Chart.js Documentation

Everything you need to know to build great looking charts using Chart.js

---

Getting started

Include Chart.js

First we need to include the Chart.js library on the page. The library occupies a global variable of `Chart`.

```
<script src="Chart.js"></script>
```

Alternatively, if you're using an AMD loader for JavaScript modules, that is also supported in the Chart.js core. Please note: the library will still occupy a global variable of `Chart`, even if it detects `define` and `define.amd`. If this is a problem, you can call `noConflict` to restore the global Chart variable to it's previous owner.

```
// Using requirejs
require(['path/to/Chartjs'], function(Chart){
    // Use Chart.js as normal here.

    // Chart.noConflict restores the Chart global variable to it's previous owner
    // The function returns what was previously Chart, allowing you to reassign.
    var Chartjs = Chart.noConflict();

});
```

You can also grab Chart.js using bower:

```
bower install Chart.js --save
```

Also, Chart.js is available from CDN:

https://cdnjs.com/libraries/chart.js

Creating a chart

To create a chart, we need to instantiate the `Chart` class. To do this, we need to pass in the 2d context of where we want to draw the chart. Here's an example.

```
<canvas id="myChart" width="400" height="400"></canvas>
```

```
// Get the context of the canvas element we want to select
var ctx = document.getElementById("myChart").getContext("2d");
var myNewChart = new Chart(ctx).PolarArea(data);
```

We can also get the context of our canvas with jQuery. To do this, we need to get the DOM node out of the jQuery collection, and call the `getContext("2d")` method on that.

```
// Get context with jQuery - using jQuery's .get() method.
var ctx = $("#myChart").get(0).getContext("2d");
// This will get the first returned node in the jQuery collection.
var myNewChart = new Chart(ctx);
```

After we've instantiated the Chart class on the canvas we want to draw on, Chart.js will handle the scaling for retina displays.

With the Chart class set up, we can go on to create one of the charts Chart.js has available. In the example below, we would be drawing a Polar area chart.

```
new Chart(ctx).PolarArea(data, options);
```

We call a method of the name of the chart we want to create. We pass in the data for that chart type, and the options for that chart as parameters. Chart.js will merge the global defaults with chart type specific defaults, then merge any options passed in as a second argument after data.

Global chart configuration

This concept was introduced in Chart.js 1.0 to keep configuration DRY, and allow for changing options globally across chart types, avoiding the need to specify options for each instance, or the default for a particular chart type.

```
Chart.defaults.global = {
    // Boolean - Whether to animate the chart
    animation: true,

    // Number - Number of animation steps
    animationSteps: 60,

    // String - Animation easing effect
    // Possible effects are:
    // [easeInOutQuart, linear, easeOutBounce, easeInBack, easeInOutQuad,
    //  easeOutQuart, easeOutQuad, easeInOutBounce, easeOutSine, easeInOutCubic,
    //  easeInExpo, easeInOutBack, easeInCirc, easeInOutElastic, easeOutBack,
    //  easeInQuad, easeInOutExpo, easeInQuart, easeOutQuint, easeInOutCirc,
    //  easeInSine, easeOutExpo, easeOutCirc, easeOutCubic, easeInQuint,
    //  easeInElastic, easeInOutSine, easeInOutQuint, easeInBounce,
    //  easeOutElastic, easeInCubic]
    animationEasing: "easeOutQuart",

    // Boolean - If we should show the scale at all
    showScale: true,

    // Boolean - If we want to override with a hard coded scale
    scaleOverride: false,

    // ** Required if scaleOverride is true **
    // Number - The number of steps in a hard coded scale
    scaleSteps: null,
    // Number - The value jump in the hard coded scale
    scaleStepWidth: null,
    // Number - The scale starting value
    scaleStartValue: null,

    // String - Colour of the scale line
    scaleLineColor: "rgba(0,0,0,.1)",

    // Number - Pixel width of the scale line
    scaleLineWidth: 1,

    // Boolean - Whether to show labels on the scale
    scaleShowLabels: true,

    // Interpolated JS string - can access value
    scaleLabel: "<%=value%>",

    // Boolean - Whether the scale should stick to integers, not floats even if dr
    scaleIntegersOnly: true,

    // Boolean - Whether the scale should start at zero, or an order of magnitude
    scaleBeginAtZero: false,

    // String - Scale label font declaration for the scale label
```

```
scaleFontFamily: "'Helvetica Neue', 'Helvetica', 'Arial', sans-serif",

// Number - Scale label font size in pixels
scaleFontSize: 12,

// String - Scale label font weight style
scaleFontStyle: "normal",

// String - Scale label font colour
scaleFontColor: "#666",

// Boolean - whether or not the chart should be responsive and resize when the
responsive: false,

// Boolean - whether to maintain the starting aspect ratio or not when respons
maintainAspectRatio: true,

// Boolean - Determines whether to draw tooltips on the canvas or not
showTooltips: true,

// Function - Determines whether to execute the customTooltips function instea
customTooltips: false,

// Array - Array of string names to attach tooltip events
tooltipEvents: ["mousemove", "touchstart", "touchmove"],

// String - Tooltip background colour
tooltipFillColor: "rgba(0,0,0,0.8)",

// String - Tooltip label font declaration for the scale label
tooltipFontFamily: "'Helvetica Neue', 'Helvetica', 'Arial', sans-serif",

// Number - Tooltip label font size in pixels
tooltipFontSize: 14,

// String - Tooltip font weight style
tooltipFontStyle: "normal",

// String - Tooltip label font colour
tooltipFontColor: "#fff",

// String - Tooltip title font declaration for the scale label
tooltipTitleFontFamily: "'Helvetica Neue', 'Helvetica', 'Arial', sans-serif",

// Number - Tooltip title font size in pixels
tooltipTitleFontSize: 14,

// String - Tooltip title font weight style
tooltipTitleFontStyle: "bold",

// String - Tooltip title font colour
tooltipTitleFontColor: "#fff",

// Number - pixel width of padding around tooltip text
tooltipYPadding: 6,

// Number - pixel width of padding around tooltip text
tooltipXPadding: 6,

// Number - Size of the caret on the tooltip
tooltipCaretSize: 8,

// Number - Pixel radius of the tooltip border
tooltipCornerRadius: 6,

// Number - Pixel offset from point x to tooltip edge
tooltipXOffset: 10,

// String - Template string for single tooltips
tooltipTemplate: "<%if (label){%><%=label%>: <%}%><%= value %>",

// String - Template string for multiple tooltips
multiTooltipTemplate: "<%= value %>",
```

```
    // Function - Will fire on animation progression.
    onAnimationProgress: function(){},

    // Function - Will fire on animation completion.
    onAnimationComplete: function(){}
}
```

If for example, you wanted all charts created to be responsive, and resize when the browser window does, the following setting can be changed:

```
Chart.defaults.global.responsive = true;
```

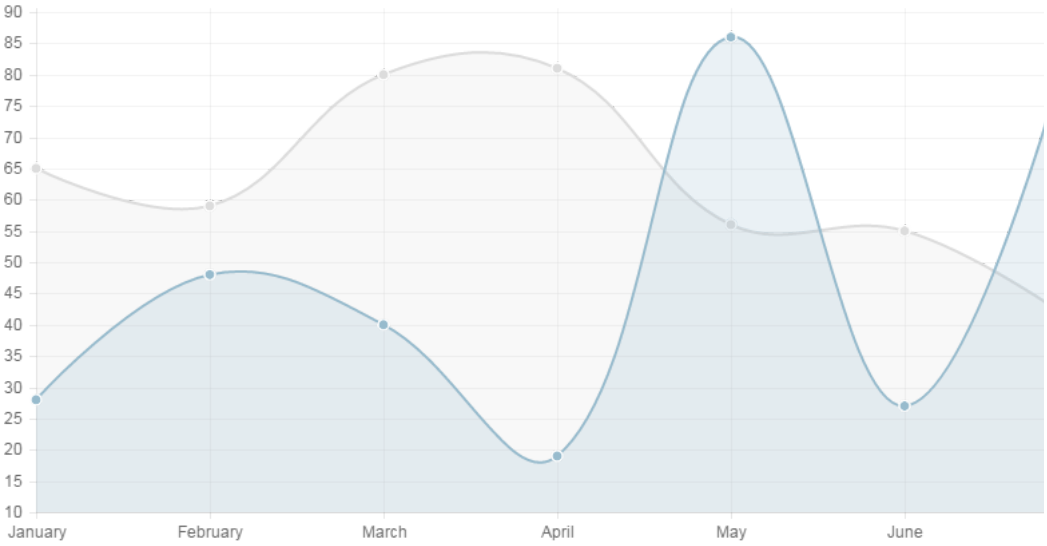
Now, every time we create a chart, `options.responsive` will be `true` .

Line Chart

Introduction

A line chart is a way of plotting data points on a line.

Often, it is used to show trend data, and the comparison of two data sets.



Example usage

```
var myLineChart = new Chart(ctx).Line(data, options);
```

Data structure

```
var data = {
    labels: ["January", "February", "March", "April", "May", "June", "July"],
    datasets: [
        {
            label: "My First dataset",
            fillColor: "rgba(220,220,220,0.2)",
            strokeColor: "rgba(220,220,220,1)",
            pointColor: "rgba(220,220,220,1)",
            pointStrokeColor: "#fff",
```

```
            pointHighlightFill: "#fff",
            pointHighlightStroke: "rgba(220,220,220,1)",
            data: [65, 59, 80, 81, 56, 55, 40]
        },
        {
            label: "My Second dataset",
            fillColor: "rgba(151,187,205,0.2)",
            strokeColor: "rgba(151,187,205,1)",
            pointColor: "rgba(151,187,205,1)",
            pointStrokeColor: "#fff",
            pointHighlightFill: "#fff",
            pointHighlightStroke: "rgba(151,187,205,1)",
            data: [28, 48, 40, 19, 86, 27, 90]
        }
    ]
};
```

The line chart requires an array of labels for each of the data points. This is shown on the X axis. The data for line charts is broken up into an array of datasets. Each dataset has a colour for the fill, a colour for the line and colours for the points and strokes of the points. These colours are strings just like CSS. You can use RGBA, RGB, HEX or HSL notation.

The label key on each dataset is optional, and can be used when generating a scale for the chart.

Chart options

These are the customisation options specific to Line charts. These options are merged with the global chart configuration options, and form the options of the chart.

```
{

    ///Boolean - Whether grid lines are shown across the chart
    scaleShowGridLines : true,

    //String - Colour of the grid lines
    scaleGridLineColor : "rgba(0,0,0,.05)",

    //Number - Width of the grid lines
    scaleGridLineWidth : 1,

    //Boolean - Whether to show horizontal lines (except X axis)
    scaleShowHorizontalLines: true,

    //Boolean - Whether to show vertical lines (except Y axis)
    scaleShowVerticalLines: true,

    //Boolean - Whether the line is curved between points
    bezierCurve : true,

    //Number - Tension of the bezier curve between points
    bezierCurveTension : 0.4,

    //Boolean - Whether to show a dot for each point
    pointDot : true,

    //Number - Radius of each point dot in pixels
    pointDotRadius : 4,

    //Number - Pixel width of point dot stroke
    pointDotStrokeWidth : 1,

    //Number - amount extra to add to the radius to cater for hit detection outsi
    pointHitDetectionRadius : 20,

    //Boolean - Whether to show a stroke for datasets
    datasetStroke : true,

    //Number - Pixel width of dataset stroke
    datasetStrokeWidth : 2,

    //Boolean - Whether to fill the dataset with a colour
```

```
    datasetFill : true,

    //String - A legend template
    legendTemplate : "<ul class=\"<%=name.toLowerCase()%>-legend\"><% for (var i=0

};
```

You can override these for your `Chart` instance by passing a second argument into the `Line` method as an object with the keys you want to override.

For example, we could have a line chart without bezier curves between points by doing the following:

```
new Chart(ctx).Line(data, {
    bezierCurve: false
});
// This will create a chart with all of the default options, merged from the globa
// and the Line chart defaults, but this particular instance will have `bezierCurv
```

We can also change these defaults values for each Line type that is created, this object is available at `Chart.defaults.Line` .

Prototype methods

.getPointsAtEvent(event)

Calling `getPointsAtEvent(event)` on your Chart instance passing an argument of an event, or jQuery event, will return the point elements that are at that the same position of that event.

```
canvas.onclick = function(evt){
    var activePoints = myLineChart.getPointsAtEvent(evt);
    // => activePoints is an array of points on the canvas that are at the same pc
};
```

This functionality may be useful for implementing DOM based tooltips, or triggering custom behaviour in your application.

.update( )

Calling `update()` on your Chart instance will re-render the chart with any updated values, allowing you to edit the value of multiple existing points, then render those in one animated render loop.

```
myLineChart.datasets[0].points[2].value = 50;
// Would update the first dataset's value of 'March' to be 50
myLineChart.update();
// Calling update now animates the position of March from 90 to 50.
```

.addData(valuesArray, label)

Calling `addData(valuesArray, label)` on your Chart instance passing an array of values for each dataset, along with a label for those points.

```
// The values array passed into addData should be one for each dataset in the char
myLineChart.addData([40, 60], "August");
// This new data will now animate at the end of the chart.
```

.removeData( )

Calling `removeData()` on your Chart instance will remove the first value for all datasets on the chart.

```
myLineChart.removeData();
```

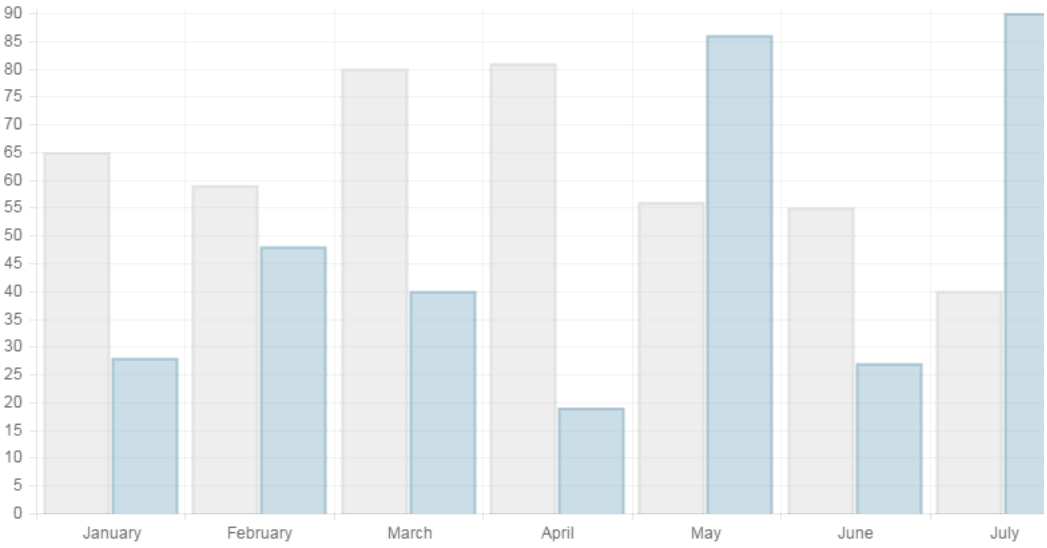
```
// The chart will remove the first point and animate other points into place
```

Bar Chart

Introduction

A bar chart is a way of showing data as bars.

It is sometimes used to show trend data, and the comparison of multiple data sets side by side.



Example usage

```
var myBarChart = new Chart(ctx).Bar(data, options);
```

Data structure

```
var data = {
    labels: ["January", "February", "March", "April", "May", "June", "July"],
    datasets: [
        {
            label: "My First dataset",
            fillColor: "rgba(220,220,220,0.5)",
            strokeColor: "rgba(220,220,220,0.8)",
            highlightFill: "rgba(220,220,220,0.75)",
            highlightStroke: "rgba(220,220,220,1)",
            data: [65, 59, 80, 81, 56, 55, 40]
        },
        {
            label: "My Second dataset",
            fillColor: "rgba(151,187,205,0.5)",
            strokeColor: "rgba(151,187,205,0.8)",
            highlightFill: "rgba(151,187,205,0.75)",
            highlightStroke: "rgba(151,187,205,1)",
            data: [28, 48, 40, 19, 86, 27, 90]
        }
    ]
};
```

The bar chart has the a very similar data structure to the line chart, and has an array of datasets,

each with colours and an array of data. Again, colours are in CSS format. We have an array of labels too for display. In the example, we are showing the same data as the previous line chart example.

The label key on each dataset is optional, and can be used when generating a scale for the chart.

Chart Options

These are the customisation options specific to Bar charts. These options are merged with the global chart configuration options, and form the options of the chart.

```
{
    //Boolean - Whether the scale should start at zero, or an order of magnitude c
    scaleBeginAtZero : true,

    //Boolean - Whether grid lines are shown across the chart
    scaleShowGridLines : true,

    //String - Colour of the grid lines
    scaleGridLineColor : "rgba(0,0,0,.05)",

    //Number - Width of the grid lines
    scaleGridLineWidth : 1,

    //Boolean - Whether to show horizontal lines (except X axis)
    scaleShowHorizontalLines: true,

    //Boolean - Whether to show vertical lines (except Y axis)
    scaleShowVerticalLines: true,

    //Boolean - If there is a stroke on each bar
    barShowStroke : true,

    //Number - Pixel width of the bar stroke
    barStrokeWidth : 2,

    //Number - Spacing between each of the X value sets
    barValueSpacing : 5,

    //Number - Spacing between data sets within X values
    barDatasetSpacing : 1,

    //String - A legend template
    legendTemplate : "<ul class=\"<%=name.toLowerCase()%>-legend\"><% for (var i=0
}
```

You can override these for your `Chart` instance by passing a second argument into the `Bar` method as an object with the keys you want to override.

For example, we could have a bar chart without a stroke on each bar by doing the following:

```
new Chart(ctx).Bar(data, {
    barShowStroke: false
});
// This will create a chart with all of the default options, merged from the globa
//  and the Bar chart defaults but this particular instance will have `barShowStrc
```

We can also change these defaults values for each Bar type that is created, this object is available at `Chart.defaults.Bar` .

Prototype methods

.getBarsAtEvent(event)

Calling `getBarsAtEvent(event)` on your Chart instance passing an argument of an event, or jQuery event, will return the bar elements that are at that the same position of that event.

```
canvas.onclick = function(evt){
    var activeBars = myBarChart.getBarsAtEvent(evt);
    // => activeBars is an array of bars on the canvas that are at the same positi
};
```

This functionality may be useful for implementing DOM based tooltips, or triggering custom behaviour in your application.

.update( )

Calling `update()` on your Chart instance will re-render the chart with any updated values, allowing you to edit the value of multiple existing points, then render those in one animated render loop.

```
myBarChart.datasets[0].bars[2].value = 50;
// Would update the first dataset's value of 'March' to be 50
myBarChart.update();
// Calling update now animates the position of March from 90 to 50.
```

.addData(valuesArray, label)

Calling `addData(valuesArray, label)` on your Chart instance passing an array of values for each dataset, along with a label for those bars.

```
// The values array passed into addData should be one for each dataset in the char
myBarChart.addData([40, 60], "August");
// The new data will now animate at the end of the chart.
```

.removeData( )

Calling `removeData()` on your Chart instance will remove the first value for all datasets on the chart.

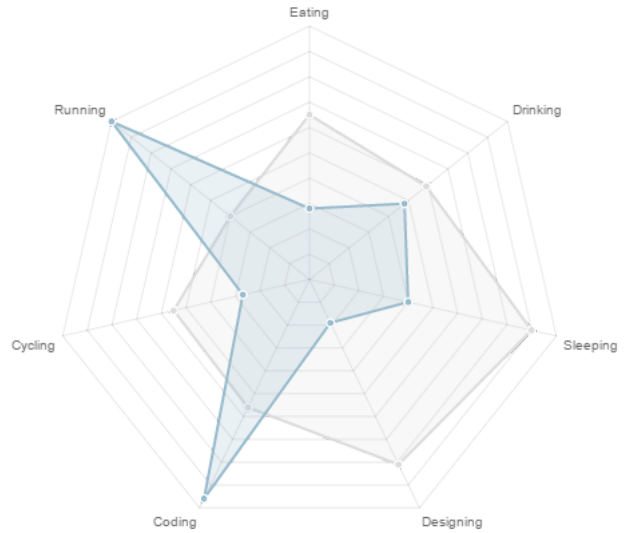
```
myBarChart.removeData();
// The chart will now animate and remove the first bar
```

Radar Chart

Introduction

A radar chart is a way of showing multiple data points and the variation between them.

They are often useful for comparing the points of two or more different data sets.



Example usage

```
var myRadarChart = new Chart(ctx).Radar(data, options);
```

Data structure

```
var data = {
    labels: ["Eating", "Drinking", "Sleeping", "Designing", "Coding", "Cycling", "
    datasets: [
        {
            label: "My First dataset",
            fillColor: "rgba(220,220,220,0.2)",
            strokeColor: "rgba(220,220,220,1)",
            pointColor: "rgba(220,220,220,1)",
            pointStrokeColor: "#fff",
            pointHighlightFill: "#fff",
            pointHighlightStroke: "rgba(220,220,220,1)",
            data: [65, 59, 90, 81, 56, 55, 40]
        },
        {
            label: "My Second dataset",
            fillColor: "rgba(151,187,205,0.2)",
            strokeColor: "rgba(151,187,205,1)",
            pointColor: "rgba(151,187,205,1)",
            pointStrokeColor: "#fff",
            pointHighlightFill: "#fff",
            pointHighlightStroke: "rgba(151,187,205,1)",
            data: [28, 48, 40, 19, 96, 27, 100]
        }
    ]
};
```

For a radar chart, to provide context of what each point means, we include an array of strings that show around each point in the chart. For the radar chart data, we have an array of datasets. Each of these is an object, with a fill colour, a stroke colour, a colour for the fill of each point, and a colour for the stroke of each point. We also have an array of data values.

The label key on each dataset is optional, and can be used when generating a scale for the chart.

Chart options

These are the customisation options specific to Radar charts. These options are merged with the global chart configuration options, and form the options of the chart.

```
{
    //Boolean - Whether to show lines for each scale point
    scaleShowLine : true,

    //Boolean - Whether we show the angle lines out of the radar
    angleShowLineOut : true,

    //Boolean - Whether to show labels on the scale
    scaleShowLabels : false,

    // Boolean - Whether the scale should begin at zero
    scaleBeginAtZero : true,

    //String - Colour of the angle line
    angleLineColor : "rgba(0,0,0,.1)",

    //Number - Pixel width of the angle line
    angleLineWidth : 1,

    //String - Point label font declaration
    pointLabelFontFamily : "'Arial'",

    //String - Point label font weight
    pointLabelFontStyle : "normal",

    //Number - Point label font size in pixels
    pointLabelFontSize : 10,

    //String - Point label font colour
    pointLabelFontColor : "#666",

    //Boolean - Whether to show a dot for each point
    pointDot : true,

    //Number - Radius of each point dot in pixels
    pointDotRadius : 3,

    //Number - Pixel width of point dot stroke
    pointDotStrokeWidth : 1,

    //Number - amount extra to add to the radius to cater for hit detection outsic
    pointHitDetectionRadius : 20,

    //Boolean - Whether to show a stroke for datasets
    datasetStroke : true,

    //Number - Pixel width of dataset stroke
    datasetStrokeWidth : 2,

    //Boolean - Whether to fill the dataset with a colour
    datasetFill : true,

    //String - A legend template
    legendTemplate : "<ul class=\"<%=name.toLowerCase()%>-legend\"><% for (var i=0

}
```

You can override these for your `Chart` instance by passing a second argument into the `Radar` method as an object with the keys you want to override.

For example, we could have a radar chart without a point for each on piece of data by doing the following:

```
new Chart(ctx).Radar(data, {
    pointDot: false
});
// This will create a chart with all of the default options, merged from the globa
//  and the Bar chart defaults but this particular instance will have `pointDot` s
```

We can also change these defaults values for each Radar type that is created, this object is available at `Chart.defaults.Radar`.

Prototype methods

.getPointsAtEvent(event)

Calling `getPointsAtEvent(event)` on your Chart instance passing an argument of an event, or jQuery event, will return the point elements that are at that the same position of that event.

```
canvas.onclick = function(evt){
    var activePoints = myRadarChart.getPointsAtEvent(evt);
    // => activePoints is an array of points on the canvas that are at the same pc
};
```

This functionality may be useful for implementing DOM based tooltips, or triggering custom behaviour in your application.

.update( )

Calling `update()` on your Chart instance will re-render the chart with any updated values, allowing you to edit the value of multiple existing points, then render those in one animated render loop.

```
myRadarChart.datasets[0].points[2].value = 50;
// Would update the first dataset's value of 'Sleeping' to be 50
myRadarChart.update();
// Calling update now animates the position of Sleeping from 90 to 50.
```

.addData(valuesArray, label)

Calling `addData(valuesArray, label)` on your Chart instance passing an array of values for each dataset, along with a label for those points.

```
// The values array passed into addData should be one for each dataset in the char
myRadarChart.addData([40, 60], "Dancing");
// The new data will now animate at the end of the chart.
```

.removeData( )

Calling `removeData()` on your Chart instance will remove the first value for all datasets on the chart.

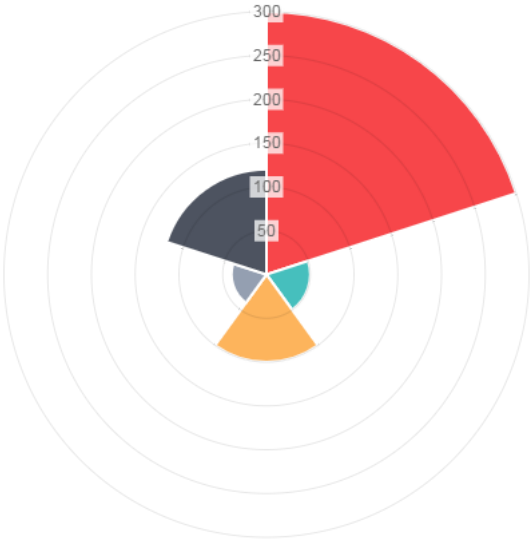
```
myRadarChart.removeData();
// Other points will now animate to their correct positions.
```

Polar Area Chart

Introduction

Polar area charts are similar to pie charts, but each segment has the same angle - the radius of the segment differs depending on the value.

This type of chart is often useful when we want to show a comparison data similar to a pie chart, but also show a scale of values for context.



Example usage

```
new Chart(ctx).PolarArea(data, options);
```

Data structure

```
var data = [
    {
        value: 300,
        color:"#F7464A",
        highlight: "#FF5A5E",
        label: "Red"
    },
    {
        value: 50,
        color: "#46BFBD",
        highlight: "#5AD3D1",
        label: "Green"
    },
    {
        value: 100,
        color: "#FDB45C",
        highlight: "#FFC870",
        label: "Yellow"
    },
    {
        value: 40,
        color: "#949FB1",
        highlight: "#A8B3C5",
        label: "Grey"
    },
    {
        value: 120,
        color: "#4D5360",
        highlight: "#616774",
        label: "Dark Grey"
    }

];
```

As you can see, for the chart data you pass in an array of objects, with a value and a colour. The value attribute should be a number, while the color attribute should be a string. Similar to CSS, for this string you can use HEX notation, RGB, RGBA or HSL.

Chart options

These are the customisation options specific to Polar Area charts. These options are merged with the global chart configuration options, and form the options of the chart.

```
{
    //Boolean - Show a backdrop to the scale label
    scaleShowLabelBackdrop : true,

    //String - The colour of the label backdrop
    scaleBackdropColor : "rgba(255,255,255,0.75)",

    // Boolean - Whether the scale should begin at zero
    scaleBeginAtZero : true,

    //Number - The backdrop padding above & below the label in pixels
    scaleBackdropPaddingY : 2,

    //Number - The backdrop padding to the side of the label in pixels
    scaleBackdropPaddingX : 2,

    //Boolean - Show line for each value in the scale
    scaleShowLine : true,

    //Boolean - Stroke a line around each segment in the chart
    segmentShowStroke : true,

    //String - The colour of the stroke on each segement.
    segmentStrokeColor : "#fff",

    //Number - The width of the stroke value in pixels
    segmentStrokeWidth : 2,

    //Number - Amount of animation steps
    animationSteps : 100,

    //String - Animation easing effect.
    animationEasing : "easeOutBounce",

    //Boolean - Whether to animate the rotation of the chart
    animateRotate : true,

    //Boolean - Whether to animate scaling the chart from the centre
    animateScale : false,

    //String - A legend template
    legendTemplate : "<ul class=\"<%=name.toLowerCase()%>-legend\"><% for (var i=0
}
```

You can override these for your `Chart` instance by passing a second argument into the `PolarArea` method as an object with the keys you want to override.

For example, we could have a polar area chart with a black stroke on each segment like so:

```
new Chart(ctx).PolarArea(data, {
    segmentStrokeColor: "#000000"
});
// This will create a chart with all of the default options, merged from the globa
// and the PolarArea chart defaults but this particular instance will have `segmer
```

We can also change these defaults values for each PolarArea type that is created, this object is available at `Chart.defaults.PolarArea`.

Prototype methods

.getSegmentsAtEvent(event)

Calling `getSegmentsAtEvent(event)` on your Chart instance passing an argument of an event, or jQuery event, will return the segment elements that are at that the same position of that event.

```
canvas.onclick = function(evt){
    var activePoints = myPolarAreaChart.getSegmentsAtEvent(evt);
    // => activePoints is an array of segments on the canvas that are at the same
};
```

This functionality may be useful for implementing DOM based tooltips, or triggering custom behaviour in your application.

.update( )

Calling `update()` on your Chart instance will re-render the chart with any updated values, allowing you to edit the value of multiple existing points, then render those in one animated render loop.

```
myPolarAreaChart.segments[1].value = 10;
// Would update the first dataset's value of 'Green' to be 10
myPolarAreaChart.update();
// Calling update now animates the position of Green from 50 to 10.
```

.addData(segmentData, index)

Calling `addData(segmentData, index)` on your Chart instance passing an object in the same format as in the constructor. There is an option second argument of 'index', this determines at what index the new segment should be inserted into the chart.

```
// An object in the same format as the original data source
myPolarAreaChart.addData({
    value: 130,
    color: "#B48EAD",
    highlight: "#C69CBE",
    label: "Purple"
});
// The new segment will now animate in.
```

.removeData(index)

Calling `removeData(index)` on your Chart instance will remove segment at that particular index. If none is provided, it will default to the last segment.

```
myPolarAreaChart.removeData();
// Other segments will update to fill the empty space left.
```

Pie & Doughnut Charts

Introduction

Pie and doughnut charts are probably the most commonly used chart there are. They are divided into segments, the arc of each segment shows the proportional value of each piece of data.

They are excellent at showing the relational proportions between data.

Pie and doughnut charts are effectively the same class in Chart.js, but have one different default value - their `percentageInnerCutout` . This equates what percentage of the inner should be cut out. This defaults to `0` for pie charts, and `50` for doughnuts.

They are also registered under two aliases in the `Chart` core. Other than their different default

value, and different alias, they are exactly the same.

Example usage

```
// For a pie chart
var myPieChart = new Chart(ctx[0]).Pie(data,options);

// And for a doughnut chart
var myDoughnutChart = new Chart(ctx[1]).Doughnut(data,options);
```

Data structure

```
var data = [
    {
        value: 300,
        color:"#F7464A",
        highlight: "#FF5A5E",
        label: "Red"
    },
    {
        value: 50,
        color: "#46BFBD",
        highlight: "#5AD3D1",
        label: "Green"
    },
    {
        value: 100,
        color: "#FDB45C",
        highlight: "#FFC870",
        label: "Yellow"
    }
]
```

For a pie chart, you must pass in an array of objects with a value and an optional color property. The value attribute should be a number, Chart.js will total all of the numbers and calculate the relative proportion of each. The color attribute should be a string. Similar to CSS, for this string you can use HEX notation, RGB, RGBA or HSL.

Chart options

These are the customisation options specific to Pie & Doughnut charts. These options are merged with the global chart configuration options, and form the options of the chart.

```
{
    //Boolean - Whether we should show a stroke on each segment
    segmentShowStroke : true,

    //String - The colour of each segment stroke
    segmentStrokeColor : "#fff",

    //Number - The width of each segment stroke
    segmentStrokeWidth : 2,

    //Number - The percentage of the chart that we cut out of the middle
```

```
    percentageInnerCutout : 50, // This is 0 for Pie charts

    //Number - Amount of animation steps
    animationSteps : 100,

    //String - Animation easing effect
    animationEasing : "easeOutBounce",

    //Boolean - Whether we animate the rotation of the Doughnut
    animateRotate : true,

    //Boolean - Whether we animate scaling the Doughnut from the centre
    animateScale : false,

    //String - A legend template
    legendTemplate : "<ul class=\"<%=name.toLowerCase()%>-legend\"><% for (var i=0

}
```

You can override these for your `Chart` instance by passing a second argument into the `Doughnut` method as an object with the keys you want to override.

For example, we could have a doughnut chart that animates by scaling out from the centre like so:

```
new Chart(ctx).Doughnut(data, {
    animateScale: true
});
// This will create a chart with all of the default options, merged from the globa
// and the Doughnut chart defaults but this particular instance will have `animate
```

We can also change these default values for each Doughnut type that is created, this object is available at `Chart.defaults.Doughnut`. Pie charts also have a clone of these defaults available to change at `Chart.defaults.Pie`, with the only difference being `percentageInnerCutout` being set to 0.

Prototype methods

.getSegmentsAtEvent(event)

Calling `getSegmentsAtEvent(event)` on your Chart instance passing an argument of an event, or jQuery event, will return the segment elements that are at the same position of that event.

```
canvas.onclick = function(evt){
    var activePoints = myDoughnutChart.getSegmentsAtEvent(evt);
    // => activePoints is an array of segments on the canvas that are at the same
};
```

This functionality may be useful for implementing DOM based tooltips, or triggering custom behaviour in your application.

.update( )

Calling `update()` on your Chart instance will re-render the chart with any updated values, allowing you to edit the value of multiple existing points, then render those in one animated render loop.

```
myDoughnutChart.segments[1].value = 10;
// Would update the first dataset's value of 'Green' to be 10
myDoughnutChart.update();
// Calling update now animates the circumference of the segment 'Green' from 50 to
// and transitions other segment widths
```

.addData(segmentData, index)

Calling `addData(segmentData, index)` on your Chart instance passing an object in the same format as in the constructor. There is an optional second argument of 'index', this determines at what index the new segment should be inserted into the chart.

```
// An object in the same format as the original data source
myDoughnutChart.addData({
    value: 130,
    color: "#B48EAD",
    highlight: "#C69CBE",
    label: "Purple"
});
// The new segment will now animate in.
```

.removeData(index)

Calling `removeData(index)` on your Chart instance will remove segment at that particular index. If none is provided, it will default to the last segment.

```
myDoughnutChart.removeData();
// Other segments will update to fill the empty space left.
```

Advanced usage

Prototype methods

For each chart, there are a set of global prototype methods on the shared `ChartType` which you may find useful. These are available on all charts created with Chart.js, but for the examples, let's use a line chart we've made.

```
// For example:
var myLineChart = new Chart(ctx).Line(data);
```

.clear()

Will clear the chart canvas. Used extensively internally between animation frames, but you might find it useful.

```
// Will clear the canvas that myLineChart is drawn on
myLineChart.clear();
// => returns 'this' for chainability
```

.stop()

Use this to stop any current animation loop. This will pause the chart during any current animation frame. Call `.render()` to re-animate.

```
// Stops the charts animation loop at its current frame
myLineChart.stop();
// => returns 'this' for chainability
```

.resize()

Use this to manually resize the canvas element. This is run each time the browser is resized, but you can call this method manually if you change the size of the canvas nodes container element.

```
// Resizes & redraws to fill its container element
myLineChart.resize();
// => returns 'this' for chainability
```

.destroy()

Use this to destroy any chart instances that are created. This will clean up any references stored to the chart object within Chart.js, along with any associated event listeners attached by Chart.js.

```
// Destroys a specific chart instance
myLineChart.destroy();
```

.toBase64Image()

This returns a base 64 encoded string of the chart in it's current state.

```
myLineChart.toBase64Image();
// => returns png data url of the image on the canvas
```

.generateLegend()

Returns an HTML string of a legend for that chart. The template for this legend is at `legendTemplate` in the chart options.

```
myLineChart.generateLegend();
// => returns HTML string of a legend for this chart
```

External Tooltips

You can enable custom tooltips in the global or chart configuration like so:

```
var myPieChart = new Chart(ctx).Pie(data, {
    customTooltips: function(tooltip) {

        // tooltip will be false if tooltip is not visible or should be hidden
        if (!tooltip) {
            return;
        }

        // Otherwise, tooltip will be an object with all tooltip properties like:

        // tooltip.caretHeight
        // tooltip.caretPadding
        // tooltip.chart
        // tooltip.cornerRadius
        // tooltip.fillColor
        // tooltip.font...
        // tooltip.text
        // tooltip.x
        // tooltip.y
        // etc...

    };
});
```

See files `sample/pie-customTooltips.html` and `sample/line-customTooltips.html` for examples on how to get started.

Writing new chart types

Chart.js 1.0 has been rewritten to provide a platform for developers to create their own custom chart types, and be able to share and utilise them through the Chart.js API.

The format is relatively simple, there are a set of utility helper methods under `Chart.helpers`, including things such as looping over collections, requesting animation frames, and easing equations.

On top of this, there are also some simple base classes of Chart elements, these all extend from `Chart.Element`, and include things such as points, bars and scales.

```
Chart.Type.extend({
    // Passing in a name registers this chart in the Chart namespace
    name: "Scatter",
    // Providing a defaults will also register the deafults in the chart namespace
    defaults : {
        options: "Here",
        available: "at this.options"
    },
    // Initialize is fired when the chart is initialized - Data is passed in as a
    // Config is automatically merged by the core of Chart.js, and is available at
    initialize:  function(data){
        this.chart.ctx // The drawing context for this chart
        this.chart.canvas // the canvas node for this chart
    },
    // Used to draw something on the canvas
    draw: function() {
    }
});

// Now we can create a new instance of our chart, using the Chart.js API
new Chart(ctx).Scatter(data);
// initialize is now run
```

Extending existing chart types

We can also extend existing chart types, and expose them to the API in the same way. Let's say for example, we might want to run some more code when we initialize every Line chart.

```
// Notice now we're extending the particular Line chart type, rather than the base
Chart.types.Line.extend({
    // Passing in a name registers this chart in the Chart namespace in the same w
    name: "LineAlt",
    initialize: function(data){
        console.log('My Line chart extension');
        Chart.types.Line.prototype.initialize.apply(this, arguments);
    }
});

// Creates a line chart in the same way
new Chart(ctx).LineAlt(data);
// but this logs 'My Line chart extension' in the console.
```

Community extensions

- Stacked Bar Chart by @Regaddi
- Error bars (bar and line charts) by @CAYdenberg
- Scatter chart (number & date scales are supported) by @dima117

Creating custom builds

Chart.js uses gulp to build the library into a single JavaScript file. We can use this same build script with custom parameters in order to build a custom version.

Firstly, we need to ensure development dependencies are installed. With node and npm installed, after cloning the Chart.js repo to a local directory, and navigating to that directory in the command line, we can run the following:

```
npm install
npm install -g gulp
```

This will install the local development dependencies for Chart.js, along with a CLI for the JavaScript task runner gulp.

Now, we can run the `gulp build` task, and pass in a comma seperated list of types as an argument to build a custom version of Chart.js with only specified chart types.

Here we will create a version of Chart.js with only Line, Radar and Bar charts included:

```
gulp build --types=Line,Radar,Bar
```

This will output to the `/custom` directory, and write two files, Chart.js, and Chart.min.js with only those chart types included.

Notes

Browser support

Browser support for the canvas element is available in all modern & major mobile browsers (caniuse.com/canvas).

For IE8 & below, I would recommend using the polyfill ExplorerCanvas - available at https://code.google.com/p/explorercanvas/. It falls back to Internet explorer's format VML when canvas support is not available. Example use:

```
<head>
    <!--[if lte IE 8]>
        <script src="excanvas.js"></script>
    <![endif]-->
</head>
```

Usually I would recommend feature detection to choose whether or not to load a polyfill, rather than IE conditional comments, however in this case, VML is a Microsoft proprietary format, so it will only work in IE.

Some important points to note in my experience using ExplorerCanvas as a fallback.

- Initialise charts on load rather than DOMContentReady when using the library, as sometimes a race condition will occur, and it will result in an error when trying to get the 2d context of a canvas.
- New VML DOM elements are being created for each animation frame and there is no hardware acceleration. As a result animation is usually slow and jerky, with flashing text. It is a good idea to dynamically turn off animation based on canvas support. I recommend using the excellent Modernizr to do this.
- When declaring fonts, the library explorercanvas requires the font name to be in single quotes inside the string. For example, instead of your scaleFontFamily property being simply "Arial", explorercanvas support, use "'Arial'" instead. Chart.js does this for default values.

Bugs & issues

Please report these on the GitHub page - at github.com/nnnick/Chart.js. If you could include a link to a simple jsbin or similar to demonstrate the issue, that'd be really helpful.

Contributing

New contributions to the library are welcome, just a couple of guidelines:

- Tabs for indentation, not spaces please.
- Please ensure you're changing the individual files in `/src`, not the concatenated output in the `Chart.js` file in the root of the repo.
- Please check that your code will pass `jshint` code standards, `gulp jshint` will run this for you.
- Please keep pull requests concise, and document new functionality in the relevant `.md` file.

- Consider whether your changes are useful for all users, or if creating a Chart.js extension would be more appropriate.

License

Chart.js is open source and available under the MIT license.