element Name	When
✓	`storageTypeName()`	Always
✓	`_nativeToString()`	Modify the `_parse()` method instead
✓	`_stringToNative()`	Modify the `_format()` method instead.
✓	`_parse()`	Always.
✓	`_format()`	Always.
✓	`supportsDates()`	Always. Return `true` if implementing date support. `false` otherwise
✓	`supportsTimes()`	Always. Return `true` if implementing time support. `false` otherwise
✓	`_reviewValue()`	Evaluates the value. Looks for illegal cases. Throws exceptions when found. It may change the value too, if needed, such as to round it.
✓	`toNumber()`	When the native value can be represented by a number.
✓	`_valChars()`	Always
✓	`_setNeutralFormat()`	Always

Key: ✓ always override; ✓ sometimes override; ✓ rarely override

TypeManagers.BaseNumber

Inherits from:	**TypeManagers.BaseCulture**
Source file:	**\jTAC\TypeManagers\BaseNumber.js**

Abstract base class for supporting number values.

Subclassing TypeManagers.BaseNumber methods

Getting started:

- All TypeManagers inherit from `jTACClassBase` in **\jTAC\jTAC.js**. It has some useful utilities for your class development. Learn more in the **\jTAC\jTAC.js** file.
- See "Adding your own classes to jTAC" to use best practices in creating your subclasses' structure.
- For an example, see TypeManagers.Integer in **\jTAC\TypeManagers\Integer.js**.

This topic will identify methods that you are likely to override or call in TypeManagers.BaseNumber. It will not describe parameters. Instead, please use the class definition in **\jTAC\TypeManagers\Base.js**.

	Element Name	When
✓	`dataTypeName()`	Always.
✓	`storageTypeName()`	Always
✓	`_nativeToString()`	Always Used by the `toString()` method to handle actual conversion between a native value and string. It can apply formatting rules from properties introduced by the subclass.
✓	`_stringToNative()`	Used by `toValue()` method to do the actual conversion from a string to the native type, applying globalization rules for the culture defined in the **cultureName** property.
✓	`_reviewValue()`	Evaluates the value. Looks for illegal cases. Throws exceptions when found. It may change the value too, if needed, such as to round it.
✓	`toValueNeutral()`	Always
✓	`_valCharRE()`	When using `isValidChar()` as implemented, this must return a RegExp object that will test a single character to determine if it is valid.

Key: ✓ always override; ✓ sometimes override; ✓ rarely override

TypeManagers.BaseString

Inherits from:	**TypeManagers.Base**
Source file:	**\jTAC\TypeManagers\BaseString.js**

Abstract base class for hosting string data. It does not use any localization library.

Its features include:

- storageTypeName = "string"
- `toString()` and `toValue()` functions return the same string passed (subclasses can override). If a non-string is passed, it is converted to a string first.
- `toNumber()` returns `null`
- `compare()` handles case insensitive values

Subclassing TypeManagers.BaseString methods

Getting started:

- All TypeManagers inherit from `jTACClassBase` in **\jTAC\jTAC.js**. It has some useful utilities for your class development. Learn more in the **\jTAC\jTAC.js** file.
- See "Adding your own classes to jTAC" to use best practices in creating your subclasses' structure.
- For an example, see TypeManagers.String in **\jTAC\TypeManagers\String.js**.

This topic will identify methods that you are likely to override or call in TypeManagers.BaseString. It will not describe parameters. Instead, please use the class definition in **\jTAC\TypeManagers\Base.js**.

	Element Name	When
✓	`dataTypeName()`	Always.
✓	`_nativeToString()`	When you want to add formatting.
✓	`_stringToNative()`	When you want to remove formatting
✓	`_reviewValue()`	Evaluates the value. Looks for illegal cases. Throws exceptions when found. It may change the value too, if needed, such as to round it. This is where you insert validation code, such as testing the string against a regular expression.
✓	`toValueNeutral()`	To remove formatting and store in it a format usable by your server side code.
✓	`isValidChar()`	When you can limit the character set.
✓	`_isCaseIns()`	When you want to support case insensitive matching with the `compare()` method.

Key: ✓ always override; ✓ sometimes override; ✓ rarely override

TypeManagers.BaseStrongPatternString

Inherits from:	**TypeManagers.BaseString**
Source file:	**\jTAC\TypeManagers\BaseStrongPatternString.js**

Abstract base class for hosting string data that is validated against a regular expression.

Subclassing TypeManagers.BaseStrongPatternString methods

Getting started:

- All TypeManagers inherit from `jTACClassBase` in **\jTAC\jTAC.js**. It has some useful utilities for your class development. Learn more in the **\jTAC\jTAC.js** file.
- See "Adding your own classes to jTAC" to use best practices in creating your subclasses' structure.
- For an example, see Url in **\jTAC\TypeManagers\Url.js**.

This topic will identify methods that you are likely to override or call in TypeManagers.BaseStrongPatternString. It will not describe parameters. Instead, please use the class definition in **\jTAC\TypeManagers\Base.js**.

	Element Name	When
✓	`dataTypeName()`	Always.
✓	`_regExp()`	Always. Return the regular expression string pattern that will be used to validate.
✓	`toValueNeutral()`	To remove formatting and store in it a format usable by your server side code.
✓	`isValidChar()`	When you can limit the character set.
✓	`_isCaseIns()`	When you want to support case insensitive matching with the regular expression pattern, return `true`.

Key: ✓ always override; ✓ sometimes override; ✓ rarely override

TypeManagers.BaseRegionString

Inherits from:	**TypeManagers.BaseString**
Source file:	**\jTAC\TypeManagers\BaseRegionString.js**

Handles characteristics of a data type that varies by the region of the world, such as phone numbers, postal codes, etc.

Each subclass defines a RegionData object on the window object and passes it to the constructor in the regiondata parameter.

The RegionData object defines each region as a property name and an object with these properties as the value:

- **name** (string) - Same name as the property. Lets consumers of this object identify it.
- **pattern** (string) - The regular expression pattern used to validate the entire string passed into the TypeManager (via toValue).

  Patterns must be created within this structure: (^your pattern$).

  This allows for multiple patterns to be merged when the user defined a pipe delimited list of region names in the TypeManager's **region** property.

  Alternatively, this can be a function that is passed the text and returns true if it is valid.
- **caseIns** (boolean) - When case insensitivity is required, define this with a value of true. Omit or set to false if not needed.
- **validChars** (string) - The regular expression pattern used to validate a single character as a legal character of the pattern.

  This is used by the `isValidChar()` function.
- **toNeutral** (function) - If you want to pass back to the server a string without some of the formatting, define this function.

  It is passed the text and returns the cleaned up text. For example, phone number may have everything except digits stripped.

  This value will be used by the `toValueNeutral()` function and is used by the DataTypeEditor widget when you have setup the hidden field that hosts a neutral format.
- **toFormat** (function) - If you want to apply formatting to a neutral format to display it to the user, define a function here. It is passed the text to format and returns the formatted text. This value is used by `toString()`. You may need to make it work with already formatted text, such as running it through the `toNeutral()` function first.

The RegionData object must always have a **defaultName** property whose value is the name of another property that will be used if the TypeManager's **region** property is left unassigned.

Sometimes a few regions use the same patterns. You can create alias properties. Define the property name for the region and the value as a string with the name of the other property that holds an object.

Example RegionData object:

```
var jTAC_PhoneNumberRegionData = {
   defaultName: "UnitedStates",

   UnitedStates: {
      pattern : "(^([1])?\\s*(\\([2-9]\\d{2}\\))?\\s*\\d{3}[\\s\\-]?\\d{4}$)",
      validChars : "[0-9 \\(\\)\\-]",
      toNeutral : function(text) {
            // clean up the text
            return text;
         }
      toFormat : function(text) {
            var r = jTAC_PhoneNumberRegionData.UnitedStates.toNeutral(text);
            // insert formatting
            return r;
         }
   }
};
```

This is designed so you can expand the list of regions, by adding new properties to the object, and replace elements of an individual regiondata node if needed, all without editing the original source code file.

To add, you can use this in your own scripts:

```
object["regionname"] = {pattern: value, more properties};
```

To replace the **pattern** property value:

```
object["regionname"].pattern = "new pattern";
```

You can also create your own full RegionData object and merge its properties with the existing one like this:

```
jTAC.extend(your object, existing RegionData object);
```

Subclassing TypeManagers.BaseRegionString methods

Getting started:

- All TypeManagers inherit from `jTACClassBase` in **\jTAC\jTAC.js**. It has some useful utilities for your class development. Learn more in the **\jTAC\jTAC.js** file.
- See "Adding your own classes to jTAC" to use best practices in creating your subclasses' structure.
- For an example, see TypeManagers.PhoneNumber in **\jTAC\TypeManagers\PhoneNumberTypeManager.js**.

This topic will identify methods that you are likely to override or call in TypeManagers.BaseRegionString. It will not describe parameters. Instead, please use the class definition in **\jTAC\TypeManagers\Base.js**.

	Element Name	When
✓	`dataTypeName()`	Always.
✓	`_defaultRegionsData()`	Return the default RegionsData object.

Key: ✓ always override; ✓ sometimes override; ✓ rarely override

Using TypeManagers in your own JavaScript

A TypeManager is a very powerful tool. You may want to use it in your own scripts to convert between native and string types.

To create one, call the `jTAC.create()` method.

```
var tm = jTAC.create("TypeManagers.Date");
var tm = jTAC.create("TypeManagers.Date", {dateFormat: 1, useUTC: true});
var tm = jTAC.create("Date.Short");       // uses an alias
```

Here are the functions available on all TypeManagers.

```
function toString(value)
```

Convert from a native value to a string, applying globalization rules for the culture defined in the **cultureName** property, if applicable.

Parameters

value

The value to convert. Its type must be supported by the subclass implementing this method or it should throw an exception. (Generally its must match the value of **storageTypeName**.)

Returns: A string. If the value represents nothing or is `null`, this will return the empty string.

Exceptions: Throws exceptions when conversion cannot happen such as when the format is not appropriate for the data type being converted.

When subclassing: Override the `_nativeToString()` method to do the actual conversion. Rarely do you override `toString()`.

```
function toValue(text)
```

Convert from a string to the native type, applying globalization rules for the culture defined in the **cultureName** property, if applicable.

Parameters

text (string)

The string to convert. It must be compatible with the formatting rules or an exception will be thrown.

Returns: native value. If the value represents nothing or is `null`, this will return `null`.

Exceptions: Throws exceptions when conversion from strings fails.

When subclassing: Override `_stringToNative()` method to do the actual conversion. Rarely do you override `toValue()`.

```
function toValueFromConnection(connection)
```

Get the value from a Connection object. If it gets a string, it is converted using the `toValue()` function. If it gets the correct native value, that value is returned.

Parameters

connection (Connection object)

The Connection that supplies the value.

Returns: The native value or `null` if the value represents `null`.

Exceptions: Throws exceptions when conversion from strings fails or the type of value the Connection supplies does not match what the TypeManager expects.

When subclassing: Rare

```
function toValueNeutral(text)
```

Conversion from string to native value. The string should be culture neutral, not following the conventions of the current culture. For example, numbers should be using period for decimal separator and dates should be in yyyy-MM-dd format.

Parameters

text (string)

The string to parse and convert.

Returns: The native value or `null` if the text is the empty string.

Exceptions: Thrown if the string cannot be converted.

When subclassing: Always. It is an ABSTRACT method in TypeManagers.Base.

```
function toStringNeutral(value)
```

Conversion from native value to culture neutral string.

Parameters

val

The native value to convert.

Returns: The native value or `null` if the text is the empty string.

When subclassing: Always. It is an ABSTRACT method in TypeManagers.Base.

```
function isValid(text, cannoteval)
```

Checks if the text passed is a valid pattern for the native type.

Parameters

text (string)

The string to check. If the string has lead or trailing spaces, they are ignored by the parser.

cannoteval (boolean)

When the text is the empty string or `null`, this is the value returned. If the parameter is undefined, it returns `false`.

Returns: `true` when text is a valid pattern and `false` when it is not. If the *text* parameter is the empty string or `null`, the result depends on the *cannoteval* parameter.

When subclassing: Rare

```
function compare(val1, val2)
```

Compare two values that represent the same data type. Either or both values can be strings or the native value.

Parameters

val1

The value to compare on the left side of the comparison operator.

If it is a string, it is passed to `toValue()` first to get it into a native value.

val2

The value to compare on the right side of the comparison operator.

If it is a string, it is passed to `toValue()` first to get it into a native value.

Returns: -1 when val1 < val2; 0 when val1 = val2; 1 when val1 > val2.

Exceptions: Throws exceptions when conversion from strings fails or either of the values is `null`.

When subclassing: The implementation in TypeManagers.Base supports primitive types, comparing using the =, <, and > operators against val1 and val2. When the values require more complex evaluation, such as for a Date object, you must subclass.

```
function isValidChar(chr)
```

Evaluates a single character to determine if it is valid in a string representing the type. It does not care about the position or quantity of this character in the string that is being created.

For example, if this is a date entry that supports only the short date format, it considers digits and the decimal separator to be valid.

Parameters

chr (string)

A single character string to evaluate. If not a single character, an exception is thrown.

Returns: `true` when the character is valid and `false` when invalid.

When subclassing: The implementation in TypeManagers.Base returns `true` for every character passed. If your TypeManager needs to offer filtering, you must subclass. You should call `this.callParent([chr])` first. It checks for an illegal string passed as the parameter and throws an exception.

```
function toNumber(value)
```

Converts the value into a number (float or integer).

Used to allow calculations even when the value is not a number. Especially useful for dates and times.

Parameters

value

The value to convert.

Returns: A number or null. `null` indicates that it is not supported. TypeManagers.Base returns `null`.

When subclassing: Override if the native value can be converted to a number. Existing subclasses have used it as follows:

When the value is a number, its value is returned.

When the value is a date, its value is the number of days since midnight Jan 01, 1970.

When the value is a time, its value is the total seconds.

When the value is a date time, its value is the number of seconds since midnight Jan 01, 1970.

Your own classes may define a differ result.

```
function dataTypeName()
```

Gets a string that identifies the real-world data type represented, such as "date", "MonthYear", or "currency".

Returns: the string.

When subclassing: Always. It is an ABSTRACT method in TypeManagers.Base.

```
function storageTypeName()
```

Gets a string that identifies the storage data type (on the server side), such as "Integer", "float", and "boolean".

Returns: the string.

When subclassing: Always. It is an ABSTRACT method in TypeManagers.Base.

Alternative parsers for TypeManagers

Each TypeManager class implements a parser for its `toValue()` method. Parsing rules often vary based on usage. The parsers for date and time oriented parsers are good cases. They handle dates in the short date pattern (like MM/dd/yyyy). Although culture sensitive, users often enter dates differently, such as in the long date pattern (MMMM dd, yyyy) or by omitting the year and assuming the website will use the current year. As a result, TypeManagers supports plug-in parsers.

jTAC includes two: PowerDateParser and PowerTimeParser. These support all date and time TypeManagers. Click below to learn more.

- TypeManagers.PowerDateParser
- TypeManagers.PowerTimeParser
- Creating your own Parsers

TypeManagers.PowerDateParser

Source file:	**\jTAC\TypeManagers\PowerDateParser.js**
Requires:	**\jTAC\TypeManagers\BaseDate.js**

This class extends the TypeManagers.Date, TypeManagers.DateTime, TypeManagers.DayMonth, and TypeManagers.MonthYear classes to parse strings using fuzzy logic. It replaces the default date parser, which has very rigid parsing rules, with something that can handle entries from users that may not match the intended pattern but are legal.

PowerDateParser supports localization, including *jquery-globalize*.

Here are its fuzzy logic cases:

- Automatically handles month names or numbers. If your default DateFormat specifies the month name and digits are found in the location, those digits determine the month.
- Matches a partial month name, such as "Ja" for "January". It matches all characters typed to the same number of characters of the first full or abbreviated month name found.
- Fills in missing year and month values with the current date's month and year.
- The user can omit date separator characters. A pattern like 01252012 will know its 01/25/2012 when using the MM/dd/yyyy pattern.
- Years can be two digits. They will convert to 2000 or 1900 based on a century break rule.
- Automatically detects dates in the culture neutral format of yyyy-MM-dd. So this is legal, unless the culture defines a date separator of "-": 2012-01-25.
- Short date pattern allows spaces around the date separators: "01 / 25 / 2012".
- Supports alternative date separators in addition to the culture's date separator character.

Caveats:

- Does not support week days.
- The server side code must also be able to handle text that was accepted by this in its server side validation and string conversion. You can either write a server side parser to match these capabilities or use the DataTypeEditor widget. When using DataTypeEditor, add a hidden input field whose ID is the same as the textbox + "_neutral". Get and set the date value in the hidden field in the culture neutral format of: yyyy-MM-dd.

Note: The parser is adapted from the author's commercial product, Peter's Date and Time. Its approach has been used in 100s of web sites since 2002.

Set up

Add this script file to the page. Place it after the other TypeManager classes.

```
<script src="/jTAC-min/TypeManagers/PowerDateParser.js" type="text/javascript"></script>
```

The act of adding the script file installs this parser in all TypeManager classes it supports automatically.

This parser adds properties to the TypeManager class that it extends.

The **parserOptions** property contains numerous properties to customize the behavior of this parser. See "parserOptions Properties".

The **nativeParser** property lets you turn off the PowerDateParser and resume using the native date parser on a case by case basis.

Set the **dateFormat** property to one of these values to tell the parser what to look for and how to convert a Date object to a string.

0 – Short date pattern, which is digits separated by the date separator character. Ex: 5/31/1965

1 – Short date pattern optionally allowing the month name (short, abbreviated or full) in place of digits for the month. Ex: "May/31/1965", "MAY/31/1965", and "5/31/1965". Reformatting uses Short month name: "May/31/1965"

2 – Same as 1, except reformatting converts the month name to uppercase.

10 – Can parse long, abbreviated, and short date patterns. Reformats to the abbreviated date format: “Aug 10, 1920”

20 – Can parse long, abbreviated, and short date patterns. Reformats to the long date format: “August 10, 1920”

Set the **twoDigitYear** to `false` if you want to require only 4 digit years. If left `true`, a 2 digit year is converted to a 4 digit year.

parserOptions Properties

These properties customize the parser. They are found on the TypeManager's **parserOptions** property. Here are ways to set them up.

```
var tm = jTAC.create("TypeManagers.Date");
tm.parserOptions.property = value;
```

In markup:

```
<input type="text" id="DateTextBox1" name="DateTextBox1"
   data-jtac-datatype="TypeManagers.Date"
   data-jtac-typemanager="{parserOptions: {property: value}}" />
```

- **defaultYear** (integer) - When there is no year entered, this is the year to use. When not defined, it uses the current year.
- **defaultMonth** (integer) - When there is no month entered, this is the month to use. Specify 0 to 11. When not defined, it uses the current year.
- **moreDateSeps** (string) – Some users may prefer alternative characters for the date separator. If you want to allow that, define this as a string containing each separator character (omitting the culture's date separator). The characters are not delimited. All characters in the string are separators.

  For example, to support "-" and ".", use **moreDateSeps**="`-.`".
- **cultureNeutral** (boolean) - When `true`, the string is checked for the culture neutral format of yyyy-MM-DD regardless of the culture's own pattern. Culture neutral dates will always require a 4 digit year, but month and day can be one digit. Month names are not allowed and the separator must be "-". Cultures that use "-" for there own separator disable this feature. It defaults to `true`.

  For example, when the culture format is "MM/dd/yyyy", these are both legal values when true: "5/31/1965" and "1965-31-05". When false, "1965-31-05" is illegal.
- **monthAllows** (integer) – When parsing the month, this determines if digits and/or month names are allowed.

  Its values are:

  - 0 - Allow either digits or month names.
  - 1 - Digits only
  - 2 - Month names only, allowing either long or abbreviated names
  - 3 - Month names only, allowing only abbreviated names

  It defaults to 0.
- **monthAsTwoCharName** (boolean) - When parsing the month and looking for month names, if there is no match to abbreviated or long, try again, matching the first two letters in the text to the first two letters in the abbreviated month name. It defaults to `true`.

  For example, when `true` these are legal values representing May 31, 1965: "Ma/31/1965", "Max/31/1965" (Max is "May" misspelled.) All are illegal when `false`.
- **insertDateSeps** (boolean) - When using the short date format (**dateFormat** < 10) and there are no date separator characters found, insert date separator characters when `true`. It defaults to `true`.

  For example, these are legal values when `true`: "05311965", "0531" (knows to use the current year), "5311965". All 3 are illegal when `false`.
- **trimParts** (boolean) - When `true` and using the short date format (**dateFormat** < 10), spaces are allowed around each part of the date in short date format. This does not impact trimming lead and trailing spaces around the entire string. That always happens. It defaults to `false`.

  For example, this is legal when `true` and illegal when `false`: "05 / 31 / 1965".

TypeManagers.PowerTimeParser

Source file:	**\jTAC\TypeManagers\PowerTimeParser.js**
Requires:	**\jTAC\TypeManagers\BaseTime.js,**

This class extends the TypeManagers.TimeOfDay, TypeManagers.DateTime, and TypeManagers.Duration classes to parse strings using fuzzy logic. It replaces the default time parser, which has very rigid parsing rules, with something that can handle entries from users that may not match the intended pattern but are legal.

PowerTimeParser supports localization, including *jquery-globalize*.

Here are its fuzzy logic cases:

- Can omit "AM" and "PM" in 12 hour format. It uses a default based on options.
- "AM" and "PM" can be shorted to their first letter and case does not matter.
- Does not require a space separator between AM/PM and the number
- Recognizes hours between 12 and 23 as PM times when "PM" is not entered.
- Can omit minutes and seconds. They will default to zero.
- Allows spaces around the time separators: "01 : 25 : 00".
- The user can omit time separator characters. A pattern like 012500 will know its 01:25:00.
- It can allow decimal format entry for time. (1.5 = 1:30:00).

Caveats:

- The server side code must also be able to handle text that was accepted by this in its server side validation and string conversion. You can either write a server side parser to match these capabilities or use the DataTypeEditor widget. When using DataTypeEditor, add a hidden input field whose ID is the same as the textbox + "_neutral". Get and set the time value in the hidden field in the culture neutral format of: HH:mm:ss.

Note: The parser is adapted from the author's commercial product, Peter's Date and Time. Its approach has been used in 100s of web sites since 2002.

Set up

Add this script file to the page. Place it after the other TypeManager classes.

```
<script src=/jTAC-min/TypeManagers/PowerTimeParser.js" type="text/javascript"></script>
```

The act of adding the script file installs this parser in all TypeManager classes it supports automatically.

This parser adds properties to the TypeManager class that it extends.

The **parserOptions** property contains numerous properties to customize the behavior of this parser. See "parserOptions Properties".

The **nativeParser** property lets you turn off the PowerDateParser and resume using the native time parser on a case by case basis.

Change the **parseTimeRequires** property to require more parts than just hours. Set it to "m" to require the user enter hours and minutes. Set it to "s" to require hours, minutes, and seconds.

parserOptions Properties

These properties customize the parser. They are found on the TypeManager's **parserOptions** property. Here are ways to set them up.

```
var tm = jTAC.create("TypeManagers.TimeOfDay");
tm.parserOptions.property = value;
```

In markup:

```
<input type="text" id="TimeTextBox1" name="TimeTextBox1"
   data-jtac-datatype="TypeManagers.TimeOfDay"
   data-jtac-typemanager="{parserOptions: {property: value}}" />
```

- **defaultAM** (boolean) - When using 12 hour format and AM/PM designator is omitted, this determines if the time means AM or PM.

  When `true`, it means AM. For example, "3:00:00" means "3:00:00 AM".

  When `false`, it means PM. For example, "3:00:00" means "3:00:00 PM".

  When `null`, it is an error to omit the AM/PM designator. It defaults to `true` (AM).

- **decimalCharacterMode** (integer) – Time entry optionally permits the decimal character to be used in two ways:
  - Allow decimal numbers for hours and minutes (1.5 = 1:30). Set this to 1.
  - An alternative for the time separator (1.30 = 1:30). Set this to 2.

  When 0 (the default), decimal characters are not used unless they are the actual time separator.

  It defaults to 0.

  Note: This option is ignored when the localized Time Separator character is the same as the Decimal Separator character.

- **noSecsEnforced** (boolean) - When **timeFormat**=1 (which means no seconds are ever shown), this determines if the presence of seconds in the string is illegal too. When `false`, the user can enter seconds and they will be kept. When `true`, if the user enters seconds of anything other than 0, it is an error. It defaults to `true`.

  For example, this is illegal when `true` and legal when `false`: "1:30:02".

- **moreTimeSeps** (string) – Some users may prefer alternative characters for the time separator. If you want to allow that, define this as a string containing each separator character (omitting the culture's time separator). The characters are not delimited. All characters in the string are separators.

  For example, to support "-" and ".", use **moreTimeSeps**="`-.`".

- **insertTimeSeps** (boolean) - When there are no time separator characters found, insert time separator characters when `true`. It defaults to `true`.

  For example, these values all mean 1:30: "0130", "013000", "13000", "130".

- **trimParts** (boolean) - When `true`, spaces are allowed around each part of the time. It defaults to `false`. This does not impact trimming lead and trailing spaces around the entire string. That always happens.

  For example, this is legal when `true` but illegal when `false`: "01 : 30 : 00".

- **fullAMPM** (boolean) - When `true`, the AM/PM designator must be the full string. When `false`, it can match with just the first character, allowing shorthand, and correcting typos. It defaults to `true`.

  For example, this is illegal when `true` but legal when `false`: "1:30:00 A"

Creating your own Parsers

If you want to create another, jTAC includes a plug-in framework.

1. Create a jTAC class with these three members (and any more you need):
   - `parse()` - The parser function. It is passed the string to parse and returns the value expected by the TypeManager class's native `_parse()` method. It throws exceptions via `this._inputError()` for errors found.
   - `valChar()` – A function that ascertains the characters allowed by the parser. It is passed a string with the characters defined natively. valChar should return the same list, a replacement or a modified version.
   - `owner` - A field that will be assigned the owning TypeManager object to use your parser.
2. Register that class with jTAC using:

```
jTAC.define("TypeManagers.classname", classobject);
```

3. Call:

```
jTAC.plugInParser("name of parser class", "TypeManager class to extend");
```

See examples in **TypeManagers****PowerDateParser.js** and **PowerTimeParser.js**.

When finished, users can either use your parser or by setting the new **nativeParser** property to `true`, can use the original parser.

The user can assign any properties on the parser object by referencing them through the **parserOptions** property on the TypeManager.

Localizing dates, times, and numbers

By default, the parsers and formatters of TypeManagers follow the United States rules for dates, times, and numbers. You can customize these rules in several ways.

- Manually assign new values
- Load values from a script file
- Use jquery-globalize
- Use another methodology

Manually assign new values

jTAC hosts an object whose properties define the date, time, and number rules in the `jTAC.cultureInfo` field. It has many properties which you modify while the page loads like this:

```
jTAC.cultureInfo.property = value;
```

For example:

```
jTAC.cultureInfo.dtShortDatePattern = "dd-MM-yyyy";
jTAC.cultureInfo.dtDateSeparator = "-";
```

See the end of the **\TypeManagers\BaseCulture.js** class for a list of all properties and their definitions.

Load values from a script file

Another way to use this is to create a separate script file that assigns all of the values for a culture. jTAC supplies several culture script files in the **\jTAC\Cultures** folder. Just add one of these files as a <script> tag after the TypeManagers are added.

```
<script src="/jTAC/Merged/core.js" type="text/javascript"></script>
<script src="/jTAC/Merged/typeManagers-date-time.js"   type="text/javascript"></script>
<script src="/jTAC/Cultures/fr-FR.js" type="text/javascript"></script>
```

You can create a new script file too.

- Clone the **Template.js** file already there and rename it to the culture code.
- Edit the file, updating any properties that require change.

Use jquery-globalize

jquery-globalize is an open source library that provides rules for numerous cultures. Despite the name, it does *not* require *jQuery*.

To use it:

1. Get the library here: https://github.com/jquery/globalize/
2. Add the **jquery-globalize** folder, containing the main script file and cultures folder, to your application.
3. Add the main script file, **globalize.js**, to your web page. Locate it after the TypeManager script files are loaded.

```
<script src="/jquery-globalize/globalize.js" type="text/javascript"></script>
```

4. Add any desired culture files. (If using the en-US culture, you can omit this step.)

```
<script src="/jquery-globalize/cultures/globalize.culture.fr-FR.js"
   type="text/javascript"></script>
```

5. Add this script file to your web page.

```
<script src="/jTAC-min/TypeManagers/Culture engine for jquery-globalize.js"
   type="text/javascript"></script>
```

Use another methodology

jTAC is designed to allow alternative culture engines by way of a plug in.

You can develop the plug-in by replacing the prototype definition for these methods on the TypeManagers.BaseCulture class:

numberFormat(rule)

currencyFormat(rule)

percentFormat(rule)

dateTimeFormat(rule)

Use the contents of **\jTAC\TypeManagers\Culture engine using jquery-globalize.js** as a guide to developing a plug-in.

Once developed, the user adds your file to the web page after TypeManager scripts are loaded.

The Condition classes

A **Condition** evaluates something and returns “success”, “failed”, and “cannot evaluate”. They are the primary tool behind a validation rule. As multipurpose objects, they are used in several ways:

- Each Condition class can be added to *jquery-validate*’s rules. All predefined Conditions are already there. You can also add your own.
- The validator’s dependency feature can use a Condition to determine if the validator is enabled.
- The Conditions.BooleanLogic class uses other Conditions to build a complex boolean expression.
- The Calculator widget can use a Condition to build boolean logic into a calculation.
- When you are writing JavaScript and need an IF statement that does the same thing as any Condition, create that condition and call its `toValue()` method. Use the result in your IF statement.

Click on any of these topics to jump to them:

Locate a Condition

Here are the Condition classes supplied by jTAC. Click on their name to learn more about them. Each of these classes is defined in files of the **jTAC****Conditions**\ folder. If you prefer merged script files, see "Loading scripts when using Conditions without jquery-validate" and "Loading scripts supporting jquery-validate".

Class name	Validation rule name	Purpose
Conditions.BooleanLogic	"booleanlogic"	Defines an AND or OR expression with other Conditions.
Conditions.CharacterCount	"charactercount"	Determines if the number of characters in a string is within a range.
Conditions.CompareToValue	"comparetovalue"	Compares the value from an element to a fixed value. It allows for all of the usual comparisons: equal, less than, etc. Handles multiple data types.
Conditions.CompareTwoElements	"comparetwoelements"	Compares two elements to each other. It allows for all of the usual comparisons: equal, less than, etc. Handles multiple data types.
Conditions.CountSelections	"countselections"	Evaluates list-style elements that support multiple selections. It counts the number of selections and compares the value to a range.
Conditions.DataTypeCheck	"datatypecheck"	Parses text to ensure it converts to a native data type. Handles multiple data types.
Conditions.Difference	"difference"	Evaluates if the difference between the values of two widget.
Conditions.DuplicateEntry	"duplicateentry"	Ensures that the values from a list of widgets are all different.
Conditions.Range	"advrange"	Compares the value from a widget to a range. Handles multiple data types.
Conditions.RegExp	"regexp"	Evaluates the value of a widget against a regular expression.
Conditions.Required	"advrequired"	Evaluates one or more widgets to determine if they are empty. With multiple widgets, use rules like "All", "One", "OneOrAll", "OneOrMore", and "Range".
Conditions.RequiredIndex	"requiredindex"	Determines if a list style widget has a selection.
Conditions.SelectedIndex	"selectedindex"	Compares an index (or list of indices) with the actual selected index of list style widgets. Supports multiselection lists too.
Conditions.WordCount	"wordcount"	Variation of CharacterCount that counts the number of words.

Here are some base classes that you may use to create your own subclasses.

Base Condition class name	Description
Conditions.Base	The top-most ancestor of all Conditions.
Conditions.BaseOneConnection	For building conditions that evaluate one element. Adds the **elementId** and **connection** properties inheriting from Conditions.Base.
Conditions.BaseTwoConnections	For building conditions that evaluate two elements. Adds the **elementId2** and **connection2** properties inheriting from Conditions.BaseOneConnection.
Conditions.BaseOneOrMoreConnections	For building conditions that evaluate a list of elements. Adds the **moreConnections** property inheriting from Conditions.BaseOneConnection.
Conditions.BaseOperator	For building conditions that use a boolean operator (=, <>, etc). Adds the **operator** property inheriting from Conditions.BaseTwoConnections.

Loading scripts when using Conditions without jquery-validate

ALERT: If already using the scripts in jTAC/Merged/jquery extensions/validate-basic/typical/all.js, do not use this section.

1. If you want to load a single script file with all of jTAC, use this:

```
<script src="/jTAC-min/Merged/jtac-all.js" type="text/javascript"></script>
```

   Otherwise continue with the next step.

2. Add **core.js** before any other jTAC script file.

```
<script src="/jTAC-min/Merged/core.js" type="text/javascript"></script>
```

3. Include one of these files defining the desired Condition objects:

```
<script src="/jTAC-min/Merged/conditions-basic.js" type="text/javascript"></script>
```
or
```
<script src="/jTAC-min/Merged/conditions-typical.js" type="text/javascript"></script>
```
or
```
<script src="/jTAC-min/Merged/conditions-all.js" type="text/javascript"></script>
```

Condition class	basic	typical	all
Conditions.BooleanLogic			x
Conditions.CharacterCount			x
Conditions.CompareToValue *		x	x
Conditions.CompareTwoElements *		x	x
Conditions.CountSelections			x
Conditions.DataTypeCheck *	x	x	x
Conditions.Difference *			x
Conditions.DuplicateEntry			x
Conditions.Range *	x	x	x
Conditions.RegExp		x	x
Conditions.Required	x	x	x
Conditions.RequiredIndex			x
Conditions.SelectedIndex			x
Conditions.WordCount			x

** requires a TypeManager*

4. If using a Condition that requires a TypeManager, load one or more of these files defining the desired TypeManager objects:

```
<script src="/jTAC-min/Merged/typemanagers-numbers.js" type="text/javascript"></script>
```
or
```
<script src="/jTAC-min/Merged/typemanagers-date-time.js" type="text/javascript"></script>
```
or
```
<script src="/jTAC-min/Merged/typemanagers-date-time-all.js" type="text/javascript"></script>
```
or
```
<script src="/jTAC-min/Merged/typemanagers-strings-common.js" type="text/javascript"></script>
```
or
```
<script src="/jTAC-min/Merged/typemanagers-strings-common-all.js"
    type="text/javascript"></script>
```
or
```
<script src="/jTAC-min/Merged/typemanagers-all.js" type="text/javascript"></script>
```

TypeManager class	numbers	date-time	d-t-all	strings common	strings all	all
TypeManagers.Boolean						x
TypeManagers.CreditCardNumber					x	x
TypeManagers.Currency	x					x
TypeManagers.DateTime		x	x			x
TypeManagers.Date		x	x			x
TypeManagers.DayMonth			x			x
TypeManagers.Duration			x			x
TypeManagers.EmailAddress				x	x	x
TypeManagers.Float	x					x
TypeManagers.Integer	x	x	x			x
TypeManagers.MonthYear			x			x
TypeManagers.Percent	x					x
TypeManagers.PhoneNumber				x	x	x
TypeManagers.PostalCode				x	x	x
TypeManagers.String			x	x	x	x
TypeManagers.TimeOfDay		x	x			x
Url					x	x

Note: If you choose to load individual condition script files, add the Merged/jquery extensions/jquery-validate-extensions.js file to install those conditions into jquery-validate. Do not use it when using the above validation-basic/typical/all.js files.

5. If you are need localization of number, date, or time TypeManagers, add the appropriate scripts. See "Localizing dates, times, and numbers".

   Here is an example using *jquery-globalize*. Add its script file first. Add any culture specific files from *jquery-globalize*. Then add the **TypeManagers\Culture engine for jquery-globalize.js** file. All of these files can be located below the TypeManager scripts.

```
<script src="/jquery-globalize/globalize.js" type="text/javascript"></script>
<script src="/jquery-globalize/cultures/globalize.culture.fr-FR.js"
   type="text/javascript"></script>
<script src="/jTAC-min/TypeManagers/Culture engine for jquery-globalize.js"
   type="text/javascript"></script>
```

6. If you are using dates and times, consider switching parsers from the default to the TypeManagers.PowerDateParser or TypeManagers.PowerTimeParser. If you do, here are the links for those classes.

```
<script src="/jTAC-min/TypeManagers/PowerDateParser.js" type="text/javascript"></script>
<script src="/jTAC-min/TypeManagers/PowerTimeParser.js" type="text/javascript"></script>
```

Example

```
<script src="/jquery/jquery-#.#.#.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.unobtrusive.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/core.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/conditions-typical.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/typemanagers-numbers.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/typemanagers-date-time.js" type="text/javascript"></script>
<script src="/jquery-globalize/globalize.js" type="text/javascript"></script>
<script src="/jquery-globalize/cultures/globalize.culture.fr-FR.js"
   type="text/javascript"></script>
<script src="/jTAC-min/TypeManagers/Culture engine for jquery-globalize.js"
   type="text/javascript"></script>
<script src="/jTAC-min/TypeManagers/PowerDateParser.js" type="text/javascript"></script>
```

Conditions.BooleanLogic

jquery-validate rule name:	**booleanlogic**
Alias names (case sensitive):	"**BooleanLogic**"
Inherits from:	**Conditions.Base**
Source file:	**\jTAC\Conditions\BooleanLogic.js**

Defines an AND or OR expression using other Conditions.

For example:

Required on TextBox1 OR CompareToValue where TextBox2 > 10.

Your child Condition objects can be any subclass of Conditions.Base including Conditions.BooleanLogic, which means you can have child boolean expressions.

Evaluation rules

If using an AND operator, all child Conditions must evaluate as "success" for this condition to return "success".

If using an OR operator, at least one child Condition must evaluate as "success" for this condition to return "success".

If not, it returns "failed".

If a child Condition cannot evaluate, it is excluded from evaluation. If all child Conditions are excluded, Conditions.BooleanLogic returns "cannot evaluate".

Set up

Create Condition objects and add them to the **conditions** property. Then set the **operator** property to "AND" or "OR".

If you want to switch operators, add a Conditions.BooleanLogic with its **operator** set differently and populate its own **conditions** property to apply against its operator.

Use the **not** property if you want to reverse the result.

Examples to evaluate the following expression:

TextBox1 is required AND (((TextBox2 holds integer) AND (TextBox2 is in range 1 to 10)) OR ((TextBox3 holds integer) AND (TextBox3 is in range 1 to 10)))

Here is pseudo code describing the above expression with Conditions.

```
Conditions.BooleanLogic  operator="AND" with these child conditions:
   Conditions.Required elementId="TextBox1"
   Conditions.BooleanLogic operator="OR" with these child conditions:
      Conditions.BooleanLogic operator="AND" with these child conditions:
         Conditions.DataTypeCheck elementId="TextBox2" datatype="Integer"
         Conditions.Range elementId="TextBox2" datatype="Integer" minimum=1 maximum=10
      Conditions.BooleanLogic operator="AND" with these child conditions:
         Conditions.DataTypeCheck elementId="TextBox3" datatype="Integer"
         Conditions.Range elementId="TextBox3" datatype="Integer" minimum=1 maximum=10
```

Example: Validation using code

```
$("TextBox1").rule("add", {
   booleanlogic: {"operator": "AND",
  "conditions": [
    {"jtacClass": "Required", "elementId" : "TextBox1" },
    {"jtacClass" : "BooleanLogic", "operator": "OR",
     "conditions": [
      { "jtacClass" : "BooleanLogic", "operator": "AND",
        "conditions": [
         {"jtacClass" : "DataTypeCheck", "elementId" : "TextBox2" },
         {"jtacClass" : "Range", "elementId" : "TextBox2", "minimum" : 1, "maximum": 10 }
        ] },
      {"jtacClass" : "BooleanLogic", "operator": "AND",
        "conditions": [
         {"jtacClass" : "DataTypeCheck", "elementId" : "TextBox3" },
         {"jtacClass" : "Range", "elementId" : "TextBox3", "minimum" : 1, "maximum": 10 }
        ] }
      ] }
 ] }
});
```

Example: Unobtrusive validation

*Note: The formatting of the **json** property should not contain carriage returns.*

```
<input type="text" id="TextBox1" name="TextBox1"
   data-val="true"
   data-val-booleanlogic=""
   data-val-booleanlogic-json="{'jtacClass': 'BooleanLogic', 'operator': 'AND',
  'conditions': [
    {'jtacClass': 'Required', 'elementId': 'TextBox1' },
    {'jtacClass': 'BooleanLogic', 'operator': 'OR',
     'conditions': [
      { 'jtacClass': 'BooleanLogic', 'operator': 'AND',
        'conditions': [
         {'jtacClass': 'DataTypeCheck', 'elementId': 'TextBox2' },
         {'jtacClass': 'Range', 'elementId': 'TextBox2', 'minimum': 1, 'maximum': 10 }
        ] },
      {'jtacClass': 'BooleanLogic', 'operator': 'AND',
        'conditions': [
         {'jtacClass': 'DataTypeCheck', 'elementId': 'TextBox3' },
         {'jtacClass': 'Range', 'elementId': 'TextBox3', 'minimum': 1, 'maximum': 10 }
        ] }
      ] }
]}"/>
```

Example: Condition using code

```
var cond = jTAC.create("BooleanLogic", {operator: "AND"});
cond.conditions.push(jTAC.create("Required", {elementId: "TextBox1"}));
var condOR = jTAC.create("BooleanLogic", {operator: "OR"});
cond.conditions.push(condOR);
var condAND1 = jTAC.create("BooleanLogic", {operator: "AND"});
condOR.conditions.push(condAND1);
condAND1.conditions.push(jTAC.create("DataTypeCheck", {elementId: "TextBox2"}));
condAND1.conditions.push(jTAC.create("Range", {elementId: "TextBox2", minimum: 1, maximum: 10}));
var condAND2 = jTAC.create("BooleanLogic", {operator: "AND"});
condOR.conditions.push(condAND2);
condAND2.conditions.push(jTAC.create("DataTypeCheck", {elementId: "TextBox3"}));
condAND2.conditions.push(jTAC.create("Range", {elementId: "TextBox3", minimum: 1, maximum: 10}));
```

Example: Condition using JavaScript object

Note: The second parameter's formatting is shown for clarity. If you want to use formatting, split and concatenate the string.

```
var cond = jTAC.create("BooleanLogic",
   {"operator": "AND", "conditions": [
    {"jtacClass": "Required", "elementId" : "TextBox1" },
    {"jtacClass": "BooleanLogic", "operator": "OR",
     "conditions": [
      { "jtacClass": "BooleanLogic", "operator": "AND",
        "conditions": [
         {"jtacClass": "DataTypeCheck", "elementId" : "TextBox2" },
         {"jtacClass": "Range", "elementId": "TextBox2", "minimum": 1, "maximum": 10 }
        ] },
      {"jtacClass": "BooleanLogic", "operator": "AND",
        "conditions": [
         {"jtacClass": "DataTypeCheck", "elementId" : "TextBox3" },
         {"jtacClass": "Range", "elementId": "TextBox3", "minimum": 1, "maximum": 10 }
        ] }
      ] }
 ] });
```

Conditions.BooleanLogic Properties

- **operator** (string) - Determines the boolean operator. Only allows "AND", "OR".

  It defaults to "OR".

- **conditions** (array of Conditions) - Add Condition objects that define elements of the logic between each AND or OR operator.

  You can add any of these items to this array:

  - A Condition object with its properties assigned, including **elementId**.
  - A JavaScript object that identifies the connection class name in the **jtacClass** property, with the remaining properties setting values on the Condition it creates.

  You can add to this property in two ways:

  - Add one item with the `push()` method on the property.

```
cond.conditions.push(jTAC.create("Required", {elementId: "TextBox1"}));
cond.conditions.push({'jtacClass': 'DataTypeCheck', 'elementId': 'TextBox1'});
```

  - Replace the array with another array.

```
cond.conditions =
    [jTAC.create("Required", {elementId: "TextBox1"}),
     {'jtacClass': 'DataTypeCheck', 'elementId': 'TextBox1'}];
```

- **enabled** (boolean) - When `false`, do not evaluate. It defaults to `true`.
- **not** (boolean) - When `true`, apply a NOT operator to the result of evaluation, so "success" = 0 and "failed" = 1. ("Cannot evaluate" stays -1). It defaults to `false`.

Conditions.BooleanLogic tokens in jquery-validate error messages

- "{LABEL}" – The text used to label the field. This comes from either the `<label for="elementid" >` tag associated with the element, or one of these attributes on the element itself: **data-msglabel** or **data-msglabel-lookupkey**. Click here for more.
- "{ERRORLABEL}" - Replaced by the **messageLabel** option passed to `$(element).validate(options)`. This is a style sheet class, which defaults to "message-label". Use this in messages that have {LABEL} tokens. Enclose them in "`<span class='{ERRORLABEL}'>{LABEL}</span>`".

Conditions.CharacterCount

jquery-validate rule name:	**charactercount**
Alias names (case sensitive):	"**CharacterCount**"
Inherits from:	**Conditions.BaseCounter**
Source file:	**\jTAC\Conditions\CharacterCount.js**

Use to ensure the number of characters in a string are within a range. It can count across several elements.

Evaluation rules

Returns "success" when the number of characters is within the range.

Returns "failed" when the number of characters is outside the range.

Set up

Assign the element's id in the **elementId** property.

For additional elements, add their ids to the **moreConnections** property, which is an array. Use either its `push()` method or assign an array of ids.

Then assign the minimum and maximum of the range in the **minimum** and **maximum** properties. If one of those properties is not used, leave it unassigned.

Example: Validation using code

```
$("TextBox1").rule("add", {
   charactercount: {maximum: 10}
});
```

Example: Unobtrusive validation

```
<input type="text" id="TextBox1" name="TextBox1"
   data-val="true" data-val-charactercount="" data-val-charactercount-json="{'maximum': 10}" />
```

Example: Condition using code

```
var cond = jTAC.create("CharacterCount");
cond.elementId = "TextBox1";
cond.maximum = 10;

if (cond.isValid())
   // code runs when true
else
   // code runs when false
```

Example: Condition using JavaScript object

```
if (jTAC.isValid({jtacClass: 'CharacterCount', elementId: 'TextBox1', maximum: 10})
   // code runs when true
else
   // code runs when false
```

Conditions.CharacterCount Properties

- **minimum** (integer) – Determines the minimum of the count to report "success". If `null`, it is not used. Otherwise it must be a positive integer.
- **maximum** (integer) - Determines the maximum of the count to report "success". If `null`, it is not used. Otherwise it must be a positive integer.
- **elementId** (string) – Set this with the id of the first element to evaluate, such as the id of an <input type="text">. Alternatively, set the **connection** property to a Connection object.

- **moreConnections** (array) - Use when there is more than one element to evaluate. The first element always goes in the **elementId** property. The rest in **moreConnections**.

  You can add any of these items to this array:

  - id of the element as a string.
  - A Connection object with its properties assigned, including **Id**.
  - A JavaScript object that identifies the connection class name in the **jtacClass** property, with the remaining properties setting values on the Connection it creates.

  You can add to this property in two ways:

  - Add one item with the `push()` method on the property.

```
cond.moreConnections.push("TextBox1");
cond.moreConnections.push(jTAC.create("Connection.FormElement", {Id: "TextBox2"}));
cond.moreConnections.push({jtacClass: "Connection.FormElement", Id: "TextBox3"});
```

  - Replace the array with another array.

```
cond.moreConnections =
   ["TextBox1",
    jTAC.create("Connection.FormElement", {Id: "TextBox2"}),
    {jtacClass: "Connection.FormElement", Id: "TextBox3"}];
```

- **ignoreNotEditable** (boolean) - When there are two or more Connections used, this determines if elements that are not editable are counted.

  When `true`, not editable Connections are not counted.

  When `false`, they are.

  It defaults to `false`.

- **not** (boolean) - When `true`, apply a NOT operator to the result of evaluation, so "success" = 0 and "failed" = 1. ("Cannot evaluate" stays -1). It defaults to `false`.
- **trim** (boolean) - When `true` (the default), string values are trimmed before evaluating. It defaults to `true`.
- **enabled** (boolean) - When `false`, do not evaluate. It defaults to `true`.

Conditions.CharacterCount tokens in jquery-validate error messages

- "{LABEL}" – The text used to label the field. This comes from either the `<label for="elementid" >` tag associated with the element, or one of these attributes on the element itself: **data-msglabel** or **data-msglabel-lookupkey**. Click here for more.
- "{ERRORLABEL}" - Replaced by the **messageLabel** option passed to `$(element).validate(options)`. This is a style sheet class, which defaults to "message-label". Use this in messages that have {LABEL} tokens. Enclose them in "`<span class='{ERRORLABEL}'>{LABEL}</span>`".
- "{VALUE}" – Replaced by the element's actual string value.
- "{COUNT}" - Show the number of characters counted.
- "{COUNT:singular:plural}" - Show one of the two strings in singular or plural positions, depending on the count. If 1, show the singular form. For example: "You entered {COUNT} {COUNT:character:characters}."
- "{MINIMUM}" - Show the value of the **minimum** property.
- "{MAXIMUM}" - Show the value of the **maximum** property.
- "{DIFF}" - The number the count exceeds the maximum or is below the minimum.
- "{DIFF:singular:plural}" - Show one of the two strings in singular or plural positions, depending on the value of {DIFF}. If 1, show the singular form.

Conditions.CompareToValue

jquery-validate rule name:	**comparetovalue**
Alias names (case sensitive):	"**CompareToValue**"
Inherits from:	**Conditions.BaseOperator**
Source file:	**\jTAC\Conditions\CompareToValue.js**

Compares the value from an element or widget to a fixed value declared in the **valueToCompare** property.

Special case: If you have a checkbox or radio style HTML input, its **checked** property can be compared to a boolean stored in **valueToCompare**.

Evaluation rules

Returns "success" when the values compare according to the **operator**.

Returns "failed" when the values do not compare correctly.

Returns "cannot evaluate" when the element's value cannot be converted into the native type.

Set up

Assign the id of the element or widget to the **elementId** property.

Assign the value to compare to the **valueToCompare** property. This property can take native types, like a number or Date object. It also can take strings that it converts to the native type, so long as you use the culture neutral format for that data type.

Use the **operator** property to define the comparison with one of these strings: "=", "<>", "<", ">", "<=", ">=".

The TypeManager is normally identified by adding either **data-jtac-datatype** or **data-jtac-typemanager** attributes to the element. If that is not done, you can use the **datatype** or **typeManager** property to define the TypeManager. See "Using TypeManagers in datatype properties".

Example: Validation using code

```
$("TextBox1").rule("add", {
   comparetovalue: {'operator': '<', 'valueToCompare': 100}
});
```

Example: Unobtrusive validation

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="Integer"
   data-val="true"
   data-val-comparetovalue=""
   data-val-comparetovalue-json="{'operator': '<', 'valueToCompare': 100}" />
```

Example: Condition using code

```
var cond = jTAC.create("CompareToValue");
cond.elementId = "TextBox1";
cond.operator = '<';
cond.valueToCompare = 100;

if (cond.isValid())
   // code runs when true
else
   // code runs when false
```

Example: Condition using JavaScript object

```
if (jTAC.isValid({jtacClass: 'CompareToValue', elementId: 'TextBox1', operator: '<',
      valueToCompare: 100}))
   // code runs when true
else
   // code runs when false
```

Conditions.CompareToValue Properties

- **elementId** (string) – Set this with the id of the element to evaluate, such as the id of an <input type="text">. Alternatively, set the **connection** property to a Connection object.
- **operator** (string) - Supports these values: "=", "<>", "<", ">", "<=", ">=". It defaults to "=".
- **valueToCompare** - The value to compare on the right side of the **operator** in the boolean expression.

  Use a string or any value that is compatible with the TypeManager for this element.

  If it is a string, its format must be culture neutral. Here are the culture neutral formats of existing TypeManagers:

Type	Pattern	Examples
Integers	[-]digits	`"1", "10000", "-1"`
Float, Currency, Percent	[-]digits.digits	`"1.0", "10000.0", "-1.0"`
Date	yyyy-MM-dd	`"2000-05-02"`
Time of Day, Duration	H:mm:ss	`"0:00:00", "16:30:21"`
Date and Time	yyyy-MM-dd H:mm:ss	`"2000-05-02 16:30:21"`
Day Month	MM-dd	`"05-02"`
Month Year	yyyy-MM	`"2000-05"`

- **datatype** (string) – Identifies the TypeManager to use in the **typeManager** property by a name.

  Some of the names supported include: "Integer", "Float", "Currency", "Percent", "Date", "DateTime", "TimeOfDay", "Duration", "DayMonth", "MonthYear", and "Boolean". See "Using TypeManagers in datatype properties" for more.

  If you don't assign either **datatype** or **typeManager**, the **typeManager** property requests the TypeManager object from the Connection object pointing to the element or widget. That object knows how to find the **data-jtac-datatype** and **data-jtac-typemanager** attributes on the input field specified by the connection, to define the TypeManager.
- **typeManager** (subclass of TypeManagers.Base) – The TypeManager object. You can define it directly with a TypeManager object, use the **datatype** property, or let this property look it up through the Connection object.
- **not** (boolean) - When `true`, apply a NOT operator to the result of evaluation, so "success" = 0 and "failed" = 1. ("Cannot evaluate" stays -1). It defaults to `false`.
- **trim** (boolean) - When `true` (the default), string values are trimmed before evaluating. It defaults to `true`.
- **enabled** (boolean) - When `false`, do not evaluate. It defaults to `true`.

Conditions.CompareToValue tokens in jquery-validate error messages

- "{LABEL}" – The text used to label the field. This comes from either the `<label for="elementid" >` tag associated with the element, or one of these attributes on the element itself: **data-msglabel** or **data-msglabel-lookupkey**. Click here for more.
- "{ERRORLABEL}" - Replaced by the **messageLabel** option passed to `$(element).validate(options)`. This is a style sheet class, which defaults to "message-label". Use this in messages that have {LABEL} tokens. Enclose them in "`<span class='{ERRORLABEL}'>{LABEL}</span>`".
- "{VALUE}" – Replaced by the element's actual string value.
- "{OPERATOR}" - Describes the **operator**. It defaults to the same strings stored in the **operator** property (such as "=" and "<>"). You can override by using the jTAC.Translations system, and editing the properties that are the same as the operator values. ("=", "<=", etc)
- "{VALUETOCOMPARE}" - Show the value of the **valueToCompare** property. If that value is not a string, it will be converted into a string applying formatting from the TypeManager's rules, which includes localization.

Conditions.CompareTwoElements

jquery-validate rule name:	**comparetwoelements**
Alias names (case sensitive):	"**CompareTwoElements**"
Inherits from:	**Conditions.BaseTwoConnections**
Source file:	**\jTAC\Conditions\CompareTwoElements.js**

Compares two elements or widgets to each other, using any comparison operator defined in the **operator** property.

Evaluation rules

Returns "success" when the values compare according to the **operator**.

Returns "failed" when the values do not compare correctly.

Returns "cannot evaluate" when either of the element's values cannot be converted into the native type.

Set up

Assign one element's id to the **elementId** property and the other to the **elementId2** property.

Use the **operator** property to define the comparison with one of these strings: "=", "<>", "<", ">", "<=", ">=".

The TypeManager is normally identified by adding either **data-jtac-datatype** or **data-jtac-typemanager** attributes to the element. If that is not done, you can use the **datatype** or **typeManager** property to define the TypeManager. See "Using TypeManagers in datatype properties".

Example: Validation using code

```
$("TextBox1").rule("add", {
   comparetwoelements: {'elementId2' :'TextBox2', 'operator': '<=' }
});
```

Example: Unobtrusive validation

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="Integer"
   data-val="true" data-val-comparetwoelements=""
   data-val-comparetwoelements-json="{'elementId2' :'TextBox2', 'operator': '<=' }" />
```

Example: Condition using code

```
var cond = jTAC.create("CompareTwoElements");
cond.elementId = "TextBox1";
cond.elementId2 = "TextBox2";
cond.operator = "<=";

if (cond.isValid())
   // code runs when true
else
   // code runs when false
```

Example: Condition using JavaScript object

```
if (jTAC.isValid({jtacClass: 'CompareTwoElements', elementId: 'TextBox1',
   elementId2: 'TextBox2', operator: '<='}))
   // code runs when true
else
   // code runs when false
```

Conditions.CompareTwoElements Properties

- **elementId** (string) – Set this with the id of the first element to evaluate, such as the id of an <input type="text">. Alternatively, set the **connection** property to a Connection object.
- **elementId2** (string) – Set this with the id of the second element to evaluate, such as the id of an <input type="text">. Alternatively, set the **connection2** property to a Connection object.
- **operator** (string) - Supports these values: "=", "<>", "<", ">", "<=", ">=". It defaults to "=".
- **datatype** (string) – Identifies the TypeManager to use in the **typeManager** property by a name.

  Some of the names supported include: "Integer", "Float", "Currency", "Percent", "Date", "DateTime", "TimeOfDay", "Duration", "DayMonth", "MonthYear", and "Boolean". See "Using TypeManagers in datatype properties" for more.

  If you don't assign either **datatype** or **typeManager**, the **typeManager** property requests the TypeManager object from the Connection object pointing to the element or widget. That object knows how to find the **data-jtac-datatype** and **data-jtac-typemanager** attributes on the input field specified by the connection, to define the TypeManager.
- **typeManager** (subclass of TypeManagers.Base) – The TypeManager object. You can define it directly with a TypeManager object, use the **datatype** property, or let this property look it up through the Connection object.
- **not** (boolean) - When `true`, apply a NOT operator to the result of evaluation, so "success" = 0 and "failed" = 1. ("Cannot evaluate" stays -1). It defaults to `false`.
- **trim** (boolean) - When `true` (the default), string values are trimmed before evaluating. It defaults to `true`.
- **enabled** (boolean) - When `false`, do not evaluate. It defaults to `true`.

Conditions.CompareTwoElements tokens in jquery-validate error messages

- "{LABEL}" – The text used to label the field. This comes from either the `<label for="elementid" >` tag associated with the element, or one of these attributes on the element itself: **data-msglabel** or **data-msglabel-lookupkey**. Click here for more.
- "{ERRORLABEL}" - Replaced by the **messageLabel** option passed to `$(element).validate(options)`. This is a style sheet class, which defaults to "message-label". Use this in messages that have {LABEL} tokens. Enclose them in "`<span class='{ERRORLABEL}'>{LABEL}</span>`".
- "{VALUE}" – Replaced by the element's actual string value.
- "{LABEL2}" - The label associated with the second element. Uses the same rules as with "{LABEL}", above.
- "{VALUE2}" - The second element's actual string value.
- "{OPERATOR}" - Describes the **operator**. It defaults to the same strings stored in the **operator** property (such as "=" and "<>"). You can override by using the jTAC.Translations system, and editing the properties that are the same as the operator values. ("=", "<=", etc)

Conditions.CountSelections

jquery-validate rule name:	**countselections**
Alias names (case sensitive):	"**CountSelections**"
Inherits from:	Conditions.BaseCounter
Source file:	**\jTAC\Conditions\CountSelections.js**

Evaluates list-style elements that support multiple selections, such as the <select> tag. It counts the number of selections and compares the value to a range. Supports Checkbox lists and Radio Button lists by using features built into the Connections.FormElement object.

Although it is typical to use a single Connection, you can total multiple connections by specifying the first in the *elementId* property and the rest in the *moreConnections* property.

Evaluation rules

Returns "success" when the number of selections is within the range.

Returns "failed" when the number of selections is outside the range.

Set up

For the first element, set the element's id in the **elementId** property.

For additional elements, add their ids to the **moreConnections** property, which is an array. Use either its `push()` method or assign an array of ids.

Then assign the minimum and maximum of the range in the **minimum** and **maximum** properties. If one of those properties is not used, leave it unassigned.

Example: Validation using code

```
$("ListBox1").rule("add", {
   countselections: {maximum: 10}
});
```

Example: Unobtrusive validation

```
<input type="text" id="TextBox1" name="TextBox1"
   data-val="true"
   data-val-countselections="" data-val-countselections-json="{'maximum': 10}" />
```

Example: Condition using code

```
var cond = jTAC.create("CountSelections");
cond.elementId = "ListBox1";
cond.maximum = 10;

if (cond.isValid())
   // code runs when true
else
   // code runs when false
```

Example: Condition using JavaScript object

```
if (jTAC.isValid({jtacClass: 'CountSelections', elementId: 'ListBox1', maximum: 10}))
   // code runs when true
else
   // code runs when false
```

Conditions.CountSelections Properties

- **minimum** (integer) – Determines the minimum of the count to report "success". If `null`, it is not used. Otherwise it must be a positive integer.
- **maximum** (integer) - Determines the maximum of the count to report "success". If `null`, it is not used. Otherwise it must be a positive integer.
- **elementId** (string) – Set this with the id of the first element to evaluate, such as the id of a <select> tag. Alternatively, set the **connection** property to a Connection object.
- **moreConnections** (array) - Use when there is more than one element to evaluate. The first element always goes in the **elementId** property. The rest in **moreConnections**.

  You can add any of these items to this array:

  - id of the element as a string.
  - A Connection object with its properties assigned, including **Id**.
  - A JavaScript object that identifies the connection class name in the **jtacClass** property, with the remaining properties setting values on the Connection it creates.

  You can add to this property in two ways:

  - Add one item with the `push()` method on the property.

```
cond.moreConnections.push("TextBox2");
cond.moreConnections.push(jTAC.create("Connection.FormElement", {Id: "ListBox3"}));
cond.moreConnections.push({jtacClass: "Connection.FormElement", Id: "ListBox4"});
```

  - Replace the array with another array.

```
cond.moreConnections =
    ["ListBox2",
     jTAC.create("Connection.FormElement", {Id: "ListBox3"}),
     {jtacClass: "Connection.FormElement", Id: "ListBox4"}];
```

- **ignoreNotEditable** (boolean) - When there are two or more Connections used, this determines if elements that are not editable are counted.

  When `true`, not editable Connections are not counted.

  When `false`, they are.

  It defaults to `false`.

- **not** (boolean) - When `true`, apply a NOT operator to the result of evaluation, so "success" = 0 and "failed" = 1. ("Cannot evaluate" stays -1). It defaults to `false`.
- **trim** (boolean) - When `true` (the default), string values are trimmed before evaluating. It defaults to `true`.
- **enabled** (boolean) - When `false`, do not evaluate. It defaults to `true`.

Conditions.CountSelections tokens in jquery-validate error messages

- "{LABEL}" – The text used to label the field. This comes from either the `<label for="elementid" >` tag associated with the element, or one of these attributes on the element itself: **data-msglabel** or **data-msglabel-lookupkey**. Click here for more.
- "{ERRORLABEL}" - Replaced by the **messageLabel** option passed to `$(element).validate(options)`. This is a style sheet class, which defaults to "message-label". Use this in messages that have {LABEL} tokens. Enclose them in "`<span class='{ERRORLABEL}'>{LABEL}</span>`".
- "{VALUE}" – Replaced by the element's actual string value.
- "{COUNT}" - Show the number of words counted.
- "{COUNT:singular:plural}" - Show one of the two strings in singular or plural positions, depending on the count. If 1, show the singular form. For example: "You entered {COUNT} {COUNT:word:words}."
- "{MINIMUM}" - Show the value of the **minimum** property.
- "{MAXIMUM}" - Show the value of the **maximum** property.
- "{DIFF}" - The number the count exceeds the maximum or is below the minimum.
- "{DIFF:singular:plural}" - Show one of the two strings in singular or plural positions, depending on the value of {DIFF}. If 1, show the singular form.

Conditions.DataTypeCheck

jquery-validate rule name:	**datatypecheck**
Alias names (case sensitive):	"**DataTypeCheck**"
Inherits from:	**Conditions.BaseOneConnection**
Source file:	**\jTAC\Conditions\DataTypeCheck.js**

Ensures the text can be converted to the native type as defined by a TypeManager.

Evaluation rules

Returns "success" when the text can be converted.

Returns "failed" when the text cannot be converted.

Returns "cannot evaluate" when the text is the empty string or represents null.

Set up

Assign the **elementId** property to the element's id whose text value will be evaluated.

The TypeManager is normally identified by adding either **data-jtac-datatype** or **data-jtac-typemanager** attributes to the element. If that is not done, you can use the **datatype** or **typeManager** property to define the TypeManager. See "Using TypeManagers in datatype properties".

Example: Validation using code

```
$("TextBox1").rule("add", {
   datatypecheck: { }
});
```

Example: Unobtrusive validation

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="date"
   data-val="true"  data-val-datatypecheck="" data-val-datatypecheck-json="{}" />
```

Also (since no parameters are common):

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="date"
   data-val="true" data-val-datatypecheck="" />
```

Example: Condition using code

```
var cond = jTAC.create("DataTypeCheck");
cond.elementId = "TextBox1";

if (cond.isValid())
   // code runs when true
else
   // code runs when false
```

Example: Condition using JavaScript object

```
if (jTAC.isValid({jtacClass: 'datatypecheck', elementId: 'TextBox1'}))
   // code runs when true
else
   // code runs when false
```

Conditions.DataTypeCheck Properties

- **elementId** (string) – Set this with the id of the first element to evaluate, such as the id of an <input type="text">. Alternatively, set the **connection** property to a Connection object.
- **datatype** (string) – Identifies the TypeManager to use in the **typeManager** property by a name.

  Some of the names supported include: “Integer”, “Float”, “Currency”, “Percent”, “Date”, “DateTime”, “TimeOfDay”, “Duration”, “DayMonth”, “MonthYear”, and “Boolean”. See “Using TypeManagers in datatype properties” for more.

  If you don’t assign either **datatype** or **typeManager**, the **typeManager** property requests the TypeManager object from the Connection object pointing to the element or widget. That object knows how to find the **data-jtac-datatype** and **data-jtac-typemanager** attributes on the input field specified by the connection, to define the TypeManager.
- **typeManager** (subclass of TypeManagers.Base) – The TypeManager object. You can define it directly with a TypeManager object, use the **datatype** property, or let this property look it up through the Connection object.
- **not** (boolean) - When `true`, apply a NOT operator to the result of evaluation, so "success" = 0 and "failed" = 1. ("Cannot evaluate" stays -1). It defaults to `false`.
- **trim** (boolean) - When `true` (the default), string values are trimmed before evaluating. It defaults to `true`.
- **enabled** (boolean) - When `false`, do not evaluate. It defaults to `true`.

Conditions.DataTypeCheck tokens in jquery-validate error messages

- “{LABEL}” – The text used to label the field. This comes from either the `<label for="elementid" >` tag associated with the element, or one of these attributes on the element itself: **data-msglabel** or **data-msglabel-lookupkey**. Click here for more.
- “{ERRORLABEL}” - Replaced by the **messageLabel** option passed to `$(element).validate(options)`. This is a style sheet class, which defaults to “message-label”. Use this in messages that have {LABEL} tokens. Enclose them in “`<span class='{ERRORLABEL}'>{LABEL}</span>`”.
- “{VALUE}” – Replaced by the element’s actual string value.
- “{DATATYPE}” - The name of the data type, as defined on the TypeManager’s **friendlyName** property. You can override it by assigning the **friendlyName** property. When you need localization, add a key to the \jTAC\Translations\ script files with the desired text. Then assign that key to the TypeManager’s **friendlyLookupKey** property.

  The easiest way to assign both of these properties is by adding the data-jtac-typemanager attribute to the HTML tag of the element. This attribute hosts JSON that specifies any properties you want to set on the TypeManager.

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="date"
   data-jtac-typemanager="{'friendlyName': 'name', 'friendlyLookupKey': 'keyname'}"
   data-val="true" data-val-datatypecheck="" />
```

Conditions.Difference

jquery-validate rule name:	**difference** (case sensitive)
Alias names (case sensitive):	"**Difference**"
Inherits from:	**Conditions.BaseTwoConnections**
Source file:	**\jTAC\Conditions\Difference.js**

Compares the difference between two values to another number in the **differenceValue** property.

The **operator** property defines the comparison rule.

Evaluation rules

Returns "success" when the values compare according to the **operator**.

Returns "failed" when the values do not compare correctly.

Returns "cannot evaluate" when either of the element's values cannot be converted into the native type.

Set up

Assign one element's id to the **elementId** property and the other to the **elementId2** property. These two will have their values calculated as `Math.abs(elementId - elementId2)`. The result is compared to the value of **differenceValue**. Assign a number to **differenceValue** and an **operator** to determine how the elements are compared to **differenceValue**.

The TypeManager is normally identified by adding either **data-jtac-datatype** or **data-jtac-typemanager** attributes to the element. If that is not done, you can use the **datatype** or **typeManager** property to define the TypeManager. See "Using TypeManagers in datatype properties".

Example: Validation using code

```
$("TextBox1").rule("add", {
   difference: {elementId2: 'TextBox2', operator: '<=', differenceValue: 5 }
});
```

Example: Unobtrusive validation

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="Integer"
   data-val="true" data-val-difference=""
   data-val-difference-json="{'elementId2': 'TextBox2', 'operator': '<=', 'differenceValue': 5 }"
/>
```

Example: Condition using code

```
var cond = jTAC.create("Difference");
cond.elementId = "TextBox1";
cond.elementId2 = "TextBox2";
cond.differenceValue = 5;
cond.operator = "<=";

if (cond.isValid())
   // code runs when true
else
   // code runs when false
```

Example: Condition using JavaScript object

```
if (jTAC.isValid({jtacClass: 'Difference', elementId: 'TextBox1',
   elementId2: 'TextBox2', operator: '<=', differenceValue: 5}))
   // code runs when true
else
   // code runs when false
```

Conditions.Difference Properties

- **elementId** (string) – Set this with the id of the first element to evaluate, such as the id of an <input type="text">. Alternatively, set the **connection** property to a Connection object.
- **elementId2** (string) – Set this with the id of the second element to evaluate, such as the id of an <input type="text">. Alternatively, set the **connection2** property to a Connection object.
- **operator** (string) - Supports these values: "=", "<>", "<", ">", "<=", ">=". It defaults to "=".
- **differenceValue** (number)- The value to compare on the right side of the **operator**. This must be an integer or float number.

  How this is used depends on the TypeManager's `toNumber()` method. For number types, 1 means 1. For Date, 1 means 1 day. For DateTime, TimeOfDay, or Duration, 1 means 1 second. For MonthYear, 1 means 1 month.

  It defaults to 1.
- **datatype** (string) – Identifies the TypeManager to use in the **typeManager** property by a name.

  Some of the names supported include: "Integer", "Float", "Currency", "Percent", "Date", "DateTime", "TimeOfDay", "Duration", "DayMonth", and "MonthYear". See "Using TypeManagers in datatype properties" for more.

  If you don't assign either **datatype** or **typeManager**, the **typeManager** property requests the TypeManager object from the Connection object pointing to the element or widget. That object knows how to find the **data-jtac-datatype** and **data-jtac-typemanager** attributes on the input field specified by the connection, to define the TypeManager.
- **typeManager** (subclass of TypeManagers.Base) – The TypeManager object. You can define it directly with a TypeManager object, use the **datatype** property, or let this property look it up through the Connection object.
- **not** (boolean) - When `true`, apply a NOT operator to the result of evaluation, so "success" = 0 and "failed" = 1. ("Cannot evaluate" stays -1). It defaults to `false`.
- **trim** (boolean) - When `true` (the default), string values are trimmed before evaluating. It defaults to `true`.
- **enabled** (boolean) - When `false`, do not evaluate. It defaults to `true`.

Conditions.Difference tokens in jquery-validate error messages

- "{LABEL}" – The text used to label the field. This comes from either the `<label for="elementid" >` tag associated with the element, or one of these attributes on the element itself: **data-msglabel** or **data-msglabel-lookupkey**. Click here for more.
- "{ERRORLABEL}" - Replaced by the **messageLabel** option passed to `$(element).validate(options)`. This is a style sheet class, which defaults to "message-label". Use this in messages that have {LABEL} tokens. Enclose them in "`<span class='{ERRORLABEL}'>{LABEL}</span>`".
- "{VALUE}" – Replaced by the element's actual string value.
- "{LABEL2}" - The label associated with the second element. Uses the same rules as with "{LABEL}", above.
- "{VALUE2}" - The second element's actual string value.
- "{OPERATOR}" - Describes the **operator**. It defaults to the same strings stored in the **operator** property (such as "=" and "<>"). You can override by using the jTAC.Translations system, and editing the properties that are the same as the operator values. ("=", "<=", etc)
- "{DIFFVALUE}" - Show the value of the **differenceValue** property. If that value is not a string, it will be converted into a string. It will not use the TypeManager assigned to the **typeManager** property because the **differenceValue** is not usually the same type as the values in elements. Instead, it uses a new TypeManagers.Float instance that strips all trailing zeros.

Conditions.DuplicateEntry

jquery-validate rule name:	**duplicateentry**
Alias names (case sensitive):	"**DuplicateEntry**"
Inherits from:	**Conditions.BaseOneOrMoreConnections**
Source file:	**\jTAC\Conditions\DuplicateEntry.js**

Ensures that the list of elements or widgets have different values.

If you are comparing list type widgets, it compares their textual values, not their index or the text shown the user.

Evaluation rules

If a pair is found to have the same string value, it evaluates as "failed".

Otherwise it evaluates as "success".

Set up

For the first Connection, set the widget's id in the **elementId** property.

For additional Connections, add their ids to the **moreConnections** property, which is an array. Use either its push() method or assign an array of ids.

Determine if comparisons are case insensitive with **caseIns** and if unassigned fields are included with **ignoreUnassigned**.

Example: Validation using code

```
$("TextBox1").rule("add", {
   duplicateentry: { moreConnections : ['TextBox2', 'TextBox3'] }
});
```

Example: Unobtrusive validation

```
<input type="text" id="TextBox1" name="TextBox1"
   data-val="true" data-val-duplicateentry=""
   data-val-duplicateentry-json="{'moreConnections': ['TextBox2', 'TextBox3'] }" />
```

Example: Condition using code

```
var cond = jTAC.create("DuplicateEntry");
cond.elementId = "TextBox1";
cond.moreConnections = ['TextBox2', 'TextBox3'];

if (cond.isValid())
   // code runs when true
else
   // code runs when false
```

Example: Condition using JavaScript object

```
if (jTAC.isValid({jtacClass: 'DuplicateEntry', elementId: 'TextBox1',
   moreConnections : ['TextBox2', 'TextBox3'] }))
   // code runs when true
else
   // code runs when false
```

Conditions.DuplicateEntry Properties

- **elementId** (string) – Set this with the id of the first element to evaluate, such as the id of an <input type="text">. Alternatively, set the **connection** property to a Connection object.
- **moreConnections** (array) – Use for the remaining elements.

  You can add any of these items to this array:

  - id of the element as a string.
  - A Connection object with its properties assigned, including **Id**.
  - A JavaScript object that identifies the connection class name in the **jtacClass** property, with the remaining properties setting values on the Connection it creates.

  You can add to this property in two ways:

  - Add one item with the `push()` method on the property.

```
cond.moreConnections.push("TextBox1");
cond.moreConnections.push(jTAC.create("Connection.FormElement", {Id: "TextBox2"}));
cond.moreConnections.push({jtacClass: "Connection.FormElement", Id: "TextBox3"});
```

  - Replace the array with another array.

```
cond.moreConnections =
    ["TextBox1",
     jTAC.create("Connection.FormElement", {Id: "TextBox2"}),
     {jtacClass: "Connection.FormElement", Id: "TextBox3"}];
```

- **caseIns** (boolean) - When `true`, strings are compared using a case insensitive match.

  It defaults to `true`.

- **ignoreUnassigned** (boolean) - When `true`, if the data entry control's value is unassigned, it is never matched. When `false`, the data entry control's value is always used, even when blank.

  It defaults to `true`.

- **not** (boolean) - When `true`, apply a NOT operator to the result of evaluation, so "success" = 0 and "failed" = 1. ("Cannot evaluate" stays -1). It defaults to `false`.
- **trim** (boolean) - When `true` (the default), string values are trimmed before evaluating. It defaults to `true`.
- **enabled** (boolean) - When `false`, do not evaluate. It defaults to `true`.

Conditions.DuplicateEntry tokens in jquery-validate error messages

- "{LABEL1}" – The label associated with the first field with a matching value. This comes from either the `<label for="elementid" >` tag associated with the element, or one of these attributes on the element itself: **data-msglabel** or **data-msglabel-lookupkey**. Click here for more.
- "{LABEL2}" - The label associated with the second field with a matching value. It uses the same rules as {LABEL1} to acquire its text.
- "{ERRORLABEL}" - Replaced by the **messageLabel** option passed to `$(element).validate(options)`. This is a style sheet class, which defaults to "message-label". Use this in messages that have {LABEL} tokens. Enclose them in "`<span class='{ERRORLABEL}'>{LABEL}</span>`".
- "{VALUE}" – Replaced by the element's actual string value.

Conditions.Range

jquery-validate rule name:	**advrange**
Alias names (case sensitive):	"**Range**"
Inherits from:	**Conditions.BaseOneConnection**
Source file:	**\jTAC\Conditions\Range.js**

Compares the value from a form element or widget to a range established by the **minimum** and **maximum** properties.

Evaluation rules

Returns "success" when the value is within the range.

Returns "failed" when the value is outside the range.

Returns "cannot evaluate" when the element's value cannot be converted into the native type.

Set up

Assign the element's id to the **elementId** property.

Assign the range with the **minimum** and **maximum** properties. This property can take native types, like a number or Date object. It also can take strings that it converts to the native type, so long as you use the culture neutral format for that data type.

The TypeManager is normally identified by adding either **data-jtac-datatype** or **data-jtac-typemanager** attributes to the element. If that is not done, you can use the **datatype** or **typeManager** property to define the TypeManager. See "Using TypeManagers in datatype properties".

The TypeManager also has **minValue** and **maxValue** properties when you include the **TypeManagers\Command extensions.js** file or **Merged\TypeManagers\datatypeeditor.js** file.

The Conditions.Range will use those properties if its own **minimum** and **maximum** are null.

Example: Validation using code

```
$("TextBox1").rule("add", {
   range: {'minimum': 1, 'maximum': 10}
});
```

Example: Unobtrusive validation

```
<input type="text" id="TextBox1" name="TextBox1"
   data-jtac-datatype="Integer"
   data-val="true" data-val-advrange="" data-val-advrange-json="{'minimum': 1, 'maximum': 10}" />
```

Example: Condition using code

```
var cond = jTAC.create("Range");
cond.elementId = "TextBox1";
cond.minimum = 1;
cond.maximum = 10;

if (cond.isValid())
   // code runs when true
else
   // code runs when false
```

Example: Condition using JavaScript object

```
if (jTAC.isValid({jtacClass: 'Range', elementId: 'TextBox1', minimum: 1, maximum: 10}))
   // code runs when true
else
   // code runs when false
```

Conditions.Range Properties

- **elementId** (string) – Set this with the id of the element to evaluate, such as the id of an <input type="text">. Alternatively, set the **connection** property to a Connection object.
- **minimum** - The minimum value. If `null`, it is not evaluated.

  Use a string or any value that is compatible with the TypeManager for this element.

  If it is a string, its format must be culture neutral. Here are the culture neutral formats of existing TypeManagers:

Type	Pattern	Examples
Integers	[-]digits	`"1", "10000", "-1"`
Float, Currency, Percent	[-]digits.digits	`"1.0", "10000.0", "-1.0"`
Date	yyyy-MM-dd	`"2000-05-02"`
Time of Day, Duration	H:mm:ss	`"0:00:00", "16:30:21"`
Date and Time	yyyy-MM-dd H:mm:ss	`"2000-05-02 16:30:21"`
Day Month	MM-dd	`"05-02"`
Month Year	yyyy-MM	`"2000-05"`

- **maximum** - The maximum value. If `null`, it is not evaluated.

  Use a string or any value that is compatible with the TypeManager for this element.

  If it is a string, its format must be culture neutral. See above.
- **lessThanMax** (boolean) - When `true`, evaluate less than the maximum. When false, evaluate less than or equals the maximum.

  It defaults to `false`.
- **datatype** (string) – Identifies the TypeManager to use in the **typeManager** property by a name.

  Some of the names supported include: “Integer”, “Float”, “Currency”, “Percent”, “Date”, “DateTime”, “TimeOfDay”, “Duration”, “DayMonth”, and “MonthYear” See “Using TypeManagers in datatype properties” for more.

  If you don’t assign either **datatype** or **typeManager**, the **typeManager** property requests the TypeManager object from the Connection object pointing to the element or widget. That object knows how to find the **data-jtac-datatype** and **data-jtac-typemanager** attributes on the input field specified by the connection, to define the TypeManager.
- **typeManager** (TypeManager object) – The TypeManager object. You can define it directly with a TypeManager object, use the **datatype** property, or let this property look it up through the Connection object.
- **not** (boolean) - When `true`, apply a NOT operator to the result of evaluation, so "success" = 0 and "failed" = 1. ("Cannot evaluate" stays -1). It defaults to `false`.
- **trim** (boolean) - When `true` (the default), string values are trimmed before evaluating. It defaults to `true`.
- **enabled** (boolean) - When `false`, do not evaluate. It defaults to `true`.

Conditions.Range tokens in jquery-validate error messages

- "{LABEL}" – The text used to label the field. This comes from either the `<label for="elementid" >` tag associated with the element, or one of these attributes on the element itself: **data-msglabel** or **data-msglabel-lookupkey**. Click here for more.
- "{ERRORLABEL}" - Replaced by the **messageLabel** option passed to `$(element).validate(options)`. This is a style sheet class, which defaults to "message-label". Use this in messages that have {LABEL} tokens. Enclose them in "`<span class='{ERRORLABEL}'>{LABEL}</span>`".
- "{VALUE}" – Replaced by the element's actual string value.
- "{MINIMUM}" - Show the value of the **minimum** property. If that value is not a string, it will be converted into a string applying formatting from the TypeManager's rules, which includes localization.
- "{MAXIMUM}" - Show the value of the **maximum** property. If that value is not a string, it will be converted into a string applying formatting from the TypeManager's rules, which includes localization.

Conditions.RegExp

jquery-validate rule name:	**regexp**
Alias names (case sensitive):	"**RegExp**"
Inherits from:	**Conditions.BaseRegExp**
Source file:	**\jTAC\Conditions\RegExp.js**

Use a regular expression to evaluate the text in the html element or widget.

This class allows the user to define the expression explicitly in its **expression** property. There may be other Conditions that also generate the correct expression for a given business rule. Consider using them as they better document the rule and are often an easier way to define the correct expression. If you want to create such a rule, subclass from Conditions.BaseRegExp.

Evaluation rules

When the text is matched by the regular expression, it evaluates as "success".

Otherwise it evaluates as "failed".

When the text is the empty string, evaluates as "cannot evaluate" so long as **ignoreBlankText** is `true`.

If you want to reverse this logic, set the **not** property to `true`.

Set up

Assign the element's id to the **elementId** property.

Develop a regular expression and assign it to the **expression** property. Hint: If you intend to evaluate the entire string, be sure to use the ^ and $ symbols. Always use "\\" whenever you normally use the special regex symbol "\".

Regex Resources:

- Regular expression special characters
- Find and test expressions

Regular expressions have options which you can set with the **caseIns**, **global**, and **multiline** properties.

Example: Validation using code

```
$("TextBox1").rule("add", {
   regexp: {'expression': '^\\d{5}$' }
});
```

Example: Unobtrusive validation

```
<input type="text" id="TextBox1" name="TextBox1"
   data-val="true" data-val-regexp="" data-val-regexp-json="{'expression': '^\\d{5}$'}" />
```

Example: Condition using code

```
var cond = jTAC.create("RegExp");
cond.elementId = "TextBox1";
cond.expression = '^\\d{5}$';

if (cond.isValid())
   // code runs when true
else
   // code runs when false
```

Example: Condition using JavaScript object

```
if (jTAC.isValid({ 'jtacClass': 'RegExp', 'elementId' : 'TextBox1', 'expression': '^\\d{5}$' }))
   // code runs when true
else
   // code runs when false
```

Conditions.RegExp Properties

- **elementId** (string) – Set this with the id of the element to evaluate, such as the id of an <input type="text">. Alternatively, set the **connection** property to a Connection object.
- **expression** (string) - A valid regular expression pattern. Hint: If you intend to evaluate the entire string, be sure to use the ^ and $ symbols. Always use "\\" whenever you normally use the special regex symbol "\".

  Regex Resources:

  - Regular expression special characters
  - Find and test expressions
- **caseIns** (boolean) - When `true`, letters are matched case insensitively.

  It defaults to `true`.
- **global** (boolean) - When `true`, use the global search option of regular expressions.

  It defaults to `false`.
- **multiline** (boolean) - When `true`, use the multiline option of regular expressions.

  It defaults to `false`.
- **ignoreBlankText** (boolean) - Determines how blank text is evaluated.

  When `true`, the Condition evaluates as "cannot evaluate".

  When `false`, the condition evaluates as "failed".

  It defaults to `true`.
- **not** (boolean) - When `true`, apply a NOT operator to the result of evaluation, so "success" = 0 and "failed" = 1. ("Cannot evaluate" stays -1). It defaults to `false`.
- **trim** (boolean) - When `true` (the default), string values are trimmed before evaluating. It defaults to `true`.
- **enabled** (boolean) - When `false`, do not evaluate. It defaults to `true`.

Conditions.RegExp tokens in jquery-validate error messages

- "{LABEL}" – The text used to label the field. This comes from either the `<label for="elementid" >` tag associated with the element, or one of these attributes on the element itself: **data-msglabel** or **data-msglabel-lookupkey**. Click here for more.
- "{ERRORLABEL}" - Replaced by the **messageLabel** option passed to `$(element).validate(options)`. This is a style sheet class, which defaults to "message-label". Use this in messages that have {LABEL} tokens. Enclose them in "`<span class='{ERRORLABEL}'>{LABEL}</span>`".
- "{VALUE}" – Replaced by the element's actual string value.
- "{PATTERN}" - Shows text defined by the **patternLabel** property which is often used to define the type of data, like "Phone number" or "Street Address". If localizing text, add a new key to the **\jTAC\Translations** script files with the desired text. Then assign that key to the **patternLookupKey** property on the Condition.

```
<input type="text" id="TextBox1" name="TextBox1"
   data-val="true" data-val-regexp=""
   data-val-regexp-json="{'expression': '^\\d{5}$', 'patternLabel': 'Postal code' }" />
```

Conditions.Required

jquery-validate rule name:	**advrequired**
Alias names (case sensitive):	"**Required**"
Inherits from:	**Conditions.BaseOneOrMoreConnections**
Source file:	**\jTAC\Conditions\Required.js**

One or more elements are evaluated to see if its value is not unassigned, empty, or null.

You can evaluate the value of it to any widget: textbox, list, checkbox, radio, calendar widget, etc.

Evaluation rules

Uses the Connection class' `isNullValue()` method to determine if the element has data or not.

With just one Connection, it returns "success" when the element's value has been assigned and "failed" and it is unassigned.

With more than one Connection, it uses the **mode** property to count the number of elements that have their values assigned. It returns "success" if the number of elements matches the **mode** property rule and "failed" when it does not.

Set up

For the first element, set the widget's id in the **elementId** property.

For additional elements, add their ids to the **moreConnections** property, which is an array. Use either its `push()` method or assign an array of ids. Then assign the **mode** property to determine how to evaluate multiple elements. **mode** has these values: "All", "OneOrMore", "AllOrNone", "One", and "Range". If using "Range", also set the **minimum** and **maximum** properties.

When working with textual or list widgets, a value of the empty string is normally considered unassigned. However, when a textbox has a watermark or a list has its first item as "No Selection", specify the text of the watermark or value of the first item in the Connection's **unassigned** property.

For example, a list has an option with the label "No selection" and value of "NONE".

```
var cond = jTAC.create("Required", {elementId: "ListBox1"});
cond.connection.unassigned = "NONE";
```

Example: Validation using code

```
$("TextBox1").rule("add", {
   required: { }
});
```

Also (since no parameters are common):

```
$("TextBox1").rule("add", {
   required: true
});
```

Using multiple textboxes:

```
$("TextBox1").rule("add", {
   required: {moreConnections: ['TextBox2', 'TextBox3'], mode: 'All'}
});
```

Example: Unobtrusive validation

```
<input type="text" id="TextBox1" name="TextBox1"
   data-val="true" data-val-advrequired="" data-val-advrequired-json="{}" />
```

Also (since no parameters are common):

```
<input type="text" id="TextBox1" name="TextBox1"
   data-val="true" data-val-advrequired="" />
```

Using multiple textboxes:

```
<input type="text" id="TextBox1" name="TextBox1" data-val="true" data-val-advrequired=""
   data-val-advrequired-json="{'moreConnections': ['TextBox2', 'TextBox3'], 'mode': 'All'}" />
```

Example: Condition using code

```
var cond = jTAC.create("Required");
cond.elementId = "TextBox1";

if (cond.isValid())
   // code runs when true
else
   // code runs when false
```

Using multiple textboxes:

```
var cond = jTAC.create("Required");
cond.elementId = "TextBox1";
cond.moreConnections: ['TextBox2', 'TextBox3'];
cond.mode = "All";
```

Example: Condition using JavaScript object

```
if (jTAC.isValid({jtacClass: 'Required', elementId: 'TextBox1',
    moreConnections: ['TextBox2', 'TextBox3'], mode: 'All'}))
   // code runs when true
else
   // code runs when false
```

Conditions.Required Properties

- **elementId** (string) – Set this with the id of the first element to evaluate, such as the id of an <input type="text">. Alternatively, set the **connection** property to a Connection object.
- **moreConnections** (array) - Use when there is more than one element to evaluate. The first element always goes in the **elementId** property. The rest in **moreConnections**.

  You can add any of these items to this array:

  - id of the element as a string.
  - A Connection object with its properties assigned, including **Id**.
  - A JavaScript object that identifies the connection class name in the **jtacClass** property, with the remaining properties setting values on the Connection it creates.

  You can add to this property in two ways:

  - Add one item with the `push()` method on the property.

  ```
  cond.moreConnections.push("TextBox2");
  cond.moreConnections.push(jTAC.create("Connection.FormElement", {Id: "TextBox3"}));
  cond.moreConnections.push({jtacClass: "Connection.FormElement", Id: "TextBox4"});
  ```

  - Replace the array with another array.

  ```
  cond.moreConnections =
     ["TextBox2",
      jTAC.create("Connection.FormElement", {Id: "TextBox3"}),
      {jtacClass: "Connection.FormElement", Id: "TextBox4"}];
  ```

- **mode** (string) - When there are multiple connections, this is used to determine how many must be not empty or `null` to report "success". Here are the mode values:

  - "`All`" - All must have text
  - "`OneOrMore`" - At least one must have text. This is the default.
  - "`AllOrNone`" - All or none must have text
  - "`One`" - Only one must have text

  - "Range" - Specify the minimum and maximum number of widgets in this Condition's **minimum** and **maximum** properties.

- **minimum** (integer) and **maximum** (integer) - When **mode** = "Range", use these to determine the number of elements with values that are not unassigned to report "success".
- **not** (boolean) - When `true`, apply a NOT operator to the result of evaluation, so "success" = 0 and "failed" = 1. ("Cannot evaluate" stays -1). It defaults to `false`.
- **trim** (boolean) - When `true` (the default), string values are trimmed before evaluating. It defaults to `true`.
- **enabled** (boolean) - When `false`, do not evaluate. It defaults to `true`.

Conditions.Required tokens in jquery-validate error messages

- "{LABEL}" – The text used to label the field. This comes from either the `<label for="elementid" >` tag associated with the element, or one of these attributes on the element itself: **data-msglabel** or **data-msglabel-lookupkey**. Click here for more.
- "{ERRORLABEL}" - Replaced by the **messageLabel** option passed to `$(element).validate(options)`. This is a style sheet class, which defaults to "message-label". Use this in messages that have {LABEL} tokens. Enclose them in "`<span class='{ERRORLABEL}'>{LABEL}</span>`".
- "{VALUE}" – Replaced by the element's actual string value.
- "{COUNT}" - Show the number of connections that were assigned.
- "{MINIMUM}" - Show the value of the **minimum** property.
- "{MAXIMUM}" - Show the value of the **maximum** property.

Conditions.RequiredIndex

jquery-validate rule name:	**requiredindex**
Alias names (case sensitive):	"**RequiredIndex**"
Inherits from:	**Conditions.BaseOneConnection**
Source file:	**\jTAC\Conditions\RequiredIndex.js**

Ensures that there is a selected item in a list type widget by evaluating the state of its **selectedIndex** property.

While often lists have a value="" representing no selection, many developers prefer to add a "---NO SELECTION--" item. As a result, sometimes "no selection" means **selectedIndex** = 0 and other times it means **selectedIndex** = -1. Therefore, the user can specify which they are using in the **unassignedIndex** property.

Note: The Conditions.Required can also be used to evaluate a list, but evaluates the textual value, not the selected index.

Evaluation rules

When the index specified in **unassignedIndex** matches the selected index, it means "failed".

Any other index means "success".

If you want to reverse this logic, set the **not** property to `true`.

Set up

Assign the list element's id to the **elementId** property.

The unassignedIndex property defaults to 0, so it is correctly setup if you have a "no selection" item first in the list. If you do not, assign the **unassignedIndex** property. Set it to -1 if it is unassigned by having no selection.

Example: Validation using code

```
$("ListBox1").rule("add", {
   requiredindex: { }
});
```

Also (since no parameters are common):

```
$("ListBox1").rule("add", {
   requiredindex: true
});
```

When there is no textual "No selection" item:

```
$("ListBox1").rule("add", {
   requiredindex: {unassignedIndex: -1 }
});
```

Example: Unobtrusive validation

```
<select id="ListBox1" name="ListBox1"
   data-val="true" data-val-requiredindex="" data-val-requiredindex-json="{}" />
```

Also (since no parameters are common):

```
<select id="ListBox1" name="ListBox1"
   data-val="true" data-val-requiredindex="" />
```

When there is no textual "No selection" item:

```
<select id="ListBox1" name="ListBox1"
   data-val="true" data-val-requiredindex=""
   data-val-requiredindex-json="{'unassignedIndex': -1}" />
```

Example: Condition using code

```
var cond = jTAC.create("RequiredIndex");
cond.elementId = "ListBox1";
```

When there is no textual “No selection” item:

```
var cond = jTAC.create("RequiredIndex");
cond.elementId = "ListBox1";
cond.unassignedIndex = -1;

if (cond.isValid())
   // code runs when true
else
   // code runs when false
```

Example: Condition using JavaScript object

```
if (jTAC.isValid(jtacClass: 'RequiredList', elementId: 'ListBox1'}))
   // code runs when true
else
   // code runs when false
```

When there is no textual “No selection” item:

```
if (jTAC.isValid({jtacClass: 'RequiredList', elementId: 'ListBox1',
   unassignedIndex: -1 })
   // code runs when true
else
   // code runs when false
```

Conditions.RequiredIndex Properties

- **elementId** (string) – Set this with the id of the element to evaluate, such as the id of a <select> tag. Alternatively, set the **connection** property to a Connection object.
- **unassignedIndex** (integer) - The index value that when selected, indicates that the list has no selection. It is typically 0 (when the first item is for "no selection") or -1 (when any selected index indicates a selection.)

  It defaults to 0.
- **not** (boolean) - When `true`, apply a NOT operator to the result of evaluation, so "success" = 0 and "failed" = 1. ("Cannot evaluate" stays -1). It defaults to `false`.
- **enabled** (boolean) - When `false`, do not evaluate. It defaults to `true`.

Conditions.RequiredIndex tokens in jquery-validate error messages

- “{LABEL}” – The text used to label the field. This comes from either the `<label for="elementid" >` tag associated with the element, or one of these attributes on the element itself: **data-msglabel** or **data-msglabel-lookupkey**. Click here for more.
- “{ERRORLABEL}” - Replaced by the **messageLabel** option passed to `$(element).validate(options)`. This is a style sheet class, which defaults to “message-label”. Use this in messages that have {LABEL} tokens. Enclose them in “`<span class='{ERRORLABEL}'>{LABEL}</span>`”.
- “{VALUE}” – Replaced by the element’s actual string value.

Conditions.SelectedIndex

jquery-validate rule name:	**selectedindex**
Alias names (case sensitive):	"**SelectedIndex**"
Inherits from:	**Conditions.BaseOneConnection**
Source file:	**\jTAC\Conditions\SelectedIndex.js**

Evaluates a list type element's selected index to one or more index values.

Supports both single selection and multiselection lists.

Evaluation rules

When the **index** property matches a selected index, it means "success".

Any other index means "failed".

If you want to reverse this logic, set the **not** property to `true`.

Set up

Assign the list element's id to the **elementId** property.

Assign one or more index values that will be compared to the selected index value in the index property. If you have just one index, assign an integer starting at 0 for the topmost item.

```
cond.index = 5;
```

If you want to match multiple values, assign an array of integers, each reflecting an index value starting at 0.

```
cond.index = [2, 3, 10];
```

When using an array, you can also add a child array of two items to reflect a range of indexes.

```
cond.index = [1, 2, [10, 20]];     // means 1, 2, and 10 through 20
```

By default, if a value in **index** matches the selected index, this Condition evaluates as "success". If you want to require that the index does not match a selected index, set the **not** property to `true`.

Example: Validation using code

```
$("ListBox1").rule("add", {
   selectedindex: {index: 2}
});
```

A range of indices that cannot be selected:

```
$("ListBox1").rule("add", {
   selectedindex: {index: [1, 2, 5], not: true}
});
```

Example: Unobtrusive validation

```
<select id="ListBox1" name="ListBox1"
   data-val="true" data-val-selectedindex="" data-val-selectedindex-json="{'index': 2 }" />
```

A range of indices that cannot be selected:

```
<select id="ListBox1" name="ListBox1"
   data-val="true" data-val-selectedindex=""
   data-val-selectedindex-json="{'index': [1, 2, 5], 'not': true }" />
```

Example: Condition using code

```
var cond = jTAC.create("SelectedIndex");
cond.elementId = "ListBox1";
cond.index = 2;
```

A range of indices that cannot be selected:

```
var cond = jTAC.create("SelectedIndex");
cond.elementId = "ListBox1";
cond.index = [1, 2, 5];
cond.not = true;

if (cond.isValid())
   // code runs when true
else
   // code runs when false
```

Example: Condition using JavaScript object

```
if (jTAC.isValid({jtacClass: 'SelectedIndex', elementId: 'ListBox1', index: 2 }))
   // code runs when true
else
   // code runs when false
```

A range of indices that cannot be selected:

```
if (jTAC.isValid({jtacClass: 'SelectedIndex', elementId: 'ListBox1', index: [1, 2, 5],
   not: true}))
   // code runs when true
else
   // code runs when false
```

Conditions.SelectedIndex Properties

- **elementId** (string) – Set this with the id of the element to evaluate, such as the id of a <select> tag. Alternatively, set the **connection** property to a Connection object.
- **index** (integer or array) - The index value to match against the selected indices of the list. The value can either be an integer or an array. When it is an array, each item either an index number or an array representing a range. A range's array has two elements, start index and end index.

  It defaults to `null`.
- **not** (boolean) – When `false`, **index** must match a selected index. When `true`, **index** must not match a selected index.

  It defaults to `false`.
- **enabled** (boolean) - When `false`, do not evaluate. It defaults to `true`.

Conditions.SelectedIndex tokens in jquery-validate error messages

- "{LABEL}" – The text used to label the field. This comes from either the `<label for="elementid">` tag associated with the element, or one of these attributes on the element itself: **data-msglabel** or **data-msglabel-lookupkey**. Click here for more.
- "{ERRORLABEL}" - Replaced by the **messageLabel** option passed to `$(element).validate(options)`. This is a style sheet class, which defaults to "message-label". Use this in messages that have {LABEL} tokens. Enclose them in "`<span class='{ERRORLABEL}'>{LABEL}</span>`".
- "{VALUE}" – Replaced by the element's actual string value.

Conditions.WordCount

jquery-validate rule name:	**wordcount**
Alias names (case sensitive):	"**WordCount**"
Inherits from:	Conditions.BaseCounter
Source file:	**\jTAC\Conditions\WordCounts.js**

Use to ensure the number of words in a string are within a range. It can count across several elements.

Words are identified by space characters or any transition between word and non-word characters.

Evaluation rules

Returns "success" when the number of characters is within the range.

Returns "failed" when the number of characters is outside the range.

Set up

For the first element, set the element's id in the **elementId** property.

For additional elements, add their ids to the **moreConnections** property, which is an array. Use either its `push()` method or assign an array of ids.

Then assign the minimum and maximum of the range in the **minimum** and **maximum** properties. If one of those properties is not used, leave it unassigned.

Example: Validation using code

```
$("TextBox1").rule("add", {
   wordcount: {maximum: 10}
});
```

Example: Unobtrusive validation

```
<input type="text" id="TextBox1" name="TextBox1"
   data-val="true" data-val-wordcount="" data-val-wordcount-json="{'maximum': 10}" />
```

Example: Condition using code

```
var cond = jTAC.create("WordCount");
cond.elementId = "TextBox1";
cond.maximum = 10;

if (cond.isValid())
   // code runs when true
else
   // code runs when false
```

Example: Condition using JavaScript object

```
if (jTAC.isValid({jtacClass: 'WordCount', elementId: 'TextBox1', maximum: 10}))
   // code runs when true
else
   // code runs when false
```

Conditions.WordCount Properties

- **minimum** (integer) – Determines the minimum of the count to report "success". If `null`, it is not used. Otherwise it must be a positive integer.
- **maximum** (integer) - Determines the maximum of the count to report "success". If `null`, it is not used. Otherwise it must be a positive integer.
- **elementId** (string) – Set this with the id of the first element to evaluate, such as the id of an <input type="text">. Alternatively, set the **connection** property to a Connection object.
- **moreConnections** (array) - Use when there is more than one element to evaluate. The first element always goes in the **elementId** property. The rest in **moreConnections**.

  You can add any of these items to this array:

  - id of the element as a string.
  - A Connection object with its properties assigned, including **Id**.
  - A JavaScript object that identifies the connection class name in the **jtacClass** property, with the remaining properties setting values on the Connection it creates.

  You can add to this property in two ways:

  - Add one item with the `push()` method on the property.

```
cond.moreConnections.push("TextBox2");
cond.moreConnections.push(jTAC.create("Connection.FormElement", {Id: "TextBox3"}));
cond.moreConnections.push({jtacClass: "Connection.FormElement", Id: "TextBox4"});
```

  - Replace the array with another array.

```
cond.moreConnections =
    ["TextBox2",
     jTAC.create("Connection.FormElement", {Id: "TextBox3"}),
     {jtacClass: "Connection.FormElement", Id: "TextBox4"}];
```

- **ignoreNotEditable** (boolean) - When there are two or more Connections used, this determines if elements that are not editable are counted.

  When `true`, not editable Connections are not counted.

  When `false`, they are.

  It defaults to `false`.

- **not** (boolean) - When `true`, apply a NOT operator to the result of evaluation, so "success" = 0 and "failed" = 1. ("Cannot evaluate" stays -1). It defaults to `false`.
- **trim** (boolean) - When `true` (the default), string values are trimmed before evaluating. It defaults to `true`.
- **enabled** (boolean) - When `false`, do not evaluate. It defaults to `true`.

Conditions.WordCount tokens in jquery-validate error messages

- "{LABEL}" – The text used to label the field. This comes from either the `<label for="elementid" >` tag associated with the element, or one of these attributes on the element itself: **data-msglabel** or **data-msglabel-lookupkey**. Click here for more.
- "{ERRORLABEL}" - Replaced by the **messageLabel** option passed to `$(element).validate(options)`. This is a style sheet class, which defaults to "message-label". Use this in messages that have {LABEL} tokens. Enclose them in "`<span class='{ERRORLABEL}'>{LABEL}</span>`".
- "{VALUE}" – Replaced by the element's actual string value.
- "{COUNT}" - Show the number of words counted.
- "{COUNT:singular:plural}" - Show one of the two strings in singular or plural positions, depending on the count. If 1, show the singular form. For example: "You entered {COUNT} {COUNT:word:words}."
- "{MINIMUM}" - Show the value of the **minimum** property.
- "{MAXIMUM}" - Show the value of the **maximum** property.
- "{DIFF}" - The number the count exceeds the maximum or is below the minimum.
- "{DIFF:singular:plural}" - Show one of the two strings in singular or plural positions, depending on the value of {DIFF}. If 1, show the singular form.

Conditions.Base

Inherits from:	**jTAC's universal base class: jTACClassBase**
Source file:	**\jTAC\Conditions\Base.js**

The abstract base class from which all Conditions inherit. Use it if you are creating your own Condition. You will override the `_evaluateRule()` method to supply your evaluation logic.

You may be better off inheriting from one of its subclasses, due to the extended functionality they offer:

Condition class name	Purpose
Conditions.BaseOneConnection	Abstract base class for building conditions that evaluate one widget.
Conditions.BaseTwoConnections	Abstract base class for building conditions that evaluate two widgets.
Conditions.BaseOneOrMoreConnections	Abstract base class for building conditions that evaluate a list of widgets.
Conditions.BaseOperator	Abstract base class for building conditions with one or two widgets and uses a boolean operator (=, <>, etc) assigned by the user.

Conditions.Base Properties

- **enabled** (boolean) - When `false`, evaluate should not be called. The `canEvaluate()` method uses it. It defaults to `true`.
- **datatype** (string) - Used when the Condition needs to convert a string to another type, like number or date. It looks up the TypeManager object to use for that conversion and assigns that object to the **typeManager** property. Alternatively, specify the TypeManager object in the **typeManager** property.

  If you don't assign either **datatype** or **typeManager**, the **typeManager** property requests the TypeManager object from the first connection object. That object knows how to find the **data-jtac-datatype** and **data-jtac-typemanager** attributes on the input field specified by the connection, to define the TypeManager.
- **typeManager** (subclass of TypeManagers.Base) – The TypeManager object used when the Condition needs to convert a string to another type. If you assign the **datatype** property, this will be setup for you.
- **not** (boolean) - When `true`, apply a NOT operator to the result of evaluation, so "success" = 0 and "failed" = 1. ("Cannot evaluate" stays -1). It defaults to `false`.
- **trim** (boolean) - When `true` (the default), string values are trimmed before evaluating. It defaults to `true`.
- **lastEvaluateResult** (integer) - Indicates the value from the last time the `evaluate()` method was run: 1 = "success", 0 = "failed", -1 = "cannot evaluate", `null` = has not been run.
- **autoDisable** (boolean) - Determines if a Connection that is attached to a non-editable widget causes this Condition to disable itself by returning `false` in its `canEvaluate()` method.

  When `true`, connections are checked. If any return `false` in their `isEditable()` method, `canEvalaute()` will return `false`.

  When `false`, connections are not checked.

  It defaults to `true`.
- **name** (string) - Gets a string that gives the class a unique name.

Subclassing Conditions.Base

Getting started:

- All Conditions inherit from `jTACClassBase` in **\jTAC\jTAC.js**. It has some useful utilities for your class development. Learn more in the **\jTAC\jTAC.js** file.
- See "Adding your own classes to jTAC" to use best practices in creating your subclasses' structure.
- For an example, see the Conditions.BaseOneConnection in **\jTAC\Conditions\BaseOneConnection.js** and Conditions.BooleanLogic in **\jTAC\Conditions\BooleanLogic.js**.

This topic will identify methods that you are likely to override or call in Conditions.Base. It will not describe parameters. Instead, please use the class definition in **\jTAC\Conditions\Base.js**.

	Element Name	When
✓	`canEvaluate()`	Call to determine if the Condition can be evaluated. If it returns `true`, you can call `evaluate()`. Override when incompletely setup. Always call the ancestor's `canEvaluate()` method and return `false` if that returns `false`. `if (!this.callParent()) return false;`
✓	`evaluate()`	User calls this to evaluate. It calls your `_evaluateRule()` method.
✓	`_evaluateRule()`	Subclasses must implement to evaluate according to the rule the Condition implements. Return an integer with one of these values: 1 = success, 0 = failed, -1 = cannot evaluate.
✓	`_createConnections()`	Subclass to add Connection objects that will be used. Conditions.BaseOneConnection, Conditions.BaseTwoConnections, and Conditions.BaseOneOrMoreConnections all implement this, so you rarely need to do it yourself.
✓	`_checkTypeManager()`	Your subclass that supports a TypeManager can call this within its `getTypeManager()` method if `this._typeManager` is `null`. It looks up the correct TypeManager from the Connection and if not found there, establishes the TypeManagers.Integer.

Key: ✓ always override; ✓ sometimes override; ✓ rarely override

Conditions.BaseOneConnection

Inherits from:	**Conditions.Base**
Source file:	**\jTAC\Conditions\BaseOneConnection.js**

An abstract base class for Conditions with one Connection object. It introduces the **connection** and **elementId** properties. Your subclass must still override the `_evaluateRule()` method to supply your evaluation logic.

Conditions.BaseOneConnection Properties

- **elementId** (string) – Set this with the id of the first element to evaluate, such as the id of an <input type="text">. When set, it resolves the **connection** property automatically.
- **connection** (Connection object) - The Connection object used to get a value from the first element, identified by the **elementId** property. Your evaluation function generally uses its `getTextValue()`, `getTypedValue()`, and `isNullValue()` methods.

  You rarely SET this value as Connection objects are determined based on the id passed automatically. If you do set it, this property accepts both a Connection object and a JavaScript object that identifies the connection class name in the **jtacClass** property, with the remaining properties setting values on the Connection it creates.

See also "Conditions.Base Properties".

Subclassing Conditions.BaseOneConnection

Getting started:

- All Conditions inherit from `jTACClassBase` in **\jTAC\jTAC.js**. It has some useful utilities for your class development. Learn more in the **\jTAC\jTAC.js** file.
- See "Adding your own classes to jTAC" to use best practices in creating your subclasses' structure.
- For an example, see the Conditions.BaseTwoConnections in **\jTAC\Conditions\BaseTwoConnections.js** and Conditions.Range in **\jTAC\Conditions\Range.js**.

This topic will identify methods that you are likely to override or call in Conditions.BaseOneConnection. It will not describe parameters. Instead, please use the class definition in **\jTAC\Conditions\Base.js**.

	Element Name	When
✓	`canEvaluate()`	Call to determine if the Condition can be evaluated. If it returns `true`, you can call `evaluate()`. Override when incompletely setup. Always call the ancestor's `canEvaluate()` method and return `false` if that returns `false`. `if (!this.callParent()) return false;`
✓	`evaluate()`	User calls this to evaluate. It calls your `_evaluateRule()` method.
✓	`_evaluateRule()`	Subclasses must implement to evaluate according to the rule the Condition implements. Return an integer with one of these values: 1 = success, 0 = failed, -1 = cannot evaluate. Your code will usually use the object defined in the **connection** property. It is a Connection object. These methods are most likely to be used: `getTextValue()`, `getTypedValue()`, and `isNullValue()`. If you have a TypeManager, call its get `toValueFromConnection()` method, passing the **connection** property. If you need to ensure that the connection has a valid native value, pass its string value to the TypeManager's `isValid()` method. If you need to convert from a string to the TypeManager's native type, call the TypeManager's `toValue()` method. If you need to compare two values whose native type you don't know, call the TypeManager's `compare()` method.
✓	`_checkTypeManager()`	Your subclass that supports a TypeManager can call this within its `getTypeManager()` method if `this.config.typeManager` is `null`. It looks up the correct TypeManager from the Connection and if not found there, establishes the TypeManagers.Integer.

Key: ✓ always override; ✓ sometimes override; ✓ rarely override

Conditions.BaseTwoConnections

Inherits from:	**Conditions.BaseOneConnection**
Source file:	**\jTAC\Conditions\BaseTwoConnections.js**

An abstract base class for Conditions with two Connection objects. It introduces the **connection2** and **elementId2** properties. Your subclass must still override the `_evaluateRule()` method to supply your evaluation logic.

Conditions.BaseTwoConnections Properties

- **elementId2** (string) – Set this with the id of the second element to evaluate, such as the id of an <input type="text">. When set, it resolves the **connection** property automatically.
- **Connection2** (Connection object) - The Connection object used to get a value from the second element, identified by the **elementId2** property. Your evaluation function generally uses its `getTextValue()`, `getTypedValue()`, and `isNullValue()` methods.

  You rarely SET this value as Connection objects are determined based on the id passed automatically. If you do set it, this property accepts both a Connection object and a JavaScript object that identifies the connection class name in the **jtacClass** property, with the remaining properties setting values on the Connection it creates.

See also "Conditions.BaseOneConnection Properties" and "Conditions.Base Properties".

Subclassing Conditions.BaseTwoConnections

Getting started:

- All Conditions inherit from `jTACClassBase` in **\jTAC\jTAC.js**. It has some useful utilities for your class development. Learn more in the **\jTAC\jTAC.js** file.
- See "Adding your own classes to jTAC" to use best practices in creating your subclasses' structure.
- For an example, see Conditions.CompareTwoElements in **\jTAC\Conditions\CompareTwoElements.js**.

This topic will identify methods that you are likely to override or call in Conditions.BaseTwoConnections. It will not describe parameters. Instead, please use the class definition in **\jTAC\Conditions\Base.js**.

	Element Name	When
✓	`canEvaluate()`	Call to determine if the Condition can be evaluated. If it returns `true`, you can call `evaluate()`. Override when incompletely setup. Always call the ancestor's `canEvaluate()` method and return `false` if that returns `false`. `if (!this.callParent()) return false;`
✓	`evaluate()`	User calls this to evaluate. It calls your `_evaluateRule()` method.
✓	`_evaluateRule()`	Subclasses must implement to evaluate according to the rule the Condition implements. Return an integer with one of these values: 1 = success, 0 = failed, -1 = cannot evaluate. Your code will usually use the object defined in the **connection** and **connection2** properties. They are Connection objects. These methods are most likely to be used: `getTextValue()`, `getTypedValue()`, and `isNullValue()`. If you have a TypeManager, call its get `toValueFromConnection()` method, passing the **connection** or **connection2** property. If you need to ensure that the connection has a valid native value, pass its string value to the TypeManager's `isValid()` method. If you need to convert from a string to the TypeManager's native type, call the TypeManager's `toValue()` method. If you need to compare two values whose native type you don't know, call the TypeManager's `compare()` method.
✓	`_checkTypeManager()`	Your subclass that supports a TypeManager can call this within its `getTypeManager()` method if `this.config.typeManager` is `null`. It looks up the correct TypeManager from the Connection and if not found there, establishes the TypeManagers.Integer.

Key: ✓ always override; ✓ sometimes override; ✓ rarely override

Conditions.BaseOneOrMoreConnections

Inherits from:	**Conditions.BaseOneConnection**
Source file:	**\jTAC\Conditions\BaseOneOrMoreConnections.js**

An abstract base class that supports one or more Connections to form elements or widgets on the page. If you only have one Connection, it can use the **elementId** or **connection** property. For any more, add an array of Connection objects to the **moreConnections** property.

Your subclass must still override the `_evaluateRule()` method to supply your evaluation logic.

Conditions.BaseOneOrMoreConnections Properties

- **moreConnections** (array) - Use when there are more than one Connection. The first connection always goes in the **connection** property. The rest in **moreConnections**.

  You can add any of these items to this array:

  - id of the element as a string.
  - A Connection object with its properties assigned, including **Id**.
  - A JavaScript object that identifies the connection class name in the **jtacClass** property, with the remaining properties setting values on the Connection it creates.

  You can add to this property in two ways:

  - Add one item with the `push()` method on the property.

```
cond.moreConnections.push("TextBox2");
cond.moreConnections.push(jTAC.create("Connection.FormElement", {Id: "TextBox3"}));
cond.moreConnections.push({jtacClass: "Connection.FormElement", Id: "TextBox4"});
```

  - Replace the array with another array.

```
cond.moreConnections =
   ["TextBox2",
    jTAC.create("Connection.FormElement", {Id: "TextBox3"}),
    {jtacClass: "Connection.FormElement", Id: "TextBox4"}];
```

- **ignoreNotEditable** (boolean) - When there are two or more Connections used, this determines if elements that are not editable are counted. For example, when **Conditions.Required.mode** = `All` and there is one non-editable connection, "`All`" means one less than the total connections.

  When `true`, not editable Connections are not counted.

  When `false`, they are.

  It defaults to `false`.

See also "Conditions.BaseOneConnection Properties" and "Conditions.Base Properties".

Subclassing Conditions.BaseOneOrMoreConnections

Getting started:

- All Conditions inherit from jTACClassBase in **\jTAC\jTAC.js**. It has some useful utilities for your class development. Learn more in the **\jTAC\jTAC.js** file.
- See "Adding your own classes to jTAC" to use best practices in creating your subclasses' structure.
- For an example, see Conditions.DuplicateEntry in **\jTAC\Conditions\DuplicateEntry.js**.

This topic will identify methods that you are likely to override or call in Conditions.BaseOneOrMoreConnections. It will not describe parameters. Instead, please use the class definition in **\jTAC\Conditions\Base.js**.

	Element Name	When
✓	`canEvaluate()`	Call to determine if the Condition can be evaluated. If it returns `true`, you can call `evaluate()`. Override when incompletely setup. Always call the ancestor's `canEvaluate()` method and return `false` if that returns `false`. `if (!this.callParent()) return false;`
✓	`evaluate()`	User calls this to evaluate. It calls your `_evaluateRule()` method.
✓	`_evaluateRule()`	Subclasses must implement to evaluate according to the rule the Condition implements. Return an integer with one of these values: 1 = success, 0 = failed, -1 = cannot evaluate. Your code will usually use Connection objects defined by the **connection** and **moreConnection** properties. Call `_cleanupConnections()` to return an array of Connections. These Connection class methods are most likely to be used: `getTextValue()`, `getTypedValue()`, and `isNullValue()`. If you have a TypeManager, call its get `toValueFromConnection()` method, passing the **connection** or **connection2** property. If you need to ensure that the connection has a valid native value, pass its string value to the TypeManager's `isValid()` method. If you need to convert from a string to the TypeManager's native type, call the TypeManager's `toValue()` method. If you need to compare two values whose native type you don't know, call the TypeManager's `compare()` method.
✓	`_checkTypeManager()`	Your subclass that supports a TypeManager can call this within its `getTypeManager()` method if `this.config.typeManager` is `null`. It looks up the correct TypeManager from the Connection and if not found there, establishes the TypeManagers.Integer.

Key: ✓ always override; ✓ sometimes override; ✓ rarely override

Conditions.BaseOperator

Inherits from:	**Conditions.BaseTwoConnections**
Source file:	**\jTAC\Conditions\BaseOperator.js**

An abstract base class that compares two values based on the rule in the **operator** property.

Your subclass must still override the `_evaluateRule()` method to supply your evaluation logic.

Conditions.BaseOperator Properties

- **operator** (string) - Supports these values: "`=`", "`<>`", "`<`", "`>`", "<=", "`>=`"

See also “”Conditions.BaseTwoConnections Properties, “Conditions.BaseOneConnection Properties” and “Conditions.Base Properties”.

Subclassing Conditions.BaseOperator

Getting started:

- All Conditions inherit from `jTACClassBase` in **\jTAC\jTAC.js**. It has some useful utilities for your class development. Learn more in the **\jTAC\jTAC.js** file.
- See "Adding your own classes to jTAC" to use best practices in creating your subclasses' structure.
- For an example, see Conditions.CompareToValue in **\jTAC\Conditions\CompareToValue.js**.

This topic will identify methods that you are likely to override or call in Conditions.BaseOperator. It will not describe parameters. Instead, please use the class definition in **\jTAC\Conditions\Base.js**.

	Element Name	When
✓	`canEvaluate()`	Call to determine if the Condition can be evaluated. If it returns `true`, you can call `evaluate()`. Override when incompletely setup. Always call the ancestor's `canEvaluate()` method and return `false` if that returns `false`. `if (!this.callParent()) return false;`
✓	`evaluate()`	User calls this to evaluate. It calls your `_evaluateRule()` method.
✓	`_evaluateRule()`	Subclasses must implement to evaluate according to the rule the Condition implements. Return an integer with one of these values: 1 = success, 0 = failed, -1 = cannot evaluate. Your code will usually use Connection objects defined by the **connection** and **moreConnection** properties. Call `_cleanupConnections()` to return an array of Connections. These Connection class methods are most likely to be used: `getTextValue()`, `getTypedValue()`, and `isNullValue()`. If you have a TypeManager, call its get `toValueFromConnection()` method, passing the **connection** or **connection2** property. After getting two native values, apply the **operator** by calling the `_compare()` method. If you need to ensure that the connection has a valid native value, pass its string value to the TypeManager's `isValid()` method. If you need to convert from a string to the TypeManager's native type, call the TypeManager's `toValue()` method.
✓	`_checkTypeManager()`	Your subclass that supports a TypeManager can call this within its `getTypeManager()` method if `this.config.typeManager` is `null`. It looks up the correct TypeManager from the Connection and if not found there, establishes the TypeManagers.Integer.
✓	`_compare()`	Utility to compare two values. Both values can be native or string types. The TypeManager's `compare()` method is used for the evaluation and operator applied against the result. Returns `true` if the values match the **operator's** setting and `false` when they do not.

Key: ✓ always override; ✓ sometimes override; ✓ rarely override

Conditions.BaseCounter

Inherits from:	**Conditions.BaseOneOrMoreConnections**
Source file:	**\jTAC\Conditions\BaseCounter.js**

An abstract base class for one or more Connections are evaluated to total something about the value of each, such as number of characters or words. That total is compared to a range defined by **minimum** and **maximum**.

Your subclass must still override the `_connCount()` method. It should evaluate the connection passed in to return a number that is added to the total by the caller.

Conditions.BaseCounter properties

- **minimum** (integer) – Determines the minimum of the count to report "success". If `null`, it is not used. Otherwise it must be a positive integer.
- **maximum** (integer) - Determines the maximum of the count to report "success". If `null`, it is not used. Otherwise it must be a positive integer.

See also "Conditions.BaseOneOrMoreConnections Properties", "Conditions.BaseOneConnection Properties" and "Conditions.Base Properties".

Subclassing Conditions.BaseCounter

Getting started:

- All Conditions inherit from `jTACClassBase` in **\jTAC\jTAC.js**. It has some useful utilities for your class development. Learn more in the **\jTAC\jTAC.js** file.
- See "Adding your own classes to jTAC" to use best practices in creating your subclasses' structure.
- For an example, see Conditions.CharacterCount in **\jTAC\Conditions\CharacterCount.js**.

This topic will identify methods that you are likely to override or call in Conditions.BaseCounter. It will not describe parameters. Instead, please use the class definition in **\jTAC\Conditions\Base.js**.

	Element Name	When
✓	`canEvaluate()`	Call to determine if the Condition can be evaluated. If it returns `true`, you can call `evaluate()`. Override when incompletely setup. Always call the ancestor's `canEvaluate()` method and return `false` if that returns `false`. `if (!this.callParent()) return false;`
✓	`evaluate()`	User calls this to evaluate. It calls your `_evaluateRule()` method.
✓	`_evaluateRule()`	Already coded for you. This calls your `_connCount()` function
✓	`_connCount()`	Returns an integer representing the count determined for the connection passed in. Must return at least a value of 0.

Key: ✓ always override; ✓ sometimes override; ✓ rarely override

Conditions.BaseRegExp

Inherits from:	**Conditions.BaseOneConnection**
Source file:	**\jTAC\Conditions\BaseRegExp.js**

An abstract base class for using regular expressions to evaluate the text of a widget. This base class requires the child to override `_getExpression()` to supply the expression. Optionally override `_getCaseIns()` and `_getGlobal()` to supply those regex flags.

This base class provides a framework for creating many expression generator Conditions. This lets the user avoid trying to create the appropriate expression for some common cases because those child Conditions will let the user set rules (as properties) and the Condition's `_getExpression()` method will generate the correct expression.

These child conditions make it easier to identify the business rule being described. You don't have to analyze the regular expression, just look at the name of the condition class and its properties.

Evaluation rules

When the text is matched by the regular expression, it evaluates as "success".

Otherwise it evaluates as "failed".

When the text is the empty string, evaluates as "cannot evaluate" so long as **ignoreBlankText** is `true`.

If you want to reverse this logic, set the **not** property to `true`.

Conditions.BaseRegExp Properties

- **ignoreBlankText** (boolean) - Determines how blank text is evaluated.

  When `true`, the Condition evaluates as “cannot evaluate”.

  When `false`, the condition evaluates as “failed”.

  It defaults to `true`.

See also “Conditions.BaseOneConnection Properties” and “Conditions.Base Properties”.

Subclassing Conditions.BaseRegExp

Getting started:

- All Conditions inherit from `jTACClassBase` in **\jTAC\jTAC.js**. It has some useful utilities for your class development. Learn more in the **\jTAC\jTAC.js** file.
- See "Adding your own classes to jTAC" to use best practices in creating your subclasses' structure.
- For an example, see Conditions.RegExp in **\jTAC\Conditions\RegExp.js**.

This topic will identify methods that you are likely to override or call in Conditions.BaseRegExp. It will not describe parameters. Instead, please use the class definition in **\jTAC\Conditions\RegExp.js**.

	Element Name	When
✓	`canEvaluate()`	Call to determine if the Condition can be evaluated. If it returns `true`, you can call `evaluate()`. This class checks that an expression is defined by calling `getExpression()`. If it is not defined, it returns `false`. Override when incompletely setup. Always call the ancestor's `canEvaluate()` method and return `false` if that returns `false`. `if (!this.callParent()) return false;`
✓	`evaluate()`	User calls this to evaluate. It calls your `_evaluateRule()` method.
✓	`_evaluateRule()`	Already coded for you. You must override `_getExpression()`, `_getCaseIns()`, `_getGlobal()`, and `_getMultiline()` to tell this method how to operate.
✓	`_getExpression()`	Return the regular expression. If it returns "", the Condition will return `false` from `canEvaluate()` and -1 from `evaluate()`.
✓	`_getCaseIns()`	Determines if the case insensitive option is used in evaluating the regular expression. It defaults to `false`. If you demand this option, return `true`.
✓	`_getGlobal()`	Determines if the Global option is used in evaluating the regular expression. It defaults to `false`. If you demand this option, return `true`.
✓	`_getMultiline()`	Determines if the Multiline option is used in evaluating the regular expression. It defaults to `false`. If you demand this option, return `true`.

Key: ✓ always override; ✓ sometimes override; ✓ rarely override

Using Conditions in your own JavaScript

Conditions are used automatically in several areas of jTAC, such as child elements of the Conditions.BooleanLogic and validation's **depends** property. You can also use them within your own JavaScript. This topic addresses how to use them in your own JavaScript.

All Conditions have these methods:

- `canEvaluate()` - Returns `true` if the Condition is enabled and ready to evaluate data. Do not call `evaluate()` or `isValid()` if `false`.
- `evaluate()` - Evaluates and returns one of these results: 1 = success, 0 = failed, -1 = cannot evaluate.
- `isValid()` – A variation of the `evaluate()` function that returns a boolean. When `true`, the condition evaluates as success or cannot evaluate. When `false`, it evaluates as failed.

There are two ways to create and evaluate a Condition.

Creating the actual object

Create any condition by calling the `jTAC.create()` function with the full class name or an alias. Optionally pass an object with properties on the Condition class that you will assign. Otherwise, assign properties to the instance created.

```
var cond = jTAC.create("Conditions.Required");
cond.elementId = "TextBox1";

var cond = jTAC.create("Conditions.Required", {elementId: "TextBox1"});
```

Call either the `isValid()` or `evaluate()` method. `isValid()` returns `true` if the condition successfully evaluates, or it cannot evaluate. It returns `false` when the condition fails. `evaluate()` returns 1 for "success", 0 for "failed" and -1 for "cannot evaluate".

```
if (cond.canEvaluate() && cond.isValid())
   // do something


if (cond.canEvaluate() && (cond.evaluate() == 1))
   // do something
```

Creating a JavaScript object

You might find it easier to do this all in one step by calling either the `jTAC.isValid()` or `jTAC.evaluate()` function. These functions are passed a JavaScript object which defines the full class name or alias name for the Condition in its **jtacClass** property. Any additional properties are assigned to the same-named properties on the Condition it creates.

```
if (jTAC.isValid({jtacClass: 'Conditions.Required', elementId: 'TextBox1'}))
   // do something


if (jTAC.evaluate({jtacClass: 'Conditions.Required', elementId: 'TextBox1'}) == 1)
   // do something
```

The Connection classes

A **Connection** provides a layer between Conditions and elements on the page. Different Connection classes handle different types of elements. This way, a Condition can work for anything you may introduce to the page, simply by creating a new Connection class. Connections know about data values, style sheet classes, editability, visibility, and attribute data of the element.

Connections can get data from other resources. Suppose you have a calculated value to use in a Condition, such as combining the text of First name and Last name textboxes. You create a Connection, overriding a few methods. Then plug it into the Condition's **connection** property. Alternatively, use the Connections.Value class with the Condition, plugging the result of your calculation into its **value** property.

The primary use is to get and set the value of something either by a string representing its value, or by the native value itself. Its primary methods are `getTextValue()`, `getTypedValue()`, `setTextValue()`, and `setTypedValue()`.

Elements on the page are often data type specific. For example, a date picker handles dates and the HTML 5 <input type="integer"> tag handles integers. Connection objects can use this information to determine the TypeManager to use with the element. For HTML elements that are not type specific, you can add the data-jtac-datatype attribute, specifying the class name or alias of a TypeManager class. From there, a Condition can identify the appropriate TypeManager without any additional setup. (Conditions have their own **datatype** and **typeManager** properties, which you would assign if the Connection doesn't have this information.)

Click on any of these topics to jump to them:

Locate a Connection

Here are the Connection classes supplied by jTAC. Click on their name to learn more about them. Each of these classes is defined in files of the **jTAC****Connections**\ folder.

Class name	Purpose
Connections.FormElement	Native HTML form elements including input, select, and textarea. Supports HTML 5 input types.
Connections.UserFunction	You write a function that supplies the Connection's value to the caller. Typically used with calculated fields.
Connections.Value	Instead of accessing a widget, it holds a constant value that any consumer can use.
Connections.InnerHtml	Works with HTML tags that hold HTML in their body, such as div and span. This is mostly used with calculations.
Connections.JQUIDatePicker	Supports the *jquery-ui* datepicker.

When developing your own Connections, here are the classes from which you can inherit.

Class	Purpose
Connections.Base	Abstract base class for all Connections. Does not know about elements on the page.
Connections.BaseElement	Abstract base class for all Connections based on elements on the page.
Connections.BaseJQUIElement	Abstract base class for all Connections that interact with *jquery-ui* widgets.

Connections.FormElement

Alias names (case sensitive):	"**FormElement**"
Inherits from:	**Connections.BaseElement**
Source file:	**\jTAC\Connections\FormElement.js**

Gets and sets the value attribute of an HTML form element, like `<input>`, `<select>`, and `<textarea>`. `getTextValue()` and `setTextValue()` methods interact with the element's **value** attribute.

`typeSupported()` supports other types depending on the element, such as checkbox supports "boolean" and <select> supports "index" and "indices".

Set up

Assign the id value of an HTML element to the **id** property.

Suppose you want to write a currency value to `<input type="text" id="TextBox1" />`. Here's how you would write that code:

```
var val = 100.0;
var tm = jTAC.create("TypeManagers.Currency");
var conn = jTAC.create("Connections.FormElement", {id: "TextBox1"});
conn.setTextValue(tm.toString(val));
```

Supporting data types

The list below identifies which HTML tags supply a TypeManager object, and which name to pass into `typeSupported()`, `getTypedValue()`, and `setTypedValue()` methods.

- "integer" – Supported by `<input type="range" />` and `<input type="number" />`.
- "float" – Supported by `<input type="number" />`.
- "date" – Supported by `<input type="date" />`, `<input type="datetime" />`, `<input type="datetime-local" />`, and `<input type="month" />`. Works with a JavaScript Date object representing only a date (the time is undefined).
- "DateTime" – Supported by `<input type="datetime" />` and `<input type="datetime-local" />`. Works with a JavaScript Date object.
- "time" – Supported by `<input type="time" />`. Works with a JavaScript Date object representing only a time (the date is undefined).
- "boolean" – Supported by `<input type="checkbox" />` and `<input type="radio" />`. Works with a boolean value.
- "index" – Supported by `<select>` to get and set the selected index. Supports a number value representing an offset in a list. -1 = no selection. 0 = first element..
- "indices" - Supported by `<select>` to get and set the selected indices in a multiple selection setup. Supports an array of integers representing multiple offsets in a list.

For any other case, or when you want to override the default TypeManager, assign the TypeManager class name or alias to the **data-jtac-datatype** attribute on the HTML tag.

Be sure to include the script files for the TypeManager needed.

Checkboxes and radio buttons

This class also can treat a group of radio buttons or checkboxes as a single widget. Radio buttons only need their **name** attribute to match. Checkboxes need you to identify which elements are in the group through its **buttons** property.

To use either of these groups, call `getTypedValue("index")` to get a selected index, or pass an index number into `setTypedValue()` to set an index.

Finally, a checkbox list also supports multiple selections by calling `getTypedValue("indices")` or passing an array of integers to `setTypedValue()`.

If working with individual checkbox or radio inputs, the `getTextValue()` and `setTextValue()` methods respect the **checked** attribute of these inputs, instead of the **value** attribute. An empty string means unchecked. Any other string means checked.

Connections.FormElement Properties

- **id** (string) - The unique id used to identify the element on the page. You can either set this or call `setElement()` to pass the actual object of the element.
- **allowNone** (boolean) - When `true`, if the id is not found when calling get or set methods, nothing happens. When `false`, an exception occurs. It defaults to `false`.
- **trim** (boolean) - Determines if strings are returned from `getTextValue()` after trimming lead and trailing spaces. `isNullValue()` also applies this property to evaluate if `null`. It defaults to `true`.
- **unassigned** (string) - Allows for watermarks in a textbox and a non-empty string value for a list to mean unselected. A watermark is text that appears when the textbox is empty.

  Lists usually use the value "" to mean an unselected state. But if you want to have a "no selection" item with a different value, assign this to the string for that value.

  When assigned, this string is compared to the textual value of the element. If it matches, `getTextValue()` returns the empty string and `isNullValue()` returns `true`.

  It defaults to `null`.
- **unassignedCase** (boolean) - When using the **unassigned** property, this determines if the 'unassigned' string is compared case sensitive or not. When `true`, it compares case sensitive. When `false`, it does not. It defaults to `false`.
- **fixLength** (boolean) - When using a `<textarea>` tag, the text length may differ between browsers for each ENTER character in the text. Many browsers report one character (%0A) while others report two (%0D%0A). In addition, all browsers post back two (%0D%0A).

  When `true`, this property ensures the `textLength()` function counts 2 characters for each ENTER character in the text.

  When `false`, it uses the exact string size of the textarea's value.

  It defaults to `true`.
- **buttons** (array or function) - When the element is part of a list of checkboxes or radio buttons from which you want to get or set by an index position, this can be used to identify the DOM input elements that form the list.

  This property supports several values:

  - *function*: Your function must return an array of either string ids to elements or the actual DOM elements. The function takes one parameter, the original element assigned to the id property.

```
conn.buttons = function(id) { return ["Button1", "Button2", "Button3"]; }
```

    When defining buttons in JSON, you can create a function globally on the page and assign its name (as a string) to **buttons**. That function will be found and used here.

```
conn.buttons = "MyFunction";
```

  - *array of DOM elements*

```
conn.buttons = [document.getElementById("Button1"),
    document.getElementById("Button2"), document.getElementById("Button3")];
```

  - array of string ids to elements.

```
conn.buttons = ["Button1", "Button2", "Button3"];
```

  When the **buttons** property is not supplied, radio buttons are handled automatically because they are grouped by their **name** attribute. However, if the order returned by `document.getElementByName()` is incorrect, that's when you use the **buttons** property.

  Grouped checkboxes always require using **buttons** because there is no built in grouping in HTML.

Connections.UserFunction

Alias names (case sensitive):	"**UserFunction**"
Inherits from:	**Connections.Base**
Source file:	**\jTAC\Connections\UserFunction.js**

Write code that supplies a value to a Condition, such as a calculated number or a string built from several other fields.

For example, this adds TextBox1 and TextBox2's values together as integers before evaluating the result within a range.

```
function MyFunction(sender, values) {
   return values[0] + values[1];
}
if (jTAC.isValid({jtacClass: 'Range', connection: {jtacClass: 'Connections.UserFunction', fnc:
'MyFunction', connections: ['TextBox1', 'TextBox2'], datatype: 'Integer'}, minimum: 1, maximum:
100}))
   // code runs when true
else
   // code runs when false
```

If using calculations, also consider using the Calculation jquery-ui widget, which has its own Connection class.

This Connection class only gets data. It does not support `setTextValue()` or `setTypedValue()` methods.

Set up

Determine the type of data that you will return. If it is not a string, define the TypeManager that will be used to prepare the values passed into your function in the **datatype** or **typeManager** property. **datatype** takes a class or alias name which **typeManager** takes an instance of the desired TypeManager class.

Identify other sources of data that are used by your calculation by defining Connection objects or ids to elements on the page in the **connections** property. This is an array. Each element of this array will be used to get a value passed to the *values* parameter of your function.

If any of those Connection objects in **connections** must have a non-null value in order to call your function, set the **required** property. This is an array of booleans. Each element in **required** is associated with an element in the **connections** collection by position.

```
var conn = jTAC.create("Connections.UserFunction");
conn.connections = ["TextBox1", "TextBox2"];
conn.required = [true, true];
```

Now define the function. It takes two parameters:

- **sender** (Connection.UserFunction) - The object that calls this. Use it to access the TypeManager with `this.getTypeManager()`.
- **values** (Array of values) - This array contains values retrieved from the **connections** and prepared with the TypeManager (if used). The values will be strings if no TypeManager is used. These values may be null when the required property is not used to block calling your function. So write your code checking for null values.

Return a value compatible with the TypeManager or a string if no TypeManager. Return `null` if there is no value available.

Example: Concatentates two strings with a space separator

```
var fnc = function(sender, values) {
   return (vals[0] ? vals[0] : "") + " " + (vals[1] ? vals[1] : "");
}
```

Example: Adds two numbers

```
var fnc = function(sender, values) {
   return (vals[0] ? vals[0] : 0) + (vals[1] ? vals[1] : 0);
}
```

Assign the function to the **fnc** property.

```
conn.fnc = fnc;
```

If using unobtrusive setup, you cannot pass a function reference through the JSON code needed. So define your function globally (as a member of the `window` object). Specify its name in the **fnc** property. See the example at the start of this topic.

Making reusable calculations

You also can subclass Connections.UserFunction and override its `userFunction()` method. Use this when creating a reusable calculation. For example, this will sum all values:

```
var members = {
   extend: "Connections.UserFunction",

   userFunction: function(sender, vals) {
      var total = 0;
      for (var i = 0; i < vals.length; i++)
         if (vals[i] != null)
            total = total + vals[i];
         return total;
   }
}
jTAC.define("Connections.Sum", members);
```

Using your new class:

```
if (jTAC.isValid({jtacClass: 'Range', connection: {jtacClass: 'Connections.Sum', connections:
['TextBox1', 'TextBox2'], datatype: 'Integer'}, minimum: 1, maximum: 100}))
   // code runs when true
else
   // code runs when false
```

To learn more on creating classes, see "Adding your own classes to jTAC".

Connections.UserFunction Properties

- **fnc** (function) - The user function to call when you do not override this class. It can be assigned a JavaScript function or the name of a globally defined function (it is a function on the window object). See "Set up" for details on this function.

  Leave it `null` if overriding the `userFunction()` method in a subclass. See "Making reusable calculations".

- **connections** (array of Connections) - Connections that are the source of data passed to your user function through its *values* parameter. The position of a Connection is the same as the position of the value in the *values* parameter.

  This array can contain:

  - Actual Connection objects
  - Strings which are the IDs of elements on the page
  - A JavaScript object with the **jtacClass** property defining the Connection class to create and other properties to assign to the instance of that Connection class.

  All elements will be converted to Connection objects by the time the user function is invoked.

- **required** (array of boolean) - If you require a non-null value from a Connection, assign the array element matching the Connection element's position in **connections** to `true`. For example, if the 2nd Connection object in **connections** is required, use required: `[false, true]`.

  Can be left unassigned.

- **datatype** (string) - The class or alias name of the TypeManager that determines the native type of values passed into your user function. (This does not determine the type returned by your function. Use **typeName** for that.)

- **typeManager** (TypeManager) - The TypeManager that determines the native type of values passed into your user function. (This does not determine the type returned by your function. Use **typeName** for that.)

  Set this if the **datatype** property cannot specify the correct TypeManager because you need to explicitly set properties. It will be set if using **datatype** by the time your user function is invoked.

- **typeName** (string) - The type that will be returned by your user function. This value is used by the `typeSupported()` method to determine if the `getTypedValue()` method is available.

  This value is normally determined by the TypeManager's `storageTypeName()` method. If there is no TypeManager, it defaults to "string".

  Set it explicitly if another type is needed. Supported values: "integer", "float", "date", "time", "datetime", "boolean", "string".

Connections.JQUIDatePicker

Alias names (case sensitive):	none
Inherits from:	**Connections.BaseJQUIElement**
Source file:	**\jTAC\Connections\JQUIDatePicker.js**

Gets and sets the value of an `<input type="text" />` when it is attached to a *jquery-ui* datePicker widget. It understands this is using a date, so it supports `typeSupported("date")` and allows getting and setting a JavaScript Date object with the `getTypedValue()` and `setTypedValue()` methods.

Set up

Assign the id value of an HTML element to the **id** property.

Connections.JQUIDatePicker Properties

- **id** (string) - The unique id used to identify the element on the page. You can either set this or call `setElement()` to pass the actual object of the element.
- **allowNone** (boolean) - When `true`, if the id is not found when calling get or set methods, nothing happens. When `false`, an exception occurs. It defaults to `false`.

Connections.Value

Alias names (case sensitive):	none
Inherits from:	**Connections.Base**
Source file:	**\jTAC\Connections\Value.js**

For holding a value without being associated with an element on the page.

Set up

Assign the value to store in the **value** property.

If the value's type is something other than a string, identify the type name that makes `typeSupported()` return `true`. The type name can be assigned to the **supportedTypeName** property or determined when the **value** property is set.

For numbers that are float values, it set **supportedTypeName** to "float" because setting the **value** property alone will use **supportedTypeName**= "integer".

Connections.Value properties

- **value** - The actual value stored.
- **nullValue** - Determines a value that means `null`. Used by `isNullValue()` to compare the **value** property to this property. It defaults to `null`.
- **supportedTypeName** (string) – The type name that makes the `typeSupported()` function return `true`.

  It is set whenever you set the **value** property, so long as the value is a number, string, or boolean. You must set it explicitly for any other type or if your number is a float.

Connections.InnerHtml

Alias names (case sensitive):	none
Inherits from:	**Connections.BaseElement**
Source file:	**\jTAC\Connections\InnerHtml.js**

This Connection interacts with an HTML tag that supports the **innerHTML** attribute, such as , , and <td>. These tags contain HTML as their value, so they are not type specific. Therefore, there is no default TypeManager class.

This Connection was developed for use with the Calculator widget so it can display the result of a calculation in one of these HTML tags (instead of textbox). Specify the id of the HTML tag in the calculator's **displayElementId** and the calculator will use this Connection in its **displayConnection** property.

You can use this similarly when you want to use a TypeManager to convert a native value to a formatted string that is assigned to one of these HTML attributes.

Set up

Assign the id value of an HTML element to the **id** property.

Suppose you want to write a currency value to `<span id="total" ></span>`. Here's how you would write that code:

```
var val = 100.0;
var tm = jTAC.create("TypeManagers.Currency");
var conn = jTAC.create("Connections.InnerHtml", {id: "total"});
conn.setTextValue(tm.toString(val));
```

Connections.InnerHtml Properties

- **id** (string) - The unique id used to identify the element on the page. You can either set this or call `setElement()` to pass the actual object of the element.
- **allowNone** (boolean) - When `true`, if the id is not found when calling get or set methods, nothing happens. When `false`, an exception occurs. It defaults to `false`.

Connections.Base

Inherits from:	**jTAC's universal base class, jTACBaseClass**
Source file:	**\jTAC\Connections\Base.js**

Abstract base class that provides a framework for getting and setting values. It does not know about elements on the page at this level.

For its methods, see "Using Connections in your own JavaScript".

Subclassing Connections.Base methods

Getting started:

- All Connections inherit from `jTACClassBase` in **\jTAC\jTAC.js**. It has some useful utilities for your class development. Learn more in the **\jTAC\jTAC.js** file.
- See "Adding your own classes to jTAC" to use best practices in creating your subclasses' structure.
- For an example, see Connections.Value in **\jTAC\Connections\Value.js**.

This topic will identify methods that you are likely to override or call in Connections.Base. It will not describe parameters. Instead, please use the class definition in **\jTAC\Connections\Base.js**.

	Element Name	When
✓	`getTextValue()`	Abstract method. Always override
✓	`setTextValue()`	Abstract method. Always override
✓	`typeSupported()`	Abstract method. Always override. If you don't support native types, return `false`.
✓	`getTypedValue()`	Abstract method. Always override. If you don't support native types, return `null`.
✓	`setTypedValue()`	Abstract method. Always override. If you don't support native types, do nothing in your method.
✓	`isNullValue()`	Abstract method. Always override
✓	`textLength()`	This method returns the length of the string from `getTextValue()`.
✓	`isEditable()`	This method returns true. Override to change.
✓	`getLabel()`	This method works with **defaultLabel** and **defaultLookupKey** properties. Override to change.
✓	`getTypeManager()`	This method returns null. Override if you can identify a native type and create its TypeManager.

Key: ✓ always override; ✓ sometimes override; ✓ rarely override

Connections.BaseElement

Inherits from:	**Connections.Base**
Source file:	**\jTAC\Connections\BaseElement.js**

Abstract base class that provides a framework for getting and setting values on elements on the page.

- Using Connections in your own JavaScript
- Methods introduced by Connections.BaseElement

Subclassing Connections.BaseElement methods

Getting started:

- All Connections inherit from `jTACClassBase` in **\jTAC\jTAC.js**. It has some useful utilities for your class development. Learn more in the **\jTAC\jTAC.js** file.
- See "Adding your own classes to jTAC" to use best practices in creating your subclasses' structure.
- For an example, see Connections.FormElement in **\jTAC\Connections\FormElement.js**.

This topic will identify methods that you are likely to override or call in Connections.BaseElement. It will not describe parameters. Instead, please use the class definition in **\jTAC\Connections\Base.js**.

	Element Name	When
✓	`getTextValue()`	Abstract method. Always override
✓	`setTextValue()`	Abstract method. Always override
✓	`typeSupported()`	Abstract method. Always override. If you don't support native types, return `false`.
✓	`getTypedValue()`	Abstract method. Always override. If you don't support native types, return `null`.
✓	`setTypedValue()`	Abstract method. Always override. If you don't support native types, do nothing in your method.
✓	`isNullValue()`	Abstract method. Always override
✓	`getElement()`	This class works with DOM elements and retrieves the ID by `document.getElementById()`. Override if you are using a different type of object for your elements (like a *jQuery* object).
✓	`_checkElement()`	Abstract method. Always override.
✓	`addEventListener()`	This method does nothing. Always override.
✓	`addSupportEventListeners()`	This method does nothing. Always override.
✓	`testElement()`	This class returns false. Write code to determine your Connection class can support the id to an element passed.
✓	`getTypeManager()`	Calls `_createTypeManager()` to do most of the work.
✓	`_createTypeManager()`	Looks for the **data-jtac-datatype** and **data-jtac-typemanager** attributes on the HTML tag to create a TypeManager. Subclass if you know the correct TypeManager to create without those attributes involvement.
✓	`getData()`	Gets attributes starting with "data-" on the HTML element. Override if you are not working with HTML elements.
✓	`setData()`	Override if you are not working with HTML elements.
✓	`isVisible()`	Works with HTML elements. Override if you are not working with HTML elements.
✓	`isEnabled()`	Works with HTML elements. Override if you are not working with HTML elements.
✓	`getLabel()`	This method works with **defaultLabel** and **defaultLookupKey** properties. Also works with **data-msglabel** and **data-msglabel-lookupkey** attributes on the HTML element. Override if you are not working with HTML elements.
✓	`getClass()`	Works with HTML elements. Override if you are not working with HTML elements.

✓	`setClass()`	Works with HTML elements. Override if you are not working with HTML elements.
✓	`addClass()`	Works with HTML elements. Override if you are not working with HTML elements.
✓	`removeClass()`	Works with HTML elements. Override if you are not working with HTML elements.

Key: ✓ always override; ✓ sometimes override; ✓ rarely override

Connections.BaseJQUIElement

Inherits from:	**Connections.BaseElement**
Source file:	**\jTAC\Connections\BaseJQUIElement.js**

Abstract base class that provides a framework for getting and setting values on elements on the page.

- Using Connections in your own JavaScript
- Methods introduced by Connections.BaseElement

Subclassing Connections.BaseJQUIElement methods

Getting started:

- All Connections inherit from `jTACClassBase` in **\jTAC\jTAC.js**. It has some useful utilities for your class development. Learn more in the **\jTAC\jTAC.js** file.
- See "Adding your own classes to jTAC" to use best practices in creating your subclasses' structure.
- For an example, see Connections.JQUIDatePicker in **\jTAC\Connections\JQUIDatePicker.js**.

This topic will identify methods that you are likely to override or call in Connections.BaseJQUIElement. It will not describe parameters. Instead, please use the class definition in **\jTAC\Connections\Base.js**.

	Element Name	When
✓	`typeSupported()`	Abstract method. Always override. If you don't support native types, return `false`.
✓	`getTypedValue()`	Abstract method. Always override. If you don't support native types, return `null`.
✓	`setTypedValue()`	Abstract method. Always override. If you don't support native types, do nothing in your method.
✓	`isNullValue()`	Abstract method. Always override
✓	`_checkElement()`	Abstract method. Always override.
✓	`testElement()`	This class returns false. Write code to determine your Connection class can support the id to an element passed.
✓	`getTypeManager()`	Calls `_createTypeManager()` to do most of the work.
✓	`_createTypeManager()`	Looks for the **data-jtac-datatype** and **data-jtac-typemanager** attributes on the HTML tag to create a TypeManager. Subclass if you know the correct TypeManager to create without those attributes involvement.

Key: ✓ always override; ✓ sometimes override; ✓ rarely override

Using Connections in your own JavaScript

To use connections, be sure to add **\jTAC\Merged\Core.js** or **\jTAC\Connections\Base.js** script file on the page.

If you want to use a Connection object associated with an element on the page, use the `jTAC.connectionResolver` object to create the Connection object for you. Pass the element's ID to its `create()` function. It determines the best Connection class to use and creates an instance of that class.

```
var conn = jTAC.connectionResolver.create("TextBox1");
```

If you know the exact Connection class, you can create it explicitly with the `jTAC.create()` method.

```
var conn = jTAC.create("Connections.FormElement");
conn.id = "TextBox1";

var conn = jTAC.create("Connections.FormElement", {id: "TextBox1"});
```

The following topics introduce you to the properties and methods you may use with your code.

- Methods defined by the base class, Connections.Base
- Properties introduced by Connections.BaseElement
- Methods introduced by Connections.BaseElement

Methods defined by the base class, Connections.Base

Here are the functions available on all Connections.

```
function getTextValue()
```

Retrieve the string representation of the value. If there is no value, return the empty string (“”).

When subclassing: Must override.

```
function setTextValue(value)
```

Assign a value represented as a string.

value (string)

The value to assign.

When subclassing: Must override.

```
function typeSupported(typename)
```

Check if a type other than string is supported. If it is, the `setTypedValue()` and `getTypedValue()` methods can be used for this type.

Subclasses may introduce other type strings, such as a rich text editor may offer "string" to return a value without any formatting codes and "html" to get the formatted version.

typename (string)

The name of the type. Here are some names already in use by the Connections.FormElement class:

- "integer" - Supports a number value that is an integer. HTML5 number and range inputs support this.
- "float" - Supports a number value that is a float. HTML5 number input supports this.
- "date" - Supports a Date object representing only a date (the time is undefined). HTML5 date, DateTime, DateTime-local, and month inputs support this.
- "DateTime" - Supports a Date object representing both date and time. HTML5 DateTime and DateTime-local inputs support this.
- "time" - Supports a Date object representing only a time (the date is undefined). HTML5 time input supports this.
- "boolean" - Supports a boolean value. checkbox and radio inputs support this.
- "index" - Supports a number value representing an offset in a list. -1 = no selection. 0 = first element. HTML's <select> element supports this.
- "indices" - Supports an array of integers representing multiple offsets in a list. HTML's <select> element supports this.

Returns: `true` when the type name is supported and `false` when it is not.

When subclassing: The base class always returns `false`.

```
function getTypedValue(typename)
```

Retrieves a value of a specific type (number, Date, boolean, string). The caller must call `typeSupported()` first to determine if this function will work.

typename (string)

> If `null`, it returns the default type (usually there is a single type supported other than string). Otherwise, one of the values that would return `true` when calling `typeSupported()`.

Returns: The value based on the expected type. If the value cannot be created in the desired type, it returns `null`.

When subclassing: When your code supports typed values, override.

```
function setTypedValue(value)
```

Sets the value represented by a specific type (number, Date, boolean, string). The caller must call `typeSupported()` first to determine if this function will work.

Subclasses should test the type of the object passed in to ensure it is still legal.

value

> The value to update. Must be a type supported by this method or it throws an exception.

When subclassing: When your code supports typed values, override. If the values type is not compatible, throw an exception.

```
function isNullValue(override)
```

Determines if the value is assigned or not. For example, HTML form elements are null when their value attribute is "".

override (boolean)

> Some widgets can either return a real value or treat that value as `null`. For example, a checkbox and radio button can treat its **checked**=`false` state as either `false` or `null`. Normally `isNullValue()` should indicate `false` in these cases. When *override* is `true`, `isNullValue()` will indicate `true` in these cases.

Returns: `true` when the value represents null. `false` when it does not.

When subclassing: Connections.Base defines an abstract method that throws an exception. Must override.

```
function isValidValue(typename)
```

Determines if the value held by the widget will create a valid value for the given type.

typename (string)

> If `null`, it returns the default type (usually there is a single type supported other than string). Otherwise, pass one of the values that would return `true` when calling `typeSupported()`.

Returns: `true` if it can convert and `false` if not. It also returns true if `isNullValue()` is `true`.

When subclassing: Rarely override

```
function textLength()
```

Determines the length of the text value.

While usually the length is the same as the string value represented by the element, there are cases where the string value is not what is saved on post back.

For example, the HTML TextArea allows ENTER in the text. Many browsers report the length of the string with 1 character (%AO) for each ENTER. However, all browsers post back two characters (%0D%A0).

So this function can return a length adjusted with the value used in post back. Connections.FormElement handles the TextArea case, although you can override it by setting its **fixLength** property to `false`.

Returns: The length as an integer.

When subclassing: Connections.Base always returns the length of the string returned by `getTextValue()`, which takes into account the **trim** property.

```
function isEditable()
```

Determines if the element allows editing.

Returns: When `true`, it is editable. When `false`, it is not.

When subclassing: Connections.Base always returns `true`. Connections.BaseElement, it evaluates the visibility and disabled states of the element by calling its `isVisible()` and `isEnabled()` methods. Those are implemented to handle HTML elements. Subclass when your widget does not use the HTML disabled attribute or HTML styles “visibility” or “block” to determine visibility.

```
function getTypeManager()
```

Returns an instance of a TypeManager that reflects the element or `null` if no TypeManager can be identified.

Connections.FormElement identifies the TypeManager by looking at the element’s attributes for the **data-jtac-datatype** and **data-jtac-typemanager** attributes. It also knows about HTML5 inputs for dates, times, and range.

When subclassing: Connections.Base always returns `null`. When subclassing from Connections.BaseElement, you get support for the **data-jtac-datatype** and **data-jtac-typemanager** attributes.

```
function getLabel()
```

Returns a string that can be displayed as the label for the element. It may return `null`.

The label can be defined by the Connection’s **defaultLabel** and **defaultLookupKey** properties. Subclasses can introduce other ways to look up the label, such as looking for a <label for=> tag or the **data-msglabel** and **data-msglabel-lookupkey** attributes on the element specified by the Connection.

When subclassing: Connections.Base only works with **defaultLabel** and **defaultLookupKey** properties. When subclassing from Connections.BaseElement, you get support for the **data-msglabel** and **data-msglabel-lookupkey** attributes.

Properties introduced by Connections.BaseElement

These properties are available to all Connections that interact with an element on the page.

- **id** (string) - The unique id used to identify the element on the page. You can either set this or call `setElement()` to pass the actual object of the element.
- **allowNone** (boolean) - When `true`, if the id is not found when calling get or set methods, nothing happens. When `false`, an exception occurs. It defaults to `false`.
- **trim** (boolean) - Determines if strings are returned from `getTextValue()` after trimming lead and trailing spaces. `isNullValue()` also applies this property to evaluate if `null`. It defaults to `true`.
- **unassigned** (string) - Allows for watermarks in a textbox and a non-empty string value for a list to mean unselected. A watermark is text that appears when the textbox is empty.

  Lists usually use the value "" to mean an unselected state. But if you want to have a "no selection" item with a different value, assign this to the string for that value.

  When assigned, this string is compared to the textual value of the element. If it matches, `getTextValue()` returns the empty string and `isNullValue()` returns `true`.

  It defaults to `null`.
- **unassignedCase** (boolean) - When using the **unassigned** property, this determines if the 'unassigned' string is compared case sensitive or not. When `true`, it compares case sensitive. When `false`, it does not. It defaults to `false`.

Methods introduced by Connections.BaseElement

These methods are available to all Connections that interact with an element on the page.

```
function getElement(noneAllowed)
```

Returns the object associated with the **id** that manages the widget. Throws exceptions if not found, unless **allowNone** is `true`.

noneAllowed (boolean)

When defined, it overrides the **allowNone** property. When `null/undefined`, use **allowNone**.

Returns: object for the element or null if not found.

```
function setElement(value)
```

Lets you set the element object itself that is returned by `getElement()`. Normally you will assign the **id** property and let `getElement()` do the work.

Use this when you have an instance of the element AND do not expect it to be deleted for the life of this Connection.

value (object)

The object reflecting the element.

```
function addEventListener(name, func, funcid)
```

Attaches a function to an event handler on the element.

name (string)

Name of the event handler. Strings are case sensitive.

Typical names: "onchange", "onfocus", "onblur", "onclick", "onkeydown".

func (function)

A function that will be called when the event is raised. Its parameters should either match those supported by the event, which is usually the event object itself, or there should be no parameters.

funcid (string)

Optional. If you want to only hook up the function once to the element, pass a name here. That name will be attached to the element's data. Later calls with the same name will not attempt to add the event.

A good usage is attaching a validation function. That function generally handles all validators. So only the first validation rule should attach the validation function. When not used, omit the parameter or pass null.

Returns: When `true`, the event was attached. When `false` it was not supported.

When subclassing: Subclasses determine how to do the actual work.

```
function getData(key)
```

Attempts to get a value associated with the data collection of the element. For an HTML element, it can get any attribute whose name starts with “data-“.

key (string)

The value key in the data collection. If specifying an attribute on the HTML tag, that attribute must start with “data-“ and this key must omit that prefix.

Returns: The value or `null` if not available. The value may be a non-string type if stored previously by `setData()`. It is a string if coming from an HTML attribute.

When subclassing: This class handles HTML elements.

```
function setData(key, value)
```

Assigns or replaces the value into the data collection of the element.

key (string)

The value key in the data collection.

value

The data to store with the key.

When subclassing: This class handles HTML elements.

```
function isVisible()
```

Determines if the element is visible.

Returns: `true` if visible and `false` if not.

When pointing to an HTML element, it looks at the **display** and **visibility** styles on the actual HTML element and up the DOM tree until it finds one invisible or nothing being invisible.

When subclassing: This class handles HTML elements.

```
function isEnabled()
```

Determines if the element is enabled.

Returns: `true` if enabled and `false` if disabled.

When pointing to an HTML element, it looks at the **disabled** attribute of the element. If no attribute exists, it returns `true`.

When subclassing: This class handles HTML elements.

```
function getClass()
```

Returns the current style sheet class name associated with the element or "" if none.

When subclassing: This class handles HTML elements.

```
function setClass(css)
```

Sets (replaces) the current style sheet class name. If the widget has multiple classes for its various parts, this impacts the part most closely associated with the data entry, such as the input tag.

css (string)

The class name to assign.

When subclassing: This class handles HTML elements.

```
function addClass(css)
```

Appends the class name provided to the existing one. Effectively it creates the pattern "[old class] [new class]".

css (string)

The class name to append.

Returns: `true` unless the class already appears. If so, no changes are made and it returns `false`.

When subclassing: This class handles HTML elements.

```
function removeClass(css)
```

Removes the style sheet class from the element's current class.

css (string)

The class name to remove.

When subclassing: This class handles HTML elements.

Using the DataTypeEditor widget

The **DataTypeEditor** widget is a *jquery-ui* widget that turns an ordinary `<input type='text' />` into a powerful editor of a specific data type.

Its primary tool is the TypeManager class. It uses any TypeManager to parse the text input and reformat it as the user leaves the field. It also filters out invalid keystrokes. Finally, it has a command processor so you can hook up special keys to change the value. For example, the TypeManagers.Date class uses the up and down arrows to increment and decrement the date.

```
<input type='text' id='TextBox1' name='TextBox1'
   data-jtac-datatype="Currency" data-jtac-datatypeeditor="" />
```

Click on any of these topics to jump to them:

Working with inline code (no unobtrusive setup)

1. Load the script files. See "Loading scripts for the DataTypeEditor widget".

```
<script src="/jquery/jquery-#.#.#.js" type="text/javascript"></script>
<script src="/jquery/jquery-ui-#.#.#.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/core.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/jquery extensions/validation-all.js"
   type="text/javascript"></script>
<script src="/jTAC-min/Merged/typemanagers-all.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/jquery extensions/textbox widgets.js"
   type="text/javascript"></script>
```

   Note: The validation-all and typemanagers-all script files are suggestions. These have merged all available scripts. If you are looking for smaller scripts, see Loading scripts for the DataTypeEditor widget.

2. Add an <input type="text" > with both its **id** and **name** attributes set to the same id value.

```
<input type="text" id="textbox1" name="textbox1" />
```

3. Add the **data-jtac-datatype** attribute to it. Its value can be any TypeManager name including a number of custom names that create TypeManagers with specific properties already set. See "Using TypeManagers in datatype properties".

```
<input type="text" id="textbox1" name="textbox1" data-jtac-datatype="Integer" />
```

   Note: This attribute's value is case sensitive.

   *Note: **data-jtac-datatype** is only valid within HTML 5. If you are using something else and demand validated markup, you can define the same value in the **datatype** property of the DataTypeEditor's options object.*

4. If you need to assign properties to the TypeManager, add the **data-jtac-typemanager** attribute. Assign it to a string containing JSON, where each property name is the property you want to update on the TypeManager.

```
<input type="text" id="textbox1" name="textbox1" data-jtac-datatype="Integer"
   data-jtac-typemanager="{'allowNegatives' : false}" />
```

   *Note: **data-jtac-typemanager** is only valid within HTML 5. If you are using something else and demand validated markup, you can define the same value in the **typemanager** property of the DataTypeEditor's options object.*

5. Add an `<input type="hidden" />` with both its id set to the same id as the textbox, but with the addition of "_neutral" (case sensitive).

```
<input type="hidden" id="textbox1_neutral" name="textbox1_neutral" />
```

   Note: This is not required but makes it much easier to communicate values with the server side. It always holds a culture neutral format of the data, so you don't have to write parsers on the server side.

6. Get the *jQuery* object for this input tag and call its `dataTypeEditor()` method, passing in options. Options are a JavaScript object with properties described in "DataTypeEditor Options".

   In `$(document).ready()`, add a rule using this syntax.

```
$("#textbox1").dataTypeEditor();
```

   This is customizing the options by passing a JavaScript object.

```
$("#textbox1").dataTypeEditor({property:value, property2:value});
```

7. Because you are allowing the user to edit a data type, always setup validation to ensure a legal value was entered. Use the rule name "datatypecheck" with *jquery-validate*.

```
$("#textbox1").rules("add", {
   datatypecheck : {}
});
```

   ALERT: Always validate the strings on the server side too! Never expect valid input even when you used these nice validators on the client-side because hackers will work around them.

8. If you want to impose range limits, also add the “advrange” rule, but instead of setting up its minimum and maximum in the rule definition, do it in the **data-jtac-typemanager** attribute on the textbox using the properties “minValue” and “maxValue”. This way, keyboard commands and range validation will both know about the limits.

```
<input type="text" id="textbox1" name="textbox1"
    data-jtac-datatype="Integer" data-jtac-typemanager="{'minValue' : 1, 'maxValue': 10}" />

$("#textbox1").rules("add", {
   datatypecheck : {},
   advrange : {}
});
```

9. On the server side, you need to write code that gets and sets the value attribute of the hidden field. The value is always a string that uses culture neutral formats without any of the formatting shown to the user in the textbox. Here are the standard culture neutral formats.

Type	Pattern	Examples
Integers	[-]digits	"1", "10000", "-1"
Float, Currency, Percent	[-]digits.digits	"1.0", "10000.0", "-1.0"
Date	yyyy-MM-dd	"2000-05-02"
Time of Day, Duration	H:mm:ss	"0:00:00", "16:30:21"
Date and Time	yyyy-MM-dd H:mm:ss	"2000-05-02 16:30:21"
Day Month	MM-dd	"05-02"
Month Year	yyyy-MM	"2000-05"

Working with unobtrusive setup

Note: If you are not working with HTML 5 but demand validated markup, do not use unobtrusive setup. It introduces attributes to the textbox that are not validated except in HTML 5.

1. Load the script files. See "Loading scripts for the DataTypeEditor widget".

```
<script src="/jquery/jquery-#.#.#.js" type="text/javascript"></script>
<script src="/jquery/jquery-ui-#.#.#.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.unobtrusive.js" type="text/javascript"></script>

<script src="/jTAC-min/Merged/core.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/jquery extensions/validation-all.js"
   type="text/javascript"></script>
<script src="/jTAC-min/Merged/typemanagers-all.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/jquery extensions/textbox widgets-unobtrusive.js"
   type="text/javascript"></script>
```

   Note: The validation-all and typemanagers-all script files are suggestions. These have merged all available scripts. If you are looking for smaller scripts, see Loading scripts for the DataTypeEditor widget.

2. Add an `<input type="text" />` tag with both its id and name attributes set to the same id value.

```
<input type="text" id="textbox1" name="textbox1" />
```

3. Add the **data-jtac-datatype** attribute to it. Its value can be any TypeManager name including a number of custom names that create TypeManagers with specific properties already set. See "Using TypeManagers in datatype properties".

```
<input type="text" id="textbox1" name="textbox1" data-jtac-datatype="Integer" />
```

   Note: This attribute's value is case sensitive.

4. If you need to assign properties to the TypeManager, add the **data-jtac-typemanager** attribute. Assign it to a string containing JSON, where each property name is the property you want to update on the TypeManager.

```
<input type="text" id="textbox1" name="textbox1" data-jtac-datatype="Integer"
   data-jtac-typemanager="{'allowNegatives': false}" />
```

5. Add an `<input type="hidden" />` tag with both its id set to the same id as the textbox, but with the addition of "_neutral" (case sensitive).

```
<input type="hidden" id="textbox1_neutral" name="textbox1_neutral" />
```

   Note: This is not required but makes it much easier to communicate values with the server side. It always holds a culture neutral format of the data, so you don't have to write parsers on the server side.

6. Add the **data-jtac-datatypeeditor** attribute to the textbox. Assign its value to a string containing the options. Options are a JSON object with properties described in "DataTypeEditor Options".

```
<input type="text" id="textbox1" name="textbox1" data-jtac-datatype="Integer"
   data-jtac-datatypeeditor="" />
```

   This is customizing the options by assigning it to JSON.

```
<input type="text" id="textbox1" name="textbox1" data-jtac-datatype="Integer"
   data-jtac-datatypeeditor="{'keyErrorClass': 'badkey'}" />
```

7. Because you are allowing the user to edit a data type, always setup validation to ensure a legal value was entered. Use the rule name "datatypecheck" with *jquery-validate*.

```
<input type="text" id="textbox1" name="textbox1" data-jtac-datatype="Integer"
   data-val="true" data-val-datatypecheck=""
   data-jtac-datatypeeditor="{'keyErrorClass': 'badkey'}" />
```

   ALERT: Always validate the strings on the server side too! Never expect valid input even when you used these nice validators on the client-side because hackers will work around them.

8. If you want to impose range limits, also add the “advrange” rule, but instead of setting up its minimum and maximum in the rule definition, do it in the **data-jtac-typemanager** attribute on the textbox using the properties “minValue” and “maxValue”. This way, keyboard commands and range validation will both know about the limits.

```
<input type="text" id="textbox1" name="textbox1" data-jtac-datatype="Integer"
    data-jtac-typemanager="{'minValue': 1, 'maxValue': 10}"
    data-val="true" data-val-datatypecheck="" data-val-advrange=""
    data-jtac-datatypeeditor="{'keyErrorClass': 'badkey'}" />
```

9. On the server side, you need to write code that gets and sets the value attribute of the hidden field. The value is always a string that uses culture neutral formats without any of the formatting shown to the user in the textbox. Here are the standard culture neutral formats.

Type	Pattern	Examples
Integers	[-]digits	"1", "10000", "-1"
Float, Currency, Percent	[-]digits.digits	"1.0", "10000.0", "-1.0"
Date	yyyy-MM-dd	"2000-05-02"
Time of Day, Duration	H:mm:ss	"0:00:00", "16:30:21"
Date and Time	yyyy-MM-dd H:mm:ss	"2000-05-02 16:30:21"
Day Month	MM-dd	"05-02"
Month Year	yyyy-MM	"2000-05"

DataTypeEditor Options

The DataTypeEditor can be customized by using a JavaScript object containing these properties.

When working with code, pass the object to the `dataTypeEditor()` method.

```
$("#ElementID").dataTypeEditor({property:value, property2:value});
```

When working with unobtrusive setup, set the **data-jtac-datatypeeditor** attribute to the object as JSON.

```
<input type="text" id="textbox1" name="textbox1" data-jtac-datatype="Integer"
   data-jtac-datatypeeditor="{'property':value, 'property2':value}" />
```

- **datatype** (string) - Alternative to using the **data-jtac-datatype** attribute assigned to the textbox. Only used when that attribute is missing. See "Using TypeManagers in datatype properties".
- **typeManager** (TypeManager object) - Used instead of the **data-jtac-datatype** and **data-jtac-typemanager** attributes on the textbox when assigned. It must host a fully setup TypeManager object. (The **data-jtac-typemanager** attribute's values will not be used to revise its properties.)
- **reformat** (boolean) - When `true`, the onchange event will reformat the contents to match the most desirable format. Effectively, it uses the TypeManager's `toValue()` and `toString()` methods. Nothing happens if there is a formatting error. It defaults to `true`.
- **filterkeys** (boolean) - When `true`, keystrokes are evaluated for valid characters. Invalid characters are filtered out. (This uses the TypeManager's `isValidChar()` method.) When `false`, no keystroke filtering is supported. It defaults to `true`.
- **checkPastes** (boolean) - On browsers that support the onpaste event, this attempts to evaluate the text from the clipboard before it is pasted. It checks that each character is valid for the TypeManager. If any invalid character is found, the entire paste is blocked. It will also block pasting non-textual (including HTML) content. It defaults to `true`.

  Supported browsers: IE, Safari, and Chrome.
- **commandKeys** (array) - An array of objects that map a keystroke to a command associated with the TypeManager's command feature.

  Requires loading **\jTAC\TypeManagers\Command extensions.js** (which is included when you load **\jTAC\Merged\jquery extensions\datatypeeditor.js**).

  When `null`, it uses the values supplied by the TypeManager object.

  Each object in the array has these properties:

  - **action** (string) - The TypeManager may supply an initial list of commands (date, time and numbers already do). As a result, this list is considered a way to modify the original list. **action** indicates how to use the values of the remaining properties.

    It supports these values:

    "`add`" (or omit the action property) - Adds the object to the end of the list.

    "`clear`" - remove the existing list. Use this to abandon items added by the TypeManager. Only use this as the first item in **commandKeys**.

    "`remove`" - remove an item that exactly matches the remaining properties. If you want to replace an existing item, remove that item then add a new item.
  - **commandName** (string) - The command name to invoke. Here are the command names available by TypeManager:

    All Date classes:

    * "NextDay", "PrevDay" - Increase or decrease by one day. Will update month at the day borders

    * "NextMonth", "PrevMonth" - Increase or decrease by one month. Will update year at the month borders

    * "NextYear", "PrevYear" - Increase or decrease by one year

    * "Today" - Assign today's date

    * "Clear" - Clear the date. Empty textbox.

Time and duration classes:

* "NextSec", "PrevSec" - Increases or decreases by one second.

* "NextMin", "PrevMin" - Increases or decreases by one minute.

* "NextHr", "PrevHr" - Increases or decreases by one hour.

* "Now" - Assign the current time of day.

* "Clear" - Clear the date. Empty textbox.

Number classes:

* "NextBy1", "PrevBy1" - Increases or decrease by 1

* "NextBy10", "PrevBy10" - Increases or decrease by 10

* "NextBy100", "PrevBy100" - Increases or decrease by 100

* "NextBy1000", "PrevBy1000" - Increases or decrease by 1000

* "NextByPt1", "PrevByPt1" - Increases or decrease by 0.1

* "NextByPt01", "PrevByPt01" - Increases or decrease by 0.01

* "Clear" - Clear the date. Empty textbox.

- **keyCode** - The keyCode to intercept. A keycode is a number returned by the DOM event object representing the key typed. It can also be a string with a single character.

  Common key codes:

  ENTER = 13, ESC = 27, PAGEUP = 33, PAGEDOWN = 35, HOME = 36, END = 35, LEFT = 37, RIGHT = 39,

  UP = 38, DOWN = 40, DELETE = 46, F1 = 112 .. F10 = 122.

- **shift** (boolean) - When true, requires the shift key pressed.
- **ctrl** (boolean) - When true, requires the control key pressed.

Example

```
commandKeys = [{action: "add", commandName: "NextDay", keyCode="+"},
    {action:"add", commandName: "PrevDay", keyCode="-"}]
```

Commands are only invoked for keystrokes that are not considered valid by the TypeManager. For example, if you define the keycode for a letter and letters are valid characters, the command will not fire. As a result, map several key combinations to the same **commandName**.

Assign to an empty array to avoid using commands.

- **getCommandName** (function) - A function hook used when evaluating a keystroke to see if it should invoke a command. When unassigned, it uses the default function which uses the commandKeys collection to determine which command to invoke. Use this when commands may change based on the situation.

  Your function must have these parameters:

  evt - the *jQuery* event object to evaluate

  cmdKeys - the commandKeys collection, which are described above.

  tm - the TypeManager object associated with this DateTextBox.

  It must return one of these values:

  - `null` - Continue processing the command.
  - commandName (string) - Use this command name to invoke the command.
  - "" (the empty string) - Stop processing this keystroke

- **keyResult** (function) - A function hook that is called after each keystroke is processed. It lets you know the result of the keystroke with one of these three strings: “valid”, “invalid”, “command”.

  Use it to modify the user interface based on keystrokes. Its primary use is to change the UI when there is an invalid keystroke. If unassigned, a default function is setup called `jTAC_DefaultKeyResult()`. It uses a default UI that changes the border using the style sheet class of the **keyErrorClass** property. Your own **keyResult** function may call `jTAC_DefaultKeyResult()`. It has the same parameters as your function.

  The keyResult function takes these parameters:

  element - the *jQuery* object for the textbox

  event (the jquery event object) - Look at its keyCode for the keystroke typed.

  result (string) - "valid", "invalid", "command"

  options - the options object with user options.

  Returns nothing.

  Set this to `null` to disable the default behavior without implementing an alternative.

  The **keyResult** property can be assigned as a string that reflects the name of a globally defined function too (for the benefit of JSON and unobtrusive setup).

- **keyErrorClass** (string) -The style sheet class added to the textbox when it needs to show an error. It defaults to "key-error".
- **keyErrorTime** (integer) - Number of milliseconds to show the **keyErrorClass** on the textbox after the user types an illegal key. It defaults to 200.

Loading scripts for the DataTypeEditor widget

jTAC offers several ways to add scripts. This section shows using the merged format. If you want minified files, specify the **jTAC-min** folder. Otherwise specify the **jTAC** folder. For more, see "Understanding the available scripts files".

1. Add support for *jquery*, *jquery-ui*, and *jquery-validate*. (The unobtrusive validation feature is optional.)

```
<script src="/jquery/jquery-#.#.#.js" type="text/javascript"></script>
<script src="/jquery/jquery-ui-#.#.#.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.unobtrusive.js" type="text/javascript"></script>
```

2. If you want to load a single script file with all of jTAC, use this:

```
<script src="/jTAC-min/Merged/jquery extensions/jtac-all.js" type="text/javascript"></script>
```

   Then skip to step 6. Otherwise continue with the next step.

3. Always load **core.js** before any other jTAC script file.

```
<script src="/jTAC-min/Merged/core.js" type="text/javascript"></script>
```

4. If using a Condition that requires a TypeManager, load one or more of these files defining the desired TypeManager objects:

```
<script src="/jTAC-min/Merged/typemanagers-numbers.js" type="text/javascript"></script>
```
or
```
<script src="/jTAC-min/Merged/typemanagers-date-time.js" type="text/javascript"></script>
```
or
```
<script src="/jTAC-min/Merged/typemanagers-date-time-all.js" type="text/javascript"></script>
```
or
```
<script src="/jTAC-min/Merged/typemanagers-strings-common.js" type="text/javascript"></script>
```
or
```
<script src="/jTAC-min/Merged/typemanagers-strings-common-all.js"
   type="text/javascript"></script>
```
or
```
<script src="/jTAC-min/Merged/typemanagers-all.js" type="text/javascript"></script>
```

TypeManager class	numbers	date-time	d-t-all	strings common	strings all	all
TypeManagers.Boolean						x
TypeManagers.CreditCardNumber					x	x
TypeManagers.Currency	x					x
TypeManagers.DateTime		x	x			x
TypeManagers.Date		x	x			x
TypeManagers.DayMonth			x			x
TypeManagers.Duration			x			x
TypeManagers.EmailAddress				x	x	x
TypeManagers.Float	x					x
TypeManagers.Integer	x	x	x			x
TypeManagers.MonthYear			x			x
TypeManagers.Percent	x					x
TypeManagers.PhoneNumber				x	x	x
TypeManagers.PostalCode				x	x	x
TypeManagers.String			x	x	x	x

TypeManagers.TimeOfDay		x	x			x
Url					x	x

5. Add one of these two script files for the DataTypeEditor widget. The second installs unobtrusive support.

```
<script src="/jTAC-min/Merged/jquery extensions/textbox widgets.js"
    type="text/javascript"></script>
```

or

```
<script src="/jTAC-min/Merged/jquery extensions/textbox widgets-unobtrusive.js"
    type="text/javascript"></script>
```

6. If you are need localization of number, date, or time TypeManagers, add the appropriate scripts. See "Localizing dates, times, and numbers".

   Here is an example using *jquery-globalize*. Add its script file first. Add any culture specific files from *jquery-globalize*. Then add the **TypeManagers\Culture engine for jquery-globalize.js** file. All of these files can be located below the TypeManager scripts.

```
<script src="/jquery-globalize/globalize.js" type="text/javascript"></script>
<script src="/jquery-globalize/cultures/globalize.culture.fr-FR.js"
    type="text/javascript"></script>
<script src="/jTAC-min/TypeManagers/Culture engine for jquery-globalize.js"
    type="text/javascript"></script>
```

7. If you are using dates and times, consider switching parsers from the default to the TypeManagers.PowerDateParser or TypeManagers.PowerTimeParser. If you do, here are the links for those classes.

```
<script src="/jTAC-min/TypeManagers/PowerDateParser.js" type="text/javascript"></script>
<script src="/jTAC-min/TypeManagers/PowerTimeParser.js" type="text/javascript"></script>
```

Example

```
<script src="/jquery/jquery-#.#.#.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.unobtrusive.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/core.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/jquery extensions/validation-typical.js"
   type="text/javascript"></script>
<script src="/jTAC-min/Merged/typemanagers-numbers.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/typemanagers-date-time.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/jquery extensions/textbox widgets-unobtrusive.js"
   type="text/javascript"></script>
<script src="/jquery-globalize/globalize.js" type="text/javascript"></script>
<script src="/jquery-globalize/cultures/globalize.culture.fr-FR.js"
   type="text/javascript"></script>
<script src="/jTAC-min/TypeManagers/Culture engine for jquery-globalize.js"
   type="text/javascript"></script>
<script src="/jTAC-min/TypeManagers/PowerDateParser.js" type="text/javascript"></script>
```

Using the DateTextBox widget

The **DateTextBox** widget is a *jquery-ui* widget that turns an ordinary `<input type="text" />` into a powerful date editor with the support of the *jquery-ui* Datepicker widget.

It effectively replaces the parser of the Datepicker with the parser from the TypeManagers supplied with jTAC. With the TypeManagers.PowerDateParser added to the page, the user gets a more fluid data entry field. DateTextBox also supports keyboard shortcut commands.

```
<input type='text' id='TextBox1' name='TextBox1' data-jtac-datatype="Date"
   data-jtac-datetextbox="{datepicker : { buttonImage : '/Images/Button.gif'} }" />
```

You will not need to declare the Datepicker itself. It is created for you within this widget. (Its options can still be set.) It is created with some differences from the default Datepicker:

- Modifies the default options of the datepicker widget to make it work gracefully with the DataTypeEditor. For example, while the Datepicker is popped up, several keystroke commands of DataTypeEditor need to be disabled.
- Defaults to a button to popup instead of focus, because the keyboard entry features make for faster entry using the textbox than having to work through a calendar. (Focus first suggests that the user goes to the calendar for most entry.)
- Applies the localization rules as described in "Localizing dates, times, and numbers".

Click on any of these topics to jump to them:

Working with inline code (no unobtrusive setup)

1. Load the script files. See "Loading scripts for the DateTextBox widget".

```
<script src="/jquery/jquery-#.#.#.js" type="text/javascript"></script>
<script src="/jquery/jquery-ui-#.#.#.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/core.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/jquery extensions/validation-all.js"
   type="text/javascript"></script>
<script src="/jTAC-min/Merged/typemanagers-all.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/jquery extensions/textbox widgets.js"
   type="text/javascript"></script>
<script src="/jTAC-min/TypeManagers/PowerDateParser.js" type="text/javascript"></script>
```

   Note: The validation-all and typemanagers-all script files are suggestions. These have merged all available scripts. If you are looking for smaller scripts, see Loading scripts for the DateTextBox widget.

2. Add an `<input type="text" />` with both its **id** and **name** attributes set to the same id value.

```
<input type="text" id="datetextbox1" name="datetextbox1" />
```

3. Add the **data-jtac-datatype** attribute to it. Here are some values to choose from:

   "Date" - Uses TypeManagers.Date in its default settings. (Identical to using Date.Short)

   "Date.Short" - Uses TypeManagers.Date in Short date format

   "Date.Abbrev" - Uses TypeManagers.Date in Abbreviated date format

   "Date.Long" - Uses TypeManagers.Date in Long date format

```
<input type="text" id="datetextbox1" name="datetextbox1" data-jtac-datatype="Date" />
```

   Note: This attribute's value is case sensitive.

   Note: Date.Abbrev and Date.Long require another parser. Switch to the TypeManagers.PowerDateParser.

   *Note: **data-jtac-datatype** is only valid within HTML 5. If you are using something else and demand validated markup, you can define the same value in the **datatype** property of the DateTextBox's options object.*

4. Add an `<input type="hidden" />` with both its id set to the same id as the textbox, but with the addition of "_neutral" (case sensitive).

```
<input type="hidden" id="datetextbox1_neutral" name="datetextbox1_neutral" />
```

   Note: This is not required but makes it much easier to communicate values with the server side. It always holds a culture neutral format of the data, so you don't have to write parsers on the server side.

5. Get the *jQuery* object for this input tag and call its `dateTextBox()` method, passing in options. Options are a JavaScript object with properties described in "DateTextBox Options".

   The options contain the property **datePicker**. Use this to pass along the options used by the datePicker itself.

   In `$(document).ready()`, add a rule using this syntax.

```
$("#datetextbox1").dateTextBox();
```

   This is customizing the options by passing a JavaScript object.

```
$("#datetextbox1").dateTextBox({property:value, property2:value,
   datePicker: {property:value, property:value}});
```

6. Because you are allowing the user to edit a data type, always setup validation to ensure a legal value was entered. Use the rule name "datatypecheck" with *jquery-validate*.

```
$("#datetextbox1").rules("add", {
   datatypecheck : {}
});
```

ALERT: Always validate the strings on the server side too! Never expect valid input even when you used these nice validators on the client-side because hackers will work around them.

7. If you want to impose range limits, also add the “advrange” rule, but instead of setting up its minimum and maximum in the rule definition, do it in the **data-jtac-typemanager** attribute on the textbox using the properties “minValue” and “maxValue”. This way, keyboard commands and range validation will both know about the limits. These values must be in the culture neutral format: "yyyy-MM-dd”.

```
<input type="text" id="datetextbox1" name="datetextbox1" data-jtac-datatype="Date"
    data-jtac-typemanager="{'minValue': '2000-01-01', 'maxValue': '2020-12-31'}" />

$("#datetextbox1").rules("add", {
   datatypecheck : {},
   advrange : {}
});
```

8. On the server side, you need to write code that gets and sets the value attribute of the hidden field. The value is always a string that uses culture neutral formats without any of the formatting shown to the user in the textbox. Here are the standard culture neutral formats.

Type	Pattern	Examples
Date	yyyy-MM-dd	"2000-05-02"

Working with unobtrusive setup

Note: If you are not working with HTML 5 but demand validated markup, do not use unobtrusive setup. It introduces attributes to the textbox that are not validated except in HTML 5.

1. Load the script files. See "Loading scripts for the DateTextBox widget".

```
<script src="/jquery/jquery-#.#.#.js" type="text/javascript"></script>
<script src="/jquery/jquery-ui-#.#.#.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.unobtrusive.js" type="text/javascript"></script>

<script src="/jTAC-min/Merged/core.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/jquery extensions/validation-all.js"
   type="text/javascript"></script>
<script src="/jTAC-min/Merged/typemanagers-all.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/jquery extensions/textbox widgets-unobtrusive.js"
   type="text/javascript"></script>
<script src="/jTAC-min/TypeManagers/PowerDateParser.js" type="text/javascript"></script>
```

   Note: The validation-all and typemanagers-all script files are suggestions. These have merged all available scripts. If you are looking for smaller scripts, see Loading scripts for the DateTextBox widget.

2. Add an `<input type="text" />` with both its **id** and **name** attributes set to the same id value.

```
<input type="text" id="datetextbox1" name="datetextbox1" />
```

3. Add the **data-jtac-datatype** attribute to it. Here are some values to choose from:

   "Date" - Uses TypeManagers.Date in its default settings. (Identical to using Date.Short)

   "Date.Short" - Uses TypeManagers.Date in Short date format

   "Date.Abbrev" - Uses TypeManagers.Date in Abbreviated date format

   "Date.Long" - Uses TypeManagers.Date in Long date format

```
<input type="text" id="datetextbox1" name="datetextbox1" data-jtac-datatype="Date" />
```

   Note: This attribute's value is case sensitive.

   Note: Date.Abbrev and Date.Long require another parser. Switch to the TypeManagers.PowerDateParser.

   *Note: **data-jtac-datatype** is only valid within HTML 5. If you are using something else and demand validated markup, you can define the same value in the **datatype** property of the DateTextBox's options object.*

4. Add an `<input type="hidden" />` with both its id set to the same id as the textbox, but with the addition of "_neutral" (case sensitive).

```
<input type="hidden" id="datetextbox1_neutral" name="datetextbox1_neutral" />
```

   Note: This is not required but makes it much easier to communicate values with the server side. It always holds a culture neutral format of the data, so you don't have to write parsers on the server side.

5. Add the **data-jtac-datetextbox** attribute to the textbox. Assign its value to a string containing the options. Options are a JSON object with properties described in "DateTextBox Options".

   The options contain the property **datepicker**. Use this to pass along the options used by the datePicker itself.

```
<input type="text" id="datetextbox1" name="datetextbox1" data-jtac-datatype="Date"
   data-jtac-datetextbox="" />
```

   This is customizing the options by assigning it to JSON.

```
<input type="text" id="datetextbox1" name="datetextbox1" data-jtac-datatype="Date"
   data-jtac-datetextbox=
   "{'keyErrorClass': 'badkey','datepicker': {'buttonImageUrl': '/images/calendar.png'}}" />
```

6. Because you are allowing the user to edit a data type, always setup validation to ensure a legal value was entered. Use the rule name “datatypecheck” with *jquery-validate*.

```
<input type="text" id="datetextbox1" name="datetextbox1" data-jtac-datatype="Date"
    data-val="true" data-val-datatypecheck=""
    data-jtac-datetextbox=
    "{'keyErrorClass': 'badkey', 'datepicker': {'buttonImageUrl': '/images/calendar.png'}}" />
```

ALERT: Always validate the strings on the server side too! Never expect valid input even when you used these nice validators on the client-side because hackers will work around them.

7. If you want to impose range limits, also add the “advrange” rule, but instead of setting up its minimum and maximum in the rule definition, do it in the **data-jtac-typemanager** attribute on the textbox using the properties “minValue” and “maxValue”. This way, keyboard commands and range validation will both know about the limits. These values must be in the culture neutral format: "yyyy-MM-dd”.

```
<input type="text" id="datetextbox1" name="datetextbox1" data-jtac-datatype="Date"
    data-jtac-typemanager="{'minValue': '2000-01-01', 'maxValue': '2020-12-31'}"
    data-val="true" data-val-datatypecheck="" data-val-advrange=""
    data-jtac-datetextbox=
    "{'keyErrorClass': 'badkey','datepicker': {'buttonImageUrl': '/images/calendar.png'}}" />
```

8. On the server side, you need to write code that gets and sets the value attribute of the hidden field. The value is always a string that uses culture neutral formats without any of the formatting shown to the user in the textbox. Here are the standard culture neutral formats.

Type	Pattern	Examples
Date	yyyy-MM-dd	"2000-05-02"

DateTextBox Options

The DateTextBox can be customized by using a JavaScript object containing these properties.

When working with code, pass the object to the `dateTextBox()` method.

```
$("#ElementID").dateTextBox({property:value, property2:value});
```

When working with unobtrusive setup, set the **data-jtac-datetextbox** attribute to the object as JSON.

```
<input type="text" id="textbox1" name="textbox1" data-jtac-datatype="Date"
   data-jtac-datetextbox="{'property':value, 'property2':value}" />
```

- **datepicker** (object) - An object that reflects the same properties as the options passed into the *jquery-ui* Datepicker. This object is passed to the datepicker as it is created.

  http://docs.jquery.com/UI/Datepicker#options

- **datatype** (string) - Alternative to using the **data-jtac-datatype** attribute assigned to the textbox. Only used when that attribute is missing. Its value can be "Date" or alias developed for the TypeManagers.Date class.

- **typeManager** (TypeManager object) - Used instead of the **data-jtac-typemanager** attribute on the textbox when assigned. It must host the TypeManagers.Base subclass, fully setup or a JavaScript object with properties to assign to the TypeManager identified by the **datatype** property.

- **reformat** (boolean) - When `true`, the onchange event will reformat the contents to match the most desirable format. Effectively, it uses the TypeManager's `toValue()` and `toString()` methods. Nothing happens if there is a formatting error. It defaults to `true`.

- **filterkeys** (boolean) - When `true`, keystrokes are evaluated for valid characters. Invalid characters are filtered out. (This uses the TypeManager's `isValidChar()` method.) When `false`, no keystroke filtering is supported. It defaults to `true`.

- **checkPastes** (boolean) - On browsers that support the onpaste event, this attempts to evaluate the text from the clipboard before it is pasted. It checks that each character is valid for the TypeManager. If any invalid character is found, the entire paste is blocked. It will also block pasting non-textual (including HTML) content. It defaults to `true`.

  Supported browsers: IE, Safari, and Chrome.

- **commandKeys** (array) - An array of objects that map a keystroke to a command associated with the TypeManager's command feature.

  Requires loading **\jTAC\TypeManagers\Command extensions.js** (which is included when you load **\jTAC\Merged\jquery extensions\datetextbox.js**).

  When `null`, it uses the values supplied by the TypeManager object.

  Each object in the array has these properties:

  - **action** (string) - The TypeManager may supply an initial list of commands (date, time and numbers already do). As a result, this list is considered a way to modify the original list. **action** indicates how to use the values of the remaining properties.

    It supports these values:

    "`add`" (or omit the action property) - Adds the object to the end of the list.

    "`clear`" - remove the existing list. Use this to abandon items added by the TypeManager. Only use this as the first item in **commandKeys**.

    "`remove`" - remove an item that exactly matches the remaining properties. If you want to replace an existing item, remove that item then add a new item.

  - **commandName** (string) - The command name to invoke. Here are the command names available:

    * "NextDay", "PrevDay" - Increase or decrease by one day. Will update month at the day borders

    * "NextMonth", "PrevMonth" - Increase or decrease by one month. Will update year at the month borders

    * "NextYear", "PrevYear" - Increase or decrease by one year

* "Today" - Assign today's date

* "Clear" - Clear the date. Empty textbox.

- **keyCode** - The keyCode to intercept. A keycode is a number returned by the DOM event object representing the key typed. It can also be a string with a single character.

  Common key codes:

  ENTER = 13, ESC = 27, PAGEUP = 33, PAGEDOWN = 35, HOME = 36, END = 35, LEFT = 37, RIGHT = 39,

  UP = 38, DOWN = 40, DELETE = 46, F1 = 112 .. F10 = 122.

- **shift** (boolean) - When true, requires the shift key pressed.
- **ctrl** (boolean) - When true, requires the control key pressed.

Example

```
commandKeys = [{action: "add", commandName: "NextDay", keyCode: "+"},
               {action: "add", commandName: "PrevDay", keyCode: "-"}]
```

Commands are only invoked for keystrokes that are not considered valid by the TypeManager. For example, if you define the keycode for a letter and letters are valid characters, the command will not fire. As a result, map several key combinations to the same **commandName**.

Assign to an empty array to avoid using commands.

- **getCommandName** (function) - A function hook used when evaluating a keystroke to see if it should invoke a command. When unassigned, it uses the default function which uses the commandKeys collection to determine which command to invoke. Use this when commands may change based on the situation.

  Your function must have these parameters:

  evt - the *jQuery* event object to evaluate

  cmdKeys - the commandKeys collection, which are described above.

  tm - the TypeManager object associated with this DateTextBox.

  It must return one of these values:

  - `null` - Continue processing the command.
  - commandName (string) - Use this command name to invoke the command.
  - "" (the empty string) - Stop processing this keystroke

- **keyResult** (function) - A function hook that is called after each keystroke is processed. It lets you know the result of the keystroke with one of these three strings: “valid”, “invalid”, “command”.

  Use it to modify the user interface based on keystrokes. Its primary use is to change the UI when there is an invalid keystroke. If unassigned, a default function is setup called `jTAC_DefaultKeyResult()`. It uses a default UI that changes the border using the style sheet class of the **keyErrorClass** property. Your own **keyResult** function may call `jTAC_DefaultKeyResult()`. It has the same parameters as your function.

  The keyResult function takes these parameters:

  element - the *jQuery* object for the textbox

  event (the *jQuery* event object) - Look at its keyCode for the keystroke typed.

  result (string) - "valid", "invalid", "command"

  options - the options object with user options.

  Returns nothing.

  Set this to `null` to disable the default behavior without implementing an alternative.

  The **keyResult** property can be assigned as a string that reflects the name of a globally defined function too (for the benefit of JSON and unobtrusive setup).

- **keyErrorClass** (string) -The style sheet class added to the textbox when it needs to show an error. It defaults to "key-error".
- **keyErrorTime** (integer) - Number of milliseconds to show the **keyErrorClass** on the textbox after the user types an illegal key. It defaults to 200.

Loading scripts for the DateTextBox widget

jTAC offers several ways to add scripts. This section shows using the merged format. If you want minified files, specify the **jTAC-min** folder. Otherwise specify the **jTAC** folder. For more, see "Understanding the available scripts files".

Note: If you already loaded scripts for the dataTypeEditor widget, skip to the last step as the earlier steps are identical.

1. Add support for *jQuery*, *jquery-ui*, and *jquery-validate*. (The unobtrusive validation feature is optional.)

```
<script src="/jquery/jquery-#.#.#.js" type="text/javascript"></script>
<script src="/jquery/jquery-ui-#.#.#.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.unobtrusive.js" type="text/javascript"></script>
```

2. If you want to load a single script file with all of jTAC, use this:

```
<script src="/jTAC-min/Merged/jquery extensions/jtac-all.js" type="text/javascript"></script>
```

   Then skip to step 6. Otherwise continue with the next step.

3. Always load **core.js** before any other jTAC script file.

```
<script src="/jTAC-min/Merged/core.js" type="text/javascript"></script>
```

4. Load one of these files defining the desired TypeManager objects:

```
<script src="/jTAC-min/Merged/typemanagers-date-time.js" type="text/javascript"></script>
```

   or

```
<script src="/jTAC-min/Merged/typemanagers-date-time-all.js" type="text/javascript"></script>
```

   or

```
<script src="/jTAC-min/Merged/typemanagers-all.js" type="text/javascript"></script>
```

5. Add one of these two script files for the DateTextBox. The second installs unobtrusive support.

```
<script src="/jTAC-min/Merged/jquery extensions/textbox widgets.js"
   type="text/javascript"></script>
```

   or

```
<script src="/jTAC-min/Merged/jquery extensions/textbox widgets-unobtrusive.js"
   type="text/javascript"></script>
```

6. If you are need localization of number, date, or time TypeManagers, add the appropriate scripts. See "Localizing dates, times, and numbers".

   Here is an example using *jquery-globalize*. Add its script file first. Add any culture specific files from *jquery-globalize*. Then add the **TypeManagers\Culture engine for jquery-globalize.js** file. All of these files can be located below the TypeManager scripts.

```
<script src="/jquery-globalize/globalize.js" type="text/javascript"></script>
<script src="/jquery-globalize/cultures/globalize.culture.fr-FR.js"
   type="text/javascript"></script>
<script src="/jTAC-min/TypeManagers/Culture engine for jquery-globalize.js"
   type="text/javascript"></script>
```

7. Consider switching parsers from the default to the TypeManagers.PowerDateParser for a better user experience. If you do, here are the links for those classes.

```
<script src="/jTAC-min/TypeManagers/PowerDateParser.js" type="text/javascript"></script>
```

Example

```
<script src="/jquery/jquery-#.#.#.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.js" type="text/javascript"></script>
<script src="/jquery-validate/jquery.validate.unobtrusive.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/core.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/jquery extensions/validation-typical.js"
   type="text/javascript"></script>
<script src="/jTAC-min/Merged/typemanagers-date-time.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/jquery extensions/textbox widgets-unobtrusive.js"
   type="text/javascript"></script>
<script src="/jquery-globalize/globalize.js" type="text/javascript"></script>
<script src="/jquery-globalize/cultures/globalize.culture.fr-FR.js"
   type="text/javascript"></script>
<script src="/jTAC-min/TypeManagers/Culture engine for jquery-globalize.js"
   type="text/javascript"></script>
<script src="/jTAC-min/TypeManagers/PowerDateParser.js" type="text/javascript"></script>
```

Using the Calculator widget

The **Calculator** widget is a *jquery-ui* widget where you can create powerful numeric calculations that interact with input elements on the page and show the result. Often used in shopping carts and grid UIs. This widget uses the Connection and TypeManager classes to quickly connect to an input widget and convert its value to a number.

Example

This form has two textboxes that accept integers (due to the DataTypeEditor feature). The hidden input hosts the calculation, which adds the values of both textboxes and displays the result in the Row1Result span tag. The **data-jtac-calculator** attribute is used for unobtrusive setup. The **expression** property defines the calculation. The **displayElementId** property identifies where the total is shown.

```
<input type='text' id='r1c1' name='r1c1' data-jtac-datatype="Integer"
   data-jtac-datatypeeditor="" />
<input type='text' id='r1c2' name='r1c2' data-jtac-datatype="Integer"
   data-jtac-datatypeeditor="" />
<span id="Row1Result"></span>
<input type="hidden" id="r1Total" name="r1Total"
   data-jtac-calculator='{ "expression": &#39;"r1c1" + "r1c2"&#39;,
      "displayElementId": "Row1Result", "useKeyEvt": true }' />
```

Note: ' is the HTML character representation of a single quote. This allows nested single quotes.

Validation often needs the result of a calculation. For example, the total of the above two fields must be less than 100. jTAC's validators and Conditions accept the id of the hidden input and will ensure the calculation is up-to-date before using the value.

Example

This is the same as above, with the addition of an advrange validation rule requiring the total to be between 0 and 100.

```
<input type="hidden" id="r1Total" name="r1Total"
   data-jtac-calculator='{ "expression": &#39;"r1c1" + "r1c2"&#39;,
      "displayElementId": "Row1Result", "useKeyEvt": true }'
   data-val="true" data-val-advrange="" data-val-advrange-json="{minimum: 0, maximum: 100}" />
```

Calculations are built upon CalcItem objects. Here are the CalcItem classes that are included with jTAC:

Class name	Purpose
CalcItems.Element	Gets the value from an element on the page, including textboxes and other Calculators.
CalcItems.Number	Holds a number.
CalcItems.Group	Groups other CalcItem objects, calculating them together like using parenthesis.
CalcItems.Conditional	Allows IF THEN logic. Uses Conditions to determine which of two CalcItems will be used.
CalcItems.Null	Returns `null`, which means nothing to calculate.
CalcItems.NaN	Returns `NaN`, which means there was an error in the calculation.
CalcItems.Round	Pass in another CalcItem or expression. This function rounds the result.
CalcItems.Abs	Pass in another CalcItem or expression. This function returns the absolute value.
CalcItems.Avg	Pass in a list of CalcItems or expressions. This function returns the average value.
CalcItems.Min	Pass in a list of CalcItems or expressions. This function returns the lowest value.
CalcItems.Fix	Use when another CalcItem may return `null` or `NaN` and you want to replace that value with something else, such as a CalcItems.Number.
CalcItems.Max	Pass in a list of CalcItems or expressions. This function returns the highest value.
CalcItems.UserFunction	Invokes your own function, passing in the value of a CalcItem or expression.

Click on any of these topics to jump to them:

Locate a CalcItem

Here are the CalcItem classes supplied by jTAC. Click on their name to learn more about them. Each of these classes is defined in files of the **jTAC****Calculations**\ folder.

Class name	Purpose
CalcItems.Element	Gets the value from an element on the page, including textboxes and other Calculators.
CalcItems.Number	Holds a number.
CalcItems.Group	Groups other CalcItem objects, calculating them together like using parenthesis.
CalcItems.Conditional	Allows IF THEN logic. Uses Conditions to determine which of two CalcItems will be used.
CalcItems.Null	Returns `null`, which means nothing to calculate.
CalcItems.NaN	Returns `NaN`, which means there was an error in the calculation.
CalcItems.Round	Pass in another CalcItem or expression. This function rounds the result.
CalcItems.Abs	Pass in another CalcItem or expression. This function returns the absolute value.
CalcItems.Avg	Pass in a list of CalcItems or expressions. This function returns the average value.
CalcItems.Min	Pass in a list of CalcItems or expressions. This function returns the lowest value.
CalcItems.Fix	Use when another CalcItem may return `null` or `NaN` and you want to replace that value with something else, such as a CalcItem.Number.
CalcItems.Max	Pass in a list of CalcItems or expressions. This function returns the highest value.
CalcItems.UserFunction	Invokes your own function, passing in the value of a CalcItem or expression.

Working with inline code (no unobtrusive setup)

1. Load the script files. See "Loading script files for the Calculator widget".

```
<script src="/jquery/jquery-#.#.#.js" type="text/javascript"></script>
<script src="/jquery/jquery-ui-#.#.#.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/core.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/typemanagers-basic.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/jquery extensions/calculations.js"
   type="text/javascript"></script>
```

   Note: The typemanagers-all script file is a suggestion. It has merged all available scripts. If you are looking for smaller scripts, see Loading script files for the Calculator widget.

2. Add an `<input type="hidden" >` with both its **id** and **name** attributes set to the same id value. This will host the calculations.

```
<input type="hidden" id="Calculator1" name="Calculator1" />
```

3. Get the *jQuery* object for this input tag and call its `calculator()` method, passing in options. Options are a JavaScript object with properties described in "Calculator Options".

   The options are where you define the calculation. You can use either its **calcItem** or **expression** property. **calcItem** uses objects while **expression** uses a string that is parsed. Usually **expression** is easier to use, but requires more processing by the browser as it parses. The syntax of **expression** is meant to resemble JavaScript.

   In `$(document).ready()`, add a rule using this syntax.

```
$("#Calculator1").calculator({property: value, property2 : value});
```

   The expression consists of these objects.

Class	Purpose	Parsing syntax
CalcItems.Element	Gets the value from an element on the page, including textboxes and other Calculators.	ID of the element in quotes: "TextBox1"
CalcItems.Number	Holds a number.	Number compatible with JavaScript syntax: 1, 1.0
CalcItems.Group	Groups other CalcItem objects, calculating them together like using parenthesis.	Parenthesis enclosing any CalcItem. Each is separated by a math operator of +, -, *, \.
CalcItems.-Conditional	Allows IF THEN logic. Uses Conditions to determine which of two CalcItems will be used.	`Condition({jtacClass:'conditionname', other properties}, "expression when success", "expression when failed")`
CalcItems.Null	Returns null, which means nothing to calculate.	`null`
CalcItems.NaN	Returns NaN, which means there was an error in the calculation.	`NaN`
CalcItems.Abs	Pass in another CalcItem or expression. This function returns the absolute value.	`Abs("expression")` or `Math.abs("expression")`
CalcItems.Avg	Pass in a list of CalcItems or expressions. This function returns the average value.	`Avg("expression", "expression2")`
CalcItems.Min	Pass in a list of CalcItems or expressions. This function returns the lowest value.	`Min("expression", "expression2")` or `Math.min("expression", "expression2")`
CalcItems.Fix	Pass in another CalcItem or expression. Then supply expressions for the null and NaN cases.	`Fix("expression", "expression when null", "expression when NaN")`
CalcItems.Max	Pass in a list of CalcItems or expressions. This function returns the highest value.	`Max("expression", "expression2")` or `Math.max("expression", "expression2")`
CalcItems.Round	Pass in another CalcItem or expression. This function rounds the result.	`Round("expression", mode, maxdecimalplaces)` `Round("TextBox1" / "TextBox2", 2, 3)`
CalcItems.-UserFunction	Invokes your own function, passing in the value of a CalcItem or expression.	`Function("function name", "expression", "expression2", etc)`

Examples

All of these examples do the same thing: add TextBox1 and TextBox2. This first case spells out everything needed in **calcItem**. Use it when you want to customize the individual objects created, as each CalcItem object has its own properties.

```
var ci1 = jTAC.create("CalcItems.Element", {elementId: "TextBox1"});
var ci2 = jTAC.create("CalcItems.Element", {elementId: "TextBox2"});
var ci = jTAC.create("CalcItems.Group");
ci.addItem(ci1);
ci.addItem(ci2);
$("#Calculator1").calculator({calcItem: ci});
```

Use an array to hold id strings and numbers. It converts to an expression summing the items.

```
$("#Calculator1").calculator({calcItem: ["TextBox1", "TextBox2"]});
```

Use the **expression** property. The parser recognizes the strings as IDs to elements on the page and converts them into the CalcItems.Element objects.

```
$("#Calculator1").calculator({expression: "'TextBox1' + 'TextBox2'"});
```

Working with unobtrusive setup

Note: If you are not working with HTML 5 but demand validated markup, do not use unobtrusive setup. It introduces attributes to the textbox that are not validated except in HTML 5.

1. Load the script files. See "Loading script files for the Calculator widget".

```
<script src="/jquery/jquery-#.#.#.js" type="text/javascript"></script>
<script src="/jquery/jquery-ui-#.#.#.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/core.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/typemanagers-basic.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/jquery extensions/calculations-unobtrusive.js"
    type="text/javascript"></script>
```

   Note: The typemanagers-all script file is a suggestion. It has merged all available scripts. If you are looking for smaller scripts, see Loading script files for the Calculator widget.

2. Add an `<input type="hidden" >` with both its **id** and **name** attributes set to the same id value. This will host the calculations.

```
<input type="hidden" id="Calculator1" name="Calculator1" />
```

3. Add the **data-jtac-calculator** attribute to the textbox. Assign its value to a string containing the options. Options are a JSON object with properties described in "Calculator Options".

   The options are where you define the calculation. You can use either its **calcItem** or **expression** property. **calcItem** uses objects while **expression** uses a string that is parsed. Usually **expression** is easier to use, but requires more processing by the browser as it parses. The syntax of **expression** is meant to resemble JavaScript. Be sure to enclose the expression in "'" (which is the HTML representation of a single quote) because you will need to use single quotes inside of your expression.

```
data-jtac-calculator="{'expression': &#39;'TextBox1' + 'TextBox2'&#39;}"
```

   The expression consists of these objects.

Class	Purpose	Parsing syntax
CalcItems.Element	Gets the value from an element on the page, including textboxes and other Calculators.	ID of the element in quotes: "TextBox1"
CalcItems.Number	Holds a number.	Number compatible with JavaScript syntax: 1, 1.0
CalcItems.Group	Groups other CalcItem objects, calculating them together like using parenthesis.	Parenthesis enclosing any CalcItem. Each is separated by a math operator of +, -, *, \.
CalcItems.-Conditional	Allows IF THEN logic. Uses Conditions to determine which of two CalcItems will be used.	Condition({jtacClass:'conditionname', other properties}, "expression when success", "expression when failed")
CalcItems.Null	Returns null, which means nothing to calculate.	null
CalcItems.NaN	Returns NaN, which means there was an error in the calculation.	NaN
CalcItems.Abs	Pass in another CalcItem or expression. This function returns the absolute value.	Abs("expression") or Math.abs("expression")
CalcItems.Avg	Pass in a list of CalcItems or expressions. This function returns the average value.	Avg("expression", "expression2")
CalcItems.Fix	Pass in another CalcItem or expression. Then supply expressions for the null and NaN cases.	Fix("expression", "expression when null", "expression when NaN")
CalcItems.Min	Pass in a list of CalcItems or expressions. This function returns the lowest value.	Min("expression", "expression2") or Math.min("expression", "expression2")
CalcItems.Max	Pass in a list of CalcItems or expressions. This function returns the highest value.	Max("expression", "expression2") or Math.max("expression", "expression2")
CalcItems.Round	Pass in another CalcItem or expression. This function rounds the result.	Round("expression", mode, maxdecimalplaces) Round("TextBox1" / "TextBox2", 2, 3)

CalcItems.-UserFunction	Invokes your own function, passing in the value of a CalcItem or expression.	Function("function name", "expression", "expression2", etc)

Examples

All of these examples do the same thing: add TextBox1 and TextBox2. This first case spells out everything needed in **calcItem**. Use it when you want to customize the individual objects created, as each CalcItem object has its own properties.

```
<input type="hidden" id="Calculator1" name="Calculator1"
   data-jtac-calculator="{'calcItem': {'jtacClass':'CalcItems.Group', 'items':
      [{'jtacClass': 'CalcItems.Element', 'elementId': 'TextBox1' },
       {'jtacClass': 'CalcItems.Element', 'elementId': 'TextBox2' } ]}" />
```

Use an array to hold id strings and numbers. It converts to an expression summing the items.

```
<input type="hidden" id="Calculator1" name="Calculator1"
   data-jtac-calculator="{'calcItem': ['TextBox1', 'TextBox2']}" />
```

Use the **expression** property. The parser recognizes the strings as IDs to elements on the page and converts them into the CalcItems.Element objects.

```
<input type="hidden" id="Calculator1" name="Calculator1"
   data-jtac-calculator="{'expression': &#39;'TextBox1' + 'TextBox2'&#39;}" />
```

Note: ' is the HTML character representation of a single quote. This allows nested single quotes.

Calculator Options

The Calculator can be customized by using a JavaScript object containing these properties.

When working with code, pass the object to the `calculator()` method.

```
$("#Calculator1").calculator({property:value, property2:value});
```

When working with unobtrusive setup, set the **data-jtac-calculator** attribute to the object as JSON.

```
<input type="hidden" id="Calculator1" name="Calculator1"
   data-jtac-calculator="{'property':value, 'property2':value}" />
```

- **calcItem** (CalcItems object) - The expression to calculate using calcItem objects.

  calcItem hosts a single CalcItem object, which is the base of the calculation. Normally you will use a CalcItems.Group here. If you don't assign it explicitly, a CalcItems.Group is created for you.

  Example when writing code. (Not supported by unobtrusive setup.)

```
var ci1 = jTAC.create("CalcItems.Element", {elementId: "TextBox1"});
var ci2 = jTAC.create("CalcItems.Element", {elementId: "TextBox2"});
var ci = jTAC.create("CalcItems.Group");
ci.addItem(ci1);
ci.addItem(ci2);
$("#Calculator1").Calculator({calcItem: ci});
```

  When using unobtrusive setup, assign it to a JavaScript object with the **jtacClass** property assigned to the name of the CalcItem class to use. The remaining properties must be valid properties on that CalcItem class, with values compatible with its properties. In many cases, a CalcItem object has its own calcItem property which also accepts all of these formats. When using unobtrusive setup, use JSON.

```
<input type="hidden" id="Calculator1" name="Calculator1"
   data-jtac-calculator="{'calcItem': {'jtacClass':'CalcItems.Group', 'items':
      [{'jtacClass': 'CalcItems.Element', 'elementId': 'TextBox1' },
       {'jtacClass': 'CalcItems.Element', 'elementId': 'TextBox2' } ]}" />
```

  There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

- **expression** (string) – The expression to calculate in a JavaScript like syntax. This is an alternative to setting the **calcItem** property.

  A parser converts the string into CalcItem objects, which then are assigned to the **calcItem** property.

  Note: Parser errors are reported to the browser's Console feature, found in their Development tools window. Review the Console when testing your code.

  When using unobtrusive set up, enclose your expression in "'". This is the HTML symbol for a single quote, since you will be using single quotes as literals inside your expression.

```
data-jtac-calculator="{'expression': &#39;'TextBox1' + 'TextBox2'&#39;}"
```

- **calcOnInit** (boolean) - When `true`, an initial calculation is run as the page is loaded, updating the **displayConnection** for the user's benefit. It defaults to `true`.
- **useChangeEvt** (boolean) - When `true`, textboxes will invoke the calculation when their onchange event fires. It defaults to `true`.
- **useKeyEvt** (boolean) - When true, textboxes will invoke the calculation as the user types. It defaults to `false`.
- **displayElementId** (string) - Assigns the ID of the element that will display the calculated value. It is usually a textbox or a tag.

  Only set this up if you want to display the value. It's common to avoid displaying the value, but still use the calculation to drive other calculations or be evaluated by a Condition object.

- **displayConnection** (Connection object) - When **displayElementId** is assigned, it creates this object, a Connection object to the element that displays the calculated value.

Typically you use this to modify the properties of the Connection object that was defined. Expect either a Connections.FormElement or Connections.InnerHtml, depending on the element being a textbox or .

- **displayDatatype** (string) – Defines a TypeManager that will format the string shown by **displayConnection**. This takes a string name of the TypeManager, including short hand names as defined in "Using TypeManagers in datatype properties".

  Avoid using TypeManagers that are not intended to handle numbers.

- **displayTypeManager** (TypeManager object) – Defines a TypeManager that will format the string shown by the **displayConnection**. This value can be set by assigning the **displayDatatype** property too. If you use **displayDataType**, you can use this to set properties on the TypeManager object that was created.

  If not assigned, it uses TypeManagers.Float.

  Avoid using TypeManagers that are not intended to handle numbers.

- **displayErrorClass** (string) - When the calculation reports an error, this style sheet class is assigned to the element identified by **displayConnection**.

  It defaults to "calcwidget-error".

- **displayNullClass** (string) - When the calculation has nothing to show, this style sheet class is assigned to the element identified by **displayConnection**.

  It defaults to "calcwidget-null".

- **displayErrorText** (string) - When the calculation reports an error, this string is assigned to the element identified by **displayConnection**.

  It defaults to "***".

- **displayNullText** (string) - When the calculation has nothing to show, this string is assigned to the element identified by **displayConnection**.

  It defaults to "".

- **preCalc** (function) – Provide a function that is called prior to calculating. It is often used to modify element values or detect errors. Its result determines if the calculation proceeds or not.

  This function must be compatible with *jQuery*'s `bind()` method. You can use a parameterless function. If you want parameters, `bind()` passes two parameters: *event* and *ui*. Both of those parameters will be passed as `null` to your preCalc function.

  If it returns `false`, the remainder of the calculation is skipped.

  It defaults to `null`.

  When using unobtrusive setup, you cannot pass a reference to a function in JSON. Instead, create your function on the page as a global function (attached to the window object). Then specify the name of your function as the value for the **preCalc** property.

- **postCalc** (function) – Provide a function that is called after calculating, but before the **displayConnection** is updated. It is often used to update the user interface.

  This function must be compatible with *jQuery*'s `bind()` method. You can use a parameterless function. If you want parameters, `bind()` passes two parameters: *event* and *ui*. *Event* will be null. *ui* will be an object with one property, **value**, which holds the numeric value of the calculation, `null`, or `NaN`.

  The function's result is ignored.

```
options.postCalc = function(evt, ui) {
    if (isNaN(ui.value))
        alert("Could not calculate");
}
```

  When using unobtrusive setup, you cannot pass a reference to a function in JSON. Instead, create your function on the page as a global function (attached to the window object). Then specify the name of your function as the value for the **postCalc** property.

- **change** (function) – Provide a function that is called after calculating when the calculated value has changed. Use it to be notified when the Calculator's value has changed. This is automatically used by *jquery-validate* when a validator is assigned to the Calculator's hidden field.

  This function must be compatible with *jQuery*'s `bind()` method. You can use a parameterless function. If you want parameters, `bind()` passes two parameters: *event* and *ui*. *Event* will be null. *ui* will be an object with one property, **value**, which holds the numeric value of the calculation, `null`, or `NaN`.

  The function result is ignored.

```
options.change = function(evt, ui) {
    // your code. ui.value is the value of the calculation.
}
```

  When using unobtrusive setup, you cannot pass a reference to a function in JSON. Instead, create your function on the page as a global function (attached to the window object). Then specify the name of your function as the value for the **change** property.

Assigning properties that accept a CalcItem object

Throughout the Calculation widget and individual CalcItem objects, there are properties that accept a CalcItem object. For example, Calculator's calcItem property and the items array in CalcItems.Group.

You don't have to work so hard to create the CalcItem object because these properties all accept a variety of values that know how to convert into a CalcItem object. Take advantage of this frequently.

- Assign a string to create a CalcItems.Element with the string as the ID of the element on the page, including textboxes and hidden fields used by other Calculators.
- Assign a number to create a CalcItems.Number with the number as the value.
- Assign `null` to creates CalcItems.Null.
- Assign `NaN` to create a CalcItems.NaN
- Assign a JavaScript object with the **jtacClass** property assigned to the class name of the CalcItem class to use. Add property names found on that class that you want to assign. In many cases, a CalcItem object has its own **calcItem** property which also accepts all of these formats. When using unobtrusive setup, use JSON.

  For example:

```
<input type="hidden" id="Calculator1" name="Calculator1"
   data-jtac-calculator="{'calcItem': {'jtacClass':'CalcItems.Group', 'items':
      [{'jtacClass': 'CalcItems.Element', 'elementId': 'TextBox1' },
       {'jtacClass': 'CalcItems.Element', 'elementId': 'TextBox2' } ]}" />
```

- Assign an Array containing any of the types described here. This creates a CalcItems.Group that sums the values it is given. Usually you give it the IDs of textboxes.

  For example:

```
<input type="hidden" id="Calculator1" name="Calculator1"
   data-jtac-calculator="{'calcItem': ['TextBox1', 'TextBox2']}" />
```

  Each child CalcItem uses a "+" operator by default. You can change that by adding a string item with "+", "-", "*", "/" preceding the CalcItem that needs is operator changed.

```
<input type="hidden" id="Calculator2" name="Calculator2"
   data-jtac-calculator="{'calcItem': ['TextBox1', '-', 'TextBox2']}" />
```

Loading script files for the Calculator widget

jTAC offers several ways to add scripts. This section shows using the merged format. If you want minified files, specify the **jTAC-min** folder. Otherwise specify the **jTAC** folder. For more, see "Understanding the available scripts files".

1. Add support for *jQuery* and *jquery-ui*. (The unobtrusive validation feature is optional.)

```
<script src="/jquery/jquery-#.#.#.js" type="text/javascript"></script>
<script src="/jquery/jquery-ui-#.#.#.js" type="text/javascript"></script>
```

2. If you want to load a single script file with all of jTAC, use this:

```
<script src="/jTAC-min/Merged/jquery extensions/jtac-all.js" type="text/javascript"></script>
```

   Otherwise continue with the next step.

3. Always load **core.js** before any other jTAC script file.

```
<script src="/jTAC-min/Merged/core.js" type="text/javascript"></script>
```

4. Ensure the numeric TypeManagers are loaded. All three merged TypeManager files provide the numeric TypeManagers which are needed.

```
<script src="/jTAC-min/Merged/typemanagers-basic.js" type="text/javascript"></script>
```

5. Add one of these two script files for the calculator widget. The second installs unobtrusive support.

```
<script src="/jTAC-min/Merged/jquery extensions/calculations.js"
   type="text/javascript"></script>
```

   or

```
<script src="/jTAC-min/Merged/jquery extensions/calculations-unobtrusive.js"
   type="text/javascript"></script>
```

Loading scripts for Calculations without jquery-ui support

1. If you want to load a single script file with all of jTAC, use this:

```
<script src="/jTAC-min/Merged/jtac-all.js" type="text/javascript"></script>
```

2. Otherwise add these jTAC script files.

```
<script src="/jTAC-min/Merged/core.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/typemanagers-numbers.js" type="text/javascript"></script>
<script src="/jTAC-min/Merged/calculations.js" type="text/javascript"></script>
```

Note: Substitute typemanagers-all.js or select individual TypeManager classes as needed. Just be sure to include them before calculations.js.

CalcItems.Element

Alias names (case sensitive):	none
Inherits from:	**CalcItems.Base**
Source file:	**\jTAC\CalcItems\Element.js**

Evaluates an element on the page, whether a textbox or another Calculator widget.

It uses a Connection object to get the value of the element and a TypeManager to convert string values to numbers.

Set up

Specify the ID of the element in the **elementId** property.

```
var ci = jTAC.create("CalcItems.Element");
ci.elementId = "TextBox1";
var ci2 = jTAC.create("CalcItems.Element", {elementId: "TextBox2"});
```

Often a textbox has an illegal value. Use the **valueWhenInvalid** property to define a CalcItem that will be returned in that case. By default, it returns a CalcItems.NaN, which tells the caller that there was an error.

When a textbox is blank, use the **valueWhenNull** property to define a CalcItem that will be returned in that case. By default, it returns a CalcItems.Null, which tells the caller that there was no value.

You can often avoid writing code to instantiate this class. Instead, you can use the ID of the element as a string in properties that accept a CalcItem, such as the **Calculator.calcItem** property and in the **CalcItems.Group.items** array.

```
var gci = jTAC.create("CalcItems.Group");
gci.items = ["TextBox1", "TextBox2"];
```

Using the expressions property

When used in an **expression** property, this has two formats:

- The ID of an element alone, as a string:

  ```
  "id"
  ```

  This value must be a case sensitive match to the id attribute on the element.

  ```
  {expression: "'TextBox1' + 'TextBox2'" }
  ```

- A function named "Element" with parameters:

  ```
  Element(id)
  ```

  ```
  {expression: "Element('TextBox1') + 'TextBox2'" }
  ```

CalcItems.Element Properties

- **elementId** (string) – Specifies the id of the element on the page to evaluate. While its common to use textboxes and the hidden input of another Calculator widget, you can use anything that provides a numeric value, so long as a Connection class supports it.
- **connection** (Connection object) - When **elementId** is assigned, it creates this object, a Connection object to the element that hosts the value.

  Typically you use this to modify the properties of the Connection object that was defined.
- **valueWhenInvalid** (CalcItem object) - If the element's value is invalid (cannot be resolved to a number), this CalcItem is evaluated.

  Typically you assign a constant of 0 using CalcItems.Number or NaN using CalcItems.NaN (that is actually the default). However, it is valid to use any CalcItem.

  If you use CalcItems.NaN, it also stops the calculation.

  It defaults to CalcItems.NaN.

There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

- **valueWhenNull** (CalcItem object) - If the element's value is unassigned (empty textbox for example), it evaluates this CalcItem.

  Typically you assign a constant of 0 using CalcItems.Number (that is actually the default) or `null` using CalcItems.Null. However, it's valid to use any CalcItem.

  It defaults to CalcItems.Number.

  There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

- **datatype** (string) – Defines a TypeManager that will parse the value from the textbox. This takes a string name of the TypeManager, including short hand names as defined in "Using TypeManagers in datatype properties".

  If rarely needs to be assigned because the Connection object can usually determine the TypeManager.

  Avoid using TypeManagers that are not intended to handle numbers.

- **typeManager** (TypeManager object) – Defines a TypeManager that will parse the value from the textbox. This value can be set by assigning the **datatype** property too. If you use **datatype**, you can use this to set properties on the TypeManager object that was created.

  If rarely needs to be assigned because the Connection object can usually determine the TypeManager.

  Avoid using TypeManagers that are not intended to handle numbers.

- **enabled** (boolean) – When `false`, this does not contribute to the calculation. It defaults to `true`.

- **operator** (string) – When this CalcItems.Element object is a child of a **CalcItems.Group.items** array, this is the math operator used between the previous CalcItem in the array and this one. Its values are all strings: "`+`", "`-`", "`*`", "`/`". It defaults to "+".

- **stopProcessing** (boolean) – When in the CalcItems.Group.**items** property and this CalcItem returns `NaN` or `null`, this property determines if the calculation is abandoned or continued with a replacement to the value from the CalcItem.

  When this property is `false`, the CalcItems.Group will replace the value of the CalcItem only. When `true`, the CalcItems.Group will replace the entire calculation with another value. It defaults to `false`.

CalcItems.Number

Alias names (case sensitive):	none
Inherits from:	**CalcItems.Base**
Source file:	**\jTAC\CalcItems\Number.js**

Holds a number that is part of the calculation.

Set up

Assign the number to the **number** property. If left unassigned, that property defaults to 0.

```
var ci = jTAC.create("CalcItems.Number");
ci.value = 10;
var ci2 = jTAC.create("CalcItems.Number"), {number: 20});
```

You can often avoid writing code to instantiate this class. Instead, you can use the number in properties that accept a CalcItem, such as the **Calculator.calcItem** property and in the **CalcItems.Group.items** array.

```
var gci = jTAC.create("CalcItems.Group");
gci.items = ["TextBox1", "TextBox2", 10, 20];
```

Using the expressions property

When used in an **expression** property, specify a number, just like you do in JavaScript.

```
{expression: "'TextBox1' + 'TextBox2' + 10.5" }
```

CalcItems.Number Properties

- **number** (number) - The numeric value to include in the calculation. It defaults to 0.
- **enabled** (boolean) – When `false`, this does not contribute to the calculation. It defaults to `true`.
- **operator** (string) – When this CalcItems.Number object is a child of a **CalcItems.Group.items** array, this is the math operator used between the previous CalcItem in the array and this one. Its values are all strings: "`+`", "`-`", "`*`", "`/`". It defaults to "+".

CalcItems.Group

Alias names (case sensitive):	none
Inherits from:	**CalcItems.Base**
Source file:	**\jTAC\CalcItems\Group.js**

Holds an array of CalcItem objects. Calculates them together as a group, like when you enclose elements in parenthesis in an expression. Each CalcItem's **operator** property is used to determine whether to add, subtract, multiply or divide it from the previous CalcItem in the array.

Handles 80 bit math rounding errors usually seen with JavaScript.

Set up

Add CalcItems to the **items** property. This is an array. You can assign an array or if you want to add, call the `addItem()` method on CalcItems.Group.

Elements of the array can use alternative formats that will be converted to the appropriate CalcItem class.

- As a number, it creates a CalcItems.Number with the number as the value.
- As a string, it creates a CalcItems.Element with the string as the ID of the element on the page.
- As null, it creates CalcItems.Null.
- As NaN, it creates CalcItems.NaN.
- As a JavaScript object, with the **jtacClass** property assigned to the name of the CalcItem class to use. The remaining properties must be valid properties on that CalcItem class, with values compatible with its properties. In many cases, a CalcItem object has its own calcItem property which also accepts all of these formats. When using unobtrusive setup, use JSON.

```
<input type="hidden" id="Calculator1" name="Calculator1"
   data-jtac-calculator="{'calcItem': {'jtacClass': 'CalcItems.Group', 'items':
      [{'jtacClass': 'CalcItems.Element', 'elementId': 'TextBox1'},
       {'jtacClass': 'CalcItems.Element', 'elementId': 'TextBox2'} ]}" />
```

- As an Array containing any of the types described here. This creates a child CalcItems.Group that sums the values it is given. Usually you give it the IDs of textboxes and numbers. Each child CalcItem uses a "+" operator. You can change that by adding a string item with "+", "-", "*", "/" preceding the CalcItem that needs is operator changed.

```
$("#Calculator1").calculator({calcItem : ["TextBox1", "TextBox2"]);
$("#Calculator2").calculator({calcItem : ["TextBox1", "-", "TextBox2"]);
```

Using the expressions property

When used in an **expression** property, specify a list of 1 or more child calculation expressions representing other CalcItems optionally enclosed by parenthesis. Each is separated by one of the math operators: + - * /

```
(calculation expression1)
(calculation expression1 + calculation expression2 - calculation expression3)
calculation expression1 + calculation expression2 - calculation expression3
```

CalcItems.Group properties

- **items** (Array of CalcItem objects) - A list of CalcItems to perform the calculations upon. Since their **operator** properties default to "+", ensure they are set correctly if you are not adding values.

  If any of these items returns NaN or null from its calculation, the **valueWhenInvalid** or **valueWhenNull** properties determine how to handle those cases.

  There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

- **valueWhenInvalid** (CalcItem object) - If a child CalcItem returns `NaN`, this provides an alternative CalcItem object.

  Its calculation either replaces the CalcItem that failed or replaces the entire Group's calculation, depending on the **stopProcessing** defined on the CalcItem object in **valueWhenInvalid**.

  If the value returned by the **valueWhenInvalid** CalcItem is either `NaN` or `null`, that value is returned by CalcItems.Group. (Only a number value can continue processing the remaining items.)

  Typically you assign a constant of 0 using CalcItems.Number or `NaN` using CalcItems.NaN (that is actually the default).

  It defaults to CalcItems.NaN.

  There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

- **valueWhenNull** (CalcItem object) - If a child CalcItem returns `null`, this provides an alternative CalcItem object.

  Its calculation either replaces the CalcItem that failed or replaces the entire Group's calculation, depending on the **stopProcessing** defined on the CalcItem object in **valueWhenNull**.

  If the value returned by the **valueWhenInvalid** CalcItem is either `NaN` or `null`, that value is returned by CalcItems.Group. (Only a number value can continue processing the remaining items.)

  Typically you assign a constant of 0 using CalcItems.Number or `null` using CalcItems.Null (that is actually the default).

  There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

- **valueWhenEmpty** (CalcItem object) - If the **items** is empty or none of its children can evaluate, it evaluates this CalcItem.

  Typically you assign a constant of 0 using CalcItems.Number or `NaN` using CalcItems.NaN. However, it is valid to use any CalcItem.

  It defaults to CalcItems.Number with a value of 0.

  There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

- **enabled** (boolean) – When `false`, this does not contribute to the calculation. It defaults to `true`.

- **operator** (string) – When this CalcItems.Group object is a child of another **CalcItems.Group.items** array, this is the math operator used between the previous CalcItem in the array and this one. Its values are all strings: "`+`", "`-`", "`*`", "`/`". It defaults to "+".

- **stopProcessing** (boolean) – When this CalcItems.Group object is a child of another **CalcItems.Group.items** array and this CalcItem returns `NaN` or `null`, this property determines if the calculation is abandoned or continued with a replacement to the value from the CalcItem.

  When this property is `false`, the CalcItems.Group will replace the value of the CalcItem only. When `true`, the CalcItems.Group will replace the entire calculation with another value. It defaults to `false`.

CalcItems.Conditional

Alias names (case sensitive):	none
Inherits from:	**CalcItems.Base**
Source file:	**\jTAC\CalcItems\Conditional.js**

Allows IF THEN logic to select between two branches of a calculation. It uses a Condition object, which evaluates something and returns "success", "failed", and "cannot evaluate". You supply CalcItem objects to run on the success and failed case. You also identify what to do in the "cannot evaluate" case.

Set up

Create a Condition object and assign it to the **condition** property. See "The Condition classes".

```
var cond = jTAC.create("Conditions.Required"), {elementId: "TextBox1"});
var ci = jTAC.create("CalcItems.Conditional");
ci.condition = cond;
```

Define the CalcItem object used when the Condition evaluates as "success" and assign it to the **success** property.

Define the CalcItem object used when the Condition evaluates as "failed" and assign it to the **failed** property. If you want this to return 0, assign it to a CalcItems.Number with the value of 0. If you want it to report an error, assign it to a CalcItems.NaN object. There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

```
ci.success = ["TextBox1", "TextBox2"]; // short hand that converts to a
        // CalcItems.Group with two CalcItems.Elements
ci.failed = 0;   // short hand that converts to CalcItems.Number(0)
```

If there is a possibility of a "cannot evaluate" result from the Condition, set the **cannotEvalMode** property to determine the action to take.

Using the expressions property

When used in an **expression** property, specify a function named "Condition" as shown here.

```
Condition(condition, success[, failed][, more])
```

The parameters are:

- *condition* (Conditions.Base subclass) – Define the Condition object using JSON where the **jtacClass** property identifies the name of the class and the remaining properties are values to assign to properties on the object created.

  ```
  Condition({"jtacClass": "Required", "elementId": "TextBox1"}, remaining parameters)
  ```

- *success* (CalcItem object) - The calculation expression to run when the Condition returns "success". If this expression is not used, assign the value `null`.

  ```
  Condition({condition}, ("TextBox1" + "TextBox2"), remaining parameters)
  ```

- *failed* (CalcItem object) (optional) - The calculation expression to run when the Condition returns "failed". If this expression is not used, assign the value `null`. It can also be omitted if the last two parameters are not needed.

  ```
  Condition({condition}, ("TextBox1" + "TextBox2"), 0, remaining parameters)
  ```

- *more* (optional)- One of two values to provide additional property values:
  - One of these strings to assign to **cannotEvalMode**: "error", "zero", "success", "failed"

    ```
    Condition({condition}, ("TextBox1" + "TextBox2"), 0, "success")
    ```

  - JSON with additional property values (optional). A JSON with property names found on the CalcItems.Conditional class, and the values to assign to those properties.

    ```
    Condition({condition}, ("TextBox1" + "TextBox2"), 0, {'cannotEvalMode' : 'success'})
    ```

CalcItems.Conditional Properties

- **condition** (Condition class) - The Condition object. Required.

  This property can be set in two ways:

  - As an instance of a Condition class. (Not supported by unobtrusive setup.)

```
var cond = jTAC.create("Conditions.Required"), {elementId: "TextBox1"});
var ci = jTAC.create("CalcItems.Conditional");
ci.condition = cond;
```

  - As a JavaScript object, with the **jtacClass** property assigned to the name of the Condition class to use. The remaining properties must be valid properties on that Condition class, with values compatible with its properties. When using unobtrusive setup, use JSON.

```
<input type="hidden" id="Calculator1" name="Calculator1"
    data-jtac-calculator="{'calcItem': {'jtacClass': 'CalcItems.Conditional',
        'condition': {'jtacClass': 'Required', 'elementId': 'TextBox1' },
        'success': value, 'failed': value />
```

- **success** (CalcItem object) - The CalcItem to evaluate when the Condition evaluates as "success".

  It's common to assign a CalcItems.Group to calculate a list of values, but it can host any type, including another CalcItems.Conditional.

  There are several ways to assign this property:

  - As an instance of a CalcItem class. (Not supported by unobtrusive setup.)

```
var ci1 = jTAC.create("CalcItems.Element"), {elementId: "TextBox1"});
var ci2 = jTAC.create("CalcItems.Element"), {elementId: "TextBox2"});
var cond = jTAC.create("Conditions.Required"), {elementId: "TextBox1"});
var ci = jTAC.create("CalcItems.Conditional"), {condition: cond});
ci.success.push(ci1);
ci.success.push(ci2);
$("#Calculator1").Calculator({calcItem: ci});
```

  - As a JavaScript object, with the **jtacClass** property assigned to the name of the CalcItem class to use. The remaining properties must be valid properties on that CalcItem class, with values compatible with its properties. In many cases, a CalcItem object has its own calcItem property which also accepts all of these formats. When using unobtrusive setup, use JSON.

```
<input type="hidden" id="Calculator1" name="Calculator1"
    data-jtac-calculator="{'calcItem': {'jtacClass': 'CalcItems.Conditional',
        'condition': {'jtacClass': 'Required', 'elementId': 'TextBox1' },
        'success': [{'jtacClass': 'CalcItems.Element', 'elementId': 'TextBox1' },
                    {'jtacClass': 'CalcItems.Element', 'elementId': 'TextBox2' } ]}" />
```

  - As an Array containing any of the types described here. This creates a CalcItems.Group that sums the values it is given. Usually you give it the IDs of textboxes. Each child CalcItem uses a "+" operator. You can change that by adding a string item with "+", "-", "*", "/" preceding the CalcItem that needs is operator changed.

```
<input type="hidden" id="Calculator1" name="Calculator1"
    data-jtac-calculator="{'calcItem': {'jtacClass': 'CalcItems.Conditional',
        'condition': {'jtacClass': 'Required', 'elementId': 'TextBox1' },
        'success': ['TextBox1', 'TextBox2']}, 'failed': value" />
```

  - As a number, it creates a CalcItems.Number with the number as the value.
  - As a string, it creates a CalcItems.Element with the string as the ID of the element on the page, including textboxes and hidden fields used by other Calculators.
  - As `null`, it creates a CalcItems.Null.
  - As `NaN`, it creates a CalcItems.NaN

  It defaults to CalcItems.Group.

- **failed** (CalcItems object) - The CalcItem to evaluate when the Condition evaluates as "failed".

  It's common to assign a CalcItems.Group to calculate a list of values, but it can host any type, including another CalcItems.Conditional.

  If you want the calculation to stop, assign a CalcItems.NaN here.

  See the **success** property documentation for several ways to set up this property.

  It defaults to CalcItems.Group.

- **cannotEvalMode** (string) - Determines how the calculation works when the Condition evaluates as "cannot evaluate".

  Some Conditions cannot evaluate data until certain values exist. For example, the Conditions.Range cannot evaluate until the text in the textbox is formatted to match what is demanded by its TypeManager.

  Legal values are:

  - "`error`" - Stop the calculation. It's an error. This is the default.
  - "`zero`" - Return 0.
  - "`success`" - Evaluate the **success** property.
  - "`failed`" - Evaluate the **failed** property.

- **enabled** (boolean) – When `false`, this does not contribute to the calculation. It defaults to `true`.

- **operator** (string) – When this CalcItems.Group object is a child of another **CalcItems.Group.items** array, this is the math operator used between the previous CalcItem in the array and this one. Its values are all strings: "`+`", "`-`", "`*`", "`/`". It defaults to "+".

- **stopProcessing** (boolean) – When this CalcItems.Conditional object is a child of a **CalcItems.Group.items** array and this CalcItem returns `NaN` or `null`, this property determines if the calculation is abandoned or continued with a replacement to the value from the CalcItem.

  When this property is `false`, the CalcItems.Group will replace the value of the CalcItem only. When `true`, the CalcItems.Group will replace the entire calculation with another value. It defaults to `false`.

CalcItems.Null

Alias names (case sensitive):	None
Inherits from:	**CalcItems.Base**
Source file:	**\jTAC\CalcItems\Null.js**

Use CalcItems.Null to return the value of `null`.

When a calculation returns the value of `null`, it means there is nothing to calculate, but it's not an error.

This is generally used in a CalcItems.Conditional on its **failed** property and in the **valueWhenNull** property found on many CalcItems.

Set up

CalcItems.Null usually has no properties to be assigned. Just add it.

```
var ci = jTAC.create("CalcItems.Null");
```

Using the expressions property

When used in an **expression** property, specify `null` (case sensitive).

```
null
```

CalcItems.Null Properties

- **enabled** (boolean) – When `false`, this does not contribute to the calculation. It defaults to `true`.

CalcItems.NaN

Alias names (case sensitive):	None
Inherits from:	**CalcItems.Base**
Source file:	**\jTAC\CalcItems\NaN.js**

Use CalcItems.NaN to return the value of `NaN`.

When a calculation returns the value of `NaN`, that means an error occurred. In most cases, this stops the calculation. However, CalcItems.Group and CalcItems.Element can use their **valueWhenInvalid** property to replace `NaN` with another value, such as 0 or the result of another calculation.

This is generally used in a CalcItems.Conditional on its **failed** property and in the **valueWhenInvalid** property found on many CalcItems.

Set up

CalcItems.NaN usually has no properties to be assigned. Just add it.

```
var ci = jTAC.create("CalcItems.NaN");
```

Using the expressions property

When used in an **expression** property, specify `NaN` (case sensitive).

```
NaN
```

CalcItems.NaN Properties

- **enabled** (boolean) – When `false`, this does not contribute to the calculation. It defaults to `true`.

CalcItems.Fix

Alias names (case sensitive):	none
Inherits from:	**CalcItems.BaseFunction**
Source file:	**\jTAC\CalcItems\Fix.js**

Use when another CalcItem may return `null` or `NaN` and you want to replace that value with something else, such as a CalcItem.Number.

*Note: Many CalcItem classes have the same functionality built in. Look for the **valueWhenNull** and **valueWhenInvalid** properties.*

This function normally returns the same value passed in. If the value is `null`, it returns the value determined by the CalcItem object in the **valueWhenNull** property.

If the value is NaN, it returns the value determined by the CalcItem object in the **valueWhenInvalid** property.

The function's **parms** property takes exactly one CalcItem object.

Set up

Create a CalcItem whose value may need fixing and assign it to the **parms** property.

```
var ci1 = jTAC.create("CalcItems.Fix"), {parms: ["TextBox2" "/" "TextBox1"});
```

```
<input type="hidden" id="Calculator1" name="Calculator1"
   data-jtac-calculator="{'calcItem': {'jtacClass': 'CalcItems.Fix',
      'parms': ['TextBox2', '/', 'TextBox1'}" />
```

Using the expressions property

When used in an **expression** property, use a function named "Fix" that takes three parameters, each being a calculation expression.

```
Fix(calculation expression, calc expression when Null, calc expression when NaN)
```

calculation expression – The expression whose value will be evaluated for `null` or `NaN`.

calculation expression when null – The expression used when `null` is determined. If not used, pass `null`.

calculation expression when NaN – The expression used when `NaN` is determined. If not used, omit the parameter.

CalcItems.Fix Properties

- **parms** – Assign a CalcItem object, or one of the formats below, which generates a value that may need rounding. If any of these parms returns `NaN` or `null` from its calculation, the **valueWhenInvalid** or **valueWhenNull** properties determine how to handle those cases.
  - As a number, it creates a CalcItems.Number with the number as the value.
  - As a string, it creates a CalcItems.Element with the string as the ID of the element on the page.
  - As a JavaScript object, with the **jtacClass** property assigned to the name of the CalcItem class to use. The remaining properties must be valid properties on that CalcItem class, with values compatible with its properties. In many cases, a CalcItem object has its own **calcItem** property which also accepts all of these formats. When using unobtrusive setup, use JSON.

```
var ci = jTAC.create("CalcItems.Fix");
ci.parms = {jtacClass: "CalcItems.Group", items: ["TextBox2", "/", "TextBox1"]};
```

```
<input type="hidden" id="Calculator1" name="Calculator1"
   data-jtac-calculator="{'calcItem': {'jtacClass': 'CalcItems.Fix',
      'parms': {'jtacClass': 'CalcItems.Group',
      'items': ['TextBox2', '/', 'TextBox1']}}" />
```

- As an Array containing any of the types described here. This creates a child CalcItems.Group that sums the values it is given. Usually you give it the IDs of textboxes and numbers. Each child CalcItem uses a "+" operator. You can change that by adding a string item with "+", "-", "*", "/" preceding the CalcItem that needs is operator changed.

```
var ci = jTAC.create("CalcItems.Fix");
ci.parms = ["TextBox1", "-", 100];
```

```
<input type="hidden" id="Calculator1" name="Calculator1"
    data-jtac-calculator="{'calcItem': {'jtacClass': 'CalcItems.Fix',
        'parms': ['TextBox2', '/', 'TextBox1'] }" />
```

- **valueWhenInvalid** (CalcItem object) - If a CalcItem in **parms** returns `NaN`, this provides an alternative CalcItem object.

  Typically you assign a constant of 0 using CalcItems.Number or NaN using CalcItems.NaN.

  It defaults to CalcItems.NaN.

  There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

- **valueWhenNull** (CalcItem object) - If a CalcItem in **parms** returns `null`, this provides an alternative CalcItem object.

  Typically you assign a constant of 0 using CalcItems.Number or `null` using CalcItems.Null.

  It defaults to CalcItems.Null.

  There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

CalcItems.Round

Alias names (case sensitive):	none
Inherits from:	**CalcItems.BaseFunction**
Source file:	**\jTAC\CalcItems\Round.js**

Pass in another CalcItem whose value may exceed a certain number of decimal places. It returns the rounded value using rules defined in the **roundMode** and **maxDecimalPlaces** properties. It can also indicate an error (return NaN) if the maximum decimal places are exceeded.

Set up

Create a CalcItem whose value may need rounding and assign it to the **parms** property.

Set the **roundMode** property to indicate how to round. It supports these values:

0 = Point5: round to the next number if .5 or higher; round down otherwise

1 = Currency: round to the nearest even number.

2 = Truncate: drop any decimals after **maxDecimalPlaces**; largest integer less than or equal to a number.

3 = Ceiling: round to the nearest even number.

4 = NextWhole: Like ceiling, but negative numbers are rounded lower instead of higher

Set the **maxDecimalPlaces** property to indicate the maximum decimal places that must be exceeded to round. It rounds back to this number of decimal places. If you are rounding to an integer, set it to 0.

```
var ci1 = jTAC.create("CalcItems.Element"), {elementId: "TextBox1"});
var ci2 = jTAC.create("CalcItems.Number"), {number: 3});
ci2.operator = "/";
var ci3 = jTAC.create("CalcItems.Group"), {items: [ci1, ci2]}); // TextBox1 / 3
var ci4 = jTAC.create("CalcItems.Round"),
   {parms: ci3, roundMode: 0, maxDecimalPlaces: 2} );   // Point5 to 2 decimal places
```

```
<input type="hidden" id="Calculator1" name="Calculator1"
   data-jtac-calculator="{'calcItem': {'jtacClass': 'CalcItems.Round',
      'parms': ['TextBox1', '/', 'TextBox2'],
      'roundMode': 0, 'maxDecimalPlaces': 2}" />
```

Using the expressions property

When used in an **expression** property, use one of these function calls:

- A function named "Round" with several parameters.

  ```
  Round(calculation expression[, roundmode, maxdecimalplaces])
  ```

  calculation expression – The expression whose value will be rounded.

  roundmode (integer) (optional) – Sets the **roundmode** property. When not assigned, it uses 0 (“Point5”).

  maxdecimalplaces (integer) (optional) - Sets the **maxDecimalPlaces** property. When 0, round as an integer. When not assigned, it uses 3.

  ```
  Round("TextBox1" + "TextBox2")

  Round("TextBox1" + "TextBox2", 2)

  Round("TextBox1" + "TextBox2", 2, 3)
  ```

- A function named "Math.round" to mimic the JavaScript function for Point5 rounding to the nearest integer. It takes one parameter, the calculation expression.

  ```
  Math.round(calculation expression)
  ```

- A function named "Math.floor" to mimic the JavaScript function for Truncate rounding to the nearest integer lower than the value provided. It takes one parameter, the calculation expression.

```
Math.floor(calculation expression)
```

- A function named "Math.ceil" to mimic the JavaScript function for Ceiling rounding to the nearest integer higher than the value provided. It takes one parameter, the calculation expression.

```
Math.ceiling(calculation expression)
```

CalcItems.Round Properties

- **parms** – Assign a CalcItem object, or one of the formats below, which generates a value that may need rounding. If any of these parms returns `NaN` or `null` from its calculation, the **valueWhenInvalid** or **valueWhenNull** properties determine how to handle those cases.
  - As a number, it creates a CalcItems.Number with the number as the value.
  - As a string, it creates a CalcItems.Element with the string as the ID of the element on the page.
  - As a JavaScript object, with the **jtacClass** property assigned to the name of the CalcItem class to use. The remaining properties must be valid properties on that CalcItem class, with values compatible with its properties. In many cases, a CalcItem object has its own **calcItem** property which also accepts all of these formats. When using unobtrusive setup, use JSON.

```
var ci = jTAC.create("CalcItems.Round");
ci.parms = {jtacClass: "CalcItems.Group", items: ["TextBox1", "/", 3]};
```

```
<input type="hidden" id="Calculator1" name="Calculator1"
    data-jtac-calculator="{'calcItem': {'jtacClass':'CalcItems.Round',
        'parms': {'jtacClass': 'CalcItems.Group', 'items': ['TextBox1', '/', 3]},
        'roundMode': 0, 'maxDecimalPlaces': 2}" />
```

  - As an Array containing any of the types described here. This creates a child CalcItems.Group that sums the values it is given. Usually you give it the IDs of textboxes and numbers. Each child CalcItem uses a "+" operator. You can change that by adding a string item with "+", "-", "*", "/" preceding the CalcItem that needs is operator changed.

```
var ci = jTAC.create("CalcItems.Round");
ci.parms = ["TextBox1", "/", 3];
```

```
<input type="hidden" id="Calculator1" name="Calculator1"
    data-jtac-calculator="{'calcItem': {'jtacClass': 'CalcItems.Round',
        'parms': ['TextBox1', '/', 3], 'roundMode': 0, 'maxDecimalPlaces': 2}" />
```

- **roundMode** (integer) - Determines the way to round.
  - 0 = Point5: round to the next number if .5 or higher; round down otherwise. It defaults to 0.
  - 1 = Currency: round to the nearest even number.
  - 2 = Truncate: drop any decimals after **maxDecimalPlaces**; largest integer less than or equal to a number.
  - 3 = Ceiling: round to the nearest even number.
  - 4 = NextWhole: Like ceiling, but negative numbers are rounded lower instead of higher
  - `null` = Report an error (return NaN).
- **maxDecimalPlaces** (integer) - Determines the maximum number of decimal digits that are legal. If there are more, it is either a validation error or rounded, depending on **roundMode**. It defaults to 3.
- **valueWhenInvalid** (CalcItem object) - If a CalcItem in **parms** returns `NaN`, this provides an alternative CalcItem object.

  Typically you assign a constant of 0 using CalcItems.Number or `NaN` using CalcItems.NaN (that is actually the default).

  It defaults to CalcItems.NaN.

There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

- **valueWhenNull** (CalcItem object) - If a CalcItem in **parms** returns `null`, this provides an alternative CalcItem object.

  Typically you assign a constant of 0 using CalcItems.Number or `null` using CalcItems.Null (that is actually the default).

  It defaults to CalcItems.Null.

  There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

CalcItems.Abs

Alias names (case sensitive):	none
Inherits from:	**CalcItems.BaseFunction**
Source file:	**\jTAC\CalcItems\Abs.js**

Returns the absolute value of the CalcItem passed to it.

Set up

Create a CalcItem whose value may need rounding and assign it to the **parms** property.

```
var ci1 = jTAC.create("CalcItems.Element"), {elementId: "TextBox1"});
var ci2 = jTAC.create("CalcItems.Abs"), {parms: ci1});
```

```
<input type="hidden" id="Calculator1" name="Calculator1"
   data-jtac-calculator="{'calcItem': {'jtacClass': 'CalcItems.Abs', 'parms': 'TextBox1'}" />
```

Using the expressions property

When used in an **expression** property, use a function named "Abs" or “Math.abs” that takes one parameter, a calculation expression.

```
Abs(calculation expression)

Math.abs(calculation expression)
```

CalcItems.Abs Properties

- **parms** – Assign a CalcItem object, or one of the formats below, which generates a value that may need rounding. If any of these parms returns `NaN` or `null` from its calculation, the **valueWhenInvalid** or **valueWhenNull** properties determine how to handle those cases.
  - As a number, it creates a CalcItems.Number with the number as the value.
  - As a string, it creates a CalcItems.Element with the string as the ID of the element on the page.
  - As a JavaScript object, with the **jtacClass** property assigned to the name of the CalcItem class to use. The remaining properties must be valid properties on that CalcItem class, with values compatible with its properties. In many cases, a CalcItem object has its own **calcItem** property which also accepts all of these formats. When using unobtrusive setup, use JSON.

```
var ci = jTAC.create("CalcItems.Abs");
ci.parms = {"jtacClass": "CalcItems.Group", "items": ["TextBox1", "-", 100]};
```

```
<input type="hidden" id="Calculator1" name="Calculator1"
   data-jtac-calculator="{'calcItem': {'jtacClass': 'CalcItems.Abs',
      'parms': {'jtacClass': 'CalcItems.Group', 'items': ['TextBox1', '-', 100]}}" />
```

  - As an Array containing any of the types described here. This creates a child CalcItems.Group that sums the values it is given. Usually you give it the IDs of textboxes and numbers. Each child CalcItem uses a "+" operator. You can change that by adding a string item with "+", "-", "*", "/" preceding the CalcItem that needs is operator changed.

```
var ci = jTAC.create("CalcItems.Abs");
ci.parms = ["TextBox1", "-", 100];
```

```
<input type="hidden" id="Calculator1" name="Calculator1"
   data-jtac-calculator="{'calcItem': {'jtacClass': 'CalcItems.Abs',
      'parms': ['TextBox1', '-', 100] }" />
```

- **valueWhenInvalid** (CalcItem object) - If a CalcItem in **parms** returns `NaN`, this provides an alternative CalcItem object.

  Typically you assign a constant of 0 using CalcItems.Number or `NaN` using CalcItems.NaN.

  It defaults to CalcItems.NaN.

There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

- **valueWhenNull** (CalcItem object) - If a CalcItem in **parms** returns `null`, this provides an alternative CalcItem object.

  Typically you assign a constant of 0 using CalcItems.Number or `null` using CalcItems.Null.

  It defaults to CalcItems.Null.

  There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

CalcItems.Min

Alias names (case sensitive):	none
Inherits from:	**CalcItems.BaseFunction**
Source file:	**\jTAC\CalcItems\Min.js**

Returns the lowest valued number from the list of CalcItems passed in.

Set up

Create a list of CalcItem objects whose value needs to be evaluated. Add them to an array that is assigned to the **parms** property.

```
var ci1 = jTAC.create("CalcItems.Element"), {elementId: "TextBox1"});
var ci2 = jTAC.create("CalcItems.Element"), {elementId: "TextBox2"});
var ci3 = jTAC.create("CalcItems.Min"), {parms: [ci1, ci2]});
```

```
<input type="hidden" id="Calculator1" name="Calculator1"
   data-jtac-calculator="{'calcItem': {'jtacClass': 'CalcItems.Min',
      'parms': ['TextBox1', 'TextBox2']}" />
```

Using the expressions property

When used in an **expression** property, use a function named "Min" or “Math.min” that takes a list of parameters, each as a calculation expression.

```
Min(calculation expression, calculation expression, calculation expression)

Math.min(calculation expression, calculation expression, calculation expression)
```

CalcItems.Min Properties

- **parms** – Assign an array of CalcItem objects, or one of the formats below, whose values will be evaluated. If any of these parms returns `NaN` or `null` from its calculation, the **valueWhenInvalid** or **valueWhenNull** properties determine how to handle those cases.
  - As a number, it creates a CalcItems.Number with the number as the value.
  - As a string, it creates a CalcItems.Element with the string as the ID of the element on the page.
  - As a JavaScript object, with the **jtacClass** property assigned to the name of the CalcItem class to use. The remaining properties must be valid properties on that CalcItem class, with values compatible with its properties. In many cases, a CalcItem object has its own **calcItem** property which also accepts all of these formats. When using unobtrusive setup, use JSON.

```
var ci = jTAC.create("CalcItems.Min"));
ci.parms = {jtacClass: "CalcItems.Group", items: ["TextBox1", "TextBox2"]};
```

```
<input type="hidden" id="Calculator1" name="Calculator1"
    data-jtac-calculator="{'calcItem': {'jtacClass':'CalcItems.Min',
        'parms': {'jtacClass': 'CalcItems.Group', 'items': ['TextBox1', 'TextBox2']}}" />
```

  - As an Array containing any of the types described here. This creates a child CalcItems.Group that sums the values it is given. Usually you give it the IDs of textboxes and numbers. Each child CalcItem uses a "+" operator. You can change that by adding a string item with "+", "-", "*", "/" preceding the CalcItem that needs is operator changed.

```
var ci = jTAC.create("CalcItems.Min");
ci.parms = ["TextBox1", "TextBox2"];
```

```
<input type="hidden" id="Calculator1" name="Calculator1"
    data-jtac-calculator="{'calcItem': {'jtacClass': 'CalcItems.Min',
        'parms': ['TextBox1', 'TextBox2'] }" />
```

- **valueWhenInvalid** (CalcItem object) - If a CalcItem in **parms** returns `NaN`, this provides an alternative CalcItem object.

  Typically you assign a constant of 0 using CalcItems.Number or `NaN` using CalcItems.NaN.

  It defaults to CalcItems.NaN.

  There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

- **valueWhenNull** (CalcItem object) - If a CalcItem in **parms** returns `null`, this provides an alternative CalcItem object.

  Typically you assign a constant of 0 using CalcItems.Number or `null` using CalcItems.Null.

  It defaults to CalcItems.Null.

  There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

CalcItems.Max

Alias names (case sensitive):	none
Inherits from:	**CalcItems.BaseFunction**
Source file:	**\jTAC\CalcItems\Max.js**

Returns the highest valued number from the list of CalcItems passed in.

Set up

Create a list of CalcItem objects whose values need to be evaluated. Add them to an array that is assigned to the **parms** property.

```
var ci1 = jTAC.create("CalcItems.Element"), {elementId: "TextBox1"});
var ci2 = jTAC.create("CalcItems.Element"), {elementId: "TextBox2"});
var ci3 = jTAC.create("CalcItems.Max"), {parms: [ci1, ci2]});
```

```
<input type="hidden" id="Calculator1" name="Calculator1"
   data-jtac-calculator="{'calcItem': {'jtacClass': 'CalcItems.Max',
      'parms': ['TextBox1', 'TextBox2']}" />
```

Using the expressions property

When used in an **expression** property, use a function named "Max" or “Math.Max” that takes a list of parameters, each as a calculation expression.

```
Max(calculation expression, calculation expression, calculation expression)

Math.Max(calculation expression, calculation expression, calculation expression)
```

CalcItems.Max Properties

- **parms** – Assign an array of CalcItem objects, or one of the formats below, whose values will be evaluated. If any of these parms returns `NaN` or `null` from its calculation, the **valueWhenInvalid** or **valueWhenNull** properties determine how to handle those cases.
  - As a number, it creates a CalcItems.Number with the number as the value.
  - As a string, it creates a CalcItems.Element with the string as the ID of the element on the page.
  - As a JavaScript object, with the **jtacClass** property assigned to the name of the CalcItem class to use. The remaining properties must be valid properties on that CalcItem class, with values compatible with its properties. In many cases, a CalcItem object has its own **calcItem** property which also accepts all of these formats. When using unobtrusive setup, use JSON.

```
var ci = jTAC.create("CalcItems.Max");
ci.parms = {jtacClass: "CalcItems.Group", items: ["TextBox1", "TextBox2"]};
```

```
<input type="hidden" id="Calculator1" name="Calculator1"
    data-jtac-calculator="{'calcItem': {'jtacClass': 'CalcItems.Max',
        'parms': {'jtacClass': 'CalcItems.Group', 'items': ['TextBox1', 'TextBox2']}}" />
```

  - As an Array containing any of the types described here. This creates a child CalcItems.Group that sums the values it is given. Usually you give it the IDs of textboxes and numbers. Each child CalcItem uses a "+" operator. You can change that by adding a string item with "+", "-", "*", "/" preceding the CalcItem that needs is operator changed.

```
var ci = jTAC.create("CalcItems.Max");
ci.parms = ["TextBox1", "TextBox2"];
```

```
<input type="hidden" id="Calculator1" name="Calculator1"
    data-jtac-calculator="{'calcItem': {'jtacClass': 'CalcItems.Max',
        'parms': ['TextBox1', 'TextBox2'] }" />
```

- **valueWhenInvalid** (CalcItem object) - If a CalcItem in **parms** returns `NaN`, this provides an alternative CalcItem object.

  Typically you assign a constant of 0 using CalcItems.Number or `NaN` using CalcItems.NaN.

  It defaults to CalcItems.NaN.

  There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

- **valueWhenNull** (CalcItem object) - If a CalcItem in **parms** returns `null`, this provides an alternative CalcItem object.

  Typically you assign a constant of 0 using CalcItems.Number or `null` using CalcItems.Null.

  It defaults to CalcItems.Null.

  There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

CalcItems.Avg

Alias names (case sensitive):	none
Inherits from:	**CalcItems.BaseFunction**
Source file:	**\jTAC\CalcItems\Avg.js**

Returns the average value from the list of CalcItems passed in.

Set up

Create a list of CalcItem objects whose values need to be evaluated. Add them to an array that is assigned to the **parms** property.

```
var ci1 = jTAC.create("CalcItems.Element"), {elementId: "TextBox1"});
var ci2 = jTAC.create("CalcItems.Element"), {elementId: "TextBox2"});
var ci3 = jTAC.create("CalcItems.Avg"), {parms: [ci1, ci2]})
```

```
<input type="hidden" id="Calculator1" name="Calculator1"
   data-jtac-calculator="{'calcItem': {'jtacClass': 'CalcItems.Avg',
      'parms': ['TextBox1', 'TextBox2']}" />
```

Using the expressions property

When used in an **expression** property, use a function named "Avg" that takes a list of parameters, each as a calculation expression.

```
Avg(calculation expression, calculation expression, calculation expression)
```

CalcItems.Avg Properties

- **parms** – Assign an array of CalcItem objects, or one of the formats below, whose values will be evaluated. If any of these parms returns `NaN` or `null` from its calculation, the **valueWhenInvalid** or **valueWhenNull** properties determine how to handle those cases.
  - As a number, it creates a CalcItems.Number with the number as the value.
  - As a string, it creates a CalcItems.Element with the string as the ID of the element on the page.
  - As a JavaScript object, with the **jtacClass** property assigned to the name of the CalcItem class to use. The remaining properties must be valid properties on that CalcItem class, with values compatible with its properties. In many cases, a CalcItem object has its own **calcItem** property which also accepts all of these formats. When using unobtrusive setup, use JSON.

```
var ci = jTAC.create("CalcItems.Avg");
ci.parms = {"jtacClass": "CalcItems.Group", "items": ["TextBox1", "TextBox2"]};
```

```
<input type="hidden" id="Calculator1" name="Calculator1"
    data-jtac-calculator="{'calcItem': {'jtacClass':'CalcItems.Avg',
        'parms': {'jtacClass': 'CalcItems.Group', 'items': ['TextBox1', 'TextBox2']}}" />
```

  - As an Array containing any of the types described here. This creates a child CalcItems.Group that sums the values it is given. Usually you give it the IDs of textboxes and numbers. Each child CalcItem uses a "+" operator. You can change that by adding a string item with "+", "-", "*", "/" preceding the CalcItem that needs is operator changed.

```
var ci = jTAC.create("CalcItems.Avg"));
ci.parms = ["TextBox1", "TextBox2"];
```

```
<input type="hidden" id="Calculator1" name="Calculator1"
    data-jtac-calculator="{'calcItem': {'jtacClass':'CalcItems.Avg',
        'parms': ['TextBox1', 'TextBox2'] }" />
```

- **valueWhenInvalid** (CalcItem object) - If a CalcItem in **parms** returns `NaN`, this provides an alternative CalcItem object.

  Typically you assign a constant of 0 using CalcItems.Number or `NaN` using CalcItems.NaN.

  It defaults to CalcItems.NaN.

  There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

- **valueWhenNull** (CalcItem object) - If a CalcItem in **parms** returns `null`, this provides an alternative CalcItem object.

  Typically you assign a constant of 0 using CalcItems.Number or `null` using CalcItems.Null.

  It defaults to CalcItems.Null.

  There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

CalcItems.UserFunction

Alias names (case sensitive):	none
Inherits from:	**CalcItems.BaseFunction**
Source file:	**\jTAC\CalcItems\UserFunction.js**

Calls your own function in its `evaluate()` method allowing you to add custom code into the calculation. You write a function and this CalcItem will call that function.

Set up

Create a function that takes one parameter and must return a number or `NaN`. The parameter will be an array of numbers, determined by evaluating the CalcItems in the **parms** property.

Your function can use or ignore the values passed in.

This example calculates the sum of the values in parms:

```
function TotalAll(parms) {
   var total = 0;
   for (var i = 0; i < parms.length; i++)
      total = total + parms[i];
   return total;
}
```

When creating the CalcItems.UserFunction object, assign either the name of your function or a reference to it in the **func** parameter.

Create a list of CalcItem objects whose value needs to be evaluated. Add them to an array that is assigned to the **parms** property.

```
var ci1 = jTAC.create("CalcItems.Element"), {elementId: "TextBox1"});
var ci2 = jTAC.create("CalcItems.Element"), {elementId: "TextBox2"});
var ci3 = jTAC.create("CalcItems.UserFunction");
ci3.func = TotalAll;
ci3.parms = [ci1, ci2];
```

```
<input type="hidden" id="Calculator1" name="Calculator1"
   data-jtac-calculator="{'calcItem': {'jtacClass':'CalcItems.UserFunction',
      'func': 'TotalAll',
      'parms': ['TextBox1', 'TextBox2']}" />
```

Using the expressions property

When used in an **expression** property, use a function named "Function" with its first parameter as the name of the function, and the remaining parameters as each calculation expression that forms its **parms** array.

```
Function(functionname, calculation expression, calculation expression, calculation expression)
```

For example:

```
Function("TotalAll", "TextBox1", "TextBox2", "TextBox3")
```

CalcItems.UserFunction Properties

- **func** (function) - Reference to a function to call. When using unobtrusive setup, define the function globally (on the window object) and specify its name as a string on this property.
- **parms** – Assign an array of CalcItem objects, or one of the formats below, whose values will be passed to your function. If any of these parms returns `NaN` or `null` from its calculation, the **valueWhenInvalid** or **valueWhenNull** properties determine how to handle those cases.
  - As a number, it creates a CalcItems.Number with the number as the value.
  - As a string, it creates a CalcItems.Element with the string as the ID of the element on the page.
  - As a JavaScript object, with the **jtacClass** property assigned to the name of the CalcItem class to use. The remaining properties must be valid properties on that CalcItem class, with values compatible with its properties. In many cases, a CalcItem object has its own **calcItem** property which also accepts all of these formats. When using unobtrusive setup, use JSON.

```
var ci = jTAC.create("CalcItems.UserFunction");
ci.parms = {"jtacClass": "CalcItems.Group", "items": ["TextBox1", "TextBox2"]};
```

```
<input type="hidden" id="Calculator1" name="Calculator1"
    data-jtac-calculator="{'calcItem': {'jtacClass':'CalcItems.UserFunction',
        'func': 'TotalAll',
        'parms': {'jtacClass': 'CalcItems.Group', 'items': ['TextBox1', 'TextBox2']}}" />
```

  - As an Array containing any of the types described here. This creates a child CalcItems.Group that sums the values it is given. Usually you give it the IDs of textboxes and numbers. Each child CalcItem uses a "+" operator. You can change that by adding a string item with "+", "-", "*", "/" preceding the CalcItem that needs is operator changed.

```
var ci = jTAC.create("CalcItems.UserFunction");
ci.func = TotalAll;
ci.parms = ["TextBox1", "TextBox2"];
```

```
<input type="hidden" id="Calculator1" name="Calculator1"
    data-jtac-calculator="{'calcItem': {'jtacClass':'CalcItems.UserFunction',
        'func': 'TotalAll', 'parms': ['TextBox1', 'TextBox2'] }" />
```

- **valueWhenInvalid** (CalcItem object) - If a CalcItem in **parms** returns `NaN`, this provides an alternative CalcItem object.

  Typically you assign a constant of 0 using CalcItems.Number or `NaN` using CalcItems.NaN.

  It defaults to CalcItems.NaN.

  There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

- **valueWhenNull** (CalcItem object) - If a CalcItem in **parms** returns `null`, this provides an alternative CalcItem object.

  Typically you assign a constant of 0 using CalcItems.Number or `null` using CalcItems.Null.

  It defaults to CalcItems.Null.

  There are several shortcuts to define a CalcItem object that work here. See "Assigning properties that accept a CalcItem object".

CalcItems.Base

Inherits from:	**jTAC's universal base class, jTACBaseClass**
Source file:	**\jTAC\CalcItems\Base.js**

Abstract base class for CalcItem objects.

Subclassing CalcItems.Base methods

Getting started:

- All CalcItems inherit from `jTACClassBase` in **\jTAC\jTAC.js**. It has some useful utilities for your class development. Learn more in the **\jTAC\jTAC.js** file.
- See "Adding your own classes to jTAC" to use best practices in creating your subclasses' structure.
- For an example, see CalcItems.Number in **\jTAC\CalcItems\Number.js**.

This topic will identify methods that you are likely to override or call in CalcItems.Base. It will not describe parameters. Instead, please use the class definition in **\jTAC\CalcItems\Base.js**.

	Element Name	When
✓	`canEvaluate()`	Base class returns the value of the **enabled** property. Override if other values may disable your CalcItem.
✓	`evaluate()`	Abstract method. Always override. This is where you do all of the calculation work, returning a number, `null` (for cannot calculate) or `NaN` (for calculation error).
✓	`parse()`	Provides support for the **expression** property specify to your CalcItem. Develop a syntax and use the parser object passed in to construct it.

Key: ✓ always override; ✓ sometimes override; ✓ rarely override

CalcItems.BaseFunction

Inherits from:	**CalcItems.Base**
Source file:	**\jTAC\CalcItems\BaseFunction.js**

Abstract base class for CalcItem objects that are functions, like Abs, Min, and Round.

Subclassing CalcItems.Base methods

Getting started:

- All CalcItems inherit from `jTACClassBase` in **\jTAC\jTAC.js**. It has some useful utilities for your class development. Learn more in the **\jTAC\jTAC.js** file.
- See "Adding your own classes to jTAC" to use best practices in creating your subclasses' structure.
- For an example, see CalcItems.Number in **\jTAC\CalcItems\Avg.js**.

This topic will identify methods that you are likely to override or call in CalcItems.Base. It will not describe parameters. Instead, please use the class definition in **\jTAC\CalcItems\Base.js**.

	Element Name	When
✓	`canEvaluate()`	Base class returns the value of the **enabled** property. Override if other values may disable your CalcItem.
✓	`evaluate()`	Already coded for you. It prepares the parameters and calls the `_func()` method.
✓	`_func()`	Abstract method. This is where you do all of the work. You are passed either an array of numbers or a single number, depending on the `_numParms()` function. Use those numbers to make your calculation. Return a number, `null` (for cannot calculate) or `NaN` (for calculation error).
✓	`_numParms()`	By default, your function can accept unlimited numbers passed in, the way CalcItems.Max uses. If you want to require a single parameter, return 1. If you want a specific number of parameters and no more, return that number.
✓	`parse()`	Already coded for you to handle this pattern: `functionname(calc expression, calcexp2, etc)` To support that format, override `_getParseName()`. If you have other values for parameters, override `parse()`. See **\jTAC\CalcItems\Round.js** for an example.
✓	`_getParseName()`	Return the name of the function used in the calculation expression.
✓	`_getParseParms()`	By default, it expects all function parameters to be calculation expressions. Override if they are not. See how CalcItems.UserFunction allows a single parameter for its function name followed by the usual list of calculation expressions in **\jTAC\CalcItems\UserFunction.js**.
✓	`_convertParseParms()`	By default, the array of values returned by `_getParseParms()` uses this to convert each into CalcItem objects that are assigned to the parms property. See **\jTAC\CalcItems\UserFunction.js** for a variation.

Key: ✓ always override; ✓ sometimes override; ✓ rarely override

jTAC class members

The global object *jTAC* provides a library of tools to define classes and create objects from them. It also hosts a number of static methods utilized by classes throughout this library.

This section documents a few key methods. For all, see **\jTAC\jTAC.js**.

```
function create( name, propertyVals, optional )
```

Creates an instance of a class, first checking for an alias name match. If not found, attempts for a full class name match.

Use `jTAC.createByClassName()` if you want to only match by class name.

Parameters

name (string)

The class name, including namespaces, or an alias name. It is case sensitive.

propertyValues (object or function)

Optional object or function used to initialize the properties of the new instance.

When an object, that object's property names will be used to set values of the same named properties created.

```
jTAC.createByClassName("Conditions.Range", {minimum:10, maximum:100});
```

When a function, it takes these parameters: new object instance, class name. Its return result is not used.

```
jTAC.createByClassName("Conditions.Range", function(obj, fullClassName) {
    // do something to obj
});
```

optional (boolean)

What happens when the name is not defined. When `true`, return `null`. When `false`, throw an exception.

```
function createByClassName( fullClassName, propertyValues, optional )
```

Creates an instance of a class. The class name must already be defined. Does not support aliases. If you have an alias to use, call `jTAC.create()`.

Parameters

fullClassName (string)

The class name, including namespaces. It is case sensitive. Does not support alias names.

propertyValues (object or function)

Optional object or function used to initialize the properties of the new instance.

When an object, that object's property names will be used to set values of the same named properties created.

```
jTAC.createByClassName("Conditions.Range", {minimum:10, maximum:100});
```

When a function, it takes these parameters: new object instance, class name. Its return result is not used.

```
jTAC.createByClassName("Conditions.Range", function(obj, fullClassName) {
    // do something to obj
});
```

optional (boolean)

What happens when the class name is not defined. When `true`, return `null`. When `false`, throw an exception.

```
function define( fullClassName, members, replace )
```

Registers a class. See also "Adding your own classes to jTAC".

Parameters

fullClassName (string)

The class name to register, including namespaces.

members (object)

An object whose properties become members of the prototype for this class. See below for details.

replace (boolean)

If the definition exists, `true` will replace its definition. `false` will throw an exception.

The *member* object defines functions that become methods on the prototype. Do not define any data storage fields here. They belong either in the **config** property (see below) or the **_internal** property, which is an object for you to use freely.

Here are predefined property names and their usage:

- **extend** (string) - The name of parent class from which this inherits. Your class will inherit all elements of the parent's prototype. Omit if there is nothing to extend.
- **constructor** (function) - A function which is run when the `jTAC.create()` method is called to initialize the object.

  The function takes one parameter, *propertyValues*, which is the same object passed to `jTAC.create(name, propertyValues)`.

  The value of `this` is the instance of the object being initialized.

  At that time it is called, the getter and setter functions for all properties defined within **config** have been added to the prototype.

  If your class extends another, call `this.callParent([propertyValues])` to invoke the parent's constructor.
- **config** (object) - An object whose properties are all actual property names to expose on the main object. These values are used a storage and hold the default value. Property names should use "camel case" (lowercase first letter; uppercase where a new word occurs; no underscores).

  Even if you don't declare it, your class will have the **config** property. It always contains properties inherited from the class it extends.

  jTAC will automatically create getter and setter functions on your class for each property name. If you want to customize those functions, define them yourself. The function names should be "get" + [property name] and "set" + [property name] where the first letter of the property name has been converted to uppercase.
- **configrules** (object) - An object that provides rules for any of the properties defined in **config**. While **config** hosts the name and default value of properties, **configrules** hosts objects with these values:
  - **valFnc** (function) - A function that takes one parameter, a value, and returns the value (possibly cleaned up) if it is a legal value, or calls `this._error("description")` if the value is not legal. (`_error()` throws an exception and writes to the browser's console for debugging assistance.)

    A validation function is automatically setup if not defined and the **config** property has a default value whose type is boolean, string, integer, array, or RegExp object. The jTAC object defines numerous functions that you can assign using just their names:

`jTAC.checkAsStr`	`jTAC.checkAsStrOrNull`
`jTAC.checkAsInt`	`jTAC.checkAsIntOrNull`
`jTAC.checkAsNumber`	`jTAC.checkAsNumberOrNull`
`jTAC.checkAsBool`	`jTAC.checkAsBoolOrNull`
`jTAC.checkAsConnection`	`jTAC.checkAsConnectionOrNull`
`jTAC.checkAsTypeManager`	`jTAC.checkAsTypeManagerOrNull`
`jTAC.checkAsCondition`	`jTAC.checkAsConditionOrNull`

jTAC.checkAsCalcItem	jTAC.checkAsCalcItemOrNull
jTAC.checkAsRegExp	jTAC.checkAsRegExpOrNull
jTAC.checkAsArray	
jTAC.checkAsFunction	jTAC.checkAsFunctionOrNull

You can also use a shorthand of just defining the function against the property name, instead of creating a child object that has **valFnc** as a property.

```
configrules: {
    propertyname:jTAC.checkAsStrOrNull,
    propertyname2: {
        valFnc: function(val) { return val == "a"; }
    }
}
```

- **legalVals** (array) - Alternative to **valFnc**. Supply an array of legal values. Values must be primitive types (strings, numbers, Booleans, `null`).

  You can also use a shorthand of just defining the array against the property name, instead of creating a child object that has **valFnc** as a property.

```
configrules: {
    propertyname: ["a", "b", "c"],
    propertyname2: {
        legalVals: [1, 2, 3, null]
    }
}
```

- **autoSet** (boolean) - When `false`, do not automatically create a missing setter function. It defaults to `true`.
- **autoGet** (boolean) - When `false`, do not automatically create a missing getter function. It defaults to `true`.

- **require** (string) - A string or array of strings identifying other classes that are required. This does not need to include any direct ancestor class. It can supply a specific class it needs and not worry about supplying all ancestor classes too.
- **abstract** (boolean) - When `true`, this class is abstract and cannot be created by `jTAC.define()`. It can still be instantiated using `new jTAC.namespace.class()`. However, that does not invoke the code defined in **constructor** nor does it initialize the **config** property.
- **_internal** (object) – Do not declare this directly in members. But you can use it to store private values.

```
function isDefined( fullClassName )
```

Checks if the class is defined with jTAC. If it is not, the script file for that class has not been loaded or there was a JavaScript error preventing it from successfully loading. Use the browser's console to view JavaScript errors.

Parameters

fullClassName (string)

The class name, including namespaces. It is case sensitive. Does not support alias names.

Returns `true` if found and `false` if not.

```
function addMembers( fullClassName, members )
```

Adds members to an existing jTAC class definition. It can include **config** and **configrules** properties to define additional properties of your class. See `jTAC.define()` for details.

Parameters

fullClassName (string)

The class name, including namespaces. It is case sensitive. Does not support alias names.

members (object)

An object whose properties will be new members. It can include **config** and **configrules** properties, each with their own members. If supplied, **config** members will be converted into properties, with getter and setter functions defined automatically if not already present.

```
function require( fullClassNames )
```

Check for one or more class names to be defined. It throws an exception if a class name is not defined.

Parameters

fullClassName (string or array)

The class name, including namespaces. It is case sensitive. Does not support alias names.

For multiple names, use an array of full class names.

```
function defineAlias( aliasName, fullClassName, propertyValues, replace, optional )
```

Adds an entry to the alias' list that lets the create function create an instance with a specific set of properties.

Parameters

aliasName (string)

The name of the alias. It does not need namespaces. It just needs to be unique. If identical to an existing name, it replaces that element.

fullClassName (string)

The class name, including namespaces, that will be created when using this alias. It is case sensitive.

propertyValues (object or function)

Optional object or function used to initialize the properties of the new instance. See `jTAC.createByClassName()` for details.

replace (boolean)

If the alias exists, `true` will replace its definition. `false` will throw an exception.

optional (boolean)

When `true`, it's safe to call this even if fullClassName isn't defined. It allows you to setup a number of calls to `defineAlias()` as a master setup, without worrying if they are present.

Adding your own classes to jTAC

jTAC is a very expandable library. You are likely to create your own TypeManager, Condition, and Connection classes. Part of its power is that the base classes do not require other libraries, like *jQuery* or *jquery-globalize*. You can subclass to introduce your own library specific support, ignoring the support already included.

If you know object oriented programming, you will relate to this approach. If you are a JS Ext v4 user, jTAC is very similar.

JavaScript is not a traditional object oriented language, but with its prototype feature, you can override and extend parent class methods. jTAC takes advantage of this, hiding the particulars of prototyping for you.

Defining a Members object

Your task is to create a JavaScript object definition. The syntax within a JavaScript object is:

```
{name1: value, name2: value, name3: value}
```

One of the trickiest issues for programmers is remembering to use commas instead of semicolons as separators.

```
var members = {
   toString: function(val) {
   },
   toValue: function(text) {
   },
   isValidChar: function(chr) {
   }
}
```

Your methods are function definitions. If you are overriding a function in an ancestor class, be sure the function definitions are compatible with those in the ancestor.

Defining properties in the config property

Do not define property values directly in the members object. Instead, create a property called **config** that is an object which will host your properties, by name and default value.

```
var members = {
   config: {
      dateFormat: 0,
      datePattern: "MM/dd/yyyy"
   },
   toString: function(val) {
   },
   toValue: function(text) {
   },
   isValidChar: function(chr) {
   }
}
```

Each property of **config** is automatically given getter and setter functions on the members object itself. This happens behind the scenes:

```
var members = {
   config: {
      dateFormat: 0,
      datePattern: "MM/dd/yyyy"
   },
   getDateFormat: function() {
      return this.config.dateFormat;
   },
   setDateFormat: function(val) {
      this.config.dateFormat = jTAC.checkAsInt(val);
   },
   getDatePattern: function() {
      return this.config.datePattern;
```

```
   },
   setDateFormat: function(val) {
      this.config.datePattern = jTAC.checkAsStr(val);
   }
}
```

If those definitions are not exactly what you want, you have two options.

1. If the setter needs alternative validation (`jTAC.checkAsInt()` and `jTAC.checkAsStr()` are both validation functions), use the **configrules** property on the members object. See the `jTAC.define()` method for details.

```
var members = {
    config: {
        dateFormat: 0,
        datePattern: "MM/dd/yyyy"
    },
    configrules: {
        dateFormat: [0, 1, 2],
        datePattern: {
            valFnc: function(val) {
                if (/^[Mdy/]*/.test(val))
                       return val;
                this._error("Illegal character");
            }
        }
    }
}
```

2. Otherwise, explicitly declare the function. This will override the default.

```
var members = {
    config: {
        dateFormat: 0,
        datePattern: "MM/dd/yyyy"
    },
    getDatePattern: function() {
        switch (this.config.dateFormat) {
            case 0:
                return this.config.datePattern;
            case 1:
                return "MMM dd, yyyy";
            case 2:
                return "MMMM dd, yyyy";
        }
    }
}
```

Inheritance

If your class should inherit the members of another, add the **extend** property. Assign its value to the full class name of the class from which it inherits.

When a method needs to call its ancestor method, call `this.callParent([parameter list])`, where the parameter list is an array of the same parameters your function is passed.

```
var members = {
   extend: "TypeManagers.Base",
   isValidChar: function(chr) {
      if (!this.callParent([chr]))
         return false;
      // do more here
   }
}
```

Developing a base class

If your class is a base to others and should not be created on its own, add the **abstract** property. Assign its value `true`. For any method that you define but must be overridden (an abstract method), call `this._AM()` from in the function.

```
var members = {
   abstract: true,
   toString: function(val) {
      this._AM();
   }
}
```

Constructor method

When you define a class in most programming languages, you define a constructor method which is called when the object is created, to initialize it. JavaScript objects also use constructors.

jTAC builds a constructor method internally for you and lets you define another function to be called as the object is created. Your function must be called **constructor**. It will be called to finish initialization. All properties are already at their defaults and getter and setter methods have already been created.

Your **constructor** function is passed on parameter, propertyValues, which is an object with properties whose values you can use in the constructor function.

It is actually unusual to need a constructor function. jTAC's **config** property handles most of the default property initialization. It also knows how to set all property names found in the propertyValues object to the same-named properties on your object. Most of the time, your constructor is used to create internal field values that are not properties. Those field values should be added to the **_internal** property on the members object.

If your object uses the **extend** property, be sure to call the ancestor constructor using `this.callParent([propertyValues])`.

```
var members = {
   extend: "TypeManagers.Base",
   constructor: function(propertyValues) {
      this.callParent([propertyValues]);
      this._internal.re = "^[\\d/]$";
   },
   isValidChar: function(chr) {
      if (!this.callParent([chr]))
         return false;
      return this._internal.re.test(chr);
   }
}
```

Requiring other classes

When your class needs to use another class, but does not inherit from that class, use the **require** property. It takes an array of strings, each of which are full class names that it requires. **require** can also take just a single class name string, without the array. You only need to define the actual class needed, not any of the ancestors to that class.

```
var members = {
   extend: "TypeManagers.Base",
   require: "Connection.Base"
}
```

Defining your class with jTAC

Once completed, give your class a name and identify its namespace. Existing namespaces include “Conditions”, “TypeManagers”, “Connections”, and “CalcItems”. Class names should always start with an uppercase letter. Be aware that this full class name is case sensitive.

Then call the `jTAC.define()` method, passing your full class name and the members object.

```
jTAC.define("TypeManagers.MyClass", members);
```

You can also create aliases to your object that do not reflect the full namespace and may provide alternative default properties with the `jTAC.defineAlias()` method.

```
jTAC.define("MyDate", "TypeManagers.MyClass");

jTAC.define("mydate", "TypeManagers.MyClass");

jTAC.define("MyDate.Full", "TypeManagers.MyClass", {dateFormat: 2});
```

See also:

- Using TypeManagers in your own JavaScript
- TypeManagers.Base
- Using Conditions in your own JavaScript
- Conditions.Base
- Using Connections in your own JavaScript
- Connections.Base

jTAC.Translations system: Replacing strings shown to the user

jTAC.Translations is a class that replaces strings shown to the user. It has a library of default strings, which you can edit. It lets you define alternative libraries that are localized.

Click on any of these topics to jump to them:

Adding to the page

The jTAC.Translations class is always loaded when using jTAC. (It is included in **jTAC****jTAC.js**) However, it is not active until you load a library of strings for it to manage.

Start it up by adding this script tag. It loads the default strings. This line must always follow loading jTAC.js and is recommended to be before anything else (because *jquery-validate* does some initialization with it when you add jTAC script files for its features.)

```
<script src="/jTAC/jTAC.js" type="text/javascript"></script>
<script src="/jTAC/Translations/Culture Neutral.js" type="text/javascript"></script>
```

Localized script files are in the **jTAC****Translations**\ folder. Add one or more of them after the <script> tag shown above. If you add more than one language, add the default language last (or call `jTAC.translations.setLanguage("culturename")` after these scripts have been loaded.)

Note: If you minify these files, specify /jTAC-min/ in these lines. See "Minifying your translations files".

jTAC provides its *jTAC.Translations system* (included in **jTAC.js**) to register key and value pairs. All jTAC code that needs to show the user a string will ask the jTAC.Translations for a string that you defined, and only if not found will it use a hard-coded default.

```
<script src="/jTAC/jTAC.js" type="text/javascript"></script>
<script src="/jTAC/Translations/Culture Neutral.js" type="text/javascript"></script>
<script src="/jTAC/Translations/fr.js" type="text/javascript"></script>
```

Editing the default strings

Open any of the script files in the **jTAC****Translations**\ folder. Edit the strings assigned to property names to reflect the desired text. Do not edit the property names themselves.

Extending the library

You can create additional strings in these files. Each must have a unique property name that becomes the lookup key elsewhere in jTAC.

For example:

```
illegalDateEM : "<span class='{ERRORLABEL}'>{LABEL}</span> requires a date " +
   "in this format: month/day/year."
```

Once defined, you can use your new key in **lookupKey** properties found on various classes throughout jTAC.

- For the error message, assign the key to the **lookupKey** property on the parameters of a validation rule.

```
advrange: {
   param: {minimum: 1, maximum: 10, lookupKey: 'newkeyname' }
}
```

  In unobtrusive validation:

```
<input type="text" id="textbox1" name="textbox1"
    data-jtac-datatype="integer" data-val="true"
    data-val-advrange=""
    data-val-advrange-json="{'minimum': 1, 'maximum': 10, 'lookupKey': 'newkeyname'}" />
```

- For the "{DATATYPE}" token on Conditions.Required, assign the key to the **friendlyLookupKey** property on the TypeManager object. Typically you add the **data-jtac-typemanager** attribute to the HTML tag, using JSON format, and specifying properties to override on the TypeManager defined by **data-jtac-datatype**.

```
<input type="text" id="textbox1" name="textbox1"
    data-jtac-datatype="integer"
    data-jtac-typemanager="{'friendlyLookupKey' : 'newkeyname'}" />
```

- For the "{PATTERN}" token, **patternLookupKey** property on the Conditions.RegExp.

```
regexp: {
   param: {expression: '^\\d{5}', patternLookupKey: 'newkeyname' }
}
```

  In unobtrusive validation:

```
<input type="text" id="textbox1" name="textbox1"
    data-val="true"
    data-val-regexp=""
    data-val-regexp-json="{'expression': '^\\d{5}', 'patternLookupKey': 'newkeyname' }" />
```

- For the "{LABEL}" token in any error message, add the **data-msglabel-lookupkey** attribute into the HTML tag that is being validated.

```
<input type="text" id="textbox1" name="textbox1"
    data-jtac-datatype="integer"
    data-msglabel-lookupkey="newkeyname" />
```

Minifying your translations files

If you want to minify these script files into the **\jTAC\Translations** folder, a Windows command line batch file has been provided. It uses the YUI compressor to do most of the work.

Locate the **Minify Translations.js** file in the **[product folder]\Preparing Scripts** folder.

Before it can be used, you must edit it.

1. Open the file in a text editor.
2. Change the file paths in the two `SET` statements.
3. Remove the first `pause` and `exit` statement.
4. Save.

Run this batch file each time you need to refresh the **\jTAC-min\Translations** folder.

Be sure that your script file paths reference these files in the **\jTAC-min** folder instead of **\jTAC**.

```
<script src="/jTAC-min/jTAC.js" type="text/javascript"></script>
<script src="/jTAC-min/Translations/Culture Neutral.js" type="text/javascript"></script>
```