/\*!

\* Chart.js

\* http://chartjs.org/

\* Version: 1.0.1

\*

\* Copyright 2015 Nick Downie

\* Released under the MIT license

\* https://github.com/nnnick/Chart.js/blob/master/LICENSE.md

\*/

(function(){

"use strict";

//Declare root variable - window in the browser, global on the server

var root = this,

previous = root.Chart;

//Occupy the global variable of Chart, and create a simple base class

var Chart = function(context){

var chart = this;

this.canvas = context.canvas;

this.ctx = context;

//Variables global to the chart

var computeDimension = function(element,dimension)

{

if (element['offset'+dimension])

{

return element['offset'+dimension];

}

else

{

return document.defaultView.getComputedStyle(element).getPropertyValue(dimension);

}

}

var width = this.width = computeDimension(context.canvas,'Width');

var height = this.height = computeDimension(context.canvas,'Height');

// Firefox requires this to work correctly

context.canvas.width = width;

context.canvas.height = height;

this.aspectRatio = this.width / this.height;

//High pixel density displays - multiply the size of the canvas height/width by the device pixel ratio, then scale.

helpers.retinaScale(this);

return this;

};

//Globally expose the defaults to allow for user updating/changing

Chart.defaults = {

global: {

// Boolean - Whether to animate the chart

animation: true,

// Number - Number of animation steps

animationSteps: 60,

// String - Animation easing effect

animationEasing: "easeOutQuart",

// Boolean - If we should show the scale at all

showScale: true,

// Boolean - If we want to override with a hard coded scale

scaleOverride: false,

// \*\* Required if scaleOverride is true \*\*

// Number - The number of steps in a hard coded scale

scaleSteps: null,

// Number - The value jump in the hard coded scale

scaleStepWidth: null,

// Number - The scale starting value

scaleStartValue: null,

// String - Colour of the scale line

scaleLineColor: "rgba(0,0,0,.1)",

// Number - Pixel width of the scale line

scaleLineWidth: 1,

// Boolean - Whether to show labels on the scale

scaleShowLabels: true,

// Interpolated JS string - can access value

scaleLabel: "<%=value%>",

// Boolean - Whether the scale should stick to integers, and not show any floats even if drawing space is there

scaleIntegersOnly: true,

// Boolean - Whether the scale should start at zero, or an order of magnitude down from the lowest value

scaleBeginAtZero: false,

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

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

// Number - Scale label font size in pixels

scaleFontSize: 12,

// String - Scale label font weight style

scaleFontStyle: "normal",

// String - Scale label font colour

scaleFontColor: "#666",

// Boolean - whether or not the chart should be responsive and resize when the browser does.

responsive: false,

// Boolean - whether to maintain the starting aspect ratio or not when responsive, if set to false, will take up entire container

maintainAspectRatio: true,

// Boolean - Determines whether to draw tooltips on the canvas or not - attaches events to touchmove & mousemove

showTooltips: true,

// Boolean - Determines whether to draw built-in tooltip or call custom tooltip function

customTooltips: false,

// Array - Array of string names to attach tooltip events

tooltipEvents: ["mousemove", "touchstart", "touchmove", "mouseout"],

// String - Tooltip background colour

tooltipFillColor: "rgba(0,0,0,0.8)",

// String - Tooltip label font declaration for the scale label

tooltipFontFamily: "'Helvetica Neue', 'Helvetica', 'Arial', sans-serif",

// Number - Tooltip label font size in pixels

tooltipFontSize: 14,

// String - Tooltip font weight style

tooltipFontStyle: "normal",

// String - Tooltip label font colour

tooltipFontColor: "#fff",

// String - Tooltip title font declaration for the scale label

tooltipTitleFontFamily: "'Helvetica Neue', 'Helvetica', 'Arial', sans-serif",

// Number - Tooltip title font size in pixels

tooltipTitleFontSize: 14,

// String - Tooltip title font weight style

tooltipTitleFontStyle: "bold",

// String - Tooltip title font colour

tooltipTitleFontColor: "#fff",

// Number - pixel width of padding around tooltip text

tooltipYPadding: 6,

// Number - pixel width of padding around tooltip text

tooltipXPadding: 6,

// Number - Size of the caret on the tooltip

tooltipCaretSize: 8,

// Number - Pixel radius of the tooltip border

tooltipCornerRadius: 6,

// Number - Pixel offset from point x to tooltip edge

tooltipXOffset: 10,

// String - Template string for single tooltips

tooltipTemplate: "<%if (label){%><%=label%>: <%}%><%= value %>",

// String - Template string for single tooltips

multiTooltipTemplate: "<%= value %>",

// String - Colour behind the legend colour block

multiTooltipKeyBackground: '#fff',

// Function - Will fire on animation progression.

onAnimationProgress: function(){},

// Function - Will fire on animation completion.

onAnimationComplete: function(){}

}

};

//Create a dictionary of chart types, to allow for extension of existing types

Chart.types = {};

//Global Chart helpers object for utility methods and classes

var helpers = Chart.helpers = {};

//-- Basic js utility methods

var each = helpers.each = function(loopable,callback,self){

var additionalArgs = Array.prototype.slice.call(arguments, 3);

// Check to see if null or undefined firstly.

if (loopable){

if (loopable.length === +loopable.length){

var i;

for (i=0; i<loopable.length; i++){

callback.apply(self,[loopable[i], i].concat(additionalArgs));

}

}

else{

for (var item in loopable){

callback.apply(self,[loopable[item],item].concat(additionalArgs));

}

}

}

},

clone = helpers.clone = function(obj){

var objClone = {};

each(obj,function(value,key){

if (obj.hasOwnProperty(key)) objClone[key] = value;

});

return objClone;

},

extend = helpers.extend = function(base){

each(Array.prototype.slice.call(arguments,1), function(extensionObject) {

each(extensionObject,function(value,key){

if (extensionObject.hasOwnProperty(key)) base[key] = value;

});

});

return base;

},

merge = helpers.merge = function(base,master){

//Merge properties in left object over to a shallow clone of object right.

var args = Array.prototype.slice.call(arguments,0);

args.unshift({});

return extend.apply(null, args);

},

indexOf = helpers.indexOf = function(arrayToSearch, item){

if (Array.prototype.indexOf) {

return arrayToSearch.indexOf(item);

}

else{

for (var i = 0; i < arrayToSearch.length; i++) {

if (arrayToSearch[i] === item) return i;

}

return -1;

}

},

where = helpers.where = function(collection, filterCallback){

var filtered = [];

helpers.each(collection, function(item){

if (filterCallback(item)){

filtered.push(item);

}

});

return filtered;

},

findNextWhere = helpers.findNextWhere = function(arrayToSearch, filterCallback, startIndex){

// Default to start of the array

if (!startIndex){

startIndex = -1;

}

for (var i = startIndex + 1; i < arrayToSearch.length; i++) {

var currentItem = arrayToSearch[i];

if (filterCallback(currentItem)){

return currentItem;

}

}

},

findPreviousWhere = helpers.findPreviousWhere = function(arrayToSearch, filterCallback, startIndex){

// Default to end of the array

if (!startIndex){

startIndex = arrayToSearch.length;

}

for (var i = startIndex - 1; i >= 0; i--) {

var currentItem = arrayToSearch[i];

if (filterCallback(currentItem)){

return currentItem;

}

}

},

inherits = helpers.inherits = function(extensions){

//Basic javascript inheritance based on the model created in Backbone.js

var parent = this;

var ChartElement = (extensions && extensions.hasOwnProperty("constructor")) ? extensions.constructor : function(){ return parent.apply(this, arguments); };

var Surrogate = function(){ this.constructor = ChartElement;};

Surrogate.prototype = parent.prototype;

ChartElement.prototype = new Surrogate();

ChartElement.extend = inherits;

if (extensions) extend(ChartElement.prototype, extensions);

ChartElement.\_\_super\_\_ = parent.prototype;

return ChartElement;

},

noop = helpers.noop = function(){},

uid = helpers.uid = (function(){

var id=0;

return function(){

return "chart-" + id++;

};

})(),

warn = helpers.warn = function(str){

//Method for warning of errors

if (window.console && typeof window.console.warn == "function") console.warn(str);

},

amd = helpers.amd = (typeof define == 'function' && define.amd),

//-- Math methods

isNumber = helpers.isNumber = function(n){

return !isNaN(parseFloat(n)) && isFinite(n);

},

max = helpers.max = function(array){

return Math.max.apply( Math, array );

},

min = helpers.min = function(array){

return Math.min.apply( Math, array );

},

cap = helpers.cap = function(valueToCap,maxValue,minValue){

if(isNumber(maxValue)) {

if( valueToCap > maxValue ) {

return maxValue;

}

}

else if(isNumber(minValue)){

if ( valueToCap < minValue ){

return minValue;

}

}

return valueToCap;

},

getDecimalPlaces = helpers.getDecimalPlaces = function(num){

if (num%1!==0 && isNumber(num)){

return num.toString().split(".")[1].length;

}

else {

return 0;

}

},

toRadians = helpers.radians = function(degrees){

return degrees \* (Math.PI/180);

},

// Gets the angle from vertical upright to the point about a centre.

getAngleFromPoint = helpers.getAngleFromPoint = function(centrePoint, anglePoint){

var distanceFromXCenter = anglePoint.x - centrePoint.x,

distanceFromYCenter = anglePoint.y - centrePoint.y,

radialDistanceFromCenter = Math.sqrt( distanceFromXCenter \* distanceFromXCenter + distanceFromYCenter \* distanceFromYCenter);

var angle = Math.PI \* 2 + Math.atan2(distanceFromYCenter, distanceFromXCenter);

//If the segment is in the top left quadrant, we need to add another rotation to the angle

if (distanceFromXCenter < 0 && distanceFromYCenter < 0){

angle += Math.PI\*2;

}

return {

angle: angle,

distance: radialDistanceFromCenter

};

},

aliasPixel = helpers.aliasPixel = function(pixelWidth){

return (pixelWidth % 2 === 0) ? 0 : 0.5;

},

splineCurve = helpers.splineCurve = function(FirstPoint,MiddlePoint,AfterPoint,t){

//Props to Rob Spencer at scaled innovation for his post on splining between points

//http://scaledinnovation.com/analytics/splines/aboutSplines.html

var d01=Math.sqrt(Math.pow(MiddlePoint.x-FirstPoint.x,2)+Math.pow(MiddlePoint.y-FirstPoint.y,2)),

d12=Math.sqrt(Math.pow(AfterPoint.x-MiddlePoint.x,2)+Math.pow(AfterPoint.y-MiddlePoint.y,2)),

fa=t\*d01/(d01+d12),// scaling factor for triangle Ta

fb=t\*d12/(d01+d12);

return {

inner : {

x : MiddlePoint.x-fa\*(AfterPoint.x-FirstPoint.x),

y : MiddlePoint.y-fa\*(AfterPoint.y-FirstPoint.y)

},

outer : {

x: MiddlePoint.x+fb\*(AfterPoint.x-FirstPoint.x),

y : MiddlePoint.y+fb\*(AfterPoint.y-FirstPoint.y)

}

};

},

calculateOrderOfMagnitude = helpers.calculateOrderOfMagnitude = function(val){

return Math.floor(Math.log(val) / Math.LN10);

},

calculateScaleRange = helpers.calculateScaleRange = function(valuesArray, drawingSize, textSize, startFromZero, integersOnly){

//Set a minimum step of two - a point at the top of the graph, and a point at the base

var minSteps = 2,

maxSteps = Math.floor(drawingSize/(textSize \* 1.5)),

skipFitting = (minSteps >= maxSteps);

var maxValue = max(valuesArray),

minValue = min(valuesArray);

// We need some degree of seperation here to calculate the scales if all the values are the same

// Adding/minusing 0.5 will give us a range of 1.

if (maxValue === minValue){

maxValue += 0.5;

// So we don't end up with a graph with a negative start value if we've said always start from zero

if (minValue >= 0.5 && !startFromZero){

minValue -= 0.5;

}

else{

// Make up a whole number above the values

maxValue += 0.5;

}

}

var valueRange = Math.abs(maxValue - minValue),

rangeOrderOfMagnitude = calculateOrderOfMagnitude(valueRange),

graphMax = Math.ceil(maxValue / (1 \* Math.pow(10, rangeOrderOfMagnitude))) \* Math.pow(10, rangeOrderOfMagnitude),

graphMin = (startFromZero) ? 0 : Math.floor(minValue / (1 \* Math.pow(10, rangeOrderOfMagnitude))) \* Math.pow(10, rangeOrderOfMagnitude),

graphRange = graphMax - graphMin,

stepValue = Math.pow(10, rangeOrderOfMagnitude),

numberOfSteps = Math.round(graphRange / stepValue);

//If we have more space on the graph we'll use it to give more definition to the data

while((numberOfSteps > maxSteps || (numberOfSteps \* 2) < maxSteps) && !skipFitting) {

if(numberOfSteps > maxSteps){

stepValue \*=2;

numberOfSteps = Math.round(graphRange/stepValue);

// Don't ever deal with a decimal number of steps - cancel fitting and just use the minimum number of steps.

if (numberOfSteps % 1 !== 0){

skipFitting = true;

}

}

//We can fit in double the amount of scale points on the scale

else{

//If user has declared ints only, and the step value isn't a decimal

if (integersOnly && rangeOrderOfMagnitude >= 0){

//If the user has said integers only, we need to check that making the scale more granular wouldn't make it a float

if(stepValue/2 % 1 === 0){

stepValue /=2;

numberOfSteps = Math.round(graphRange/stepValue);

}

//If it would make it a float break out of the loop

else{

break;

}

}

//If the scale doesn't have to be an int, make the scale more granular anyway.

else{

stepValue /=2;

numberOfSteps = Math.round(graphRange/stepValue);

}

}

}

if (skipFitting){

numberOfSteps = minSteps;

stepValue = graphRange / numberOfSteps;

}

return {

steps : numberOfSteps,

stepValue : stepValue,

min : graphMin,

max : graphMin + (numberOfSteps \* stepValue)

};

},

/\* jshint ignore:start \*/

// Blows up jshint errors based on the new Function constructor

//Templating methods

//Javascript micro templating by John Resig - source at http://ejohn.org/blog/javascript-micro-templating/

template = helpers.template = function(templateString, valuesObject){

// If templateString is function rather than string-template - call the function for valuesObject

if(templateString instanceof Function){

return templateString(valuesObject);

}

var cache = {};

function tmpl(str, data){

// Figure out if we're getting a template, or if we need to

// load the template - and be sure to cache the result.

var fn = !/\W/.test(str) ?

cache[str] = cache[str] :

// Generate a reusable function that will serve as a template

// generator (and which will be cached).

new Function("obj",

"var p=[],print=function(){p.push.apply(p,arguments);};" +

// Introduce the data as local variables using with(){}

"with(obj){p.push('" +

// Convert the template into pure JavaScript

str

.replace(/[\r\t\n]/g, " ")

.split("<%").join("\t")

.replace(/((^|%>)[^\t]\*)'/g, "$1\r")

.replace(/\t=(.\*?)%>/g, "',$1,'")

.split("\t").join("');")

.split("%>").join("p.push('")

.split("\r").join("\\'") +

"');}return p.join('');"

);

// Provide some basic currying to the user

return data ? fn( data ) : fn;

}

return tmpl(templateString,valuesObject);

},

/\* jshint ignore:end \*/

generateLabels = helpers.generateLabels = function(templateString,numberOfSteps,graphMin,stepValue){

var labelsArray = new Array(numberOfSteps);

if (labelTemplateString){

each(labelsArray,function(val,index){

labelsArray[index] = template(templateString,{value: (graphMin + (stepValue\*(index+1)))});

});

}

return labelsArray;

},

//--Animation methods

//Easing functions adapted from Robert Penner's easing equations

//http://www.robertpenner.com/easing/

easingEffects = helpers.easingEffects = {

linear: function (t) {

return t;

},

easeInQuad: function (t) {

return t \* t;

},

easeOutQuad: function (t) {

return -1 \* t \* (t - 2);

},

easeInOutQuad: function (t) {

if ((t /= 1 / 2) < 1) return 1 / 2 \* t \* t;

return -1 / 2 \* ((--t) \* (t - 2) - 1);

},

easeInCubic: function (t) {

return t \* t \* t;

},

easeOutCubic: function (t) {

return 1 \* ((t = t / 1 - 1) \* t \* t + 1);

},

easeInOutCubic: function (t) {

if ((t /= 1 / 2) < 1) return 1 / 2 \* t \* t \* t;

return 1 / 2 \* ((t -= 2) \* t \* t + 2);

},

easeInQuart: function (t) {

return t \* t \* t \* t;

},

easeOutQuart: function (t) {

return -1 \* ((t = t / 1 - 1) \* t \* t \* t - 1);

},

easeInOutQuart: function (t) {

if ((t /= 1 / 2) < 1) return 1 / 2 \* t \* t \* t \* t;

return -1 / 2 \* ((t -= 2) \* t \* t \* t - 2);

},

easeInQuint: function (t) {

return 1 \* (t /= 1) \* t \* t \* t \* t;

},

easeOutQuint: function (t) {

return 1 \* ((t = t / 1 - 1) \* t \* t \* t \* t + 1);

},

easeInOutQuint: function (t) {

if ((t /= 1 / 2) < 1) return 1 / 2 \* t \* t \* t \* t \* t;

return 1 / 2 \* ((t -= 2) \* t \* t \* t \* t + 2);

},

easeInSine: function (t) {

return -1 \* Math.cos(t / 1 \* (Math.PI / 2)) + 1;

},

easeOutSine: function (t) {

return 1 \* Math.sin(t / 1 \* (Math.PI / 2));

},

easeInOutSine: function (t) {

return -1 / 2 \* (Math.cos(Math.PI \* t / 1) - 1);

},

easeInExpo: function (t) {

return (t === 0) ? 1 : 1 \* Math.pow(2, 10 \* (t / 1 - 1));

},

easeOutExpo: function (t) {

return (t === 1) ? 1 : 1 \* (-Math.pow(2, -10 \* t / 1) + 1);

},

easeInOutExpo: function (t) {

if (t === 0) return 0;

if (t === 1) return 1;

if ((t /= 1 / 2) < 1) return 1 / 2 \* Math.pow(2, 10 \* (t - 1));

return 1 / 2 \* (-Math.pow(2, -10 \* --t) + 2);

},

easeInCirc: function (t) {

if (t >= 1) return t;

return -1 \* (Math.sqrt(1 - (t /= 1) \* t) - 1);

},

easeOutCirc: function (t) {

return 1 \* Math.sqrt(1 - (t = t / 1 - 1) \* t);

},

easeInOutCirc: function (t) {

if ((t /= 1 / 2) < 1) return -1 / 2 \* (Math.sqrt(1 - t \* t) - 1);

return 1 / 2 \* (Math.sqrt(1 - (t -= 2) \* t) + 1);

},

easeInElastic: function (t) {

var s = 1.70158;

var p = 0;

var a = 1;

if (t === 0) return 0;

if ((t /= 1) == 1) return 1;

if (!p) p = 1 \* 0.3;

if (a < Math.abs(1)) {

a = 1;

s = p / 4;

} else s = p / (2 \* Math.PI) \* Math.asin(1 / a);

return -(a \* Math.pow(2, 10 \* (t -= 1)) \* Math.sin((t \* 1 - s) \* (2 \* Math.PI) / p));

},

easeOutElastic: function (t) {

var s = 1.70158;

var p = 0;

var a = 1;

if (t === 0) return 0;

if ((t /= 1) == 1) return 1;

if (!p) p = 1 \* 0.3;

if (a < Math.abs(1)) {

a = 1;

s = p / 4;

} else s = p / (2 \* Math.PI) \* Math.asin(1 / a);

return a \* Math.pow(2, -10 \* t) \* Math.sin((t \* 1 - s) \* (2 \* Math.PI) / p) + 1;

},

easeInOutElastic: function (t) {

var s = 1.70158;

var p = 0;

var a = 1;

if (t === 0) return 0;

if ((t /= 1 / 2) == 2) return 1;

if (!p) p = 1 \* (0.3 \* 1.5);

if (a < Math.abs(1)) {

a = 1;

s = p / 4;

} else s = p / (2 \* Math.PI) \* Math.asin(1 / a);

if (t < 1) return -0.5 \* (a \* Math.pow(2, 10 \* (t -= 1)) \* Math.sin((t \* 1 - s) \* (2 \* Math.PI) / p));

return a \* Math.pow(2, -10 \* (t -= 1)) \* Math.sin((t \* 1 - s) \* (2 \* Math.PI) / p) \* 0.5 + 1;

},

easeInBack: function (t) {

var s = 1.70158;

return 1 \* (t /= 1) \* t \* ((s + 1) \* t - s);

},

easeOutBack: function (t) {

var s = 1.70158;

return 1 \* ((t = t / 1 - 1) \* t \* ((s + 1) \* t + s) + 1);

},

easeInOutBack: function (t) {

var s = 1.70158;

if ((t /= 1 / 2) < 1) return 1 / 2 \* (t \* t \* (((s \*= (1.525)) + 1) \* t - s));

return 1 / 2 \* ((t -= 2) \* t \* (((s \*= (1.525)) + 1) \* t + s) + 2);

},

easeInBounce: function (t) {

return 1 - easingEffects.easeOutBounce(1 - t);

},

easeOutBounce: function (t) {

if ((t /= 1) < (1 / 2.75)) {

return 1 \* (7.5625 \* t \* t);

} else if (t < (2 / 2.75)) {

return 1 \* (7.5625 \* (t -= (1.5 / 2.75)) \* t + 0.75);

} else if (t < (2.5 / 2.75)) {

return 1 \* (7.5625 \* (t -= (2.25 / 2.75)) \* t + 0.9375);

} else {

return 1 \* (7.5625 \* (t -= (2.625 / 2.75)) \* t + 0.984375);

}

},

easeInOutBounce: function (t) {

if (t < 1 / 2) return easingEffects.easeInBounce(t \* 2) \* 0.5;

return easingEffects.easeOutBounce(t \* 2 - 1) \* 0.5 + 1 \* 0.5;

}

},

//Request animation polyfill - http://www.paulirish.com/2011/requestanimationframe-for-smart-animating/

requestAnimFrame = helpers.requestAnimFrame = (function(){

return window.requestAnimationFrame ||

window.webkitRequestAnimationFrame ||

window.mozRequestAnimationFrame ||

window.oRequestAnimationFrame ||

window.msRequestAnimationFrame ||

function(callback) {

return window.setTimeout(callback, 1000 / 60);

};

})(),

cancelAnimFrame = helpers.cancelAnimFrame = (function(){

return window.cancelAnimationFrame ||

window.webkitCancelAnimationFrame ||

window.mozCancelAnimationFrame ||

window.oCancelAnimationFrame ||

window.msCancelAnimationFrame ||

function(callback) {

return window.clearTimeout(callback, 1000 / 60);

};

})(),

animationLoop = helpers.animationLoop = function(callback,totalSteps,easingString,onProgress,onComplete,chartInstance){

var currentStep = 0,

easingFunction = easingEffects[easingString] || easingEffects.linear;

var animationFrame = function(){

currentStep++;

var stepDecimal = currentStep/totalSteps;

var easeDecimal = easingFunction(stepDecimal);

callback.call(chartInstance,easeDecimal,stepDecimal, currentStep);

onProgress.call(chartInstance,easeDecimal,stepDecimal);

if (currentStep < totalSteps){

chartInstance.animationFrame = requestAnimFrame(animationFrame);

} else{

onComplete.apply(chartInstance);

}

};

requestAnimFrame(animationFrame);

},

//-- DOM methods

getRelativePosition = helpers.getRelativePosition = function(evt){

var mouseX, mouseY;

var e = evt.originalEvent || evt,

canvas = evt.currentTarget || evt.srcElement,

boundingRect = canvas.getBoundingClientRect();

if (e.touches){

mouseX = e.touches[0].clientX - boundingRect.left;

mouseY = e.touches[0].clientY - boundingRect.top;

}

else{

mouseX = e.clientX - boundingRect.left;

mouseY = e.clientY - boundingRect.top;

}

return {

x : mouseX,

y : mouseY

};

},

addEvent = helpers.addEvent = function(node,eventType,method){

if (node.addEventListener){

node.addEventListener(eventType,method);

} else if (node.attachEvent){

node.attachEvent("on"+eventType, method);

} else {

node["on"+eventType] = method;

}

},

removeEvent = helpers.removeEvent = function(node, eventType, handler){

if (node.removeEventListener){

node.removeEventListener(eventType, handler, false);

} else if (node.detachEvent){

node.detachEvent("on"+eventType,handler);

} else{

node["on" + eventType] = noop;

}

},

bindEvents = helpers.bindEvents = function(chartInstance, arrayOfEvents, handler){

// Create the events object if it's not already present

if (!chartInstance.events) chartInstance.events = {};

each(arrayOfEvents,function(eventName){

chartInstance.events[eventName] = function(){

handler.apply(chartInstance, arguments);

};

addEvent(chartInstance.chart.canvas,eventName,chartInstance.events[eventName]);

});

},

unbindEvents = helpers.unbindEvents = function (chartInstance, arrayOfEvents) {

each(arrayOfEvents, function(handler,eventName){

removeEvent(chartInstance.chart.canvas, eventName, handler);

});

},

getMaximumWidth = helpers.getMaximumWidth = function(domNode){

var container = domNode.parentNode;

// TODO = check cross browser stuff with this.

return container.clientWidth;

},

getMaximumHeight = helpers.getMaximumHeight = function(domNode){

var container = domNode.parentNode;

// TODO = check cross browser stuff with this.

return container.clientHeight;

},

getMaximumSize = helpers.getMaximumSize = helpers.getMaximumWidth, // legacy support

retinaScale = helpers.retinaScale = function(chart){

var ctx = chart.ctx,

width = chart.canvas.width,

height = chart.canvas.height;

if (window.devicePixelRatio) {

ctx.canvas.style.width = width + "px";

ctx.canvas.style.height = height + "px";

ctx.canvas.height = height \* window.devicePixelRatio;

ctx.canvas.width = width \* window.devicePixelRatio;

ctx.scale(window.devicePixelRatio, window.devicePixelRatio);

}

},

//-- Canvas methods

clear = helpers.clear = function(chart){

chart.ctx.clearRect(0,0,chart.width,chart.height);

},

fontString = helpers.fontString = function(pixelSize,fontStyle,fontFamily){

return fontStyle + " " + pixelSize+"px " + fontFamily;

},

longestText = helpers.longestText = function(ctx,font,arrayOfStrings){

ctx.font = font;

var longest = 0;

each(arrayOfStrings,function(string){

var textWidth = ctx.measureText(string).width;

longest = (textWidth > longest) ? textWidth : longest;

});

return longest;

},

drawRoundedRectangle = helpers.drawRoundedRectangle = function(ctx,x,y,width,height,radius){

ctx.beginPath();

ctx.moveTo(x + radius, y);

ctx.lineTo(x + width - radius, y);

ctx.quadraticCurveTo(x + width, y, x + width, y + radius);

ctx.lineTo(x + width, y + height - radius);

ctx.quadraticCurveTo(x + width, y + height, x + width - radius, y + height);

ctx.lineTo(x + radius, y + height);

ctx.quadraticCurveTo(x, y + height, x, y + height - radius);

ctx.lineTo(x, y + radius);

ctx.quadraticCurveTo(x, y, x + radius, y);

ctx.closePath();

};

//Store a reference to each instance - allowing us to globally resize chart instances on window resize.

//Destroy method on the chart will remove the instance of the chart from this reference.

Chart.instances = {};

Chart.Type = function(data,options,chart){

this.options = options;

this.chart = chart;

this.id = uid();

//Add the chart instance to the global namespace

Chart.instances[this.id] = this;

// Initialize is always called when a chart type is created

// By default it is a no op, but it should be extended

if (options.responsive){

this.resize();

}

this.initialize.call(this,data);

};

//Core methods that'll be a part of every chart type

extend(Chart.Type.prototype,{

initialize : function(){return this;},

clear : function(){

clear(this.chart);

return this;

},

stop : function(){

// Stops any current animation loop occuring

helpers.cancelAnimFrame.call(root, this.animationFrame);

return this;

},

resize : function(callback){

this.stop();

var canvas = this.chart.canvas,

newWidth = getMaximumWidth(this.chart.canvas),

newHeight = this.options.maintainAspectRatio ? newWidth / this.chart.aspectRatio : getMaximumHeight(this.chart.canvas);

canvas.width = this.chart.width = newWidth;

canvas.height = this.chart.height = newHeight;

retinaScale(this.chart);

if (typeof callback === "function"){

callback.apply(this, Array.prototype.slice.call(arguments, 1));

}

return this;

},

reflow : noop,

render : function(reflow){

if (reflow){

this.reflow();

}

if (this.options.animation && !reflow){

helpers.animationLoop(

this.draw,

this.options.animationSteps,

this.options.animationEasing,

this.options.onAnimationProgress,

this.options.onAnimationComplete,

this

);

}

else{

this.draw();

this.options.onAnimationComplete.call(this);

}

return this;

},

generateLegend : function(){

return template(this.options.legendTemplate,this);

},

destroy : function(){

this.clear();

unbindEvents(this, this.events);

var canvas = this.chart.canvas;

// Reset canvas height/width attributes starts a fresh with the canvas context

canvas.width = this.chart.width;

canvas.height = this.chart.height;

// < IE9 doesn't support removeProperty

if (canvas.style.removeProperty) {

canvas.style.removeProperty('width');

canvas.style.removeProperty('height');

} else {

canvas.style.removeAttribute('width');

canvas.style.removeAttribute('height');

}

delete Chart.instances[this.id];

},

showTooltip : function(ChartElements, forceRedraw){

// Only redraw the chart if we've actually changed what we're hovering on.

if (typeof this.activeElements === 'undefined') this.activeElements = [];

var isChanged = (function(Elements){

var changed = false;

if (Elements.length !== this.activeElements.length){

changed = true;

return changed;

}

each(Elements, function(element, index){

if (element !== this.activeElements[index]){

changed = true;

}

}, this);

return changed;

}).call(this, ChartElements);

if (!isChanged && !forceRedraw){

return;

}

else{

this.activeElements = ChartElements;

}

this.draw();

if(this.options.customTooltips){

this.options.customTooltips(false);

}

if (ChartElements.length > 0){

// If we have multiple datasets, show a MultiTooltip for all of the data points at that index

if (this.datasets && this.datasets.length > 1) {

var dataArray,

dataIndex;

for (var i = this.datasets.length - 1; i >= 0; i--) {

dataArray = this.datasets[i].points || this.datasets[i].bars || this.datasets[i].segments;

dataIndex = indexOf(dataArray, ChartElements[0]);

if (dataIndex !== -1){

break;

}

}

var tooltipLabels = [],

tooltipColors = [],

medianPosition = (function(index) {

// Get all the points at that particular index

var Elements = [],

dataCollection,

xPositions = [],

yPositions = [],

xMax,

yMax,

xMin,

yMin;

helpers.each(this.datasets, function(dataset){

dataCollection = dataset.points || dataset.bars || dataset.segments;

if (dataCollection[dataIndex] && dataCollection[dataIndex].hasValue()){

Elements.push(dataCollection[dataIndex]);

}

});

helpers.each(Elements, function(element) {

xPositions.push(element.x);

yPositions.push(element.y);

//Include any colour information about the element

tooltipLabels.push(helpers.template(this.options.multiTooltipTemplate, element));

tooltipColors.push({

fill: element.\_saved.fillColor || element.fillColor,

stroke: element.\_saved.strokeColor || element.strokeColor

});

}, this);

yMin = min(yPositions);

yMax = max(yPositions);

xMin = min(xPositions);

xMax = max(xPositions);

return {

x: (xMin > this.chart.width/2) ? xMin : xMax,

y: (yMin + yMax)/2

};

}).call(this, dataIndex);

new Chart.MultiTooltip({

x: medianPosition.x,

y: medianPosition.y,

xPadding: this.options.tooltipXPadding,

yPadding: this.options.tooltipYPadding,

xOffset: this.options.tooltipXOffset,

fillColor: this.options.tooltipFillColor,

textColor: this.options.tooltipFontColor,

fontFamily: this.options.tooltipFontFamily,

fontStyle: this.options.tooltipFontStyle,

fontSize: this.options.tooltipFontSize,

titleTextColor: this.options.tooltipTitleFontColor,

titleFontFamily: this.options.tooltipTitleFontFamily,

titleFontStyle: this.options.tooltipTitleFontStyle,

titleFontSize: this.options.tooltipTitleFontSize,

cornerRadius: this.options.tooltipCornerRadius,

labels: tooltipLabels,

legendColors: tooltipColors,

legendColorBackground : this.options.multiTooltipKeyBackground,

title: ChartElements[0].label,

chart: this.chart,

ctx: this.chart.ctx,

custom: this.options.customTooltips

}).draw();

} else {

each(ChartElements, function(Element) {

var tooltipPosition = Element.tooltipPosition();

new Chart.Tooltip({

x: Math.round(tooltipPosition.x),

y: Math.round(tooltipPosition.y),

xPadding: this.options.tooltipXPadding,

yPadding: this.options.tooltipYPadding,

fillColor: this.options.tooltipFillColor,

textColor: this.options.tooltipFontColor,

fontFamily: this.options.tooltipFontFamily,

fontStyle: this.options.tooltipFontStyle,

fontSize: this.options.tooltipFontSize,

caretHeight: this.options.tooltipCaretSize,

cornerRadius: this.options.tooltipCornerRadius,

text: template(this.options.tooltipTemplate, Element),

chart: this.chart,

custom: this.options.customTooltips

}).draw();

}, this);

}

}

return this;

},

toBase64Image : function(){

return this.chart.canvas.toDataURL.apply(this.chart.canvas, arguments);

}

});

Chart.Type.extend = function(extensions){

var parent = this;

var ChartType = function(){

return parent.apply(this,arguments);

};

//Copy the prototype object of the this class

ChartType.prototype = clone(parent.prototype);

//Now overwrite some of the properties in the base class with the new extensions

extend(ChartType.prototype, extensions);

ChartType.extend = Chart.Type.extend;

if (extensions.name || parent.prototype.name){

var chartName = extensions.name || parent.prototype.name;

//Assign any potential default values of the new chart type

//If none are defined, we'll use a clone of the chart type this is being extended from.

//I.e. if we extend a line chart, we'll use the defaults from the line chart if our new chart

//doesn't define some defaults of their own.

var baseDefaults = (Chart.defaults[parent.prototype.name]) ? clone(Chart.defaults[parent.prototype.name]) : {};

Chart.defaults[chartName] = extend(baseDefaults,extensions.defaults);

Chart.types[chartName] = ChartType;

//Register this new chart type in the Chart prototype

Chart.prototype[chartName] = function(data,options){

var config = merge(Chart.defaults.global, Chart.defaults[chartName], options || {});

return new ChartType(data,config,this);

};

} else{

warn("Name not provided for this chart, so it hasn't been registered");

}

return parent;

};

Chart.Element = function(configuration){

extend(this,configuration);

this.initialize.apply(this,arguments);

this.save();

};

extend(Chart.Element.prototype,{

initialize : function(){},

restore : function(props){

if (!props){

extend(this,this.\_saved);

} else {

each(props,function(key){

this[key] = this.\_saved[key];

},this);

}

return this;

},

save : function(){

this.\_saved = clone(this);

delete this.\_saved.\_saved;

return this;

},

update : function(newProps){

each(newProps,function(value,key){

this.\_saved[key] = this[key];

this[key] = value;

},this);

return this;

},

transition : function(props,ease){

each(props,function(value,key){

this[key] = ((value - this.\_saved[key]) \* ease) + this.\_saved[key];

},this);

return this;

},

tooltipPosition : function(){

return {

x : this.x,

y : this.y

};

},

hasValue: function(){

return isNumber(this.value);

}

});

Chart.Element.extend = inherits;

Chart.Point = Chart.Element.extend({

display: true,

inRange: function(chartX,chartY){

var hitDetectionRange = this.hitDetectionRadius + this.radius;

return ((Math.pow(chartX-this.x, 2)+Math.pow(chartY-this.y, 2)) < Math.pow(hitDetectionRange,2));

},

draw : function(){

if (this.display){

var ctx = this.ctx;

ctx.beginPath();

ctx.arc(this.x, this.y, this.radius, 0, Math.PI\*2);

ctx.closePath();

ctx.strokeStyle = this.strokeColor;

ctx.lineWidth = this.strokeWidth;

ctx.fillStyle = this.fillColor;

ctx.fill();

ctx.stroke();

}

//Quick debug for bezier curve splining

//Highlights control points and the line between them.

//Handy for dev - stripped in the min version.

// ctx.save();

// ctx.fillStyle = "black";

// ctx.strokeStyle = "black"

// ctx.beginPath();

// ctx.arc(this.controlPoints.inner.x,this.controlPoints.inner.y, 2, 0, Math.PI\*2);

// ctx.fill();

// ctx.beginPath();

// ctx.arc(this.controlPoints.outer.x,this.controlPoints.outer.y, 2, 0, Math.PI\*2);

// ctx.fill();

// ctx.moveTo(this.controlPoints.inner.x,this.controlPoints.inner.y);

// ctx.lineTo(this.x, this.y);

// ctx.lineTo(this.controlPoints.outer.x,this.controlPoints.outer.y);

// ctx.stroke();

// ctx.restore();

}

});

Chart.Arc = Chart.Element.extend({

inRange : function(chartX,chartY){

var pointRelativePosition = helpers.getAngleFromPoint(this, {

x: chartX,

y: chartY

});

//Check if within the range of the open/close angle

var betweenAngles = (pointRelativePosition.angle >= this.startAngle && pointRelativePosition.angle <= this.endAngle),

withinRadius = (pointRelativePosition.distance >= this.innerRadius && pointRelativePosition.distance <= this.outerRadius);

return (betweenAngles && withinRadius);

//Ensure within the outside of the arc centre, but inside arc outer

},

tooltipPosition : function(){

var centreAngle = this.startAngle + ((this.endAngle - this.startAngle) / 2),

rangeFromCentre = (this.outerRadius - this.innerRadius) / 2 + this.innerRadius;

return {

x : this.x + (Math.cos(centreAngle) \* rangeFromCentre),

y : this.y + (Math.sin(centreAngle) \* rangeFromCentre)

};

},

draw : function(animationPercent){

var easingDecimal = animationPercent || 1;

var ctx = this.ctx;

ctx.beginPath();

ctx.arc(this.x, this.y, this.outerRadius, this.startAngle, this.endAngle);

ctx.arc(this.x, this.y, this.innerRadius, this.endAngle, this.startAngle, true);

ctx.closePath();

ctx.strokeStyle = this.strokeColor;

ctx.lineWidth = this.strokeWidth;

ctx.fillStyle = this.fillColor;

ctx.fill();

ctx.lineJoin = 'bevel';

if (this.showStroke){

ctx.stroke();

}

}

});

Chart.Rectangle = Chart.Element.extend({

draw : function(){

var ctx = this.ctx,

halfWidth = this.width/2,

leftX = this.x - halfWidth,

rightX = this.x + halfWidth,

top = this.base - (this.base - this.y),

halfStroke = this.strokeWidth / 2;

// Canvas doesn't allow us to stroke inside the width so we can

// adjust the sizes to fit if we're setting a stroke on the line

if (this.showStroke){

leftX += halfStroke;

rightX -= halfStroke;

top += halfStroke;

}

ctx.beginPath();

ctx.fillStyle = this.fillColor;

ctx.strokeStyle = this.strokeColor;

ctx.lineWidth = this.strokeWidth;

// It'd be nice to keep this class totally generic to any rectangle

// and simply specify which border to miss out.

ctx.moveTo(leftX, this.base);

ctx.lineTo(leftX, top);

ctx.lineTo(rightX, top);

ctx.lineTo(rightX, this.base);

ctx.fill();

if (this.showStroke){

ctx.stroke();

}

},

height : function(){

return this.base - this.y;

},

inRange : function(chartX,chartY){

return (chartX >= this.x - this.width/2 && chartX <= this.x + this.width/2) && (chartY >= this.y && chartY <= this.base);

}

});

Chart.Tooltip = Chart.Element.extend({

draw : function(){

var ctx = this.chart.ctx;

ctx.font = fontString(this.fontSize,this.fontStyle,this.fontFamily);

this.xAlign = "center";

this.yAlign = "above";

//Distance between the actual element.y position and the start of the tooltip caret

var caretPadding = this.caretPadding = 2;

var tooltipWidth = ctx.measureText(this.text).width + 2\*this.xPadding,

tooltipRectHeight = this.fontSize + 2\*this.yPadding,

tooltipHeight = tooltipRectHeight + this.caretHeight + caretPadding;

if (this.x + tooltipWidth/2 >this.chart.width){

this.xAlign = "left";

} else if (this.x - tooltipWidth/2 < 0){

this.xAlign = "right";

}

if (this.y - tooltipHeight < 0){

this.yAlign = "below";

}

var tooltipX = this.x - tooltipWidth/2,

tooltipY = this.y - tooltipHeight;

ctx.fillStyle = this.fillColor;

// Custom Tooltips

if(this.custom){

this.custom(this);

}

else{

switch(this.yAlign)

{

case "above":

//Draw a caret above the x/y

ctx.beginPath();

ctx.moveTo(this.x,this.y - caretPadding);

ctx.lineTo(this.x + this.caretHeight, this.y - (caretPadding + this.caretHeight));

ctx.lineTo(this.x - this.caretHeight, this.y - (caretPadding + this.caretHeight));

ctx.closePath();

ctx.fill();

break;

case "below":

tooltipY = this.y + caretPadding + this.caretHeight;

//Draw a caret below the x/y

ctx.beginPath();

ctx.moveTo(this.x, this.y + caretPadding);

ctx.lineTo(this.x + this.caretHeight, this.y + caretPadding + this.caretHeight);

ctx.lineTo(this.x - this.caretHeight, this.y + caretPadding + this.caretHeight);

ctx.closePath();

ctx.fill();

break;

}

switch(this.xAlign)

{

case "left":

tooltipX = this.x - tooltipWidth + (this.cornerRadius + this.caretHeight);

break;

case "right":

tooltipX = this.x - (this.cornerRadius + this.caretHeight);

break;

}

drawRoundedRectangle(ctx,tooltipX,tooltipY,tooltipWidth,tooltipRectHeight,this.cornerRadius);

ctx.fill();

ctx.fillStyle = this.textColor;

ctx.textAlign = "center";

ctx.textBaseline = "middle";

ctx.fillText(this.text, tooltipX + tooltipWidth/2, tooltipY + tooltipRectHeight/2);

}

}

});

Chart.MultiTooltip = Chart.Element.extend({

initialize : function(){

this.font = fontString(this.fontSize,this.fontStyle,this.fontFamily);

this.titleFont = fontString(this.titleFontSize,this.titleFontStyle,this.titleFontFamily);

this.height = (this.labels.length \* this.fontSize) + ((this.labels.length-1) \* (this.fontSize/2)) + (this.yPadding\*2) + this.titleFontSize \*1.5;

this.ctx.font = this.titleFont;

var titleWidth = this.ctx.measureText(this.title).width,

//Label has a legend square as well so account for this.

labelWidth = longestText(this.ctx,this.font,this.labels) + this.fontSize + 3,

longestTextWidth = max([labelWidth,titleWidth]);

this.width = longestTextWidth + (this.xPadding\*2);

var halfHeight = this.height/2;

//Check to ensure the height will fit on the canvas

//The three is to buffer form the very

if (this.y - halfHeight < 0 ){

this.y = halfHeight;

} else if (this.y + halfHeight > this.chart.height){

this.y = this.chart.height - halfHeight;

}

//Decide whether to align left or right based on position on canvas

if (this.x > this.chart.width/2){

this.x -= this.xOffset + this.width;

} else {

this.x += this.xOffset;

}

},

getLineHeight : function(index){

var baseLineHeight = this.y - (this.height/2) + this.yPadding,

afterTitleIndex = index-1;

//If the index is zero, we're getting the title

if (index === 0){

return baseLineHeight + this.titleFontSize/2;

} else{

return baseLineHeight + ((this.fontSize\*1.5\*afterTitleIndex) + this.fontSize/2) + this.titleFontSize \* 1.5;

}

},

draw : function(){

// Custom Tooltips

if(this.custom){

this.custom(this);

}

else{

drawRoundedRectangle(this.ctx,this.x,this.y - this.height/2,this.width,this.height,this.cornerRadius);

var ctx = this.ctx;

ctx.fillStyle = this.fillColor;

ctx.fill();

ctx.closePath();

ctx.textAlign = "left";

ctx.textBaseline = "middle";

ctx.fillStyle = this.titleTextColor;

ctx.font = this.titleFont;

ctx.fillText(this.title,this.x + this.xPadding, this.getLineHeight(0));

ctx.font = this.font;

helpers.each(this.labels,function(label,index){

ctx.fillStyle = this.textColor;

ctx.fillText(label,this.x + this.xPadding + this.fontSize + 3, this.getLineHeight(index + 1));

//A bit gnarly, but clearing this rectangle breaks when using explorercanvas (clears whole canvas)

//ctx.clearRect(this.x + this.xPadding, this.getLineHeight(index + 1) - this.fontSize/2, this.fontSize, this.fontSize);

//Instead we'll make a white filled block to put the legendColour palette over.

ctx.fillStyle = this.legendColorBackground;

ctx.fillRect(this.x + this.xPadding, this.getLineHeight(index + 1) - this.fontSize/2, this.fontSize, this.fontSize);

ctx.fillStyle = this.legendColors[index].fill;

ctx.fillRect(this.x + this.xPadding, this.getLineHeight(index + 1) - this.fontSize/2, this.fontSize, this.fontSize);

},this);

}

}

});

Chart.Scale = Chart.Element.extend({

initialize : function(){

this.fit();

},

buildYLabels : function(){

this.yLabels = [];

var stepDecimalPlaces = getDecimalPlaces(this.stepValue);

for (var i=0; i<=this.steps; i++){

this.yLabels.push(template(this.templateString,{value:(this.min + (i \* this.stepValue)).toFixed(stepDecimalPlaces)}));

}

this.yLabelWidth = (this.display && this.showLabels) ? longestText(this.ctx,this.font,this.yLabels) : 0;

},

addXLabel : function(label){

this.xLabels.push(label);

this.valuesCount++;

this.fit();

},

removeXLabel : function(){

this.xLabels.shift();

this.valuesCount--;

this.fit();

},

// Fitting loop to rotate x Labels and figure out what fits there, and also calculate how many Y steps to use

fit: function(){

// First we need the width of the yLabels, assuming the xLabels aren't rotated

// To do that we need the base line at the top and base of the chart, assuming there is no x label rotation

this.startPoint = (this.display) ? this.fontSize : 0;

this.endPoint = (this.display) ? this.height - (this.fontSize \* 1.5) - 5 : this.height; // -5 to pad labels

// Apply padding settings to the start and end point.

this.startPoint += this.padding;

this.endPoint -= this.padding;

// Cache the starting height, so can determine if we need to recalculate the scale yAxis

var cachedHeight = this.endPoint - this.startPoint,

cachedYLabelWidth;

// Build the current yLabels so we have an idea of what size they'll be to start

/\*

\* This sets what is returned from calculateScaleRange as static properties of this class:

\*

this.steps;

this.stepValue;

this.min;

this.max;

\*

\*/

this.calculateYRange(cachedHeight);

// With these properties set we can now build the array of yLabels

// and also the width of the largest yLabel

this.buildYLabels();

this.calculateXLabelRotation();

while((cachedHeight > this.endPoint - this.startPoint)){

cachedHeight = this.endPoint - this.startPoint;

cachedYLabelWidth = this.yLabelWidth;

this.calculateYRange(cachedHeight);

this.buildYLabels();

// Only go through the xLabel loop again if the yLabel width has changed

if (cachedYLabelWidth < this.yLabelWidth){

this.calculateXLabelRotation();

}

}

},

calculateXLabelRotation : function(){

//Get the width of each grid by calculating the difference

//between x offsets between 0 and 1.

this.ctx.font = this.font;

var firstWidth = this.ctx.measureText(this.xLabels[0]).width,

lastWidth = this.ctx.measureText(this.xLabels[this.xLabels.length - 1]).width,

firstRotated,

lastRotated;

this.xScalePaddingRight = lastWidth/2 + 3;

this.xScalePaddingLeft = (firstWidth/2 > this.yLabelWidth + 10) ? firstWidth/2 : this.yLabelWidth + 10;

this.xLabelRotation = 0;

if (this.display){

var originalLabelWidth = longestText(this.ctx,this.font,this.xLabels),

cosRotation,

firstRotatedWidth;

this.xLabelWidth = originalLabelWidth;

//Allow 3 pixels x2 padding either side for label readability

var xGridWidth = Math.floor(this.calculateX(1) - this.calculateX(0)) - 6;

//Max label rotate should be 90 - also act as a loop counter

while ((this.xLabelWidth > xGridWidth && this.xLabelRotation === 0) || (this.xLabelWidth > xGridWidth && this.xLabelRotation <= 90 && this.xLabelRotation > 0)){

cosRotation = Math.cos(toRadians(this.xLabelRotation));

firstRotated = cosRotation \* firstWidth;

lastRotated = cosRotation \* lastWidth;

// We're right aligning the text now.

if (firstRotated + this.fontSize / 2 > this.yLabelWidth + 8){

this.xScalePaddingLeft = firstRotated + this.fontSize / 2;

}

this.xScalePaddingRight = this.fontSize/2;

this.xLabelRotation++;

this.xLabelWidth = cosRotation \* originalLabelWidth;

}

if (this.xLabelRotation > 0){

this.endPoint -= Math.sin(toRadians(this.xLabelRotation))\*originalLabelWidth + 3;

}

}

else{

this.xLabelWidth = 0;

this.xScalePaddingRight = this.padding;

this.xScalePaddingLeft = this.padding;

}

},

// Needs to be overidden in each Chart type

// Otherwise we need to pass all the data into the scale class

calculateYRange: noop,

drawingArea: function(){

return this.startPoint - this.endPoint;

},

calculateY : function(value){

var scalingFactor = this.drawingArea() / (this.min - this.max);

return this.endPoint - (scalingFactor \* (value - this.min));

},

calculateX : function(index){

var isRotated = (this.xLabelRotation > 0),

// innerWidth = (this.offsetGridLines) ? this.width - offsetLeft - this.padding : this.width - (offsetLeft + halfLabelWidth \* 2) - this.padding,

innerWidth = this.width - (this.xScalePaddingLeft + this.xScalePaddingRight),

valueWidth = innerWidth/(this.valuesCount - ((this.offsetGridLines) ? 0 : 1)),

valueOffset = (valueWidth \* index) + this.xScalePaddingLeft;

if (this.offsetGridLines){

valueOffset += (valueWidth/2);

}

return Math.round(valueOffset);

},

update : function(newProps){

helpers.extend(this, newProps);

this.fit();

},

draw : function(){

var ctx = this.ctx,

yLabelGap = (this.endPoint - this.startPoint) / this.steps,

xStart = Math.round(this.xScalePaddingLeft);

if (this.display){

ctx.fillStyle = this.textColor;

ctx.font = this.font;

each(this.yLabels,function(labelString,index){

var yLabelCenter = this.endPoint - (yLabelGap \* index),

linePositionY = Math.round(yLabelCenter),

drawHorizontalLine = this.showHorizontalLines;

ctx.textAlign = "right";

ctx.textBaseline = "middle";

if (this.showLabels){

ctx.fillText(labelString,xStart - 10,yLabelCenter);

}

// This is X axis, so draw it

if (index === 0 && !drawHorizontalLine){

drawHorizontalLine = true;

}

if (drawHorizontalLine){

ctx.beginPath();

}

if (index > 0){

// This is a grid line in the centre, so drop that

ctx.lineWidth = this.gridLineWidth;

ctx.strokeStyle = this.gridLineColor;

} else {

// This is the first line on the scale

ctx.lineWidth = this.lineWidth;

ctx.strokeStyle = this.lineColor;

}

linePositionY += helpers.aliasPixel(ctx.lineWidth);

if(drawHorizontalLine){

ctx.moveTo(xStart, linePositionY);

ctx.lineTo(this.width, linePositionY);

ctx.stroke();

ctx.closePath();

}

ctx.lineWidth = this.lineWidth;

ctx.strokeStyle = this.lineColor;

ctx.beginPath();

ctx.moveTo(xStart - 5, linePositionY);

ctx.lineTo(xStart, linePositionY);

ctx.stroke();

ctx.closePath();

},this);

each(this.xLabels,function(label,index){

var xPos = this.calculateX(index) + aliasPixel(this.lineWidth),

// Check to see if line/bar here and decide where to place the line

linePos = this.calculateX(index - (this.offsetGridLines ? 0.5 : 0)) + aliasPixel(this.lineWidth),

isRotated = (this.xLabelRotation > 0),

drawVerticalLine = this.showVerticalLines;

// This is Y axis, so draw it

if (index === 0 && !drawVerticalLine){

drawVerticalLine = true;

}

if (drawVerticalLine){

ctx.beginPath();

}

if (index > 0){

// This is a grid line in the centre, so drop that

ctx.lineWidth = this.gridLineWidth;

ctx.strokeStyle = this.gridLineColor;

} else {

// This is the first line on the scale

ctx.lineWidth = this.lineWidth;

ctx.strokeStyle = this.lineColor;

}

if (drawVerticalLine){

ctx.moveTo(linePos,this.endPoint);

ctx.lineTo(linePos,this.startPoint - 3);

ctx.stroke();

ctx.closePath();

}

ctx.lineWidth = this.lineWidth;

ctx.strokeStyle = this.lineColor;

// Small lines at the bottom of the base grid line

ctx.beginPath();

ctx.moveTo(linePos,this.endPoint);

ctx.lineTo(linePos,this.endPoint + 5);

ctx.stroke();

ctx.closePath();

ctx.save();

ctx.translate(xPos,(isRotated) ? this.endPoint + 12 : this.endPoint + 8);

ctx.rotate(toRadians(this.xLabelRotation)\*-1);

ctx.font = this.font;

ctx.textAlign = (isRotated) ? "right" : "center";

ctx.textBaseline = (isRotated) ? "middle" : "top";

ctx.fillText(label, 0, 0);

ctx.restore();

},this);

}

}

});

Chart.RadialScale = Chart.Element.extend({

initialize: function(){

this.size = min([this.height, this.width]);

this.drawingArea = (this.display) ? (this.size/2) - (this.fontSize/2 + this.backdropPaddingY) : (this.size/2);

},

calculateCenterOffset: function(value){

// Take into account half font size + the yPadding of the top value

var scalingFactor = this.drawingArea / (this.max - this.min);

return (value - this.min) \* scalingFactor;

},

update : function(){

if (!this.lineArc){

this.setScaleSize();

} else {

this.drawingArea = (this.display) ? (this.size/2) - (this.fontSize/2 + this.backdropPaddingY) : (this.size/2);

}

this.buildYLabels();

},

buildYLabels: function(){

this.yLabels = [];

var stepDecimalPlaces = getDecimalPlaces(this.stepValue);

for (var i=0; i<=this.steps; i++){

this.yLabels.push(template(this.templateString,{value:(this.min + (i \* this.stepValue)).toFixed(stepDecimalPlaces)}));

}

},

getCircumference : function(){

return ((Math.PI\*2) / this.valuesCount);

},

setScaleSize: function(){

/\*

\* Right, this is really confusing and there is a lot of maths going on here

\* The gist of the problem is here: https://gist.github.com/nnnick/696cc9c55f4b0beb8fe9

\*

\* Reaction: https://dl.dropboxusercontent.com/u/34601363/toomuchscience.gif

\*

\* Solution:

\*

\* We assume the radius of the polygon is half the size of the canvas at first

\* at each index we check if the text overlaps.

\*

\* Where it does, we store that angle and that index.

\*

\* After finding the largest index and angle we calculate how much we need to remove

\* from the shape radius to move the point inwards by that x.

\*

\* We average the left and right distances to get the maximum shape radius that can fit in the box

\* along with labels.

\*

\* Once we have that, we can find the centre point for the chart, by taking the x text protrusion

\* on each side, removing that from the size, halving it and adding the left x protrusion width.

\*

\* This will mean we have a shape fitted to the canvas, as large as it can be with the labels

\* and position it in the most space efficient manner

\*

\* https://dl.dropboxusercontent.com/u/34601363/yeahscience.gif

\*/

// Get maximum radius of the polygon. Either half the height (minus the text width) or half the width.

// Use this to calculate the offset + change. - Make sure L/R protrusion is at least 0 to stop issues with centre points

var largestPossibleRadius = min([(this.height/2 - this.pointLabelFontSize - 5), this.width/2]),

pointPosition,

i,

textWidth,

halfTextWidth,

furthestRight = this.width,

furthestRightIndex,

furthestRightAngle,

furthestLeft = 0,

furthestLeftIndex,

furthestLeftAngle,

xProtrusionLeft,

xProtrusionRight,

radiusReductionRight,

radiusReductionLeft,

maxWidthRadius;

this.ctx.font = fontString(this.pointLabelFontSize,this.pointLabelFontStyle,this.pointLabelFontFamily);

for (i=0;i<this.valuesCount;i++){

// 5px to space the text slightly out - similar to what we do in the draw function.

pointPosition = this.getPointPosition(i, largestPossibleRadius);

textWidth = this.ctx.measureText(template(this.templateString, { value: this.labels[i] })).width + 5;

if (i === 0 || i === this.valuesCount/2){

// If we're at index zero, or exactly the middle, we're at exactly the top/bottom

// of the radar chart, so text will be aligned centrally, so we'll half it and compare

// w/left and right text sizes

halfTextWidth = textWidth/2;

if (pointPosition.x + halfTextWidth > furthestRight) {

furthestRight = pointPosition.x + halfTextWidth;

furthestRightIndex = i;

}

if (pointPosition.x - halfTextWidth < furthestLeft) {

furthestLeft = pointPosition.x - halfTextWidth;

furthestLeftIndex = i;

}

}

else if (i < this.valuesCount/2) {

// Less than half the values means we'll left align the text

if (pointPosition.x + textWidth > furthestRight) {

furthestRight = pointPosition.x + textWidth;

furthestRightIndex = i;

}

}

else if (i > this.valuesCount/2){

// More than half the values means we'll right align the text

if (pointPosition.x - textWidth < furthestLeft) {

furthestLeft = pointPosition.x - textWidth;

furthestLeftIndex = i;

}

}

}

xProtrusionLeft = furthestLeft;

xProtrusionRight = Math.ceil(furthestRight - this.width);

furthestRightAngle = this.getIndexAngle(furthestRightIndex);

furthestLeftAngle = this.getIndexAngle(furthestLeftIndex);

radiusReductionRight = xProtrusionRight / Math.sin(furthestRightAngle + Math.PI/2);

radiusReductionLeft = xProtrusionLeft / Math.sin(furthestLeftAngle + Math.PI/2);

// Ensure we actually need to reduce the size of the chart

radiusReductionRight = (isNumber(radiusReductionRight)) ? radiusReductionRight : 0;

radiusReductionLeft = (isNumber(radiusReductionLeft)) ? radiusReductionLeft : 0;

this.drawingArea = largestPossibleRadius - (radiusReductionLeft + radiusReductionRight)/2;

//this.drawingArea = min([maxWidthRadius, (this.height - (2 \* (this.pointLabelFontSize + 5)))/2])

this.setCenterPoint(radiusReductionLeft, radiusReductionRight);

},

setCenterPoint: function(leftMovement, rightMovement){

var maxRight = this.width - rightMovement - this.drawingArea,

maxLeft = leftMovement + this.drawingArea;

this.xCenter = (maxLeft + maxRight)/2;

// Always vertically in the centre as the text height doesn't change

this.yCenter = (this.height/2);

},

getIndexAngle : function(index){

var angleMultiplier = (Math.PI \* 2) / this.valuesCount;

// Start from the top instead of right, so remove a quarter of the circle

return index \* angleMultiplier - (Math.PI/2);

},

getPointPosition : function(index, distanceFromCenter){

var thisAngle = this.getIndexAngle(index);

return {

x : (Math.cos(thisAngle) \* distanceFromCenter) + this.xCenter,

y : (Math.sin(thisAngle) \* distanceFromCenter) + this.yCenter

};

},

draw: function(){

if (this.display){

var ctx = this.ctx;

each(this.yLabels, function(label, index){

// Don't draw a centre value

if (index > 0){

var yCenterOffset = index \* (this.drawingArea/this.steps),

yHeight = this.yCenter - yCenterOffset,

pointPosition;

// Draw circular lines around the scale

if (this.lineWidth > 0){

ctx.strokeStyle = this.lineColor;

ctx.lineWidth = this.lineWidth;

if(this.lineArc){

ctx.beginPath();

ctx.arc(this.xCenter, this.yCenter, yCenterOffset, 0, Math.PI\*2);

ctx.closePath();

ctx.stroke();

} else{

ctx.beginPath();

for (var i=0;i<this.valuesCount;i++)

{

pointPosition = this.getPointPosition(i, this.calculateCenterOffset(this.min + (index \* this.stepValue)));

if (i === 0){

ctx.moveTo(pointPosition.x, pointPosition.y);

} else {

ctx.lineTo(pointPosition.x, pointPosition.y);

}

}

ctx.closePath();

ctx.stroke();

}

}

if(this.showLabels){

ctx.font = fontString(this.fontSize,this.fontStyle,this.fontFamily);

if (this.showLabelBackdrop){

var labelWidth = ctx.measureText(label).width;

ctx.fillStyle = this.backdropColor;

ctx.fillRect(

this.xCenter - labelWidth/2 - this.backdropPaddingX,

yHeight - this.fontSize/2 - this.backdropPaddingY,

labelWidth + this.backdropPaddingX\*2,

this.fontSize + this.backdropPaddingY\*2

);

}

ctx.textAlign = 'center';

ctx.textBaseline = "middle";

ctx.fillStyle = this.fontColor;

ctx.fillText(label, this.xCenter, yHeight);

}

}

}, this);

if (!this.lineArc){

ctx.lineWidth = this.angleLineWidth;

ctx.strokeStyle = this.angleLineColor;

for (var i = this.valuesCount - 1; i >= 0; i--) {

if (this.angleLineWidth > 0){

var outerPosition = this.getPointPosition(i, this.calculateCenterOffset(this.max));

ctx.beginPath();

ctx.moveTo(this.xCenter, this.yCenter);

ctx.lineTo(outerPosition.x, outerPosition.y);

ctx.stroke();

ctx.closePath();

}

// Extra 3px out for some label spacing

var pointLabelPosition = this.getPointPosition(i, this.calculateCenterOffset(this.max) + 5);

ctx.font = fontString(this.pointLabelFontSize,this.pointLabelFontStyle,this.pointLabelFontFamily);

ctx.fillStyle = this.pointLabelFontColor;

var labelsCount = this.labels.length,

halfLabelsCount = this.labels.length/2,

quarterLabelsCount = halfLabelsCount/2,

upperHalf = (i < quarterLabelsCount || i > labelsCount - quarterLabelsCount),

exactQuarter = (i === quarterLabelsCount || i === labelsCount - quarterLabelsCount);

if (i === 0){

ctx.textAlign = 'center';

} else if(i === halfLabelsCount){

ctx.textAlign = 'center';

} else if (i < halfLabelsCount){

ctx.textAlign = 'left';

} else {

ctx.textAlign = 'right';

}

// Set the correct text baseline based on outer positioning

if (exactQuarter){

ctx.textBaseline = 'middle';

} else if (upperHalf){

ctx.textBaseline = 'bottom';

} else {

ctx.textBaseline = 'top';

}

ctx.fillText(this.labels[i], pointLabelPosition.x, pointLabelPosition.y);

}

}

}

}

});

// Attach global event to resize each chart instance when the browser resizes

helpers.addEvent(window, "resize", (function(){

// Basic debounce of resize function so it doesn't hurt performance when resizing browser.

var timeout;

return function(){

clearTimeout(timeout);

timeout = setTimeout(function(){

each(Chart.instances,function(instance){

// If the responsive flag is set in the chart instance config

// Cascade the resize event down to the chart.

if (instance.options.responsive){

instance.resize(instance.render, true);

}

});

}, 50);

};

})());

if (amd) {

define(function(){

return Chart;

});

} else if (typeof module === 'object' && module.exports) {

module.exports = Chart;

}

root.Chart = Chart;

Chart.noConflict = function(){

root.Chart = previous;

return Chart;

};

}).call(this);

(function(){

"use strict";

var root = this,

Chart = root.Chart,

helpers = Chart.helpers;

var defaultConfig = {

//Boolean - Whether the scale should start at zero, or an order of magnitude down from the lowest value

scaleBeginAtZero : true,

//Boolean - Whether grid lines are shown across the chart

scaleShowGridLines : true,

//String - Colour of the grid lines

scaleGridLineColor : "rgba(0,0,0,.05)",

//Number - Width of the grid lines

scaleGridLineWidth : 1,

//Boolean - Whether to show horizontal lines (except X axis)

scaleShowHorizontalLines: true,

//Boolean - Whether to show vertical lines (except Y axis)

scaleShowVerticalLines: true,

//Boolean - If there is a stroke on each bar

barShowStroke : true,

//Number - Pixel width of the bar stroke

barStrokeWidth : 2,

//Number - Spacing between each of the X value sets

barValueSpacing : 5,

//Number - Spacing between data sets within X values

barDatasetSpacing : 1,

//String - A legend template

legendTemplate : "<ul class=\"<%=name.toLowerCase()%>-legend\"><% for (var i=0; i<datasets.length; i++){%><li><span style=\"background-color:<%=datasets[i].fillColor%>\"></span><%if(datasets[i].label){%><%=datasets[i].label%><%}%></li><%}%></ul>"

};

Chart.Type.extend({

name: "Bar",

defaults : defaultConfig,

initialize: function(data){

//Expose options as a scope variable here so we can access it in the ScaleClass

var options = this.options;

this.ScaleClass = Chart.Scale.extend({

offsetGridLines : true,

calculateBarX : function(datasetCount, datasetIndex, barIndex){

//Reusable method for calculating the xPosition of a given bar based on datasetIndex & width of the bar

var xWidth = this.calculateBaseWidth(),

xAbsolute = this.calculateX(barIndex) - (xWidth/2),

barWidth = this.calculateBarWidth(datasetCount);

return xAbsolute + (barWidth \* datasetIndex) + (datasetIndex \* options.barDatasetSpacing) + barWidth/2;

},

calculateBaseWidth : function(){

return (this.calculateX(1) - this.calculateX(0)) - (2\*options.barValueSpacing);

},

calculateBarWidth : function(datasetCount){

//The padding between datasets is to the right of each bar, providing that there are more than 1 dataset

var baseWidth = this.calculateBaseWidth() - ((datasetCount - 1) \* options.barDatasetSpacing);

return (baseWidth / datasetCount);

}

});

this.datasets = [];

//Set up tooltip events on the chart

if (this.options.showTooltips){

helpers.bindEvents(this, this.options.tooltipEvents, function(evt){

var activeBars = (evt.type !== 'mouseout') ? this.getBarsAtEvent(evt) : [];

this.eachBars(function(bar){

bar.restore(['fillColor', 'strokeColor']);

});

helpers.each(activeBars, function(activeBar){

activeBar.fillColor = activeBar.highlightFill;

activeBar.strokeColor = activeBar.highlightStroke;

});

this.showTooltip(activeBars);

});

}

//Declare the extension of the default point, to cater for the options passed in to the constructor

this.BarClass = Chart.Rectangle.extend({

strokeWidth : this.options.barStrokeWidth,

showStroke : this.options.barShowStroke,

ctx : this.chart.ctx

});

//Iterate through each of the datasets, and build this into a property of the chart

helpers.each(data.datasets,function(dataset,datasetIndex){

var datasetObject = {

label : dataset.label || null,

fillColor : dataset.fillColor,

strokeColor : dataset.strokeColor,

bars : []

};

this.datasets.push(datasetObject);

helpers.each(dataset.data,function(dataPoint,index){

//Add a new point for each piece of data, passing any required data to draw.

datasetObject.bars.push(new this.BarClass({

value : dataPoint,

label : data.labels[index],

datasetLabel: dataset.label,

strokeColor : dataset.strokeColor,

fillColor : dataset.fillColor,

highlightFill : dataset.highlightFill || dataset.fillColor,

highlightStroke : dataset.highlightStroke || dataset.strokeColor

}));

},this);

},this);

this.buildScale(data.labels);

this.BarClass.prototype.base = this.scale.endPoint;

this.eachBars(function(bar, index, datasetIndex){

helpers.extend(bar, {

width : this.scale.calculateBarWidth(this.datasets.length),

x: this.scale.calculateBarX(this.datasets.length, datasetIndex, index),

y: this.scale.endPoint

});

bar.save();

}, this);

this.render();

},

update : function(){

this.scale.update();

// Reset any highlight colours before updating.

helpers.each(this.activeElements, function(activeElement){

activeElement.restore(['fillColor', 'strokeColor']);

});

this.eachBars(function(bar){

bar.save();

});

this.render();

},

eachBars : function(callback){

helpers.each(this.datasets,function(dataset, datasetIndex){

helpers.each(dataset.bars, callback, this, datasetIndex);

},this);

},

getBarsAtEvent : function(e){

var barsArray = [],

eventPosition = helpers.getRelativePosition(e),

datasetIterator = function(dataset){

barsArray.push(dataset.bars[barIndex]);

},

barIndex;

for (var datasetIndex = 0; datasetIndex < this.datasets.length; datasetIndex++) {

for (barIndex = 0; barIndex < this.datasets[datasetIndex].bars.length; barIndex++) {

if (this.datasets[datasetIndex].bars[barIndex].inRange(eventPosition.x,eventPosition.y)){

helpers.each(this.datasets, datasetIterator);

return barsArray;

}

}

}

return barsArray;

},

buildScale : function(labels){

var self = this;

var dataTotal = function(){

var values = [];

self.eachBars(function(bar){

values.push(bar.value);

});

return values;

};

var scaleOptions = {

templateString : this.options.scaleLabel,

height : this.chart.height,

width : this.chart.width,

ctx : this.chart.ctx,

textColor : this.options.scaleFontColor,

fontSize : this.options.scaleFontSize,

fontStyle : this.options.scaleFontStyle,

fontFamily : this.options.scaleFontFamily,

valuesCount : labels.length,

beginAtZero : this.options.scaleBeginAtZero,

integersOnly : this.options.scaleIntegersOnly,

calculateYRange: function(currentHeight){

var updatedRanges = helpers.calculateScaleRange(

dataTotal(),

currentHeight,

this.fontSize,

this.beginAtZero,

this.integersOnly

);

helpers.extend(this, updatedRanges);

},

xLabels : labels,

font : helpers.fontString(this.options.scaleFontSize, this.options.scaleFontStyle, this.options.scaleFontFamily),

lineWidth : this.options.scaleLineWidth,

lineColor : this.options.scaleLineColor,

showHorizontalLines : this.options.scaleShowHorizontalLines,

showVerticalLines : this.options.scaleShowVerticalLines,

gridLineWidth : (this.options.scaleShowGridLines) ? this.options.scaleGridLineWidth : 0,

gridLineColor : (this.options.scaleShowGridLines) ? this.options.scaleGridLineColor : "rgba(0,0,0,0)",

padding : (this.options.showScale) ? 0 : (this.options.barShowStroke) ? this.options.barStrokeWidth : 0,

showLabels : this.options.scaleShowLabels,

display : this.options.showScale

};

if (this.options.scaleOverride){

helpers.extend(scaleOptions, {

calculateYRange: helpers.noop,

steps: this.options.scaleSteps,

stepValue: this.options.scaleStepWidth,

min: this.options.scaleStartValue,

max: this.options.scaleStartValue + (this.options.scaleSteps \* this.options.scaleStepWidth)

});

}

this.scale = new this.ScaleClass(scaleOptions);

},

addData : function(valuesArray,label){

//Map the values array for each of the datasets

helpers.each(valuesArray,function(value,datasetIndex){

//Add a new point for each piece of data, passing any required data to draw.

this.datasets[datasetIndex].bars.push(new this.BarClass({

value : value,

label : label,

x: this.scale.calculateBarX(this.datasets.length, datasetIndex, this.scale.valuesCount+1),

y: this.scale.endPoint,

width : this.scale.calculateBarWidth(this.datasets.length),

base : this.scale.endPoint,

strokeColor : this.datasets[datasetIndex].strokeColor,

fillColor : this.datasets[datasetIndex].fillColor

}));

},this);

this.scale.addXLabel(label);

//Then re-render the chart.

this.update();

},

removeData : function(){

this.scale.removeXLabel();

//Then re-render the chart.

helpers.each(this.datasets,function(dataset){

dataset.bars.shift();

},this);

this.update();

},

reflow : function(){

helpers.extend(this.BarClass.prototype,{

y: this.scale.endPoint,

base : this.scale.endPoint

});

var newScaleProps = helpers.extend({

height : this.chart.height,

width : this.chart.width

});

this.scale.update(newScaleProps);

},

draw : function(ease){

var easingDecimal = ease || 1;

this.clear();

var ctx = this.chart.ctx;

this.scale.draw(easingDecimal);

//Draw all the bars for each dataset

helpers.each(this.datasets,function(dataset,datasetIndex){

helpers.each(dataset.bars,function(bar,index){

if (bar.hasValue()){

bar.base = this.scale.endPoint;

//Transition then draw

bar.transition({

x : this.scale.calculateBarX(this.datasets.length, datasetIndex, index),

y : this.scale.calculateY(bar.value),

width : this.scale.calculateBarWidth(this.datasets.length)

}, easingDecimal).draw();

}

},this);

},this);

}

});

}).call(this);

(function(){

"use strict";

var root = this,

Chart = root.Chart,

//Cache a local reference to Chart.helpers

helpers = Chart.helpers;

var defaultConfig = {

//Boolean - Whether we should show a stroke on each segment

segmentShowStroke : true,

//String - The colour of each segment stroke

segmentStrokeColor : "#fff",

//Number - The width of each segment stroke

segmentStrokeWidth : 2,

//The percentage of the chart that we cut out of the middle.

percentageInnerCutout : 50,

//Number - Amount of animation steps

animationSteps : 100,

//String - Animation easing effect

animationEasing : "easeOutBounce",

//Boolean - Whether we animate the rotation of the Doughnut

animateRotate : true,

//Boolean - Whether we animate scaling the Doughnut from the centre

animateScale : false,

//String - A legend template

legendTemplate : "<ul class=\"<%=name.toLowerCase()%>-legend\"><% for (var i=0; i<segments.length; i++){%><li><span style=\"background-color:<%=segments[i].fillColor%>\"></span><%if(segments[i].label){%><%=segments[i].label%><%}%></li><%}%></ul>"

};

Chart.Type.extend({

//Passing in a name registers this chart in the Chart namespace

name: "Doughnut",

//Providing a defaults will also register the deafults in the chart namespace

defaults : defaultConfig,

//Initialize is fired when the chart is initialized - Data is passed in as a parameter

//Config is automatically merged by the core of Chart.js, and is available at this.options

initialize: function(data){

//Declare segments as a static property to prevent inheriting across the Chart type prototype

this.segments = [];

this.outerRadius = (helpers.min([this.chart.width,this.chart.height]) - this.options.segmentStrokeWidth/2)/2;

this.SegmentArc = Chart.Arc.extend({

ctx : this.chart.ctx,

x : this.chart.width/2,

y : this.chart.height/2

});

//Set up tooltip events on the chart

if (this.options.showTooltips){

helpers.bindEvents(this, this.options.tooltipEvents, function(evt){

var activeSegments = (evt.type !== 'mouseout') ? this.getSegmentsAtEvent(evt) : [];

helpers.each(this.segments,function(segment){

segment.restore(["fillColor"]);

});

helpers.each(activeSegments,function(activeSegment){

activeSegment.fillColor = activeSegment.highlightColor;

});

this.showTooltip(activeSegments);

});

}

this.calculateTotal(data);

helpers.each(data,function(datapoint, index){

this.addData(datapoint, index, true);

},this);

this.render();

},

getSegmentsAtEvent : function(e){

var segmentsArray = [];

var location = helpers.getRelativePosition(e);

helpers.each(this.segments,function(segment){

if (segment.inRange(location.x,location.y)) segmentsArray.push(segment);

},this);

return segmentsArray;

},

addData : function(segment, atIndex, silent){

var index = atIndex || this.segments.length;

this.segments.splice(index, 0, new this.SegmentArc({

value : segment.value,

outerRadius : (this.options.animateScale) ? 0 : this.outerRadius,

innerRadius : (this.options.animateScale) ? 0 : (this.outerRadius/100) \* this.options.percentageInnerCutout,

fillColor : segment.color,

highlightColor : segment.highlight || segment.color,

showStroke : this.options.segmentShowStroke,

strokeWidth : this.options.segmentStrokeWidth,

strokeColor : this.options.segmentStrokeColor,

startAngle : Math.PI \* 1.5,

circumference : (this.options.animateRotate) ? 0 : this.calculateCircumference(segment.value),

label : segment.label

}));

if (!silent){

this.reflow();

this.update();

}

},

calculateCircumference : function(value){

return (Math.PI\*2)\*(value / this.total);

},

calculateTotal : function(data){

this.total = 0;

helpers.each(data,function(segment){

this.total += segment.value;

},this);

},

update : function(){

this.calculateTotal(this.segments);

// Reset any highlight colours before updating.

helpers.each(this.activeElements, function(activeElement){

activeElement.restore(['fillColor']);

});

helpers.each(this.segments,function(segment){

segment.save();

});

this.render();

},

removeData: function(atIndex){

var indexToDelete = (helpers.isNumber(atIndex)) ? atIndex : this.segments.length-1;

this.segments.splice(indexToDelete, 1);

this.reflow();

this.update();

},

reflow : function(){

helpers.extend(this.SegmentArc.prototype,{

x : this.chart.width/2,

y : this.chart.height/2

});

this.outerRadius = (helpers.min([this.chart.width,this.chart.height]) - this.options.segmentStrokeWidth/2)/2;

helpers.each(this.segments, function(segment){

segment.update({

outerRadius : this.outerRadius,

innerRadius : (this.outerRadius/100) \* this.options.percentageInnerCutout

});

}, this);

},

draw : function(easeDecimal){

var animDecimal = (easeDecimal) ? easeDecimal : 1;

this.clear();

helpers.each(this.segments,function(segment,index){

segment.transition({

circumference : this.calculateCircumference(segment.value),

outerRadius : this.outerRadius,

innerRadius : (this.outerRadius/100) \* this.options.percentageInnerCutout

},animDecimal);

segment.endAngle = segment.startAngle + segment.circumference;

segment.draw();

if (index === 0){

segment.startAngle = Math.PI \* 1.5;

}

//Check to see if it's the last segment, if not get the next and update the start angle

if (index < this.segments.length-1){

this.segments[index+1].startAngle = segment.endAngle;

}

},this);

}

});

Chart.types.Doughnut.extend({

name : "Pie",

defaults : helpers.merge(defaultConfig,{percentageInnerCutout : 0})

});

}).call(this);

(function(){

"use strict";

var root = this,

Chart = root.Chart,

helpers = Chart.helpers;

var defaultConfig = {

///Boolean - Whether grid lines are shown across the chart

scaleShowGridLines : true,

//String - Colour of the grid lines

scaleGridLineColor : "rgba(0,0,0,.05)",

//Number - Width of the grid lines

scaleGridLineWidth : 1,

//Boolean - Whether to show horizontal lines (except X axis)

scaleShowHorizontalLines: true,

//Boolean - Whether to show vertical lines (except Y axis)

scaleShowVerticalLines: true,

//Boolean - Whether the line is curved between points

bezierCurve : true,

//Number - Tension of the bezier curve between points

bezierCurveTension : 0.4,

//Boolean - Whether to show a dot for each point

pointDot : true,

//Number - Radius of each point dot in pixels

pointDotRadius : 4,

//Number - Pixel width of point dot stroke

pointDotStrokeWidth : 1,

//Number - amount extra to add to the radius to cater for hit detection outside the drawn point

pointHitDetectionRadius : 20,

//Boolean - Whether to show a stroke for datasets

datasetStroke : true,

//Number - Pixel width of dataset stroke

datasetStrokeWidth : 2,

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

datasetFill : true,

//String - A legend template

legendTemplate : "<ul class=\"<%=name.toLowerCase()%>-legend\"><% for (var i=0; i<datasets.length; i++){%><li><span style=\"background-color:<%=datasets[i].strokeColor%>\"></span><%if(datasets[i].label){%><%=datasets[i].label%><%}%></li><%}%></ul>"

};

Chart.Type.extend({

name: "Line",

defaults : defaultConfig,

initialize: function(data){

//Declare the extension of the default point, to cater for the options passed in to the constructor

this.PointClass = Chart.Point.extend({

strokeWidth : this.options.pointDotStrokeWidth,

radius : this.options.pointDotRadius,

display: this.options.pointDot,

hitDetectionRadius : this.options.pointHitDetectionRadius,

ctx : this.chart.ctx,

inRange : function(mouseX){

return (Math.pow(mouseX-this.x, 2) < Math.pow(this.radius + this.hitDetectionRadius,2));

}

});

this.datasets = [];

//Set up tooltip events on the chart

if (this.options.showTooltips){

helpers.bindEvents(this, this.options.tooltipEvents, function(evt){

var activePoints = (evt.type !== 'mouseout') ? this.getPointsAtEvent(evt) : [];

this.eachPoints(function(point){

point.restore(['fillColor', 'strokeColor']);

});

helpers.each(activePoints, function(activePoint){

activePoint.fillColor = activePoint.highlightFill;

activePoint.strokeColor = activePoint.highlightStroke;

});

this.showTooltip(activePoints);

});

}

//Iterate through each of the datasets, and build this into a property of the chart

helpers.each(data.datasets,function(dataset){

var datasetObject = {

label : dataset.label || null,

fillColor : dataset.fillColor,

strokeColor : dataset.strokeColor,

pointColor : dataset.pointColor,

pointStrokeColor : dataset.pointStrokeColor,

points : []

};

this.datasets.push(datasetObject);

helpers.each(dataset.data,function(dataPoint,index){

//Add a new point for each piece of data, passing any required data to draw.

datasetObject.points.push(new this.PointClass({

value : dataPoint,

label : data.labels[index],

datasetLabel: dataset.label,

strokeColor : dataset.pointStrokeColor,

fillColor : dataset.pointColor,

highlightFill : dataset.pointHighlightFill || dataset.pointColor,

highlightStroke : dataset.pointHighlightStroke || dataset.pointStrokeColor

}));

},this);

this.buildScale(data.labels);

this.eachPoints(function(point, index){

helpers.extend(point, {

x: this.scale.calculateX(index),

y: this.scale.endPoint

});

point.save();

}, this);

},this);

this.render();

},

update : function(){

this.scale.update();

// Reset any highlight colours before updating.

helpers.each(this.activeElements, function(activeElement){

activeElement.restore(['fillColor', 'strokeColor']);

});

this.eachPoints(function(point){

point.save();

});

this.render();

},

eachPoints : function(callback){

helpers.each(this.datasets,function(dataset){

helpers.each(dataset.points,callback,this);

},this);

},

getPointsAtEvent : function(e){

var pointsArray = [],

eventPosition = helpers.getRelativePosition(e);

helpers.each(this.datasets,function(dataset){

helpers.each(dataset.points,function(point){

if (point.inRange(eventPosition.x,eventPosition.y)) pointsArray.push(point);

});

},this);

return pointsArray;

},

buildScale : function(labels){

var self = this;

var dataTotal = function(){

var values = [];

self.eachPoints(function(point){

values.push(point.value);

});

return values;

};

var scaleOptions = {

templateString : this.options.scaleLabel,

height : this.chart.height,

width : this.chart.width,

ctx : this.chart.ctx,

textColor : this.options.scaleFontColor,

fontSize : this.options.scaleFontSize,

fontStyle : this.options.scaleFontStyle,

fontFamily : this.options.scaleFontFamily,

valuesCount : labels.length,

beginAtZero : this.options.scaleBeginAtZero,

integersOnly : this.options.scaleIntegersOnly,

calculateYRange : function(currentHeight){

var updatedRanges = helpers.calculateScaleRange(

dataTotal(),

currentHeight,

this.fontSize,

this.beginAtZero,

this.integersOnly

);

helpers.extend(this, updatedRanges);

},

xLabels : labels,

font : helpers.fontString(this.options.scaleFontSize, this.options.scaleFontStyle, this.options.scaleFontFamily),

lineWidth : this.options.scaleLineWidth,

lineColor : this.options.scaleLineColor,

showHorizontalLines : this.options.scaleShowHorizontalLines,

showVerticalLines : this.options.scaleShowVerticalLines,

gridLineWidth : (this.options.scaleShowGridLines) ? this.options.scaleGridLineWidth : 0,

gridLineColor : (this.options.scaleShowGridLines) ? this.options.scaleGridLineColor : "rgba(0,0,0,0)",

padding: (this.options.showScale) ? 0 : this.options.pointDotRadius + this.options.pointDotStrokeWidth,

showLabels : this.options.scaleShowLabels,

display : this.options.showScale

};

if (this.options.scaleOverride){

helpers.extend(scaleOptions, {

calculateYRange: helpers.noop,

steps: this.options.scaleSteps,

stepValue: this.options.scaleStepWidth,

min: this.options.scaleStartValue,

max: this.options.scaleStartValue + (this.options.scaleSteps \* this.options.scaleStepWidth)

});

}

this.scale = new Chart.Scale(scaleOptions);

},

addData : function(valuesArray,label){

//Map the values array for each of the datasets

helpers.each(valuesArray,function(value,datasetIndex){

//Add a new point for each piece of data, passing any required data to draw.

this.datasets[datasetIndex].points.push(new this.PointClass({

value : value,

label : label,

x: this.scale.calculateX(this.scale.valuesCount+1),

y: this.scale.endPoint,

strokeColor : this.datasets[datasetIndex].pointStrokeColor,

fillColor : this.datasets[datasetIndex].pointColor

}));

},this);

this.scale.addXLabel(label);

//Then re-render the chart.

this.update();

},

removeData : function(){

this.scale.removeXLabel();

//Then re-render the chart.

helpers.each(this.datasets,function(dataset){

dataset.points.shift();

},this);

this.update();

},

reflow : function(){

var newScaleProps = helpers.extend({

height : this.chart.height,

width : this.chart.width

});

this.scale.update(newScaleProps);

},

draw : function(ease){

var easingDecimal = ease || 1;

this.clear();

var ctx = this.chart.ctx;

// Some helper methods for getting the next/prev points

var hasValue = function(item){

return item.value !== null;

},

nextPoint = function(point, collection, index){

return helpers.findNextWhere(collection, hasValue, index) || point;

},

previousPoint = function(point, collection, index){

return helpers.findPreviousWhere(collection, hasValue, index) || point;

};

this.scale.draw(easingDecimal);

helpers.each(this.datasets,function(dataset){

var pointsWithValues = helpers.where(dataset.points, hasValue);

//Transition each point first so that the line and point drawing isn't out of sync

//We can use this extra loop to calculate the control points of this dataset also in this loop

helpers.each(dataset.points, function(point, index){

if (point.hasValue()){

point.transition({

y : this.scale.calculateY(point.value),

x : this.scale.calculateX(index)

}, easingDecimal);

}

},this);

// Control points need to be calculated in a seperate loop, because we need to know the current x/y of the point

// This would cause issues when there is no animation, because the y of the next point would be 0, so beziers would be skewed

if (this.options.bezierCurve){

helpers.each(pointsWithValues, function(point, index){

var tension = (index > 0 && index < pointsWithValues.length - 1) ? this.options.bezierCurveTension : 0;

point.controlPoints = helpers.splineCurve(

previousPoint(point, pointsWithValues, index),

point,

nextPoint(point, pointsWithValues, index),

tension

);

// Prevent the bezier going outside of the bounds of the graph

// Cap puter bezier handles to the upper/lower scale bounds

if (point.controlPoints.outer.y > this.scale.endPoint){

point.controlPoints.outer.y = this.scale.endPoint;

}

else if (point.controlPoints.outer.y < this.scale.startPoint){

point.controlPoints.outer.y = this.scale.startPoint;

}

// Cap inner bezier handles to the upper/lower scale bounds

if (point.controlPoints.inner.y > this.scale.endPoint){

point.controlPoints.inner.y = this.scale.endPoint;

}

else if (point.controlPoints.inner.y < this.scale.startPoint){

point.controlPoints.inner.y = this.scale.startPoint;

}

},this);

}

//Draw the line between all the points

ctx.lineWidth = this.options.datasetStrokeWidth;

ctx.strokeStyle = dataset.strokeColor;

ctx.beginPath();

helpers.each(pointsWithValues, function(point, index){

if (index === 0){

ctx.moveTo(point.x, point.y);

}

else{

if(this.options.bezierCurve){

var previous = previousPoint(point, pointsWithValues, index);

ctx.bezierCurveTo(

previous.controlPoints.outer.x,

previous.controlPoints.outer.y,

point.controlPoints.inner.x,

point.controlPoints.inner.y,

point.x,

point.y

);

}

else{

ctx.lineTo(point.x,point.y);

}

}

}, this);

ctx.stroke();

if (this.options.datasetFill && pointsWithValues.length > 0){

//Round off the line by going to the base of the chart, back to the start, then fill.

ctx.lineTo(pointsWithValues[pointsWithValues.length - 1].x, this.scale.endPoint);

ctx.lineTo(pointsWithValues[0].x, this.scale.endPoint);

ctx.fillStyle = dataset.fillColor;

ctx.closePath();

ctx.fill();

}

//Now draw the points over the line

//A little inefficient double looping, but better than the line

//lagging behind the point positions

helpers.each(pointsWithValues,function(point){

point.draw();

});

},this);

}

});

}).call(this);

(function(){

"use strict";

var root = this,

Chart = root.Chart,

//Cache a local reference to Chart.helpers

helpers = Chart.helpers;

var defaultConfig = {

//Boolean - Show a backdrop to the scale label

scaleShowLabelBackdrop : true,

//String - The colour of the label backdrop

scaleBackdropColor : "rgba(255,255,255,0.75)",

// Boolean - Whether the scale should begin at zero

scaleBeginAtZero : true,

//Number - The backdrop padding above & below the label in pixels

scaleBackdropPaddingY : 2,

//Number - The backdrop padding to the side of the label in pixels

scaleBackdropPaddingX : 2,

//Boolean - Show line for each value in the scale

scaleShowLine : true,

//Boolean - Stroke a line around each segment in the chart

segmentShowStroke : true,

//String - The colour of the stroke on each segement.

segmentStrokeColor : "#fff",

//Number - The width of the stroke value in pixels

segmentStrokeWidth : 2,

//Number - Amount of animation steps

animationSteps : 100,

//String - Animation easing effect.

animationEasing : "easeOutBounce",

//Boolean - Whether to animate the rotation of the chart

animateRotate : true,

//Boolean - Whether to animate scaling the chart from the centre

animateScale : false,

//String - A legend template

legendTemplate : "<ul class=\"<%=name.toLowerCase()%>-legend\"><% for (var i=0; i<segments.length; i++){%><li><span style=\"background-color:<%=segments[i].fillColor%>\"></span><%if(segments[i].label){%><%=segments[i].label%><%}%></li><%}%></ul>"

};

Chart.Type.extend({

//Passing in a name registers this chart in the Chart namespace

name: "PolarArea",

//Providing a defaults will also register the deafults in the chart namespace

defaults : defaultConfig,

//Initialize is fired when the chart is initialized - Data is passed in as a parameter

//Config is automatically merged by the core of Chart.js, and is available at this.options

initialize: function(data){

this.segments = [];

//Declare segment class as a chart instance specific class, so it can share props for this instance

this.SegmentArc = Chart.Arc.extend({

showStroke : this.options.segmentShowStroke,

strokeWidth : this.options.segmentStrokeWidth,

strokeColor : this.options.segmentStrokeColor,

ctx : this.chart.ctx,

innerRadius : 0,

x : this.chart.width/2,

y : this.chart.height/2

});

this.scale = new Chart.RadialScale({

display: this.options.showScale,

fontStyle: this.options.scaleFontStyle,

fontSize: this.options.scaleFontSize,

fontFamily: this.options.scaleFontFamily,

fontColor: this.options.scaleFontColor,

showLabels: this.options.scaleShowLabels,

showLabelBackdrop: this.options.scaleShowLabelBackdrop,

backdropColor: this.options.scaleBackdropColor,

backdropPaddingY : this.options.scaleBackdropPaddingY,

backdropPaddingX: this.options.scaleBackdropPaddingX,

lineWidth: (this.options.scaleShowLine) ? this.options.scaleLineWidth : 0,

lineColor: this.options.scaleLineColor,

lineArc: true,

width: this.chart.width,

height: this.chart.height,

xCenter: this.chart.width/2,

yCenter: this.chart.height/2,

ctx : this.chart.ctx,

templateString: this.options.scaleLabel,

valuesCount: data.length

});

this.updateScaleRange(data);

this.scale.update();

helpers.each(data,function(segment,index){

this.addData(segment,index,true);

},this);

//Set up tooltip events on the chart

if (this.options.showTooltips){

helpers.bindEvents(this, this.options.tooltipEvents, function(evt){

var activeSegments = (evt.type !== 'mouseout') ? this.getSegmentsAtEvent(evt) : [];

helpers.each(this.segments,function(segment){

segment.restore(["fillColor"]);

});

helpers.each(activeSegments,function(activeSegment){

activeSegment.fillColor = activeSegment.highlightColor;

});

this.showTooltip(activeSegments);

});

}

this.render();

},

getSegmentsAtEvent : function(e){

var segmentsArray = [];

var location = helpers.getRelativePosition(e);

helpers.each(this.segments,function(segment){

if (segment.inRange(location.x,location.y)) segmentsArray.push(segment);

},this);

return segmentsArray;

},

addData : function(segment, atIndex, silent){

var index = atIndex || this.segments.length;

this.segments.splice(index, 0, new this.SegmentArc({

fillColor: segment.color,

highlightColor: segment.highlight || segment.color,

label: segment.label,

value: segment.value,

outerRadius: (this.options.animateScale) ? 0 : this.scale.calculateCenterOffset(segment.value),

circumference: (this.options.animateRotate) ? 0 : this.scale.getCircumference(),

startAngle: Math.PI \* 1.5

}));

if (!silent){

this.reflow();

this.update();

}

},

removeData: function(atIndex){

var indexToDelete = (helpers.isNumber(atIndex)) ? atIndex : this.segments.length-1;

this.segments.splice(indexToDelete, 1);

this.reflow();

this.update();

},

calculateTotal: function(data){

this.total = 0;

helpers.each(data,function(segment){

this.total += segment.value;

},this);

this.scale.valuesCount = this.segments.length;

},

updateScaleRange: function(datapoints){

var valuesArray = [];

helpers.each(datapoints,function(segment){

valuesArray.push(segment.value);

});

var scaleSizes = (this.options.scaleOverride) ?

{

steps: this.options.scaleSteps,

stepValue: this.options.scaleStepWidth,

min: this.options.scaleStartValue,

max: this.options.scaleStartValue + (this.options.scaleSteps \* this.options.scaleStepWidth)

} :

helpers.calculateScaleRange(

valuesArray,

helpers.min([this.chart.width, this.chart.height])/2,

this.options.scaleFontSize,

this.options.scaleBeginAtZero,

this.options.scaleIntegersOnly

);

helpers.extend(

this.scale,

scaleSizes,

{

size: helpers.min([this.chart.width, this.chart.height]),

xCenter: this.chart.width/2,

yCenter: this.chart.height/2

}

);

},

update : function(){

this.calculateTotal(this.segments);

helpers.each(this.segments,function(segment){

segment.save();

});

this.render();

},

reflow : function(){

helpers.extend(this.SegmentArc.prototype,{

x : this.chart.width/2,

y : this.chart.height/2

});

this.updateScaleRange(this.segments);

this.scale.update();

helpers.extend(this.scale,{

xCenter: this.chart.width/2,

yCenter: this.chart.height/2

});

helpers.each(this.segments, function(segment){

segment.update({

outerRadius : this.scale.calculateCenterOffset(segment.value)

});

}, this);

},

draw : function(ease){

var easingDecimal = ease || 1;

//Clear & draw the canvas

this.clear();

helpers.each(this.segments,function(segment, index){

segment.transition({

circumference : this.scale.getCircumference(),

outerRadius : this.scale.calculateCenterOffset(segment.value)

},easingDecimal);

segment.endAngle = segment.startAngle + segment.circumference;

// If we've removed the first segment we need to set the first one to

// start at the top.

if (index === 0){

segment.startAngle = Math.PI \* 1.5;

}

//Check to see if it's the last segment, if not get the next and update the start angle

if (index < this.segments.length - 1){

this.segments[index+1].startAngle = segment.endAngle;

}

segment.draw();

}, this);

this.scale.draw();

}

});

}).call(this);

(function(){

"use strict";

var root = this,

Chart = root.Chart,

helpers = Chart.helpers;

Chart.Type.extend({

name: "Radar",

defaults:{

//Boolean - Whether to show lines for each scale point

scaleShowLine : true,

//Boolean - Whether we show the angle lines out of the radar

angleShowLineOut : true,

//Boolean - Whether to show labels on the scale

scaleShowLabels : false,

// Boolean - Whether the scale should begin at zero

scaleBeginAtZero : true,

//String - Colour of the angle line

angleLineColor : "rgba(0,0,0,.1)",

//Number - Pixel width of the angle line

angleLineWidth : 1,

//String - Point label font declaration

pointLabelFontFamily : "'Arial'",

//String - Point label font weight

pointLabelFontStyle : "normal",

//Number - Point label font size in pixels

pointLabelFontSize : 10,

//String - Point label font colour

pointLabelFontColor : "#666",

//Boolean - Whether to show a dot for each point

pointDot : true,

//Number - Radius of each point dot in pixels

pointDotRadius : 3,

//Number - Pixel width of point dot stroke

pointDotStrokeWidth : 1,

//Number - amount extra to add to the radius to cater for hit detection outside the drawn point

pointHitDetectionRadius : 20,

//Boolean - Whether to show a stroke for datasets

datasetStroke : true,

//Number - Pixel width of dataset stroke

datasetStrokeWidth : 2,

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

datasetFill : true,

//String - A legend template

legendTemplate : "<ul class=\"<%=name.toLowerCase()%>-legend\"><% for (var i=0; i<datasets.length; i++){%><li><span style=\"background-color:<%=datasets[i].strokeColor%>\"></span><%if(datasets[i].label){%><%=datasets[i].label%><%}%></li><%}%></ul>"

},

initialize: function(data){

this.PointClass = Chart.Point.extend({

strokeWidth : this.options.pointDotStrokeWidth,

radius : this.options.pointDotRadius,

display: this.options.pointDot,

hitDetectionRadius : this.options.pointHitDetectionRadius,

ctx : this.chart.ctx

});

this.datasets = [];

this.buildScale(data);

//Set up tooltip events on the chart

if (this.options.showTooltips){

helpers.bindEvents(this, this.options.tooltipEvents, function(evt){

var activePointsCollection = (evt.type !== 'mouseout') ? this.getPointsAtEvent(evt) : [];

this.eachPoints(function(point){

point.restore(['fillColor', 'strokeColor']);

});

helpers.each(activePointsCollection, function(activePoint){

activePoint.fillColor = activePoint.highlightFill;

activePoint.strokeColor = activePoint.highlightStroke;

});

this.showTooltip(activePointsCollection);

});

}

//Iterate through each of the datasets, and build this into a property of the chart

helpers.each(data.datasets,function(dataset){

var datasetObject = {

label: dataset.label || null,

fillColor : dataset.fillColor,

strokeColor : dataset.strokeColor,

pointColor : dataset.pointColor,

pointStrokeColor : dataset.pointStrokeColor,

points : []

};

this.datasets.push(datasetObject);

helpers.each(dataset.data,function(dataPoint,index){

//Add a new point for each piece of data, passing any required data to draw.

var pointPosition;

if (!this.scale.animation){

pointPosition = this.scale.getPointPosition(index, this.scale.calculateCenterOffset(dataPoint));

}

datasetObject.points.push(new this.PointClass({

value : dataPoint,

label : data.labels[index],

datasetLabel: dataset.label,

x: (this.options.animation) ? this.scale.xCenter : pointPosition.x,

y: (this.options.animation) ? this.scale.yCenter : pointPosition.y,

strokeColor : dataset.pointStrokeColor,

fillColor : dataset.pointColor,

highlightFill : dataset.pointHighlightFill || dataset.pointColor,

highlightStroke : dataset.pointHighlightStroke || dataset.pointStrokeColor

}));

},this);

},this);

this.render();

},

eachPoints : function(callback){

helpers.each(this.datasets,function(dataset){

helpers.each(dataset.points,callback,this);

},this);

},

getPointsAtEvent : function(evt){

var mousePosition = helpers.getRelativePosition(evt),

fromCenter = helpers.getAngleFromPoint({

x: this.scale.xCenter,

y: this.scale.yCenter

}, mousePosition);

var anglePerIndex = (Math.PI \* 2) /this.scale.valuesCount,

pointIndex = Math.round((fromCenter.angle - Math.PI \* 1.5) / anglePerIndex),

activePointsCollection = [];

// If we're at the top, make the pointIndex 0 to get the first of the array.

if (pointIndex >= this.scale.valuesCount || pointIndex < 0){

pointIndex = 0;

}

if (fromCenter.distance <= this.scale.drawingArea){

helpers.each(this.datasets, function(dataset){

activePointsCollection.push(dataset.points[pointIndex]);

});

}

return activePointsCollection;

},

buildScale : function(data){

this.scale = new Chart.RadialScale({

display: this.options.showScale,

fontStyle: this.options.scaleFontStyle,

fontSize: this.options.scaleFontSize,

fontFamily: this.options.scaleFontFamily,

fontColor: this.options.scaleFontColor,

showLabels: this.options.scaleShowLabels,

showLabelBackdrop: this.options.scaleShowLabelBackdrop,

backdropColor: this.options.scaleBackdropColor,

backdropPaddingY : this.options.scaleBackdropPaddingY,

backdropPaddingX: this.options.scaleBackdropPaddingX,

lineWidth: (this.options.scaleShowLine) ? this.options.scaleLineWidth : 0,

lineColor: this.options.scaleLineColor,

angleLineColor : this.options.angleLineColor,

angleLineWidth : (this.options.angleShowLineOut) ? this.options.angleLineWidth : 0,

// Point labels at the edge of each line

pointLabelFontColor : this.options.pointLabelFontColor,

pointLabelFontSize : this.options.pointLabelFontSize,

pointLabelFontFamily : this.options.pointLabelFontFamily,

pointLabelFontStyle : this.options.pointLabelFontStyle,

height : this.chart.height,

width: this.chart.width,

xCenter: this.chart.width/2,

yCenter: this.chart.height/2,

ctx : this.chart.ctx,

templateString: this.options.scaleLabel,

labels: data.labels,

valuesCount: data.datasets[0].data.length

});

this.scale.setScaleSize();

this.updateScaleRange(data.datasets);

this.scale.buildYLabels();

},

updateScaleRange: function(datasets){

var valuesArray = (function(){

var totalDataArray = [];

helpers.each(datasets,function(dataset){

if (dataset.data){

totalDataArray = totalDataArray.concat(dataset.data);

}

else {

helpers.each(dataset.points, function(point){

totalDataArray.push(point.value);

});

}

});

return totalDataArray;

})();

var scaleSizes = (this.options.scaleOverride) ?

{

steps: this.options.scaleSteps,

stepValue: this.options.scaleStepWidth,

min: this.options.scaleStartValue,

max: this.options.scaleStartValue + (this.options.scaleSteps \* this.options.scaleStepWidth)

} :

helpers.calculateScaleRange(

valuesArray,

helpers.min([this.chart.width, this.chart.height])/2,

this.options.scaleFontSize,

this.options.scaleBeginAtZero,

this.options.scaleIntegersOnly

);

helpers.extend(

this.scale,

scaleSizes

);

},

addData : function(valuesArray,label){

//Map the values array for each of the datasets

this.scale.valuesCount++;

helpers.each(valuesArray,function(value,datasetIndex){

var pointPosition = this.scale.getPointPosition(this.scale.valuesCount, this.scale.calculateCenterOffset(value));

this.datasets[datasetIndex].points.push(new this.PointClass({

value : value,

label : label,

x: pointPosition.x,

y: pointPosition.y,

strokeColor : this.datasets[datasetIndex].pointStrokeColor,

fillColor : this.datasets[datasetIndex].pointColor

}));

},this);

this.scale.labels.push(label);

this.reflow();

this.update();

},

removeData : function(){

this.scale.valuesCount--;

this.scale.labels.shift();

helpers.each(this.datasets,function(dataset){

dataset.points.shift();

},this);

this.reflow();

this.update();

},

update : function(){

this.eachPoints(function(point){

point.save();

});

this.reflow();

this.render();

},

reflow: function(){

helpers.extend(this.scale, {

width : this.chart.width,

height: this.chart.height,

size : helpers.min([this.chart.width, this.chart.height]),

xCenter: this.chart.width/2,

yCenter: this.chart.height/2

});

this.updateScaleRange(this.datasets);

this.scale.setScaleSize();

this.scale.buildYLabels();

},

draw : function(ease){

var easeDecimal = ease || 1,

ctx = this.chart.ctx;

this.clear();

this.scale.draw();

helpers.each(this.datasets,function(dataset){

//Transition each point first so that the line and point drawing isn't out of sync

helpers.each(dataset.points,function(point,index){

if (point.hasValue()){

point.transition(this.scale.getPointPosition(index, this.scale.calculateCenterOffset(point.value)), easeDecimal);

}

},this);

//Draw the line between all the points

ctx.lineWidth = this.options.datasetStrokeWidth;

ctx.strokeStyle = dataset.strokeColor;

ctx.beginPath();

helpers.each(dataset.points,function(point,index){

if (index === 0){

ctx.moveTo(point.x,point.y);

}

else{

ctx.lineTo(point.x,point.y);

}

},this);

ctx.closePath();

ctx.stroke();

ctx.fillStyle = dataset.fillColor;

ctx.fill();

//Now draw the points over the line

//A little inefficient double looping, but better than the line

//lagging behind the point positions

helpers.each(dataset.points,function(point){

if (point.hasValue()){

point.draw();

}

});

},this);

}

});

}).call(this);