Perfect Widgets Documentation

Last modified on: June 29, 2012

Table of Content

Preface

This document contains instructions on how to add Perfect Widgets gauge to the page as well as description of setup features of the gauge.

This user guide is prepared by Perpetuum Software team for Perfect Widgets users.

System Requirements

Gauges are displayed in the browser in SVG format. That is why browser should support SVG 1.1 in order to use the product.

Windows OS with installed .NET Framework 4 is required to run Instrument designer.

Supported Browsers

Internet Explorer 9+

Mozilla Firefox 4+

Google Chrome 7+

Safari 5.1+

Opera 11.6+

Supported Platforms

PC, Mac, iOS, Android, Windows Phone 7

Widget Designer Requirements

- Windows XP, 7, 8
- .NET Framework 4

Product Description

Before we proceed to the process of creation of a gauge on the page it is necessary to mention some peculiarities in Perfect Widgets use.

Perfect Widgets is a set of JavaScript libraries and a stand-alone application (gauge designer) intended to create gauges from base primitives. All this together helps create and add new unique gauges to your page in just a couple of minutes.

Gauge view format

Information about gauge is stored in JSON format. JSON object is got as a result of work in the gauge designer and passed to the gauge constructor for its creation.

Basic ways of getting gauge JSON

Basic ways of getting JSON object are:

- Creation of gauge in the designer;
- Purchase of separate gauges from our web site.

About JSON object

As you know there is a SharpShooter Gauges product which is a great .NET windows forms solution with instrument designer. JSON helps us to connect JavaScript-based Perfect Widgets with SharpShooter Gauges. We can say that JSON describes instrument in the platform independent way.

There is a stand-alone Windows application for gauge design called Instrument Designer in the package. You can design a widget in Instrument Designer, export it into JSON model and load this model in Perfect Widgets product.

Of course it's not a must and you can create your widget without JSON by assembling instrument part by part. It's possible, but not trivial as WYSWYG SharpShooter Gauges designer.

JSON is created by standard .NET `DataContractJsonSerializer`. Classes are marked with *DataContract* attribute and members are marked with *DataMember* attribute. Those marked will be available in the resulting JSON.

When you get JSON from SharpShooter Gauges you may change some settings in this JSON and use this modified JSON in Perfect Widgets product.

Development

Adding gauge to the web page

Add gauge to the web page will require only basic knowledge of HTML and JavaScript.

Note: To continue development you need to get JSON view of the gauge in one of the described ways.

Step 1. Adding JavaScript libraries

Add reference to the "PerfectWidgets.js" of Perfect Widgets in the page title.

Step 2. Adding container for the element

Separate div element should be added on a page for each gauge.

```
<div id="widget"></div>
```

After the gauge is added it should be the only content of this div element.

Step 3. Adding gauge to the page

To add gauge to the page you need to create new instance of the object passing name of the div that will be used to contain gauge and JSON description of this gauge to the constructor.

*Widget*class constructor looks as follows:

```
public Widget(id, jsonModel, additionalParams, tool, view)
```

Parameter name	Class	Comments
id	string	ID div that will contain gauge
jsonModel	object	Object containing description of basic gauge elements
additionalParams	object	Optional parameter. Associative array containing additional gauge parameters
tool	Tool	Optional parameter. Not used
view	AbstractView	Optional parameter. Not used

Thus, you can add gauge to the page using the following code:

```
<script type="text/javascript">
	$(document).ready(function () {
		var model = { <Widget Description Object> };
		var widget = new PerfectWidgets.Widget("widget", model);
	}
</script>
```

Note: jQuery is used to simplify sample, but this library is not required to make Perfect Widgets work.

Loading the page will result in the following view:

Configuring gauge with the help of additional parameters

It is possible to pass to the constructor additional parameter (jsonParams) that will allow changing of the gauge.

```
var model = { "Active": true, "BreakEventsBubbling":… <very long string> …};
var additionalParams = {
          "uniqueClassName": "widget_id",
          "keepRatio": false,
          "Scale1.Minimum": -50,
          "RangedLevel2.Value": "50%",
          "Slider1.Value": 30
        };

new PerfectWidgets.Widget("widget", model, additionalParams);
```

Most parameters have quite simple format `[gauge element name].[property name]`.

Note: To set gauge properties that allow percent values it is possible to use string value in the following format "`value%`". For example: `"RangedLevel2.Value": "50%"`.

So, it is possible to set any value for the gauge element properties. For example, by adding the following property `"Scale1.Minimum":-50` we set minimum value of the *Scale1* element.

Keeping proportions

Gauge keeps proportions when setting div size by default (without using JSON with parameters). If it is necessary to stretch gauge to fit the whole div without keeping proportions, it is possible to use *keepRatio* property in the array of additional parameters. For example:

```
<div id="widgetContainer" style="width:200px;height:200px"></div>

new PerfectWidgets.Widget("widgetContainer", model, { keepRatio": false });
```

Gauge interactivity

Gauge is interactive by default, i.e. user can set values not only from code, but also by drag-and-drop.

When it is necessary to disable drag-and-drop functionality of some gauge you need either to pass additional value to JSON with parameters `"interactive": false`, or call gauge method `public void SetInteractive(bool interactive)`.

Setting values

Adding gauge to the page is not just enough. It is necessary to set values for its elements.

Slider object is responsible for the position of the pointers and other interactive gauge elements. Its *Value* property is changed within limits set by the corresponding *Scale* object and is responsible for the position of the nested visual elements, for example, *Needle*, *Circle* etc.

To set value of the gauge element you need to know the name of the corresponding slider. If it wasn't changed, most likely it is *Slider1*, *Slider2* etc.

To find necessary *Slider* and set its value, it is possible to use gauge designer (in case .IMK file is available), or online Widget JSON Inspector, if you've got only JSON.

Slider–type element should be nested in the *Scale*-type element, that is in its turn is nested in *Guide* (for linear gauges) or *Joint* (for radial gauges).

Knowing its name (let it be *Slider1*), we can use the following code to set values:

```
aWidget.getByName("Slider1").setValue(4);
```

Animation set for *Slider1* will run automatically. If it necessary to set value at once without animation, you need to pass the *false* value as a second parameter:

```
aWidget.getByName("Slider1").setValue(4, false);
```

Such manner of setting value is useful when it is necessary to set start values of the gauge directly after it is drawn.

Event handling

List of available events

If interactive gauge is created, user should be able to catch change of value of any gauge slider. You need to add handler for several events generated by a gauge.

Table 1. Events generated by a gauge

Event	Object	How to subscribe	Argument	Description
valueChanged	Slider	`void addValueChangedHandler (EventHandler handler)` `void removeValueChangedHandler (EventHandler handler)`	**handler** – function taking two parameters **sender** and **e**). **sender** -- object calling this event, **e** – EventArgs.	Works at every change of Slider value (without animation)
valueChanged	Slider	`void addAnimationValueChangedHandler(EventHandler handler)` `void removeAnimationValueChangedHandler(EventHandler handler)`	**handler** – function taking two parameters **sender** and **e**). **sender** -- object calling this event, **e** – EventArgs.	Works at every change of Slider value (and in animation)
animationStarting	Slider	`void addAnimationStartingHandler (EventHandler handler)` `void removeAnimationStartingHandler (EventHandler handler)`	**handler** – – function taking two parameters **sender** and **e**). **sender** -- object calling this event, **e** – CancelEventArgs.	Works right before animation starts. Animation can be disabled by setting some value `e.cancel = true;`
animationFinished	Slider	`void addAnimationFinishedHandler (EventHandler handler)` `void removeAnimationFinishedHandler (EventHandler handler)`	**handler** – function taking two parameters **sender** and **e**). **sender** -- object calling this event, **e** – CancelEventArgs.	Works directly after animation ends.

Utilization Samples

To continue development you need a page with a gauge. If you don't have it, please review item **"Adding gauge to the web page"**.

valueChanged

```
<script type="text/javascript">
        window.onload = function () {
            var model = { <Widget Description Object> };
            var widget = new PerfectWidgets.Widget("widget", model);
            var slider = widget.getByName("Slider1");

            slider.addValueChangedHandler(
                function (sender, e) {
                    alert(sender.getValue());
                }
            )
        }
    </script>
```

animationStarting

```
<script type="text/javascript">
        window.onload = function () {
            var model = { <Widget Description Object> };
            var widget = new PerfectWidgets.Widget("widget", model);

            var slider1 = widget.getByName("Slider1");
            slider1.addAnimationStartingHandler(animationStarting);
        }

        function animationStarting(sender, e) {
            alert("animation starting");

            if (/*CONDITION FOR INTERRUPT ANIMATION*/) {
                e.cancel = true;
            } else {
                /*DO SOMETHING*/
            }

        }
    </script>
```

animationFinished

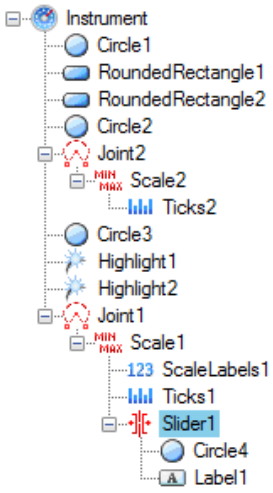
```
<script type="text/javascript">
        window.onload = function () {
            var model = { <Widget Description Object> };
            var widget = new PerfectWidgets.Widget("root", model);

            var slider1 = widget.getByName("Slider1");
            slider1.addAnimationFinishedHandler(animationFinished);
```

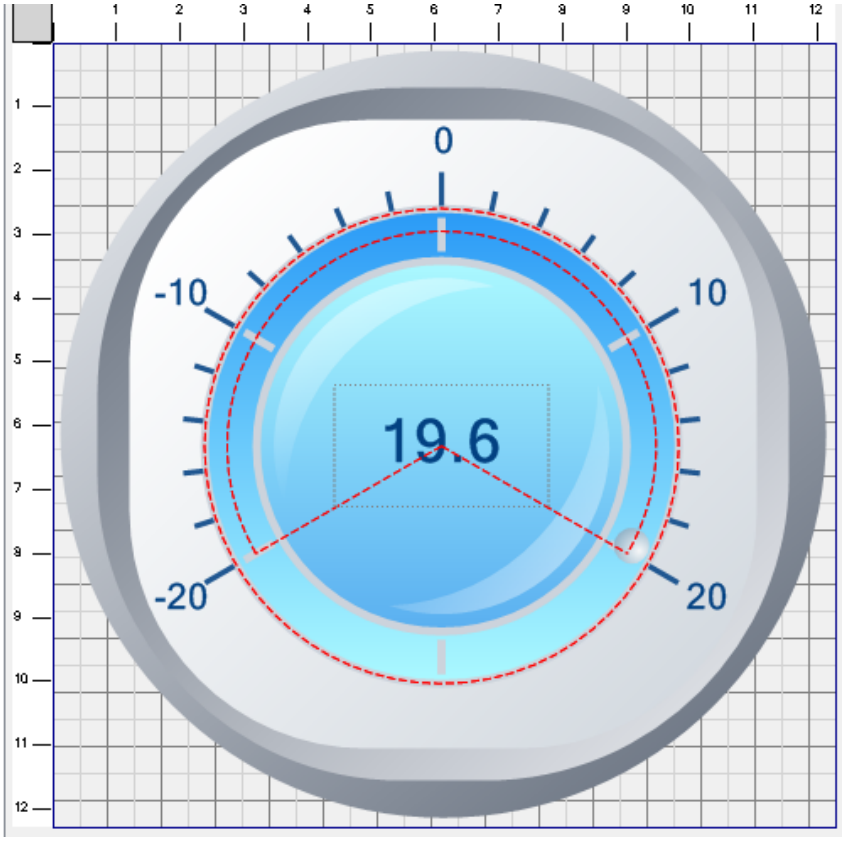
```
    }

    function animationFinished(sender, e) {
        alert("animation finished");
    }
</script>
```

Inner gauge view

General gauge structure

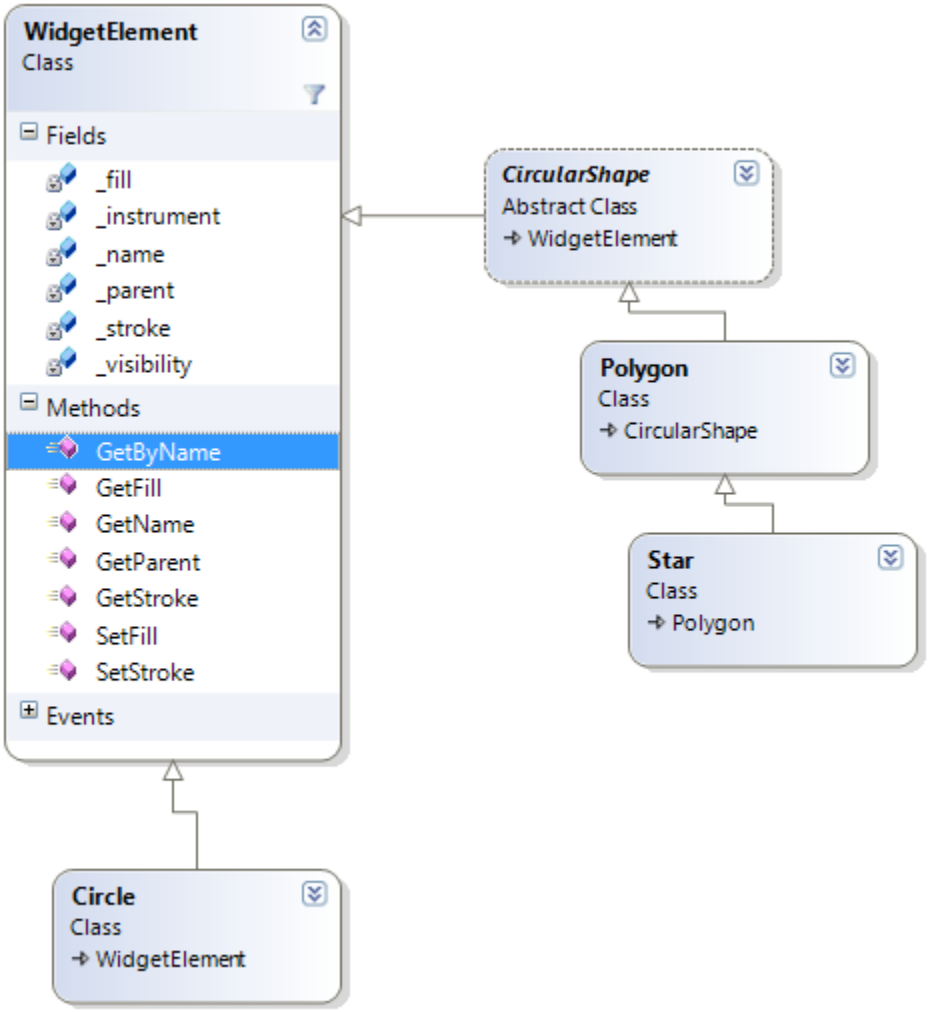
Gauge has tree-like structure, i.e. any gauge created in Perfect Widgets is a composition of some other elements. Gauge is a data model (hierarchy of elements) and a set of parameters that configure this model. Having a model and additional parameters you can describe every gauge in a unique way.





Selected common properties of the gauge elements

All gauge elements are in strict hierarchy and are descendants from the base element WidgetElement. It provides base interface of every element.



On the picture, you can see some WidgetElement properties. Two methods are used to access every gauge property: getter and setter. For example, gauge has `fill` property setting fill of the element. To get value of this property you need to use `getFill()` method, and to set new fill – `setFill(<new fill>)` method.

Special attention should be paid to `Name` property. This very property allows identification of every element inside gauge as well as getting access to the element using getByName method.

getByName(. . .) method

This method `public WidgetElement GetByName(string name)` takes element name as the only parameter and gets it if there is such an element in the gauge.

Sample

One of the most frequently used values in the gauge is value of the `Value` property in the `Slider` object:

```
<script type="text/javascript">
        window.onload = function () {
            var model = { <Widget Description Object> };
            var clock = new PerfectWidgets.Widget("div1", model);
            var hours = clock.getByName("Slider1").getValue();
            clock.getByName("Slider1").setValue(12);

        };
</script>
```

Information on the gauge structure

If you have only description of the gauge in JSON format and it is not possible to open the corresponding IMK file in the designer, you can inspect its structure using online Widget JSON Inspector

This service allow getting of the gauge structure based only on its description in JSON format.

Gauge customization with CSS

Use of CSS styles makes it possible to quickly change gauge appearance. You just need to slightly change a web page where gauge is used.

Create CSS file add reference to it in the project; or just add Inline styles.

```
<head>
    <title></title>
    <link href="StyleSheet1.css" rel="stylesheet" type="text/css" />
    <script src="scripts/mscorlib.js" type="text/javascript"></script>
    <script src="scripts/PerfectWidgets.js" type="text/javascript"></script>
    <script type="text/javascript">
        window.onload = function () {
            var model = { <Widget Description Object> };
            new PerfectWidgets.Widget("widget", model);
        }
    </script>
</head>
```

Methods of setting element style

It is possible to apply one of the methods of setting styles in Perfect Widgets.

Method № 1

Every gauge element is automatically bound to some CSS class which name reflects element position in the hierarchy of gauge elements. As a sample, we will use a gauge with the following structure: `Instrument` → `Joint1` → `Scale1` → `Slider1` → `Circle4`. This structure can be inspected using online Widget JSON Inspector. `Circle4` belongs to `Intrument_Joint1_Scale1_Slider1_Circle4` class. In the same way you can define class of any other element.

In order to change `Circle4` element color, you need to add the following line in the Styles.css file:

```
.widget_Instrument_Joint1_Scale1_Slider1_Circle4
{
    fill:lime;
    stroke:black;
    stroke-width:4;
}
```

Method № 2

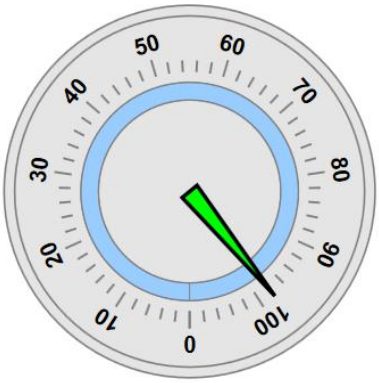
The second method of gauge customization is that some gauge elements are assigned with classes with intuitive names on the design stage. For example: `foreground`, `backround`, `ticks`, etc. In this case there is no necessity to inspect gauge structure to get name of class of some element. Besides, you can apply single style to several elements. Class names are set by the user when creating gauge in the designer. (CssClass property of the selected element in the propertyGrid).

The following content of `Styles.css` gets the same results as the method № 1:

```
.widget_arrow
{
    fill:lime;
    stroke:black;
    stroke-width:4;
}
```

Or this variant that can be easily applied to add gauges on the page with corresponding mark up:

```
.arrow
{
    fill:lime;
    stroke:black;
    stroke-width:4;
}
```



Some notes about naming classes

All methods of appearance customization with the help of CSS use some of the below classes:

- **Id**_structureName
- **Id**_className
- className

Prefix **id** matches with the id value of the element-container containing gauge. It is also possible to change it to the other matching id of other gauges (for example, to create common style for a definite group of elements) using the following method:

```
public void SetUniqueClassName(string id).
```

Notes about gradient fill

Unfortunately, SVG format has some limitations preventing setup of gradient fill directly in CSS file. If necessary, it is possible to set gradient fill from JS code using the following method:

```
public void SetFill(Fill FillValue)
```

It is necessary to configure `FillValue` at first

```
var fill = new PerfectWidgets.Framework.Drawing.LinearGradientFill();
fill.setStartColor(PerfectWidgets.Framework.Drawing.Color.fromArgb(255, 255, 0, 0));
fill.setEndColor(PerfectWidgets.Framework.Drawing.Color.fromArgb(255, 0, 255, 0));
widget.getByName("Circle7").setFill(fill);
```

Animation

Animation has become a must for today applications. By using build-in animation you get your application feel and look.

Animation is basically used by gauge slider element. Animation is enabled by default with the following settings:

```
Duration = 3
EasingFunction = "swing"
PropertyName = “AnimationValue”
Direction = “normal”
```

It’s quite easy to configure animation settings via configureAnimation method of the slider class.

Here is the div element where we want to place our widget:

```
widget = new PerfectWidgets.Widget("div1", model);
```

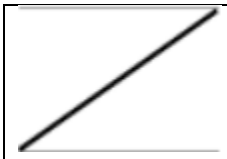
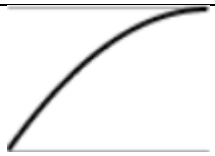
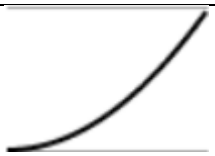
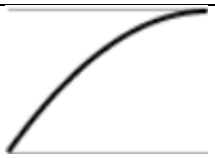
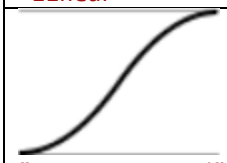
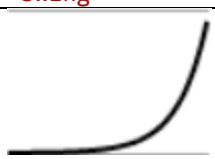
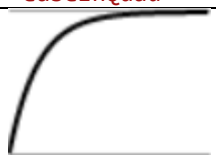
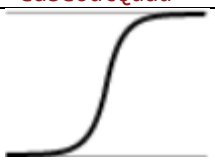
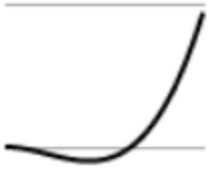
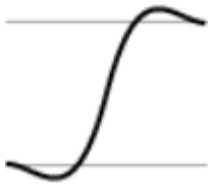
Firstly, we need to retrieve slider. Its name is Slider1:

```
widget.getByName("Slider1")
```

Then we need to call configuration method and specify some parameters in order to override default settings: " `widget.getByName("Slider1").configureAnimation({"enabled": true, "ease": "easeInQuad", "duration": 4, "direction": "CCW" });`", where:

- “*enabled*” is intended to enable or disable animation;
- “*ease*” allows setup of animation easing functions;
- “*duration*” impacts animation speed and means the period of time in seconds which takes slider to pass whole value range from maximum to minimum.
- “*direction*” allows to manage animation direction and intended for managing cycles in your gauge.

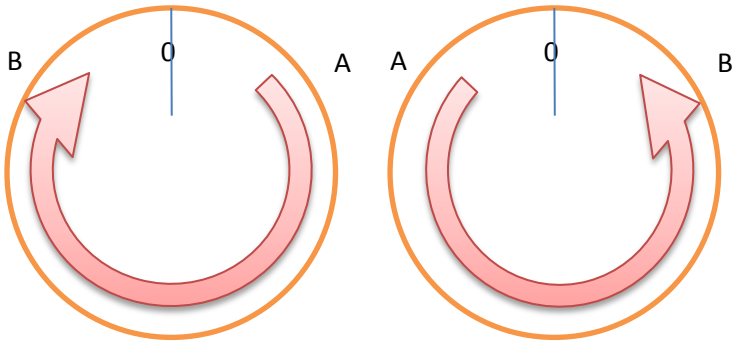
You can find possible easing function names here:

 "linear"	 "swing"	 "easeInQuad"	 "easeOutQuad"
 "easeInOutQuad"	 "easeInExpo"	 "easeOutExpo"	 "easeInOutExpo"
 "easeInBack"	 "easeOutBack"	 "easeInOutBack"	

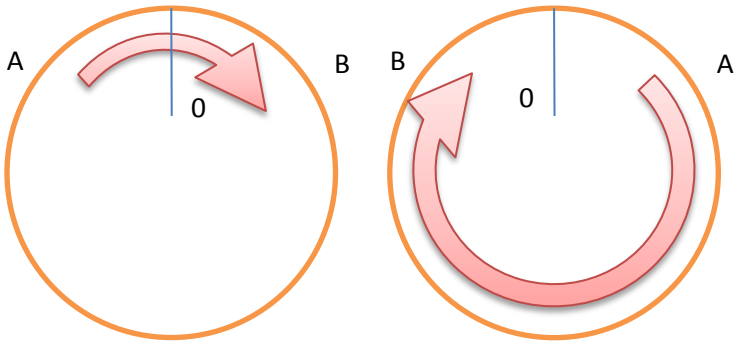
You can see demonstration of these functions here: http://jqueryui.com/demos/effect/easing.html

As for direction, there could be the following options with respective aliases:

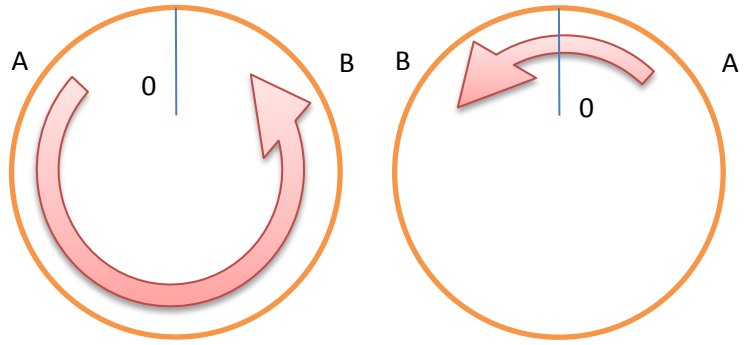
"normal" – there are no cycles in the widget.



"CW" or "clockwise" – there is a cycle in the widget and I want the slider to move in clockwise direction.



`"ACW", "CCW", "anticlockwise"` or `"counterclockwise"` – there is a cycle in the widget and I want the slider to move in counterclockwise direction.



`"nearest"` – there is a cycle in the widget and I want the slider to move to the specified value choosing the shortest way.