Chart.js

API Documentation

Getting started

Download Chart.js

You can download the latest version of Chart.js on GitHub or just use these Chart.js CDN links. If you download or clone the repository, you must run `gulp build` to generate the dist files. Chart.js no longer comes with prebuilt release versions, so an alternative option to downloading the repo is strongly advised.

Installation

npm

```
npm install chart.js --save
```

bower

```
bower install chart.js --save
```

Selecting the Correct Build

Chart.js provides two different builds that are available for your use. The `Chart.js` and `Chart.min.js` files include Chart.js and the accompanying color parsing library. If this version is used and you require the use of the time axis, Moment.js will need to be included before Chart.js.

The `Chart.bundle.js` and `Chart.bundle.min.js` builds include Moment.js in a single file. This version should be used if you require time axes and want a single file to include, select this version. Do not use this build if your application already includes Moment.js. If you do, Moment.js will be included twice, increasing the page load time and potentially introducing version issues.

Usage

To import Chart.js using an old-school script tag:

```
<script src="Chart.js"></script>
<script>
    var myChart = new Chart({...})
</script>
```

To import Chart.js using an awesome module loader:

```
// Using CommonJS
var Chart = require('src/chart.js')
var myChart = new Chart({...})

// ES6
import Chart from 'src/chart.js'
let myChart = new Chart({...})
```

```
// Using requirejs
require(['path/to/Chartjs'], function(Chart){
 var myChart = new Chart({...})
})
```

Creating a Chart

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

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

```
// Any of the following formats may be used
var ctx = document.getElementById("myChart");
var ctx = document.getElementById("myChart").getContext("2d");
var ctx = $("#myChart");
var ctx = "myChart";
```

Once you have the element or context, you're ready to instantiate a pre-defined chart-type or create your own!

The following example instantiates a bar chart showing the number of votes for different colors and the y-axis starting at 0.

```
<canvas id="myChart" width="400" height="400"></canvas>
<script>
var ctx = document.getElementById("myChart");
var myChart = new Chart(ctx, {
    type: 'bar',
    data: {
        labels: ["Red", "Blue", "Yellow", "Green", "Purple", "Orange"],
        datasets: [{
            label: '# of Votes',
            data: [12, 19, 3, 5, 2, 3],
            backgroundColor: [
                'rgba(255, 99, 132, 0.2)',
                'rgba(54, 162, 235, 0.2)',
                'rgba(255, 206, 86, 0.2)',
                'rgba(75, 192, 192, 0.2)',
                'rgba(153, 102, 255, 0.2)',
                'rgba(255, 159, 64, 0.2)'
            ],
            borderColor: [
                'rgba(255,99,132,1)',
                'rgba(54, 162, 235, 1)',
                'rgba(255, 206, 86, 1)',
                'rgba(75, 192, 192, 1)',
                'rgba(153, 102, 255, 1)',
                'rgba(255, 159, 64, 1)'
            ],
            borderWidth: 1
        }]
    },
    options: {
        scales: {
            yAxes: [{
                ticks: {
                    beginAtZero:true
                }
            }]
        }
    }
});
</script>
```

It's that easy to get started using Chart.js! From here you can explore the many options that can help you customise your charts with scales, tooltips, labels, colors, custom actions, and much more.

There are many examples of Chart.js that are available in the `/samples` folder of `Chart.js.zip` that is attatched to every release.

Chart Configuration

Chart.js provides a number of options for changing the behaviour of created charts. These configuration options can be changed on a per chart basis by passing in an options object when creating the chart. Alternatively, the global configuration can be changed which will be used by all charts created after that point.

Chart Data

To display data, the chart must be passed a data object that contains all of the information needed by the chart. The data object can contain the following parameters

Name	Type	Description
datasets	Array[object]	Contains data for each dataset. See the documentation for each chart type to determine the valid options that can be attached to the dataset
labels	Array[string]	Optional parameter that is used with the category axis.
xLabels	Array[string]	Optional parameter that is used with the category axis and is used if the axis is horizontal
yLabels	Array[string]	Optional parameter that is used with the category axis and is used if the axis is vertical

Creating a Chart with Options

To create a chart with configuration options, simply pass an object containing your configuration to the constructor. In the example below, a line chart is created and configured to not be responsive.

```
var chartInstance = new Chart(ctx, {
    type: 'line',
    data: data,
    options: {
        responsive: false
    }
});
```

Global 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.js merges the options object passed to the chart with the global configuration using chart type defaults and scales defaults appropriately. This way you can be as specific as you would like in your individual chart configuration, while still changing the defaults for all chart types where applicable. The global general options are defined in `Chart.defaults.global`. The defaults for each chart type are discussed in the documentation for that chart type.

The following example would set the hover mode to 'nearest' for all charts where this was not overridden by the chart type defaults or the options passed to the constructor on creation.

```
Chart.defaults.global.hover.mode = 'nearest';

// Hover mode is set to nearest because it was not overridden here
var chartInstanceHoverModeNearest = new Chart(ctx, {
    type: 'line',
    data: data,
});
```

```
// This chart would have the hover mode that was passed in
var chartInstanceDifferentHoverMode = new Chart(ctx, {
    type: 'line',
    data: data,
    options: {
        hover: {
            // Overrides the global setting
            mode: 'index'
        }
    }
})
```

Global Font Settings

There are 4 special global settings that can change all of the fonts on the chart. These options are in `Chart.defaults.global`.

Name	Type	Default	Description
defaultFontColor	Color	'#666'	Default font color for all text
defaultFontFamily	String	"'Helvetica Neue', 'Helvetica', 'Arial', sans-serif"	Default font family for all text
defaultFontSize	Number	12	Default font size (in px) for text. Does not apply to radialLinear scale point labels
defaultFontStyle	String	'normal'	Default font style. Does not apply to tooltip title or footer. Does not apply to chart title

Common Chart Configuration

The following options are applicable to all charts. The can be set on the global configuration, or they can be passed to the chart constructor.

Name	Type	Default	Description
responsive	Boolean	true	Resizes the chart canvas when its container does.
responsiveAnimationDuration	Number	0	Duration in milliseconds it takes to animate to new size after a resize event.
maintainAspectRatio	Boolean	true	Maintain the original canvas aspect ratio `(width / height)` when resizing
events	Array[String]	`["mousemove", "mouseout", "click", "touchstart", "touchmove", "touchend"]`	Events that the chart should listen to for tooltips and hovering
onClick	Function	null	Called if the event is of type 'mouseup' or 'click'. Called in the context of the chart and passed an array of active elements

Name	Type	Default	Description
legendCallback	Function	`function (chart) { }`	Function to generate a legend. Receives the chart object to generate a legend from. Default implementation returns an HTML string.
onResize	Function	null	Called when a resize occurs. Gets passed two arguments: the chart instance and the new size.

Layout Configuration

The layout configuration is passed into the `options.layout` namespace. The global options for the chart layout is defined in `Chart.defaults.global.layout`.

Name	Type	Default	Description
padding	Number or Object	0	The padding to add inside the chart. If this value is a number, it is applied to all sides of the chart (left, top, right, bottom). If this value is an object, the `left` property defines the left padding. Similarly the `right`, `top`, and `bottom` properties can also be specified.

Title Configuration

The title configuration is passed into the `options.title` namespace. The global options for the chart title is defined in `Chart.defaults.global.title`.

Name	Type	Default	Description
display	Boolean	false	Display the title block
position	String	'top'	Position of the title. Possible values are 'top', 'left', 'bottom' and 'right'.
fullWidth	Boolean	true	Marks that this box should take the full width of the canvas (pushing down other boxes)
fontSize	Number	12	Font size inherited from global configuration
fontFamily	String	"'Helvetica Neue', 'Helvetica', 'Arial', sans-serif"	Font family inherited from global configuration
fontColor	Color	"#666"	Font color inherited from global configuration
fontStyle	String	'bold'	Font styling of the title.
padding	Number	10	Number of pixels to add above and below the title text
text	String	''	Title text

Example Usage

The example below would enable a title of 'Custom Chart Title' on the chart that is created.

```
var chartInstance = new Chart(ctx, {
    type: 'line',
    data: data,
    options: {
        title: {
            display: true,
            text: 'Custom Chart Title'
        }
    }
})
```

Legend Configuration

The legend configuration is passed into the **options.legend** namespace. The global options for the chart legend is defined in **Chart.defaults.global.legend**.

Name	Type	Default	Description
display	Boolean	true	Is the legend displayed
position	String	'top'	Position of the legend. Possible values are 'top', 'left', 'bottom' and 'right'.
fullWidth	Boolean	true	Marks that this box should take the full width of the canvas (pushing down other boxes)
onClick	Function	`function(event, legendItem) {}`	A callback that is called when a 'click' event is registered on top of a label item
onHover	Function	`function(event, legendItem) {}`	A callback that is called when a 'mousemove' event is registered on top of a label item
labels	Object	-	See the Legend Label Configuration section below.
reverse	Boolean	false	Legend will show datasets in reverse order

Legend Label Configuration

The legend label configuration is nested below the legend configuration using the **labels** key.

Name	Type	Default	Description
boxWidth	Number	40	Width of coloured box
fontSize	Number	12	Font size inherited from global configuration
fontStyle	String	"normal"	Font style inherited from global configuration
fontColor	Color	"#666"	Font color inherited from global configuration
fontFamily	String	"'Helvetica Neue', 'Helvetica', 'Arial', sans-serif"	Font family inherited from global configuration
padding	Number	10	Padding between rows of colored boxes
generateLabels:	Function	`function(chart) { }`	Generates legend items for each thing in the legend. Default implementation returns the text + styling for the color box. See Legend Item for details.

Name	Type	Default	Description
usePointStyle	Boolean	false	Label style will match corresponding point style (size is based on fontSize, boxWidth is not used in this case).

Legend Item Interface

Items passed to the legend `onClick` function are the ones returned from `labels.generateLabels`. These items must implement the following interface.

```
{
    // Label that will be displayed
    text: String,

    // Fill style of the legend box
    fillStyle: Color,

    // If true, this item represents a hidden dataset. Label will be rendered with
    hidden: Boolean,

    // For box border. See https://developer.mozilla.org/en/docs/Web/API/CanvasRend
    lineCap: String,

    // For box border. See https://developer.mozilla.org/en-US/docs/Web/API/CanvasR
    lineDash: Array[Number],

    // For box border. See https://developer.mozilla.org/en-US/docs/Web/API/CanvasR
    lineDashOffset: Number,

    // For box border. See https://developer.mozilla.org/en-US/docs/Web/API/CanvasR
    lineJoin: String,

    // Width of box border
    lineWidth: Number,

    // Stroke style of the legend box
    strokeStyle: Color

    // Point style of the legend box (only used if usePointStyle is true)
    pointStyle: String
}
```

Example

The following example will create a chart with the legend enabled and turn all of the text red in color.

```
var chartInstance = new Chart(ctx, {
    type: 'bar',
    data: data,
    options: {
        legend: {
            display: true,
            labels: {
                fontColor: 'rgb(255, 99, 132)'
            }
        }
}
});
```

Tooltip Configuration

The tooltip configuration is passed into the `options.tooltips` namespace. The global options for the chart tooltips is defined in `Chart.defaults.global.tooltips`.

Name	Type	Default	Description
enabled	Boolean	true	Are tooltips enabled
custom	Function	null	See section below
mode	String	'nearest'	Sets which elements appear in the tooltip. See Interaction Modes for details
intersect	Boolean	true	if true, the tooltip mode applies only when the mouse position intersects with an element. If false, the mode will be applied at all times.
position	String	'average'	The mode for positioning the tooltip. 'average' mode will place the tooltip at the average position of the items displayed in the tooltip. 'nearest' will place the tooltip at the position of the element closest to the event position. New modes can be defined by adding functions to the Chart.Tooltip.positioners map.
itemSort	Function	undefined	Allows sorting of tooltip items. Must implement at minimum a function that can be passed to Array.prototype.sort. This function can also accept a third parameter that is the data object passed to the chart.
filter	Function	undefined	Allows filtering of tooltip items. Must implement at minimum a function that can be passed to Array.prototype.filter. This function can also accept a second parameter that is the data object passed to the chart.
backgroundColor	Color	'rgba(0,0,0,0.8)'	Background color of the tooltip
titleFontFamily	String	"'Helvetica Neue', 'Helvetica', 'Arial', sans-serif"	Font family for tooltip title inherited from global font family
titleFontSize	Number	12	Font size for tooltip title inherited from global font size
titleFontStyle	String	"bold"	
titleFontColor	Color	"#fff"	Font color for tooltip title
titleSpacing	Number	2	Spacing to add to top and bottom of each title line.
titleMarginBottom	Number	6	Margin to add on bottom of title section
bodyFontFamily	String	"'Helvetica Neue', 'Helvetica', 'Arial', sans-serif"	Font family for tooltip items inherited from global font family
bodyFontSize	Number	12	Font size for tooltip items inherited from global font size
bodyFontStyle	String	"normal"	
bodyFontColor	Color	"#fff"	Font color for tooltip items.

Name	Type	Default	Description
bodySpacing	Number	2	Spacing to add to top and bottom of each tooltip item
footerFontFamily	String	"'Helvetica Neue', 'Helvetica', 'Arial', sans-serif"	Font family for tooltip footer inherited from global font family.
footerFontSize	Number	12	Font size for tooltip footer inherited from global font size.
footerFontStyle	String	"bold"	Font style for tooltip footer.
footerFontColor	Color	"#fff"	Font color for tooltip footer.
footerSpacing	Number	2	Spacing to add to top and bottom of each footer line.
footerMarginTop	Number	6	Margin to add before drawing the footer
xPadding	Number	6	Padding to add on left and right of tooltip
yPadding	Number	6	Padding to add on top and bottom of tooltip
caretSize	Number	5	Size, in px, of the tooltip arrow
cornerRadius	Number	6	Radius of tooltip corner curves
multiKeyBackground	Color	"#fff"	Color to draw behind the colored boxes when multiple items are in the tooltip
displayColors	Boolean	true	if true, color boxes are shown in the tooltip
callbacks	Object		See the callbacks section below

Tooltip Callbacks

The tooltip label configuration is nested below the tooltip configuration using the `callbacks` key. The tooltip has the following callbacks for providing text. For all functions, 'this' will be the tooltip object created from the Chart.Tooltip constructor.

All functions are called with the same arguments: a tooltip item and the data object passed to the chart. All functions must return either a string or an array of strings. Arrays of strings are treated as multiple lines of text.

Callback	Arguments	Description
beforeTitle	`Array[tooltipItem], data`	Text to render before the title
title	`Array[tooltipItem], data`	Text to render as the title
afterTitle	`Array[tooltipItem], data`	Text to render after the title
beforeBody	`Array[tooltipItem], data`	Text to render before the body section
beforeLabel	`tooltipItem, data`	Text to render before an individual label

Callback	Arguments	Description
label	`tooltipItem, data`	Text to render for an individual item in the tooltip
labelColor	`tooltipItem, chartInstance`	Returns the colors to render for the tooltip item. Return as an object containing two parameters: `borderColor` and `backgroundColor`.
afterLabel	`tooltipItem, data`	Text to render after an individual label
afterBody	`Array[tooltipItem], data`	Text to render after the body section
beforeFooter	`Array[tooltipItem], data`	Text to render before the footer section
footer	`Array[tooltipItem], data`	Text to render as the footer
afterFooter	`Array[tooltipItem], data`	Text to render after the footer section
dataPoints	`Array[tooltipItem]`	List of matching point informations.

Tooltip Item Interface

The tooltip items passed to the tooltip callbacks implement the following interface.

```
{
    // X Value of the tooltip as a string
    xLabel: String,

    // Y value of the tooltip as a string
    yLabel: String,

    // Index of the dataset the item comes from
    datasetIndex: Number,

    // Index of this data item in the dataset
    index: Number,

    // X position of matching point
    x: Number,

    // Y position of matching point
    y: Number,
}
```

Hover Configuration

The hover configuration is passed into the `options.hover` namespace. The global hover configuration is at `Chart.defaults.global.hover`.

Name	Type	Default	Description
mode	String	'nearest'	Sets which elements appear in the tooltip. See Interaction Modes for details
intersect	Boolean	true	if true, the hover mode only applies when the mouse position intersects an item on the chart
animationDuration	Number	400	Duration in milliseconds it takes to animate hover style changes

Name	Type	Default	Description
onHover	Function	null	Called when any of the events fire. Called in the context of the chart and passed an array of active elements (bars, points, etc)

Interaction Modes

When configuring interaction with the graph via hover or tooltips, a number of different modes are available.

The following table details the modes and how they behave in conjunction with the `intersect` setting

Mode	Behaviour
point	Finds all of the items that intersect the point
nearest	Gets the item that is nearest to the point. The nearest item is determined based on the distance to the center of the chart item (point, bar). If 2 or more items are at the same distance, the one with the smallest area is used. If `intersect` is true, this is only triggered when the mouse position intersects an item in the graph. This is very useful for combo charts where points are hidden behind bars.
single (deprecated)	Finds the first item that intersects the point and returns it. Behaves like 'nearest' mode with intersect = true.
label (deprecated)	See `'index'` mode
index	Finds item at the same index. If the `intersect` setting is true, the first intersecting item is used to determine the index in the data. If `intersect` false the nearest item is used to determine the index.
x-axis (deprecated)	Behaves like `'index'` mode with `intersect = true`
dataset	Finds items in the same dataset. If the `intersect` setting is true, the first intersecting item is used to determine the index in the data. If `intersect` false the nearest item is used to determine the index.
x	Returns all items that would intersect based on the X coordinate of the position only. Would be useful for a vertical cursor implementation. Note that this only applies to cartesian charts
y	Returns all items that would intersect based on the Y coordinate of the position. This would be useful for a horizontal cursor implementation. Note that this only applies to cartesian charts.

Animation Configuration

The following animation options are available. The global options for are defined in `Chart.defaults.global.animation`.

Name	Type	Default	Description
duration	Number	1000	The number of milliseconds an animation takes.

Name	Type	Default	Description
easing	String	"easeOutQuart"	Easing function to use. Available options are: `'linear'`, `'easeInQuad'`, `'easeOutQuad'`, `'easeInOutQuad'`, `'easeInCubic'`, `'easeOutCubic'`, `'easeInOutCubic'`, `'easeInQuart'`, `'easeOutQuart'`, `'easeInOutQuart'`, `'easeInQuint'`, `'easeOutQuint'`, `'easeInOutQuint'`, `'easeInSine'`, `'easeOutSine'`, `'easeInOutSine'`, `'easeInExpo'`, `'easeOutExpo'`, `'easeInOutExpo'`, `'easeInCirc'`, `'easeOutCirc'`, `'easeInOutCirc'`, `'easeInElastic'`, `'easeOutElastic'`, `'easeInOutElastic'`, `'easeInBack'`, `'easeOutBack'`, `'easeInOutBack'`, `'easeInBounce'`, `'easeOutBounce'`, `'easeInOutBounce'`. See Robert Penner's easing equations.
onProgress	Function	none	Callback called on each step of an animation. Passed a single argument, an object, containing the chart instance and an object with details of the animation.
onComplete	Function	none	Callback called at the end of an animation. Passed the same arguments as `onProgress`

Animation Callbacks

The `onProgress` and `onComplete` callbacks are useful for synchronizing an external draw to the chart animation. The callback is passed an object that implements the following interface. An example usage of these callbacks can be found on Github. This sample displays a progress bar showing how far along the animation is.

```
{
    // Chart object
    chartInstance,

    // Contains details of the on-going animation
    animationObject,
}
```

Animation Object

The animation object passed to the callbacks is of type `Chart.Animation`. The object has the following parameters.

```
{
    // Current Animation frame number
    currentStep: Number,

    // Number of animation frames
    numSteps: Number,

    // Animation easing to use
    easing: String,

    // Function that renders the chart
    render: Function,

    // User callback
    onAnimationProgress: Function,

    // User callback
    onAnimationComplete: Function
}
```

Element Configuration

The global options for elements are defined in `Chart.defaults.global.elements`.

Options can be configured for four different types of elements: arc, lines, points, and rectangles. When set, these options apply to all objects of that type unless specifically overridden by the configuration attached to a dataset.

Arc Configuration

Arcs are used in the polar area, doughnut and pie charts. They can be configured with the following options. The global arc options are stored in `Chart.defaults.global.elements.arc`.

Name	Type	Default	Description
backgroundColor	Color	'rgba(0,0,0,0.1)'	Default fill color for arcs. Inherited from the global default
borderColor	Color	'#fff'	Default stroke color for arcs
borderWidth	Number	2	Default stroke width for arcs

Line Configuration

Line elements are used to represent the line in a line chart. The global line options are stored in `Chart.defaults.global.elements.line`.

Name	Type	Default	Description
tension	Number	0.4	Default bezier curve tension. Set to `0` for no bezier curves.
backgroundColor	Color	'rgba(0,0,0,0.1)'	Default line fill color
borderWidth	Number	3	Default line stroke width
borderColor	Color	'rgba(0,0,0,0.1)'	Default line stroke color
borderCapStyle	String	'butt'	Default line cap style. See MDN
borderDash	Array	`[]`	Default line dash. See MDN
borderDashOffset	Number	0.0	Default line dash offset. See MDN
borderJoinStyle	String	'miter'	Default line join style. See MDN
capBezierPoints	Boolean	true	If true, bezier control points are kept inside the chart. If false, no restriction is enforced.
fill	Boolean or String	true	If true, the fill is assumed to be to zero. String values are 'zero', 'top', and 'bottom' to fill to different locations. If `false`, no fill is added
stepped	Boolean	false	If true, the line is shown as a stepped line and 'tension' will be ignored

Point Configuration

Point elements are used to represent the points in a line chart or a bubble chart. The global point options are stored in `Chart.defaults.global.elements.point`.

Name	Type	Default	Description

radius	Number	3	Default point radius
pointStyle	String	'circle'	Default point style
backgroundColor	Color	'rgba(0,0,0,0.1)'	Default point fill color
borderWidth	Number	1	Default point stroke width
borderColor	Color	'rgba(0,0,0,0.1)'	Default point stroke color
hitRadius	Number	1	Extra radius added to point radius for hit detection
hoverRadius	Number	4	Default point radius when hovered
hoverBorderWidth	Number	1	Default stroke width when hovered

Rectangle Configuration

Rectangle elements are used to represent the bars in a bar chart. The global rectangle options are stored in `Chart.defaults.global.elements.rectangle`.

Name	Type	Default	Description
backgroundColor	Color	'rgba(0,0,0,0.1)'	Default bar fill color
borderWidth	Number	0	Default bar stroke width
borderColor	Color	'rgba(0,0,0,0.1)'	Default bar stroke color
borderSkipped	String	'bottom'	Default skipped (excluded) border for rectangle. Can be one of `bottom`, `left`, `top`, `right`

Colors

When supplying colors to Chart options, you can use a number of formats. You can specify the color as a string in hexadecimal, RGB, or HSL notations. If a color is needed, but not specified, Chart.js will use the global default color. This color is stored at `Chart.defaults.global.defaultColor`. It is initially set to 'rgba(0, 0, 0, 0.1)';

You can also pass a CanvasGradient object. You will need to create this before passing to the chart, but using it you can achieve some interesting effects.

Patterns

An alternative option is to pass a CanvasPattern object. For example, if you wanted to fill a dataset with a pattern from an image you could do the following.

```
var img = new Image();
img.src = 'https://example.com/my_image.png';
img.onload = function() {
    var ctx = document.getElementById('canvas').getContext('2d');
    var fillPattern = ctx.createPattern(img, 'repeat');

    var chart = new Chart(ctx, {
        data: {
            labels: ['Item 1', 'Item 2', 'Item 3'],
            datasets: [{
                data: [10, 20, 30],
                backgroundColor: fillPattern
            }]
        }
    })
}
```

Using pattern fills for data graphics can help viewers with vision deficiencies (e.g. color-blindness or partial sight) to more easily understand your data.

Using the Patternomaly library you can generate patterns to fill datasets.

```
var chartData = {
    datasets: [{
        data: [45, 25, 20, 10],
        backgroundColor: [
            pattern.draw('square', '#ff6384'),
            pattern.draw('circle', '#36a2eb'),
            pattern.draw('diamond', '#cc65fe'),
            pattern.draw('triangle', '#ffce56'),
        ]
    }],
    labels: ['Red', 'Blue', 'Purple', 'Yellow']
};
```

Mixed Chart Types

When creating a chart, you have the option to overlay different chart types on top of each other as separate datasets.

To do this, you must set a `type` for each dataset individually. You can create mixed chart types with bar and line chart types.

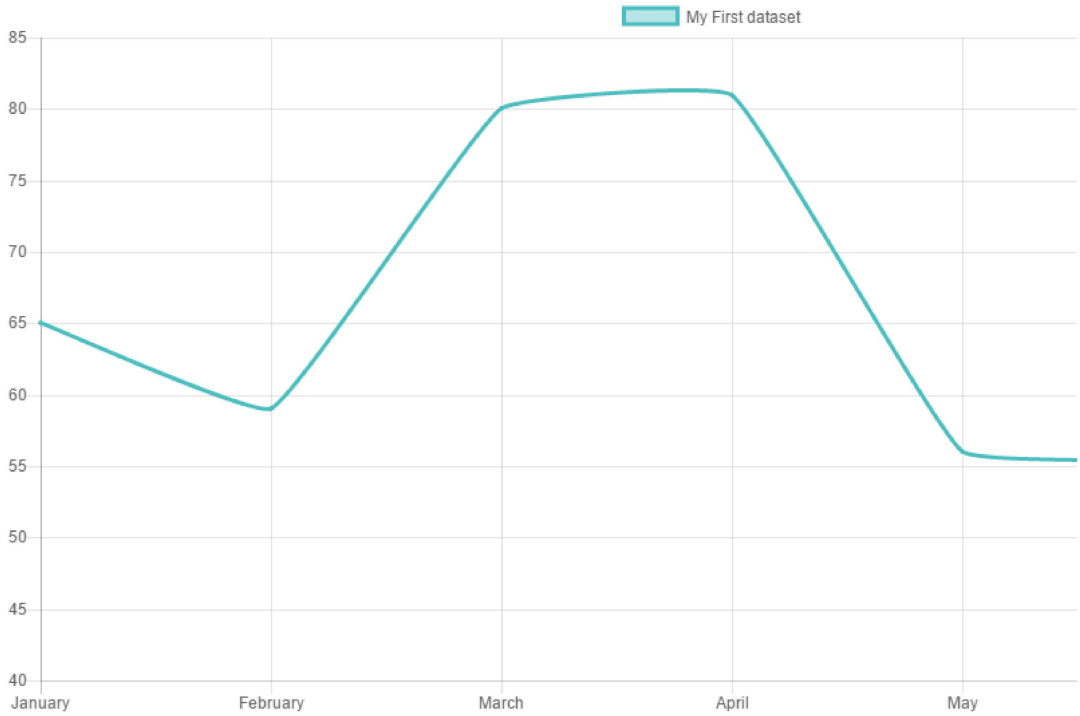
When creating the chart you must set the overall `type` as `bar`.

```
var myChart = new Chart(ctx, {
    type: 'bar',
    data: {
        labels: ['Item 1', 'Item 2', 'Item 3'],
        datasets: [
            {
                type: 'bar',
                label: 'Bar Component',
                data: [10, 20, 30],
            },
            {
                type: 'line',
                label: 'Line Component',
                data: [30, 20, 10],
            }
        ]
    }
});
```

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, {
    type: 'line',
    data: data,
    options: options
});
```

Alternatively a line chart can be created using syntax similar to the v1.0 syntax

```
var myLineChart = Chart.Line(ctx, {
    data: data,
    options: options
});
```

Dataset Structure

The following options can be included in a line chart dataset to configure options for that specific dataset.

All point* properties can be specified as an array. If these are set to an array value, the first value applies to the first point, the second value to the second point, and so on.

Property	Type	Usage
data	See data point section	The data to plot in a line
label	`String`	The label for the dataset which appears in the legend and tooltips
xAxisID	`String`	The ID of the x axis to plot this dataset on
yAxisID	`String`	The ID of the y axis to plot this dataset on
fill	`Boolean`	If true, fill the area under the line

Property	Type	Usage
cubicInterpolationMode	`String`	Algorithm used to interpolate a smooth curve from th discrete data points. Options are 'default' and 'monotone'. The 'default' algorithm uses a custom weighted cubic interpolation, which produces pleasa curves for all types of datasets. The 'monotone' algorithm is more suited to `y = f(x)` datasets : it preserves monotonicity (or piecewise monotonicity) c the dataset being interpolated, and ensures local extremums (if any) stay at input data points. If left untouched (`undefined`), the global `options.elements.line.cubicInterpolationM` property is used.
lineTension	`Number`	Bezier curve tension of the line. Set to 0 to draw straightlines. This option is ignored if monotone cubi interpolation is used. *Note* This was renamed from 'tension' but the old name still works.
backgroundColor	`Color`	The fill color under the line. See Colors
borderWidth	`Number`	The width of the line in pixels
borderColor	`Color`	The color of the line.
borderCapStyle	`String`	Cap style of the line. See MDN
borderDash	`Array<Number>`	Length and spacing of dashes. See MDN
borderDashOffset	`Number`	Offset for line dashes. See MDN
borderJoinStyle	`String`	Line joint style. See MDN
pointBorderColor	`Color or Array<Color>`	The border color for points.
pointBackgroundColor	`Color or Array<Color>`	The fill color for points
pointBorderWidth	`Number or Array<Number>`	The width of the point border in pixels
pointRadius	`Number or Array<Number>`	The radius of the point shape. If set to 0, nothing is rendered.
pointHoverRadius	`Number or Array<Number>`	The radius of the point when hovered
pointHitRadius	`Number or Array<Number>`	The pixel size of the non-displayed point that reacts t mouse events
pointHoverBackgroundColor	`Color or Array<Color>`	Point background color when hovered
pointHoverBorderColor	`Color or Array<Color>`	Point border color when hovered
pointHoverBorderWidth	`Number or Array<Number>`	Border width of point when hovered
pointStyle	`String, Array<String>, Image, Array<Image>`	The style of point. Options are 'circle', 'triangle', 'rect', 'rectRot', 'cross', 'crossRot', 'star', 'line', and 'dash'. If the option is an image, that image is drawn on the canva using `drawImage`.
showLine	`Boolean`	If false, the line is not drawn for this dataset

Property	Type	Usage
spanGaps	`Boolean`	If true, lines will be drawn between points with no or data
steppedLine	`Boolean`	If true, the line is shown as a stepped line and 'lineTension' will be ignored

An example data object using these attributes is shown below.

```
var data = {
    labels: ["January", "February", "March", "April", "May", "June", "July"],
    datasets: [
        {
            label: "My First dataset",
            fill: false,
            lineTension: 0.1,
            backgroundColor: "rgba(75,192,192,0.4)",
            borderColor: "rgba(75,192,192,1)",
            borderCapStyle: 'butt',
            borderDash: [],
            borderDashOffset: 0.0,
            borderJoinStyle: 'miter',
            pointBorderColor: "rgba(75,192,192,1)",
            pointBackgroundColor: "#fff",
            pointBorderWidth: 1,
            pointHoverRadius: 5,
            pointHoverBackgroundColor: "rgba(75,192,192,1)",
            pointHoverBorderColor: "rgba(220,220,220,1)",
            pointHoverBorderWidth: 2,
            pointRadius: 1,
            pointHitRadius: 10,
            data: [65, 59, 80, 81, 56, 55, 40],
            spanGaps: false,
        }
    ]
};
```

The line chart usually requires an array of labels. This labels are shown on the X axis. There must be one label for each data point. More labels than datapoints are allowed, in which case the line ends at the last data point. 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.

When `spanGaps` is set to true, the gaps between points in sparse datasets are filled in. By default, it is off.

Data Points

The data passed to the chart can be passed in two formats. The most common method is to pass the data array as an array of numbers. In this case, the `data.labels` array must be specified and must contain a label for each point or, in the case of labels to be displayed over multiple lines an array of labels (one for each line) i.e `[["June","2015"], "July"]`.

The alternate is used for sparse datasets. Data is specified using an object containing `x` and `y` properties. This is used for scatter charts as documented below.

Scatter Line Charts

Scatter line charts can be created by changing the X axis to a linear axis. To use a scatter chart, data must be passed as objects containing X and Y properties. The example below creates a scatter chart with 3 points.

```
var scatterChart = new Chart(ctx, {
    type: 'line',
    data: {
```

```
        datasets: [{
            label: 'Scatter Dataset',
            data: [{
                x: -10,
                y: 0
            }, {
                x: 0,
                y: 10
            }, {
                x: 10,
                y: 5
            }]
        }]
    },
    options: {
        scales: {
            xAxes: [{
                type: 'linear',
                position: 'bottom'
            }]
        }
    }
});
```

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.

Name	Type	Default	Description
showLines	Boolean	true	If false, the lines between points are not drawn
spanGaps	Boolean	false	If true, NaN data does not break the line

You can override these for your `Chart` instance by passing a member `options` into the `Line` method.

For example, we could have a line chart display without an X axis by doing the following. The config merge is smart enough to handle arrays so that you do not need to specify all axis settings to change one thing.

```
new Chart(ctx, {
    type: 'line',
    data: data,
    options: {
        scales: {
            xAxes: [{
                display: false
            }]
        }
    }
});
```

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

Stacked Charts

Stacked area charts can be created by setting the Y axis to a stacked configuration. The following example would have stacked lines.

```
var stackedLine = new Chart(ctx, {
    type: 'line',
    data: data,
    options: {
        scales: {
            yAxes: [{
```

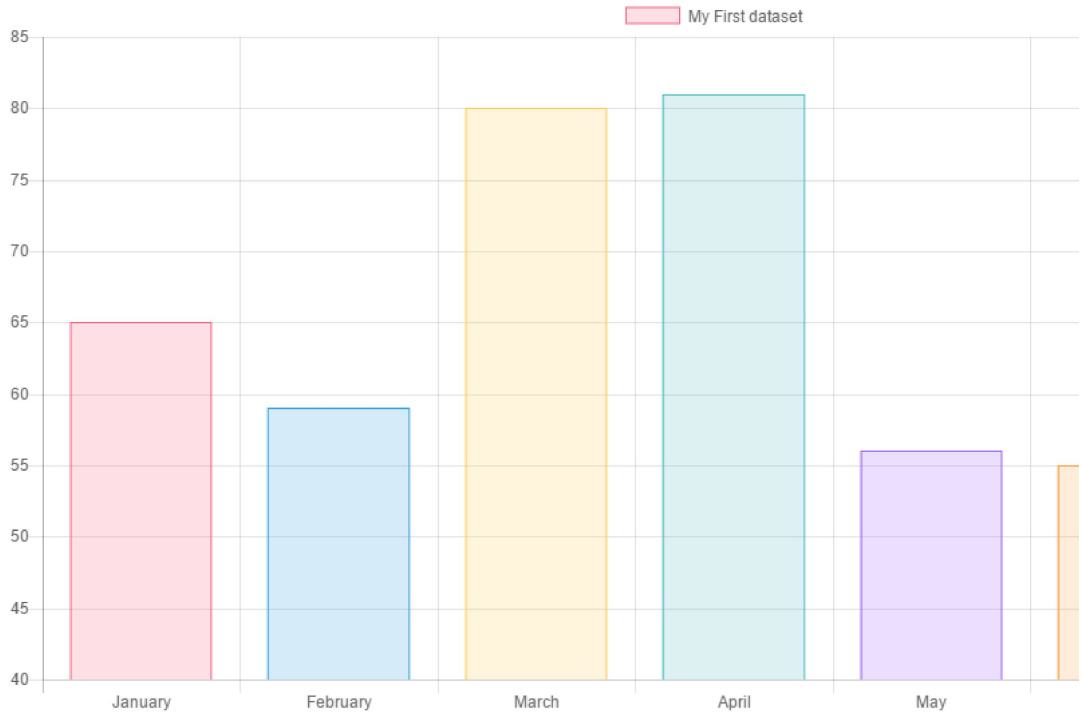
```
                stacked: true
            }]
        }
    }
});
```

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, {
    type: 'bar',
    data: data,
    options: options
});
```

Or if you want horizontal bars.

```
var myBarChart = new Chart(ctx, {
    type: 'horizontalBar',
    data: data,
    options: options
});
```

Dataset Structure

The following options can be included in a bar chart dataset to configure options for that specific dataset.

Some properties can be specified as an array. If these are set to an array value, the first value applies to the first bar, the second value to the second bar, and so on.

Property	Type	Usage
data	`Array<Number>`	The data to plot as bars
label	`String`	The label for the dataset which appears in the legend and tooltips
xAxisID	`String`	The ID of the x axis to plot this dataset on
yAxisID	`String`	The ID of the y axis to plot this dataset on
backgroundColor	`Color or Array<Color>`	The fill color of the bars. See Colors
borderColor	`Color or Array<Color>`	Bar border color
borderWidth	`Number or Array<Number>`	Border width of bar in pixels
borderSkipped	`String or Array<String>`	Which edge to skip drawing the border for. Options are 'bottom', 'left', 'top', and 'right'
hoverBackgroundColor	`Color or Array<Color>`	Bar background color when hovered
hoverBorderColor	`Color or Array<Color>`	Bar border color when hovered
hoverBorderWidth	`Number or Array<Number>`	Border width of bar when hovered

An example data object using these attributes is shown below.

```
var data = {
    labels: ["January", "February", "March", "April", "May", "June", "July"],
    datasets: [
        {
            label: "My First dataset",
            backgroundColor: [
                'rgba(255, 99, 132, 0.2)',
                'rgba(54, 162, 235, 0.2)',
                'rgba(255, 206, 86, 0.2)',
                'rgba(75, 192, 192, 0.2)',
                'rgba(153, 102, 255, 0.2)',
                'rgba(255, 159, 64, 0.2)'
            ],
            borderColor: [
                'rgba(255,99,132,1)',
                'rgba(54, 162, 235, 1)',
                'rgba(255, 206, 86, 1)',
                'rgba(75, 192, 192, 1)',
                'rgba(153, 102, 255, 1)',
                'rgba(255, 159, 64, 1)'
            ],
            borderWidth: 1,
            data: [65, 59, 80, 81, 56, 55, 40],
        }
    ]
};
```

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. We have an array of labels too for display. In the example, we are showing the same data as the previous line chart example.

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.

The default options for bar chart are defined in `Chart.defaults.bar`.

Name	Type	Default	Description
hover.mode	String	"label"	Label's hover mode. "label" is used since the x axis displays data by the index in the dataset.
scales	Object	-	-
scales.xAxes	Array		The bar chart officially supports only 1 x-axis but uses an array to keep the API consistent. Use a scatter chart if you need multiple x axes.
Options for xAxes			
type	String	"Category"	As defined in Scales.
display	Boolean	true	If true, show the scale.
id	String	"x-axis-0"	Id of the axis so that data can bind to it
stacked	Boolean	false	If true, bars are stacked on the x-axis
barThickness	Number		Manually set width of each bar in pixels. If not set, the bars are sized automatically.
categoryPercentage	Number	0.8	Percent (0-1) of the available width (the space between the gridlines for small datasets) for each data-point to use for the bars. Read More
barPercentage	Number	0.9	Percent (0-1) of the available width each bar should be within the category percentage. 1.0 will take the whole category width and put the bars right next to each other. Read More
gridLines	Object	See Scales	
gridLines.offsetGridLines	Boolean	true	If true, the bars for a particular data point fall between the grid lines. If false, the grid line will go right down the middle of the bars.
scales.yAxes	Array	`[{ type: "linear" }]`	
Options for yAxes			
type	String	"linear"	As defined in Scales.
display	Boolean	true	If true, show the scale.
id	String	"y-axis-0"	Id of the axis so that data can bind to it.
stacked	Boolean	false	If true, bars are stacked on the y-axis
barThickness	Number		Manually set height of each bar in pixels. If not set, the bars are sized automatically.

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, {
    type: "bar",
    data: data,
    options: {
        scales: {
            xAxes: [{
                stacked: true
            }],
            yAxes: [{
                stacked: true
            }]
        }
    }
});
// This will create a chart with all of the default options, merged from the global
//  and the Bar chart defaults but this particular instance will have `stacked` set
// for both x and y axes.
```

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

The default options for horizontal bar charts are defined in `Chart.defaults.horizontalBar` and are same as those of the bar chart, but with `xAxes` and `yAxes` swapped and the following additional options.

Name	Type	Default	Description
Options for xAxes			
position	String	"bottom"	
Options for yAxes			
position	String	"left"	

barPercentage vs categoryPercentage

The following shows the relationship between the bar percentage option and the category percentage option.

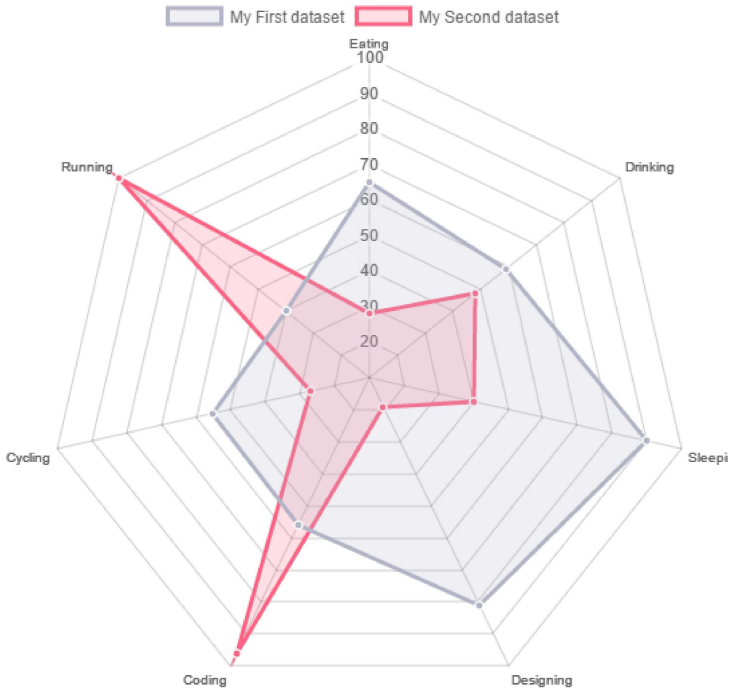
```
// categoryPercentage: 1.0
// barPercentage: 1.0
Bar:        | 1.0 | 1.0 |
Category:   |    1.0    |
Sample:     |===========|

// categoryPercentage: 1.0
// barPercentage: 0.5
Bar:          |.5|  |.5|
Category:  |      1.0     |
Sample:    |==============|

// categoryPercentage: 0.5
// barPercentage: 1.0
Bar:            |1.||1.|
Category:       |  .5  |
Sample:     |==============|
```

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, {
    type: 'radar',
    data: data,
    options: options
});
```

Dataset Structure

The following options can be included in a radar chart dataset to configure options for that specific dataset.

All point* properties can be specified as an array. If these are set to an array value, the first value applies to the first point, the second value to the second point, and so on.

Property	Type	Usage
data	`Array<Number>`	The data to plot in a line
label	`String`	The label for the dataset which appears in the legend and tooltips
fill	`Boolean`	If true, fill the area under the line
lineTension	`Number`	Bezier curve tension of the line. Set to 0 to draw straightlines. *Note* This was renamed from 'tension' but the old name still works.
backgroundColor	`Color`	The fill color under the line. See Colors
borderWidth	`Number`	The width of the line in pixels
borderColor	`Color`	The color of the line.

Property	Type	Usage
borderCapStyle	`String`	Cap style of the line. See MDN
borderDash	`Array<Number>`	Length and spacing of dashes. See MDN
borderDashOffset	`Number`	Offset for line dashes. See MDN
borderJoinStyle	`String`	Line joint style. See MDN
pointBorderColor	`Color or Array<Color>`	The border color for points.
pointBackgroundColor	`Color or Array<Color>`	The fill color for points
pointBorderWidth	`Number or Array<Number>`	The width of the point border in pixels
pointRadius	`Number or Array<Number>`	The radius of the point shape. If set to 0, nothing is rendered.
pointHoverRadius	`Number or Array<Number>`	The radius of the point when hovered
hitRadius	`Number or Array<Number>`	The pixel size of the non-displayed point that reacts to mouse events
pointHoverBackgroundColor	`Color or Array<Color>`	Point background color when hovered
pointHoverBorderColor	`Color or Array<Color>`	Point border color when hovered
pointHoverBorderWidth	`Number or Array<Number>`	Border width of point when hovered
pointStyle	`String or Array<String>`	The style of point. Options include 'circle', 'triangle', 'rect', 'rectRot', 'cross', 'crossRot', 'star', 'line', and 'dash'

An example data object using these attributes is shown below.

```
var data = {
    labels: ["Eating", "Drinking", "Sleeping", "Designing", "Coding", "Cycling", "R
    datasets: [
        {
            label: "My First dataset",
            backgroundColor: "rgba(179,181,198,0.2)",
            borderColor: "rgba(179,181,198,1)",
            pointBackgroundColor: "rgba(179,181,198,1)",
            pointBorderColor: "#fff",
            pointHoverBackgroundColor: "#fff",
            pointHoverBorderColor: "rgba(179,181,198,1)",
            data: [65, 59, 90, 81, 56, 55, 40]
        },
        {
            label: "My Second dataset",
            backgroundColor: "rgba(255,99,132,0.2)",
            borderColor: "rgba(255,99,132,1)",
            pointBackgroundColor: "rgba(255,99,132,1)",
            pointBorderColor: "#fff",
            pointHoverBackgroundColor: "#fff",
            pointHoverBorderColor: "rgba(255,99,132,1)",
            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.

The default options for radar chart are defined in `Chart.defaults.radar`.

Name	Type	Default	Description
scale	Object	See Scales and Defaults for Radial Linear Scale	Options for the one scale used on the chart. Use this to style the ticks, labels, and grid lines.
scale.type	String	"radialLinear"	As defined in "Radial Linear".
elements.line	Object		Options for all line elements used on the chart, as defined in the global elements, duplicated here to show Radar chart specific defaults.
elements.line.lineTension	Number	0	Tension exhibited by lines when calculating splineCurve. Setting to 0 creates straight lines.
startAngle	Number	0	The number of degrees to rotate the chart clockwise.

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, {
    type: "radar",
    data: data,
    options: {
            scale: {
                reverse: true,
                ticks: {
                    beginAtZero: true
                }
            }
    }
});
// This will create a chart with all of the default options, merged from the global
//  and the Radar chart defaults but this particular instance's scale will be rever
// well as the ticks beginning at zero.
```

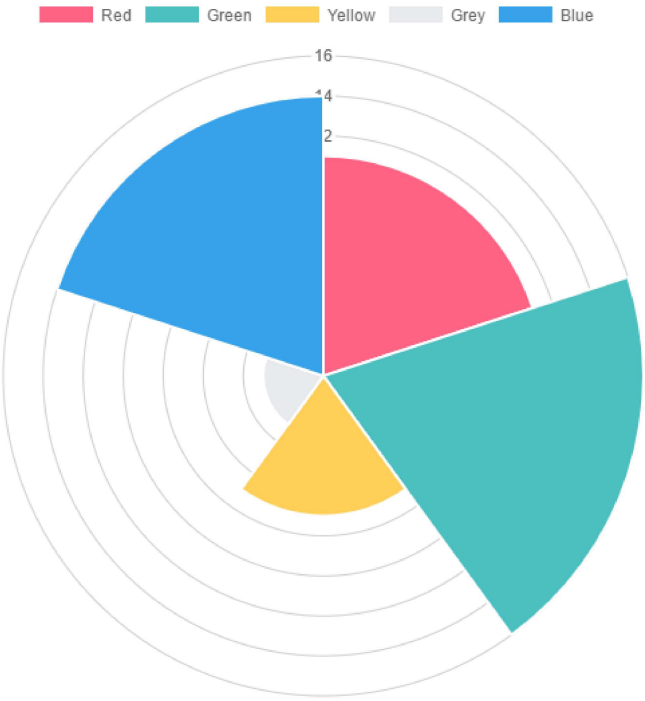
We can also change these defaults values for each Radar type that is created, this object is available at `Chart.defaults.radar`.

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, {
    data: data,
    type: 'polarArea',
    options: options
});
```

Dataset Structure

The following options can be included in a polar area chart dataset to configure options for that specific dataset.

Some properties are specified as arrays. The first value applies to the first bar, the second value to the second bar, and so on.

Property	Type	Usage
data	`Array<Number>`	The data to plot as arcs
label	`String`	The label for the dataset which appears in the legend and tooltips
backgroundColor	`Array<Color>`	The fill color of the arcs. See Colors
borderColor	`Array<Color>`	Arc border color
borderWidth	`Array<Number>`	Border width of arcs in pixels
hoverBackgroundColor	`Array<Color>`	Arc background color when hovered

Property	Type	Usage
hoverBorderColor	`Array<Color>`	Arc border color when hovered
hoverBorderWidth	`Array<Number>`	Border width of arc when hovered

An example data object using these attributes is shown below.

```
var data = {
    datasets: [{
        data: [
            11,
            16,
            7,
            3,
            14
        ],
        backgroundColor: [
            "#FF6384",
            "#4BC0C0",
            "#FFCE56",
            "#E7E9ED",
            "#36A2EB"
        ],
        label: 'My dataset' // for legend
    }],
    labels: [
        "Red",
        "Green",
        "Yellow",
        "Grey",
        "Blue"
    ]
};
```

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.

Name	Type	Default	Description
startAngle	Number	-0.5 * Math.PI	Sets the starting angle for the first item in a dataset
scale	Object	See Scales and Defaults for Radial Linear Scale	Options for the one scale used on the chart. Use this to style the ticks, labels, and grid.
scale.type	String	"radialLinear"	As defined in "Radial Linear".
scale.lineArc	Boolean	true	When true, lines are circular.
animation.animateRotate	Boolean	true	If true, will animate the rotation of the chart.
animation.animateScale	Boolean	true	If true, will animate scaling the chart.
legend.labels.generateLabels	Function	`function(data) {}`	Returns labels for each the legend

Name	Type	Default	Description
legend.onClick	Function	function(event, legendItem) {} `	Handles clicking an individual legend item
legendCallback	Function	`function(chart)`	Generates the HTML legend via calls to `generateLegend`

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, {
    data: data,
    type: "polarArea",
    options: {
        elements: {
            arc: {
                borderColor: "#000000"
            }
        }
    }
});
// This will create a chart with all of the default options, merged from the global
// and the PolarArea chart defaults but this particular instance will have `element
```

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

Pie & Doughnut Charts

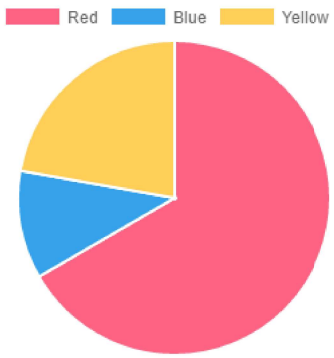
Introduction

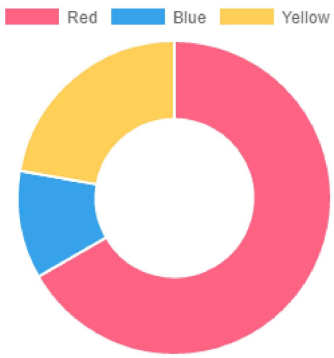
Pie and doughnut charts are probably the most commonly used charts there are. They are divided into segments, the arc of each segment shows the proportional value of each piece of data.

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 `cutoutPercentage`. 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,{
    type: 'pie',
```

```
    data: data,
    options: options
});


// And for a doughnut chart
var myDoughnutChart = new Chart(ctx, {
    type: 'doughnut',
    data: data,
    options: options
});
```

Dataset Structure

Property	Type	Usage
data	`Array<Number>`	The data to plot as arcs
label	`String`	The label for the dataset which appears in the legend and tooltips
backgroundColor	`Array<Color>`	The fill color of the arcs. See Colors
borderColor	`Array<Color>`	Arc border color
borderWidth	`Array<Number>`	Border width of arcs in pixels
hoverBackgroundColor	`Array<Color>`	Arc background color when hovered
hoverBorderColor	`Array<Color>`	Arc border color when hovered
hoverBorderWidth	`Array<Number>`	Border width of arc when hovered

An example data object using these attributes is shown below.

```
var data = {
    labels: [
        "Red",
        "Blue",
        "Yellow"
    ],
    datasets: [
        {
            data: [300, 50, 100],
            backgroundColor: [
                "#FF6384",
                "#36A2EB",
                "#FFCE56"
            ],
            hoverBackgroundColor: [
                "#FF6384",
                "#36A2EB",
                "#FFCE56"
            ]
        }]
};
```

For a pie chart, datasets need to contain an array of data points. The data points should be a number, Chart.js will total all of the numbers and calculate the relative proportion of each. You can also add an array of background colors. The color attributes 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.

Name	Type	Default	Description
cutoutPercentage	Number	50 - for doughnut, 0 - for pie	The percentage of the chart that is cut out of the middle.
rotation	Number	-0.5 * Math.PI	Starting angle to draw arcs from
circumference	Number	2 * Math.PI	Sweep to allow arcs to cover
animation.animateRotate	Boolean	true	If true, will animate the rotation of the chart.
animation.animateScale	Boolean	false	If true, will animate scaling the Doughnut from the centre.
legend.labels.generateLabels	Function	`function(chart) {}`	Returns a label for each item to be displayed on the legend.
legend.onClick	Function	function(event, legendItem) {} `	Handles clicking an individual legend item

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:

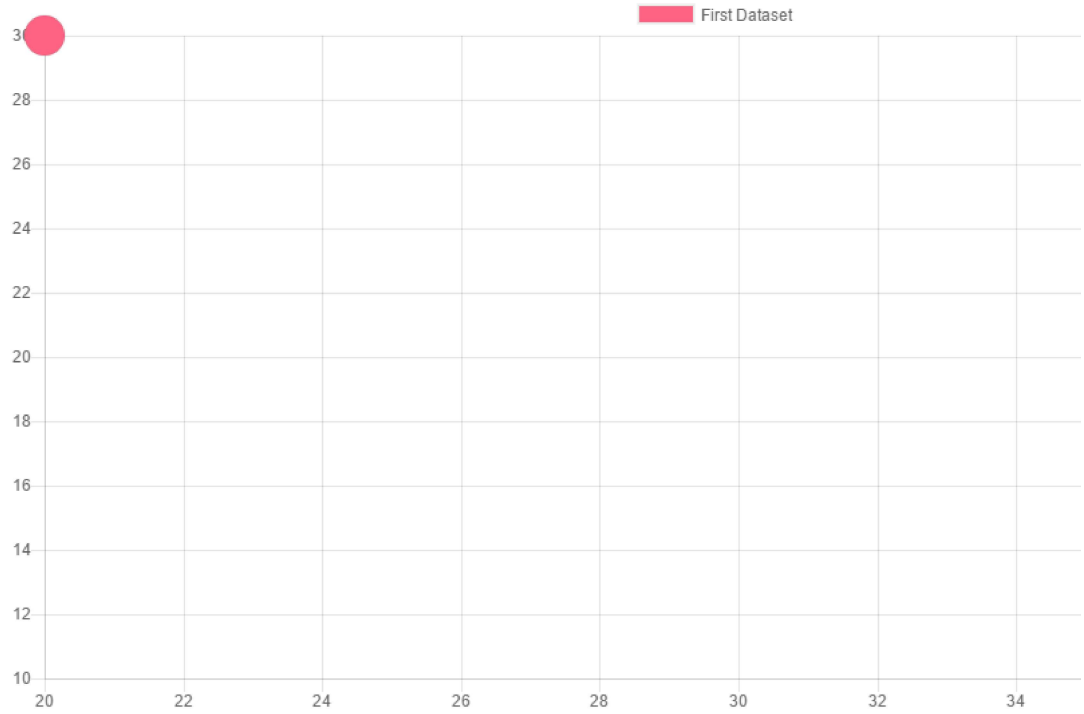
```
new Chart(ctx,{
    type:"doughnut",
    animation:{
        animateScale:true
    }
});
// This will create a chart with all of the default options, merged from the global
// and the Doughnut chart defaults but this particular instance will have `animateS
```

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 `cutoutPercentage` being set to 0.

Bubble Chart

Introduction

A bubble chart is used to display three dimensions of data at the same time. The location of the bubble is determined by the first two dimensions and the corresponding horizontal and vertical axes. The third dimension is represented by the size of the individual bubbles.



Example Usage

```
// For a bubble chart
var myBubbleChart = new Chart(ctx,{
    type: 'bubble',
    data: data,
    options: options
});
```

Dataset Structure

Property	Type	Usage
data	`Array<BubbleDataObject>`	The data to plot as bubbles. See Data format
label	`String`	The label for the dataset which appears in the legend and tooltips
backgroundColor	`Color Array<Color>`	The fill color of the bubbles. See Colors
borderColor	`Color or Array<Color>`	The stroke color of the bubbles.
borderWidth	`Number or Array<Number>`	The stroke width of bubble in pixels.
hoverBackgroundColor	`Color or Array<Color>`	The fill color of the bubbles when hovered.
hoverBorderColor	`Color or Array<Color>`	The stroke color of the bubbles when hovered.
hoverBorderWidth	`Number or Array<Number>`	The stroke width of the bubbles when hovered.
hoverRadius	`Number or Array<Number>`	Additional radius to add to data radius on hover.

An example data object using these attributes is shown below. This example creates a single dataset with 2 different bubbles.

```
var data = {
    datasets: [
        {
            label: 'First Dataset',
            data: [
                {
                    x: 20,
                    y: 30,
                    r: 15
                },
                {
                    x: 40,
                    y: 10,
                    r: 10
                }
            ],
            backgroundColor:"#FF6384",
            hoverBackgroundColor: "#FF6384",
        }]
};
```

Data Object

Data for the bubble chart is passed in the form of an object. The object must implement the following interface. It is important to note that the radius property, `r` is not scaled by the chart. It is the raw radius in pixels of the bubble that is drawn on the canvas.

```
{
    // X Value
    x: <Number>,

    // Y Value
    y: <Number>,

    // Radius of bubble. This is not scaled.
    r: <Number>
}
```

Chart Options

The bubble chart has no unique configuration options. To configure options common to all of the bubbles, the point element options are used.

For example, to give all bubbles a 1px wide black border, the following options would be used.

```
new Chart(ctx,{
    type:"bubble",
    options: {
        elements: {
            points: {
                borderWidth: 1,
                borderColor: 'rgb(0, 0, 0)'
            }
        }
    }
});
```

We can also change the default values for the Bubble chart type. Doing so will give all bubble charts created after this point the new defaults. The default configuration for the bubble chart can be accessed at `Chart.defaults.bubble`.

Scales

Scales in v2.0 of Chart.js are significantly more powerful, but also different than those of v1.0.

- Multiple X & Y axes are supported.
- A built-in label auto-skip feature detects would-be overlapping ticks and labels and removes every nth label to keep things displaying normally.
- Scale titles are supported
- New scale types can be extended without writing an entirely new chart type

Common Configuration

Every scale extends a core scale class with the following options:

Name	Type	Default	Description
type	String	Chart specific.	Type of scale being employed. Custom scales can be created and registered with a string key. Options: "category", "linear", "logarithmic", "time", "radialLinear"
display	Boolean	true	If true, show the scale including gridlines, ticks, and labels. Overrides *gridLines.display*, *scaleLabel.display*, and *ticks.display*.
position	String	"left"	Position of the scale. Possible values are 'top', 'left', 'bottom' and 'right'.
id	String		The ID is used to link datasets and scale axes together. The properties `datasets.xAxisID` or `datasets.yAxisID` have to match the scale properties `scales.xAxes.id` or `scales.yAxes.id`. This is especially needed if multi-axes charts are used.
beforeUpdate	Function	undefined	Callback called before the update process starts. Passed a single argument, the scale instance.
beforeSetDimensions	Function	undefined	Callback that runs before dimensions are set. Passed a single argument, the scale instance.
afterSetDimensions	Function	undefined	Callback that runs after dimensions are set. Passed a single argument, the scale instance.
beforeDataLimits	Function	undefined	Callback that runs before data limits are determined. Passed a single argument, the scale instance.
afterDataLimits	Function	undefined	Callback that runs after data limits are determined. Passed a single argument, the scale instance.
beforeBuildTicks	Function	undefined	Callback that runs before ticks are created. Passed a single argument, the scale instance.
afterBuildTicks	Function	undefined	Callback that runs after ticks are created. Useful for filtering ticks. Passed a single argument, the scale instance.
beforeTickToLabelConversion	Function	undefined	Callback that runs before ticks are converted into strings. Passed a single argument, the scale instance.

Name	Type	Default	Description
afterTickToLabelConversion	Function	undefined	Callback that runs after ticks are converted into strings. Passed a single argument, the scale instance.
beforeCalculateTickRotation	Function	undefined	Callback that runs before tick rotation is determined. Passed a single argument, the scale instance.
afterCalculateTickRotation	Function	undefined	Callback that runs after tick rotation is determined. Passed a single argument, the scale instance.
beforeFit	Function	undefined	Callback that runs before the scale fits to the canvas. Passed a single argument, the scale instance.
afterFit	Function	undefined	Callback that runs after the scale fits to the canvas. Passed a single argument, the scale instance.
afterUpdate	Function	undefined	Callback that runs at the end of the update process. Passed a single argument, the scale instance.
gridLines	Object	-	See grid line configuration section.
scaleLabel	Object		See scale title configuration section.
ticks	Object		See ticks configuration section.

Grid Line Configuration

The grid line configuration is nested under the scale configuration in the **`gridLines`** key. It defines options for the grid lines that run perpendicular to the axis.

Name	Type	Default	Description
display	Boolean	true	
color	Color or Array[Color]	"rgba(0, 0, 0, 0.1)"	Color of the grid lines.
borderDash	Array[Number]	[]	Length and spacing of dashes. See MDN
borderDashOffset	Number	0.0	Offset for line dashes. See MDN
lineWidth	Number or Array[Number]	1	Stroke width of grid lines
drawBorder	Boolean	true	If true draw border on the edge of the chart
drawOnChartArea	Boolean	true	If true, draw lines on the chart area inside the axis lines. This is useful when there are multiple axes and you need to control which grid lines are drawn
drawTicks	Boolean	true	If true, draw lines beside the ticks in the axis area beside the chart.
tickMarkLength	Number	10	Length in pixels that the grid lines will draw into the axis area.

Name	Type	Default	Description
zeroLineWidth	Number	1	Stroke width of the grid line for the first index (index 0).
zeroLineColor	Color	"rgba(0, 0, 0, 0.25)"	Stroke color of the grid line for the first index (index 0).
offsetGridLines	Boolean	false	If true, labels are shifted to be between grid lines. This is used in the bar chart.

Scale Title Configuration

The scale label configuration is nested under the scale configuration in the `scaleLabel` key. It defines options for the scale title.

Name	Type	Default	Description
display	Boolean	false	
labelString	String	""	The text for the title. (i.e. "# of People", "Response Choices")
fontColor	Color	"#666"	Font color for the scale title.
fontFamily	String	"'Helvetica Neue', 'Helvetica', 'Arial', sans-serif"	Font family for the scale title, follows CSS font-family options.
fontSize	Number	12	Font size for the scale title.
fontStyle	String	"normal"	Font style for the scale title, follows CSS font-style options (i.e. normal, italic, oblique, initial, inherit).

Tick Configuration

The tick configuration is nested under the scale configuration in the `ticks` key. It defines options for the tick marks that are generated by the axis.

Name	Type	Default	Description
autoSkip	Boolean	true	If true, automatically calculates how many labels that can be shown and hides labels accordingly. Turn it off to show all labels no matter what
autoSkipPadding	Number	0	Padding between the ticks on the horizontal axis when `autoSkip` is enabled. *Note: Only applicable to horizontal scales.*
callback	Function	`function(value) { return helpers.isArray(value) ? value : '' + value; }`	Returns the string representation of the tick value as it should be displayed on the chart. See callback section below.
display	Boolean	true	If true, show the ticks.
fontColor	Color	"#666"	Font color for the tick labels.

Name	Type	Default	Description
fontFamily	String	"'Helvetica Neue', 'Helvetica', 'Arial', sans-serif"	Font family for the tick labels, follows CSS font-family options.
fontSize	Number	12	Font size for the tick labels.
fontStyle	String	"normal"	Font style for the tick labels, follows CSS font-style options (i.e. normal, italic, oblique, initial, inherit).
labelOffset	Number	0	Distance in pixels to offset the label from the centre point of the tick (in the y direction for the x axis, and the x direction for the y axis). *Note: this can cause labels at the edges to be cropped by the edge of the canvas*
maxRotation	Number	90	Maximum rotation for tick labels when rotating to condense labels. Note: Rotation doesn't occur until necessary. *Note: Only applicable to horizontal scales.*
minRotation	Number	0	Minimum rotation for tick labels. *Note: Only applicable to horizontal scales.*
mirror	Boolean	false	Flips tick labels around axis, displaying the labels inside the chart instead of outside. *Note: Only applicable to vertical scales.*
padding	Number	10	Padding between the tick label and the axis. *Note: Only applicable to horizontal scales.*
reverse	Boolean	false	Reverses order of tick labels.

Creating Custom Tick Formats

The `callback` method may be used for advanced tick customization. In the following example, every label of the Y axis would be displayed in scientific notation.

If the callback returns `null` or `undefined` the associated grid line will be hidden.

```
var chartInstance = new Chart(ctx, {
    type: 'line',
    data: data,
    options: {
        scales: {
            yAxes: [{
                ticks: {
                    // Create scientific notation labels
                    callback: function(value, index, values) {
                        return value.toExponential();
                    }
                }
            }]
        }
    }
});
```

Category Scale

The category scale will be familiar to those who have used v1.0. Labels are drawn from one of the label arrays included in the chart data. If only `data.labels` is defined, this will be used. If `data.xLabels` is defined and the axis is horizontal, this will be used. Similarly, if `data.yLabels` is defined and the axis is

vertical, this property will be used. Using both `xLabels` and `yLabels` together can create a chart that uses strings for both the X and Y axes.

Configuration Options

The category scale has the following additional options that can be set.

Name	Type	Default	Description
ticks.min	String	-	The minimum item to display. Must be a value in the `labels` array
ticks.max	String	-	The maximum item to display. Must be a value in the `labels` array

Linear Scale

The linear scale is use to chart numerical data. It can be placed on either the x or y axis. The scatter chart type automatically configures a line chart to use one of these scales for the x axis. As the name suggests, linear interpolation is used to determine where a value lies on the axis.

Configuration Options

The following options are provided by the linear scale. They are all located in the `ticks` sub options.

Name	Type	Default	Description
beginAtZero	Boolean	-	if true, scale will include 0 if it is not already included.
min	Number	-	User defined minimum number for the scale, overrides minimum value from data.
max	Number	-	User defined maximum number for the scale, overrides maximum value from data.
maxTicksLimit	Number	11	Maximum number of ticks and gridlines to show. If not defined, it will limit to 11 ticks but will show all gridlines.
fixedStepSize	Number	-	User defined fixed step size for the scale. If set, the scale ticks will be enumerated by multiple of stepSize, having one tick per increment. If not set, the ticks are labeled automatically using the nice numbers algorithm.
stepSize	Number	-	if defined, it can be used along with the min and the max to give a custom number of steps. See the example below.
suggestedMax	Number	-	User defined maximum number for the scale, overrides maximum value *except for if* it is lower than the maximum value.
suggestedMin	Number	-	User defined minimum number for the scale, overrides minimum value *except for if* it is higher than the minimum value.

Example Configuration

The following example creates a line chart with a vertical axis that goes from 0 to 5 in 0.5 sized steps.

```
var chartInstance = new Chart(ctx, {
    type: 'line',
    data: data,
    options: {
        scales: {
            yAxes: [{
                ticks: {
                    max: 5,
                    min: 0,
```

```
                    stepSize: 0.5
                }
            }]
        }
    }
});
```

Logarithmic Scale

The logarithmic scale is use to chart numerical data. It can be placed on either the x or y axis. As the name suggests, logarithmic interpolation is used to determine where a value lies on the axis.

Configuration Options

The following options are provided by the logarithmic scale. They are all located in the `ticks` sub options.

Name	Type	Default	Description
min	Number	-	User defined minimum number for the scale, overrides minimum value from data.
max	Number	-	User defined maximum number for the scale, overrides maximum value from data.

Example Configuration

The following example creates a chart with a logarithmic X axis that ranges from 1 to 1000.

```
var chartInstance = new Chart(ctx, {
    type: 'line',
    data: data,
    options: {
        scales: {
            xAxes: [{
                type: 'logarithmic',
                position: 'bottom',
                ticks: {
                    min: 1,
                    max: 1000
                }
            }]
        }
    }
})
```

Time Scale

The time scale is used to display times and dates. It can only be placed on the X axis. When building its ticks, it will automatically calculate the most comfortable unit base on the size of the scale.

Configuration Options

The following options are provided by the time scale. They are all located in the `time` sub options.

Name	Type	Default	Description
displayFormats	Object	-	See Display Formats section below.
isoWeekday	Boolean	false	If true and the unit is set to 'week', iso weekdays will be used.
max	Time	-	If defined, this will override the data maximum

Name	Type	Default	Description
min	Time	-	If defined, this will override the data minimum
parser	String or Function	-	If defined as a string, it is interpreted as a custom format to be used by moment to parse the date. If this is a function, it must return a moment.js object given the appropriate data value.
round	String	-	If defined, dates will be rounded to the start of this unit. See Time Units below for the allowed units.
tooltipFormat	String	''	The moment js format string to use for the tooltip.
unit	String	-	If defined, will force the unit to be a certain type. See Time Units section below for details.
unitStepSize	Number	1	The number of units between grid lines.
minUnit	String	'millisecond'	The minimum display format to be used for a time unit

Date Formats

When providing data for the time scale, Chart.js supports all of the formats that Moment.js accepts. See Moment.js docs for details.

Display Formats

The following display formats are used to configure how different time units are formed into strings for the axis tick marks. See moment.js for the allowable format strings.

Name	Default
millisecond	'SSS [ms]'
second	'h:mm:ss a'
minute	'h:mm:ss a'
hour	'MMM D, hA'
day	'll'
week	'll'
month	'MMM YYYY'
quarter	'[Q]Q - YYYY'
year	'YYYY'

For example, to set the display format for the 'quarter' unit to show the month and year, the following config would be passed to the chart constructor.

```
var chartInstance = new Chart(ctx, {
    type: 'line',
    data: data,
    options: {
        scales: {
            xAxes: [{
                type: 'time',
                time: {
```

```
                    displayFormats: {
                        quarter: 'MMM YYYY'
                    }
                }
            }]
        }
    }
})
```

Time Units

The following time measurements are supported. The names can be passed as strings to the `time.unit` config option to force a certain unit.

- millisecond
- second
- minute
- hour
- day
- week
- month
- quarter
- year

For example, to create a chart with a time scale that always displayed units per month, the following config could be used.

```
var chartInstance = new Chart(ctx, {
    type: 'line',
    data: data,
    options: {
        scales: {
            xAxes: [{
                time: {
                    unit: 'month'
                }
            }]
        }
    }
})
```

Radial Linear Scale

The radial linear scale is used specifically for the radar and polar area chart types. It overlays the chart area, rather than being positioned on one of the edges.

Configuration Options

The following additional configuration options are provided by the radial linear scale.

Name	Type	Default	Description
lineArc	Boolean	false	If true, circular arcs are used else straight lines are used. The former is used by the polar area chart and the latter by the radar chart
angleLines	Object	-	See the Angle Line Options section below for details.
pointLabels	Object	-	See the Point Label Options section below for details.
ticks	Object	-	See the Ticks table below for options.

Angle Line Options

The following options are used to configure angled lines that radiate from the center of the chart to the point labels. They can be found in the `angleLines` sub options. Note that these options only apply if `lineArc` is false.

Name	Type	Default	Description
display	Boolean	true	If true, angle lines are shown.
color	Color	'rgba(0, 0, 0, 0.1)'	Color of angled lines
lineWidth	Number	1	Width of angled lines

Point Label Options

The following options are used to configure the point labels that are shown on the perimeter of the scale. They can be found in the `pointLabels` sub options. Note that these options only apply if `lineArc` is false.

Name	Type	Default	Description
callback	Function	-	Callback function to transform data label to axis label
fontColor	Color	'#666'	Font color
fontFamily	String	"'Helvetica Neue', 'Helvetica', 'Arial', sans-serif"	Font family to render
fontSize	Number	10	Font size in pixels
fontStyle	String	'normal'	Font Style to use

Tick Options

Name	Type	Default	Description
backdropColor	Color	'rgba(255, 255, 255, 0.75)'	Color of label backdrops
backdropPaddingX	Number	2	Horizontal padding of label backdrop
backdropPaddingY	Number	2	Vertical padding of label backdrop
beginAtZero	Boolean	-	if true, scale will include 0 if it is not already included.
min	Number	-	User defined minimum number for the scale, overrides minimum value from data.
max	Number	-	User defined maximum number for the scale, overrides maximum value from data.
maxTicksLimit	Number	11	Maximum number of ticks and gridlines to show. If not defined, it will limit to 11 ticks but will show all gridlines.
showLabelBackdrop	Boolean	true	If true, draw a background behind the tick labels

Name	Type	Default	Description
fixedStepSize	Number	-	User defined fixed step size for the scale. If set, the scale ticks will be enumerated by multiple of stepSize, having one tick per increment. If not set, the ticks are labeled automatically using the nice numbers algorithm.
stepSize	Number	-	if defined, it can be used along with the min and the max to give a custom number of steps. See the example below.
suggestedMax	Number	-	User defined maximum number for the scale, overrides maximum value *except for if* it is lower than the maximum value.
suggestedMin	Number	-	User defined minimum number for the scale, overrides minimum value *except for if* it is higher than the minimum value.

Update Default Scale config

The default configuration for a scale can be easily changed using the scale service. Pass in a partial configuration that will be merged with the current scale default configuration.

For example, to set the minimum value of 0 for all linear scales, you would do the following. Any linear scales created after this time would now have a minimum of 0.

```
Chart.scaleService.updateScaleDefaults('linear', {
    ticks: {
        min: 0
    }
})
```

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, config);
```

.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. This must be called before the canvas is reused for a new chart.

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

.update(duration, lazy)

Triggers an update of the chart. This can be safely called after replacing the entire data object. This will update all scales, legends, and then re-render the chart.

```
// duration is the time for the animation of the redraw in milliseconds
// lazy is a boolean. if true, the animation can be interrupted by other animations
myLineChart.data.datasets[0].data[2] = 50; // Would update the first dataset's valu
myLineChart.update(); // Calling update now animates the position of March from 90
```

.reset()

Reset the chart to it's state before the initial animation. A new animation can then be triggered using `update`.

```
myLineChart.reset();
```

.render(duration, lazy)

Triggers a redraw of all chart elements. Note, this does not update elements for new data. Use `.update()` in that case.

```
// duration is the time for the animation of the redraw in milliseconds
// lazy is a boolean. if true, the animation can be interrupted by other animations
myLineChart.render(duration, lazy);
```

.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 canvas container 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
```

.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
```

.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 legend is generated from the `legendCallback` in the options.

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

.getElementAtEvent(e)

Calling `getElementAtEvent(event)` on your Chart instance passing an argument of an event, or jQuery event, will return the single element at the event position. If there are multiple items within range, only the first is returned

```
myLineChart.getElementAtEvent(e);
// => returns the first element at the event point.
```

.getElementsAtEvent(e)

Looks for the element under the event point, then returns all elements at the same data index. This is used internally for 'label' mode highlighting.

Calling `getElementsAtEvent(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.getElementsAtEvent(evt);
    // => activePoints is an array of points on the canvas that are at the same pos
};
```

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

.getDatasetAtEvent(e)

Looks for the element under the event point, then returns all elements from that dataset. This is used internally for 'dataset' mode highlighting

```
myLineChart.getDatasetAtEvent(e);
// => returns an array of elements
```

.getDatasetMeta(index)

Looks for the dataset that matches the current index and returns that metadata. This returned data has all of the metadata that is used to construct the chart.

The `data` property of the metadata will contain information about each point, rectangle, etc. depending on the chart type.

Extensive examples of usage are available in the Chart.js tests.

```
var meta = myChart.getDatasetMeta(0);
var x = meta.data[0]._model.x
```

External Tooltips

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

```
var myPieChart = new Chart(ctx, {
    type: 'pie',
    data: data,
    options: {
```

```
        tooltips: {
            custom: function(tooltip) {
                // tooltip will be false if tooltip is not visible or should be hid
                if (!tooltip) {
                    return;
                }

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

                // tooltip.caretSize
                // tooltip.caretPadding
                // tooltip.chart
                // tooltip.cornerRadius
                // tooltip.fillColor
                // tooltip.font...
                // tooltip.text
                // tooltip.x
                // tooltip.y
                // tooltip.caretX
                // tooltip.caretY
                // etc...
            }
        }
    }
});
```

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

Writing New Scale Types

Starting with Chart.js 2.0 scales can be individually extended. Scales should always derive from Chart.Scale.

```
var MyScale = Chart.Scale.extend({
    /* extensions ... */
});

// MyScale is now derived from Chart.Scale
```

Once you have created your scale class, you need to register it with the global chart object so that it can be used. A default config for the scale may be provided when registering the constructor. The first parameter to the register function is a string key that is used later to identify which scale type to use for a chart.

```
Chart.scaleService.registerScaleType('myScale', MyScale, defaultConfigObject);
```

To use the new scale, simply pass in the string key to the config when creating a chart.

```
var lineChart = new Chart(ctx, {
    data: data,
    type: 'line',
    options: {
        scales: {
            yAxes: [{
                type: 'myScale' // this is the same key that was passed to the regi
            }]
        }
    }
})
```

Scale Properties

Scale instances are given the following properties during the fitting process.

```
{
    left: Number, // left edge of the scale bounding box
    right: Number, // right edge of the bounding box'
    top: Number,
    bottom: Number,
    width: Number, // the same as right - left
    height: Number, // the same as bottom - top

    // Margin on each side. Like css, this is outside the bounding box.
    margins: {
        left: Number,
        right: Number,
        top: Number,
        bottom: Number,
    },

    // Amount of padding on the inside of the bounding box (like CSS)
    paddingLeft: Number,
    paddingRight: Number,
    paddingTop: Number,
    paddingBottom: Number,
}
```

Scale Interface

To work with Chart.js, custom scale types must implement the following interface.

```
{
    // Determines the data limits. Should set this.min and this.max to be the data
    determineDataLimits: function() {},

    // Generate tick marks. this.chart is the chart instance. The data object can b
    // buildTicks() should create a ticks array on the axis instance, if you intend
    buildTicks: function() {},

    // Get the value to show for the data at the given index of the the given datas
    getLabelForIndex: function(index, datasetIndex) {},

    // Get the pixel (x coordinate for horizontal axis, y coordinate for vertical a
    // @param index: index into the ticks array
    // @param includeOffset: if true, get the pixel halway between the given tick a
    getPixelForTick: function(index, includeOffset) {},

    // Get the pixel (x coordinate for horizontal axis, y coordinate for vertical a
    // @param value : the value to get the pixel for
    // @param index : index into the data array of the value
    // @param datasetIndex : index of the dataset the value comes from
    // @param includeOffset : if true, get the pixel halway between the given tick
    getPixelForValue: function(value, index, datasetIndex, includeOffset) {}

    // Get the value for a given pixel (x coordinate for horizontal axis, y coordin
    // @param pixel : pixel value
    getValueForPixel: function(pixel) {}
}
```

Optionally, the following methods may also be overwritten, but an implementation is already provided by the `Chart.Scale` base class.

```
    // Transform the ticks array of the scale instance into strings. The default im
    convertTicksToLabels: function() {},

    // Determine how much the labels will rotate by. The default implementation wil
    calculateTickRotation: function() {},

    // Fits the scale into the canvas.
    // this.maxWidth and this.maxHeight will tell you the maximum dimensions the sc
    // this.margins is the amount of space you have on either side of your scale th
    // You must set this.minSize to be the size of your scale. It must be an object
    // You must set this.width to be the width and this.height to be the height of
    fit: function() {},
```

```
    // Draws the scale onto the canvas. this.(left|right|top|bottom) will have been
    // @param chartArea : an object containing four properties: left, right, top, b
    draw: function(chartArea) {},
```

The Core.Scale base class also has some utility functions that you may find useful.

```
{
    // Returns true if the scale instance is horizontal
    isHorizontal: function() {},

    // Get the correct value from the value from this.chart.data.datasets[x].data[]
    // If dataValue is an object, returns .x or .y depending on the return of isHor
    // If the value is undefined, returns NaN
    // Otherwise returns the value.
    // Note that in all cases, the returned value is not guaranteed to be a Number
    getRightValue: function(dataValue) {},
}
```

Writing New Chart Types

Chart.js 2.0 introduces the concept of controllers for each dataset. Like scales, new controllers can be written as needed.

```
Chart.controllers.MyType = Chart.DatasetController.extend({

});


// Now we can create a new instance of our chart, using the Chart.js API
new Chart(ctx, {
    // this is the string the constructor was registered at, ie Chart.controllers.M
    type: 'MyType',
    data: data,
    options: options
});
```

Dataset Controller Interface

Dataset controllers must implement the following interface.

```
{
    // Create elements for each piece of data in the dataset. Store elements in an
    addElements: function() {},

    // Create a single element for the data at the given index and reset its state
    addElementAndReset: function(index) {},

    // Draw the representation of the dataset
    // @param ease : if specified, this number represents how far to transition ele
    draw: function(ease) {},

    // Remove hover styling from the given element
    removeHoverStyle: function(element) {},

    // Add hover styling to the given element
    setHoverStyle: function(element) {},

    // Update the elements in response to new data
    // @param reset : if true, put the elements into a reset state so they can anim
    update: function(reset) {},
}
```

The following methods may optionally be overridden by derived dataset controllers

```
{
    // Initializes the controller
    initialize: function(chart, datasetIndex) {},

    // Ensures that the dataset represented by this controller is linked to a scale
    // chart types using a single scale
    linkScales: function() {},

    // Called by the main chart controller when an update is triggered. The default
    buildOrUpdateElements: function() {}
}
```

Extending Existing Chart Types

Extending or replacing an existing controller type is easy. Simply replace the constructor for one of the built in types with your own.

The built in controller types are:

- `Chart.controllers.line`
- `Chart.controllers.bar`
- `Chart.controllers.radar`
- `Chart.controllers.doughnut`
- `Chart.controllers.polarArea`
- `Chart.controllers.bubble`

Bar Controller

The bar controller has a special property that you should be aware of. To correctly calculate the width of a bar, the controller must determine the number of datasets that map to bars. To do this, the bar controller attaches a property `bar` to the dataset during initialization. If you are creating a replacement or updated bar controller, you should do the same. This will ensure that charts with regular bars and your new derived bars will work seamlessly.

Creating Plugins

Starting with v2.1.0, you can create plugins for chart.js. To register your plugin, simply call `Chart.plugins.register` and pass your plugin in. Plugins will be called at the following times

- Start of initialization
- End of initialization
- Start of update
- After the chart scales have calculated
- Start of datasets update
- End of datasets update
- End of update (before render occurs)
- Start of draw
- End of draw
- Before datasets draw
- After datasets draw
- Resize
- Before an animation is started

Plugins should derive from Chart.PluginBase and implement the following interface

```
{
    beforeInit: function(chartInstance) { },
    afterInit: function(chartInstance) { },
```

```
    resize: function(chartInstance, newChartSize) { },

    beforeUpdate: function(chartInstance) { },
    afterScaleUpdate: function(chartInstance) { }
    beforeDatasetsUpdate: function(chartInstance) { }
    afterDatasetsUpdate: function(chartInstance) { }
    afterUpdate: function(chartInstance) { },

    // This is called at the start of a render. It is only called once, even if the
    // to do something on each animation frame
    beforeRender: function(chartInstance) { },

    // Easing is for animation
    beforeDraw: function(chartInstance, easing) { },
    afterDraw: function(chartInstance, easing) { },
    // Before the datasets are drawn but after scales are drawn
    beforeDatasetsDraw: function(chartInstance, easing) { },
    afterDatasetsDraw: function(chartInstance, easing) { },

    destroy: function(chartInstance) { }
}
```

Building Chart.js

Chart.js uses gulp to build the library into a single JavaScript file.

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.

```
gulp build
```

Notes

Previous versions

Version 2 has a completely different API than earlier versions.

Most earlier version options have current equivalents or are the same.

Please use the documentation that is available on chartjs.org for the current version of Chart.js.

Please note - documentation for previous versions are available on the GitHub repo.

- 1.x Documentation

Browser support

Chart.js offers support for all browsers where canvas is supported.

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

Thanks to BrowserStack for allowing our team to test on thousands of browsers.

Bugs & issues

Please report these on the GitHub page - at github.com/chartjs/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, but we ask that you please follow these guidelines:

- Use tabs for indentation, not spaces.
- Only change the individual files in `/src`.
- Check that your code will pass `eslint` code standards, `gulp lint` will run this for you.
- Check that your code will pass tests, `gulp test` will run tests for you.
- 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 plugin would be more appropriate.

License

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

Charting Library Comparison

Library Features

Feature	Chart.js	D3	HighCharts	Chartist
Completely Free	✓	✓		✓
Canvas	✓			
SVG		✓	✓	✓
Built-in Charts	✓		✓	✓
8+ Chart Types	✓	✓	✓	
Extendable to Custom Charts	✓	✓		
Supports Modern Browsers	✓	✓	✓	✓
Extensive Documentation	✓	✓	✓	✓
Open Source	✓	✓	✓	✓

Built in Chart Types

Type	Chart.js	HighCharts	Chartist
Combined Types	✓	✓	
Line	✓	✓	✓
Bar	✓	✓	✓
Horizontal Bar	✓	✓	✓
Pie/Doughnut	✓	✓	✓
Polar Area	✓	✓	

Type	Chart.js	HighCharts	Chartist
Radar	✓		
Scatter	✓	✓	✓
Bubble	✓		
Gauges		✓	
Maps (Heat/Tree/etc.)		✓	

Popular Plugins

There are many plugins that add additional functionality to Chart.js. Some particularly notable ones are listed here. In addition, many plugins can be found on the Chart.js GitHub organization.

- Chart.Zoom.js - Enable zooming and panning on charts
- Chart.Annotation.js - Draw lines and boxes on chart area
- Chart.BarFunnel.js - Adds a bar funnel chart type
- Chart.Deferred.js - Defer initial chart update until chart scrolls into viewport
- Chart.Smith.js - Adds a smith chart type
- Chart.LinearGauge.js - Adds a linear gauge chart type

Popular Extensions

There are many extensions which are available for use with popular frameworks. Some particularly notable ones are listed here.

Angular

- angular-chart.js
- tc-angular-chartjs
- angular-chartjs
- Angular Chart-js Directive

React

- react-chartjs2
- react-chartjs-2

Django

- Django Chartjs

Ruby on Rails

- chartjs-ror

Laravel

- laravel-chartjs

Chart.js was built from the hard work of all these contributors.