JIDE Charts Developer Guide

Contents

Purpose of This Document

This developer guide is for those who want to develop applications using *JIDE Charts*. JIDE Charts offers an extremely powerful Swing charting capability that breathes life into your data and those of your clients. It generates some great looking charts, but it does much more than that too – by following this guide you can write applications that enable your users not only to see their data, but to interactively *explore* it. You can easily add features such as data point tooltips and advanced selection behaviours. You can add mouse-wheel zooming and mouse-drag panning to a chart with a single line of code. You can easily switch from one visual paradigm to another – like switching from a bar chart to a pie chart – with minimal coding effort. You can make the charts visually appealing with colours, shapes, shadow effects and animations. We provide the most common choices to make it easy for you to create the chart you want to see, but because we cannot anticipate all requirements we take a similar approach to other advanced Swing components such as JTable, and allow you to customize chart appearance with your own sets of renderers.

JIDE Charts Background and Philosophy

We have used many different charting components in many different projects, and found them to be of varying quality and usefulness. It was the frustrations with the flexibility of these other components that led directly to the development of JIDE Charts.

We designed JIDE Charts to:

- embrace the Swing MVC approach to offer maximum power and flexibility
- make it possible for a Swing developer to start working with charts within minutes
- generate great looking charts
- support the interactive exploring of data

The point about embracing the Swing MVC approach is an important one, since some charting components that are available have been translated from another programming language, do not embrace MVC, and therefore do not offer the same level of flexibility as JIDE Charts. A great deal of thought has gone into the design – it has been conceived as a component that can continue to support your needs as your project requirements grow.

JIDE Charts Quick Start

This section contains some examples that demonstrate how easy it is to get up and running with JIDE Charts. The following sections provide much more detail about how to configure your charts and which features are available, but most developers are eager to get something working quickly, so here is a quick working example.

The data displayed in a chart is held in a ChartModel instance. ChartModel is a Java interface, so there can be many different kinds of ChartModel, specialized for different tasks (for example, adapting from other data structures or retrieving data from a database). There is a ready-made implementation called DefaultChartModel, which is easy to use.

Solving a Pair of Simultaneous Equations

Suppose we wish to find a solution to the pair of simultaneous equations:

(a) $y = x$

(b) $y = 1-x$

We can do this graphically by plotting two lines and looking for the intersection. We create a DefaultChartModel instance for (a), and another for (b):

```
DefaultChartModel modelA = new DefaultChartModel("ModelA");
DefaultChartModel modelB = new DefaultChartModel("ModelB");
```

Two data points are all that is needed to draw a straight line. We observe that (0, 0) and (1, 1) both lie on the line $y = x$ and therefore add those points to the model:

```
modelA.addPoint(0, 0);
modelA.addPoint(1, 1);
```

Similarly, we observe that (0, 1) and (1, 0) both lie on the line for (b), so we add those points to the model for (b):

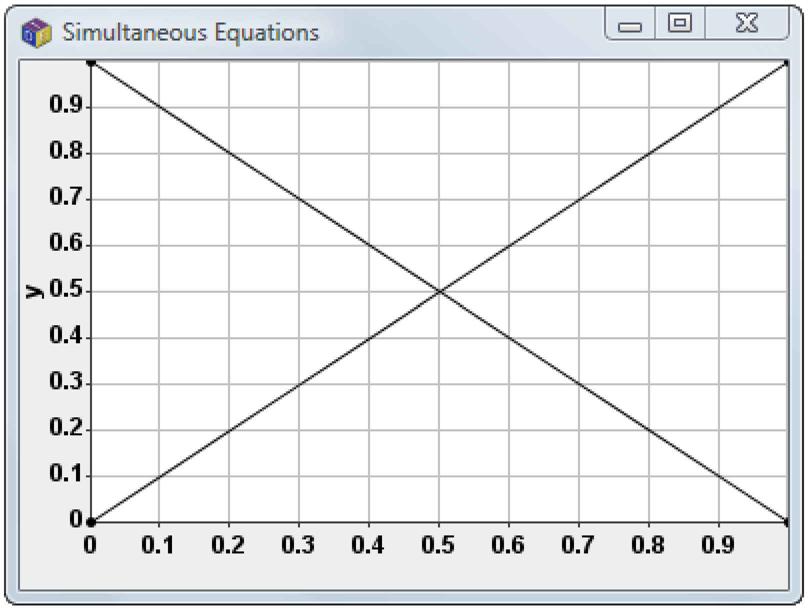
```
modelB.addPoint(0, 1);
modelB.addPoint(1, 0);
```

Now we have something to plot and look at. To do this, we create a Chart and add the chart models to it. Chart is a visual component, so we can add it to a Swing JPanel or set it as the content pane of a JFrame. The complete code (apart from package and import declarations) for this example is as follows:

```
public class SimultaneousEquations {
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        JFrame frame = new JFrame("Simultaneous Equations");
        frame.setIconImage(JideIconsFactory.getImageIcon(
                                JideIconsFactory.JIDE32).getImage());
```

```
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setSize(400, 300);
            DefaultChartModel modelA = new DefaultChartModel("ModelA");
            DefaultChartModel modelB = new DefaultChartModel("ModelB");
            modelA.addPoint(0, 0);
            modelA.addPoint(1, 1);
            modelB.addPoint(0, 1);
            modelB.addPoint(1, 0);
            Chart chart = new Chart();
            chart.addModel(modelA);
            chart.addModel(modelB);
            frame.setContentPane(chart);
            frame.setVisible(true);
         }
      });
   }
}
```

This produces the following window (screenshot is from Windows Vista):



You can resize the window and the chart will resize accordingly.

The chart shows that the lines cross at the point (0.5, 0.5), so x = 0.5, y = 0.5 is the solution to the simultaneous equations. By default, Chart uses axes from 0 to 1, but we can easily redefine them as follows:

```
Axis xAxis = chart.getXAxis();
xAxis.setRange(new NumericRange(-2, 2));
Axis yAxis = chart.getYAxis();
yAxis.setRange(new NumericRange(-2, 2));
```

For this chart, perhaps we prefer the axes to be placed in the center rather than the edges of the plot area, so we can specify this as follows:

```
xAxis.setPlacement(AxisPlacement.CENTER);
```

```
yAxis.setPlacement(AxisPlacement.CENTER);
```

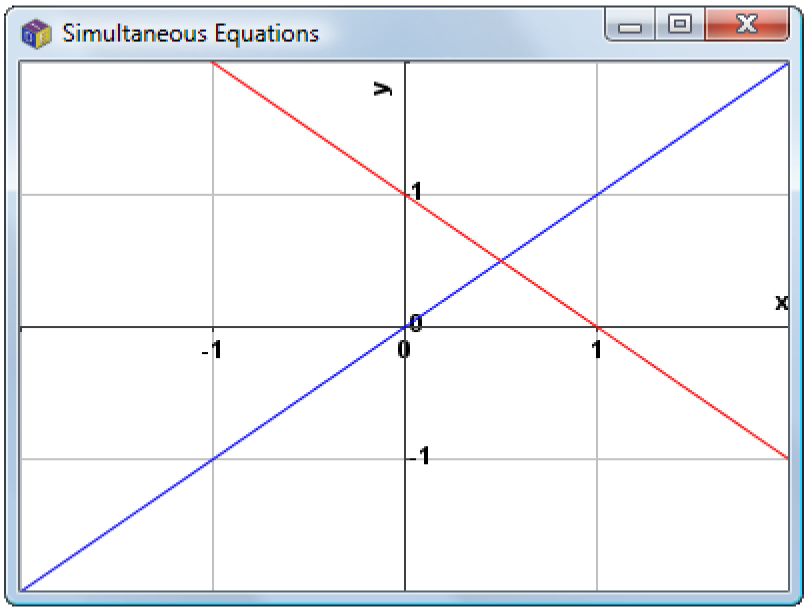
Finally, we are interested in seeing only the lines and not the points that make up the lines. We can specify this by using a ChartStyle. For example, to specify a ChartStyle in which you would like to see blue lines but no points, you create a ChartStyle and use it when adding the ChartModel to the Chart:

```
ChartStyle styleA = new ChartStyle();
chart.addModel(modelA, styleA);
```

Our example now becomes:

```
DefaultChartModel modelA = new DefaultChartModel("ModelA");
DefaultChartModel modelB = new DefaultChartModel("ModelB");
modelA.addPoint(-2, -2).addPoint(0, 0).addPoint(2, 2);
modelB.addPoint(-2, 3).addPoint(0, 1).addPoint(1, 0).addPoint(2, -1);
Chart chart = new Chart();
ChartStyle styleA = new ChartStyle(Color.blue, false, true);
ChartStyle styleB = new ChartStyle(Color.red, false, true);
chart.addModel(modelA, styleA).addModel(modelB, styleB);
Axis xAxis = chart.getXAxis();
xAxis.setPlacement(AxisPlacement.CENTER);
xAxis.setRange(new NumericRange(-2, 2));
Axis yAxis = chart.getYAxis();
yAxis.setPlacement(AxisPlacement.CENTER);
yAxis.setRange(new NumericRange(-2, 2));
```

and generates the following chart:



A Simple Bar Chart

Suppose we wish to create a chart of sales figures for an ice cream parlour that sells three flavours of ice cream: chocolate, vanilla and strawberry. The x values on the chart correspond to one of these flavours and the y values correspond to sales volume. This situation is different from the simultaneous equations situation described above because chocolate, vanilla and strawberry are not numeric values, and yet we need those values to relate to a position on the chart. We do this by defining a CategoryRange that contains the possible category values. Here are the definitions of the category values:

```
ChartCategory<String> chocolate   = new ChartCategory<String>("Chocolate");
ChartCategory<String> vanilla     = new ChartCategory<String>("Vanilla");
ChartCategory<String> strawberry  = new ChartCategory<String>("Strawberry");
```

The category values themselves are defined using Java generics, so instances of any class can be turned into categorical values and used in a chart. For this example, we could define a Flavor class (or perhaps an enum) with the instances chocolate, vanilla and strawberry. However, to keep the example simple we have used string values.

The CategoryRange is defined by creating a new range and adding the possible values:

```
CategoryRange<String> flavours = new CategoryRange<String>();
flavours.add(chocolate).add(vanilla).add(strawberry);
```

Now we can create a DefaultChartModel and use the values chocolate, vanilla and strawberry as x coordinate values, just as if they were numbers:

```
DefaultChartModel salesModel = new DefaultChartModel("Sales");
salesModel.addPoint(chocolate, 300);
salesModel.addPoint(vanilla, 500);
salesModel.addPoint(strawberry, 250);
```

Next, we create a Chart component and set the ranges for the axes. The x axis uses the CategoryRange whereas the y axis uses a numeric range.

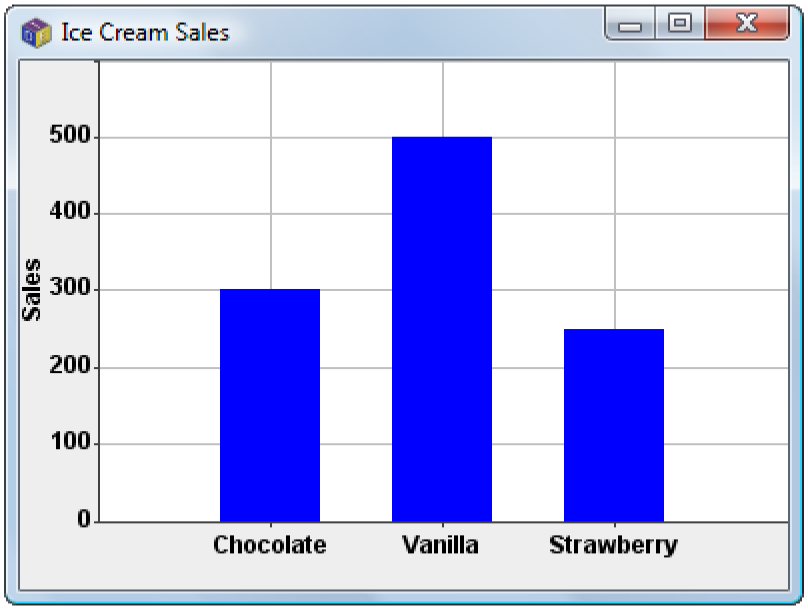
```
Chart chart = new Chart();
chart.setXAxis(new CategoryAxis(flavors, "Flavors"));
chart.setYAxis(new Axis(new NumericRange(0, 600), "Sales"));
```

Notice that in this example we have used a CategoryAxis for the x axis. A CategoryAxis is an Axis that knows to render the tick labels using the toString() method of the category object, rather than with a number. There are equivalent classes for numeric and time-based axes, namely, NumericAxis and TimeAxis.

Lastly, we specify the style to use for the display. We want to see bars, rather than lines or points so we set the values accordingly, and also set the width to use for the bars. Finally, we add the chart model to the chart using this style.

```
ChartStyle style = new ChartStyle(Color.blue);
style.setLinesVisible(false);
style.setPointsVisible(false);
style.setBarsVisible(true);
chart.addModel(salesModel, style);
```

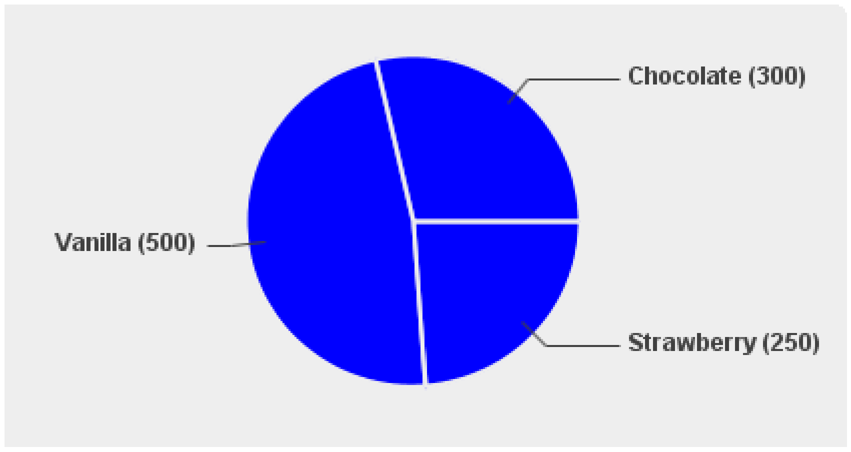
This generates the following bar chart:



We use the same technique to prepare data for displaying as a pie chart, so to modify the display to be a pie chart instead of a bar chart we need only add the line:

```
chart.setChartType(ChartType.PIE);
```

Then instead of the bar chart shown above we generate the following:



We shall see later how to change the colouring and rendering style of bar charts and pie charts.

XY Charts

XY Charts are plotted on a two dimensional rectangular plot area and are the most common chart type. Line charts, point-and-line charts, area charts, scatter plots and even bar charts are all examples of XY charts. (However, as bar charts have some special properties, they are described in the next section.)

Although each point in an XY chart is plotted using a numerical value (actually a double precision number), the values that you use a developer need not all be numeric. We also support categorical ranges and time series.

Chart Style

We have already seen how to create a chart display by first adding points to a ChartModel and then adding the ChartModel to a Chart. We can customize the appearance of the chart by using a ChartStyle. A ChartStyle is the styling applied to a single ChartModel and can be supplied when adding the ChartModel to the Chart or later by using the Chart.setStyle() method.

The ChartStyle has two roles: firstly, it determines whether we wish to display a ChartModel with points, lines and/or bars; and secondly, it determines the sizes and colors of those elements.

For example, suppose we create a simple ChartModel:

```
DefaultChartModel model = new DefaultChartModel();
model.addPoint(1, 2).addPoint(2, 3).addPoint(3, 4).addPoint(5, 2.5);
```

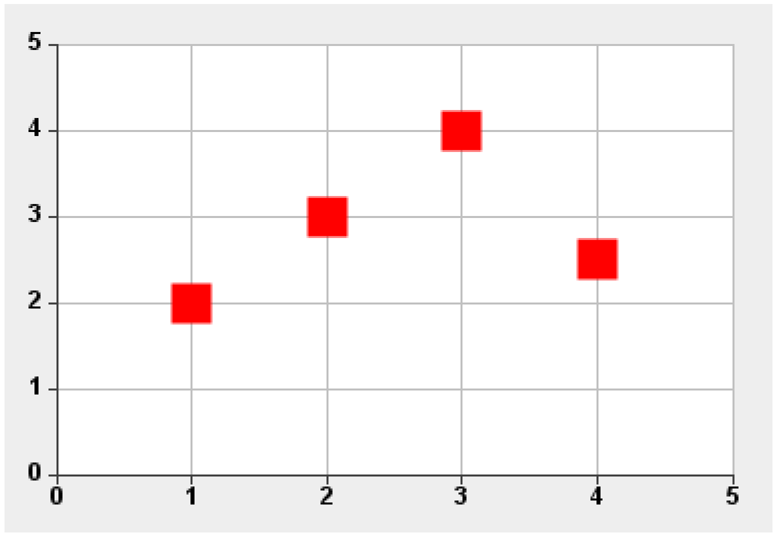
Point Plots

We can display this as a points-only plot as follows:

```
ChartStyle style = new ChartStyle(Color.red, PointShape.BOX);
style.setPointSize(20);
Chart chart = new Chart();
chart.setXAxis(new Axis(0, 5));
chart.setYAxis(new Axis(0, 5));
chart.addModel(model, style);
```

The style specified is a points-only style made with red box-shaped points of size 20 pixels.

This produces the following:

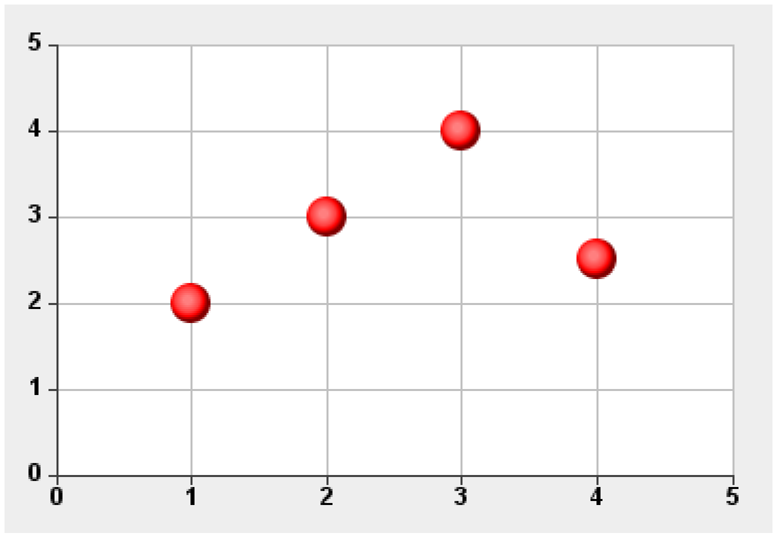


The predefined point shapes are CIRCLE, DISC, SQUARE, BOX, DIAMOND, DOWN_TRIANGLE, UP_TRIANGLE, HORIZONTAL_LINE, VERTICAL_LINE and UPRIGHT_CROSS.

For custom effects, it is also possible to use a point renderer. Or to be more precise, the example shown uses the default point renderer. We have also defined another renderer called SphericalPointRenderer. To use this, add the following line:

```
chart.setPointRenderer(new SphericalPointRenderer());
```

Then the output will be as follows:

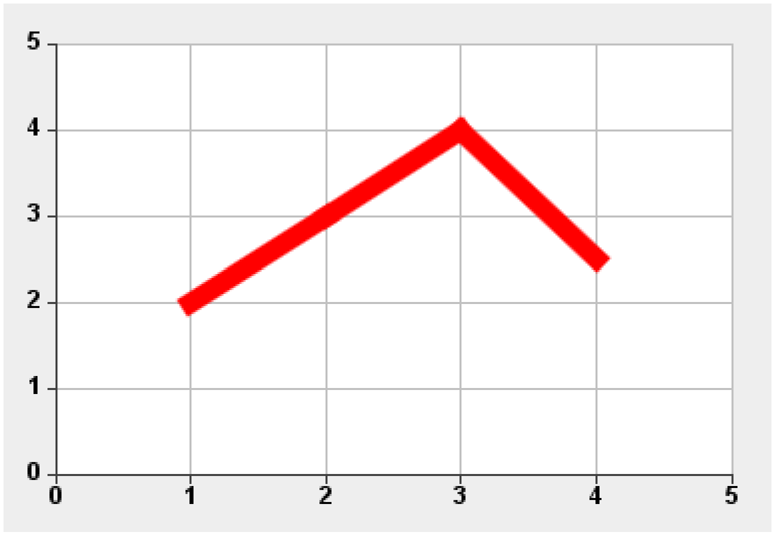


Line Plots

For line plots, we can set up a style as follows:

```
ChartStyle style = new ChartStyle(Color.red);
style.setLineWidth(10);
```

This would produce the following line chart:



Note that you can produce much more detailed charts by using a thinner line width and a model containing many more points. For example, the same technique has been used for producing Electrocardiogram (ECG) plots from using data collected from a heart sensor.

As with points, there is the option of changing the renderer for lines. The default line renderer draws the line by connecting the points of the model with a straight line, but there is another line renderer that we provide that joins the points of the model with a curve, producing a smooth line. This is the aptly named SmoothLineRenderer.

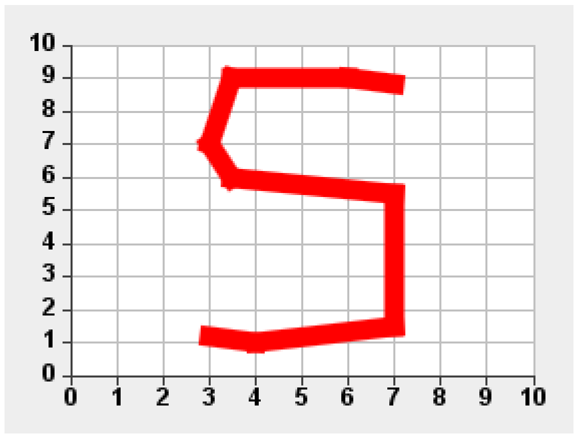
To invoke the smooth line renderer you do the following:

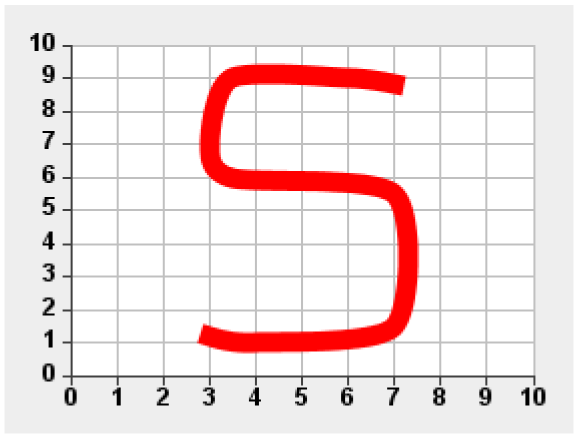
```
chart.setLineRenderer(new SmoothLineRenderer(chart));
```

or if you only want to apply the smooth line renderer to particular chart models, you can do that as follows:

```
SmoothLineRenderer renderer = new SmoothLineRenderer(chart);
chart.setLineRenderer(model1, renderer);
chart.setLineRenderer(model2, renderer);
```

Here is an example line chart that has been plotted on the left with the default line renderer and on the right with the smooth line renderer:

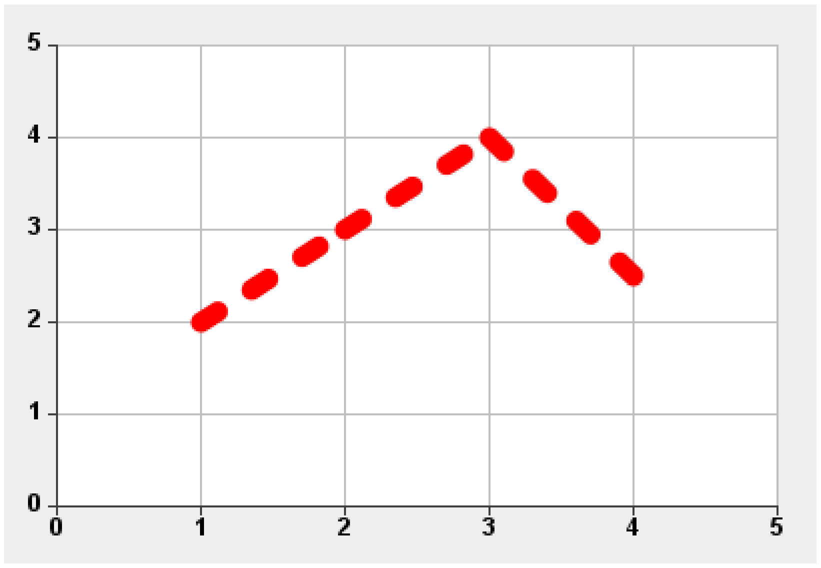




As well as changing the color, you can also customize the output by changing the Stroke of the line. So for example, you can specify a dotted line as follows:

```
style.setLineStroke(new BasicStroke(10f,
                                    BasicStroke.CAP_ROUND,
                                    BasicStroke.JOIN_ROUND,
                                    10f, new float[] {10f, 20f}, 0f));
```

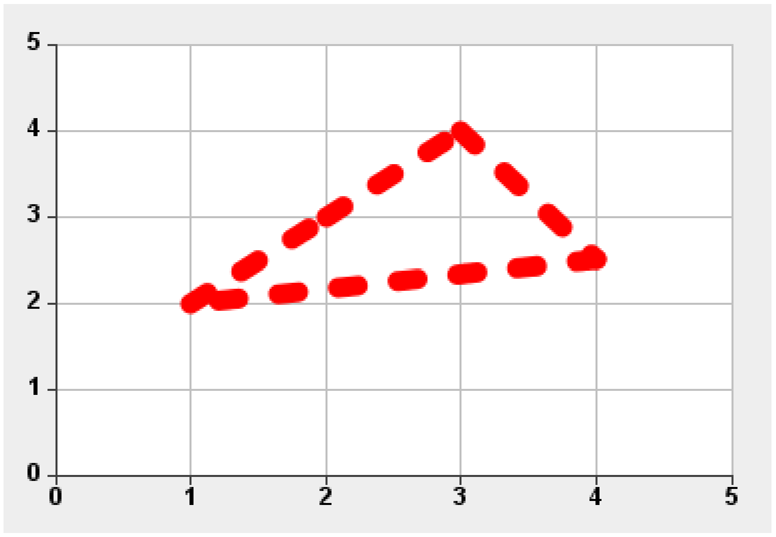
This generates the following chart:



Now is a good time to mention a feature of ChartModel that also affects the presentation. By default a line plot is plotted by drawing line segments between points starting from the first point and ending at the last. If you also wish a line to be drawn from the last back to the first you can mark the model as being cyclical as follows:

```
model.setCyclical(true);
```

By adding this line, the output changes to:

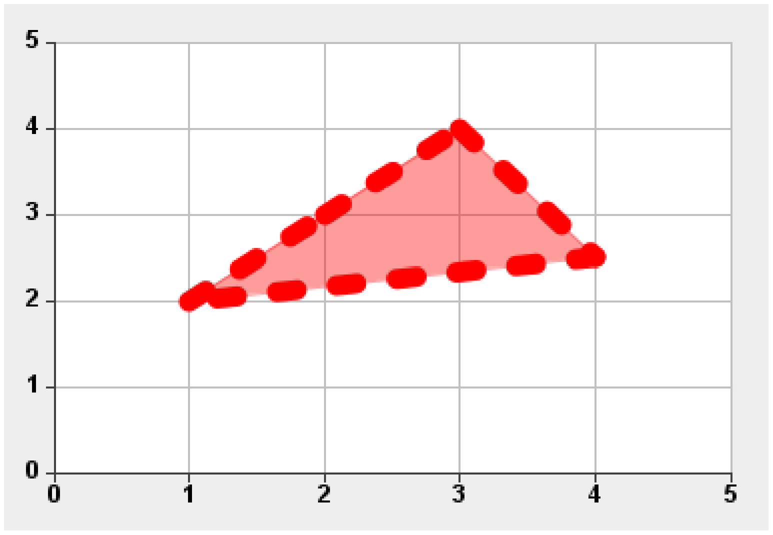


NEW: It is now also possible to fill the shape enclosed by a cyclical model. To do this, you call setLineFill() on the ChartStyle object that is used by the cyclical model, and provide as a parameter the Color or Paint style with which you would like the shape filled.

For example, by adding the following line to the example above:

```
style.setLineFill(new Color(255, 0, 0, 100));
```

we get this chart:



Time Series Charts

Time Series charts are really no different from other XY charts – except that they use a time range, usually on the X axis. Although we often use java.util.Date objects to express points in time, Java also maintains a numeric value expressed as the number of milliseconds since midnight on January 1st 1970. We also use this numeric representation in generating charts.

For example, we can create some time points as follows:

```
DateFormat format = new SimpleDateFormat("dd-MMM-yyyy");
final long mar = format.parse("15-Mar-2009").getTime();
final long apr = format.parse("15-Apr-2009").getTime();
final long may = format.parse("15-May-2009").getTime();
final long jun = format.parse("15-Jun-2009").getTime();
final long jul = format.parse("15-Jul-2009").getTime();
final long aug = format.parse("15-Aug-2009").getTime();
```

Then we can create a simple time series chart model as follows:

```
DefaultChartModel model = new DefaultChartModel();
model.addPoint(apr, 2);
model.addPoint(may, 3);
model.addPoint(jun, 4);
model.addPoint(jul, 2.5);
```

We set up a chart style to use both lines and points (with the spherical point renderer):

```
ChartStyle style = new ChartStyle(new Color(200, 50, 50), true, true);
style.setLineWidth(5);
style.setPointSize(20);
chart.setPointRenderer(new SphericalPointRenderer());
```

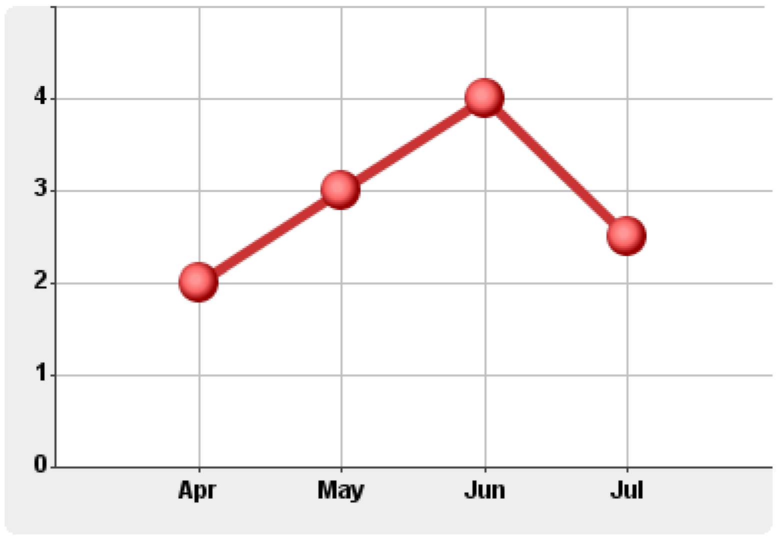
Lastly, we create and set a time range for the x axis of the chart:

```
TimeRange timeRange = new TimeRange(mar, aug);
Axis xAxis = new TimeAxis(timeRange);
chart.setXAxis(xAxis);
```

Although the above code works, it looks even better if we make sure that the tick marks on the x axis match the time points that we are interested in. You can customize the generation of tick marks with a tick calculator:

```
xAxis.setTickCalculator(new DefaultTimeTickCalculator() {
  @Override
  public Tick[] calculateTicks(Range<Date> r) {
    return new Tick[] {
      new Tick(apr, "Apr"),
      new Tick(may, "May"),
      new Tick(jun, "Jun"),
      new Tick(jul, "Jul")};
  }
});
```

This produces the following (somehow familiar-looking) time series chart:



Category Charts

As we saw earlier, Category ranges can be used to place string values along an axis, making them plottable in an XY chart. Now I will show how to apply the same technique to some other class.

Suppose we had a class Person

```
class Person {
  private String name;

  public Person(String name) {
    this.name = name;
  }

  @Override
  public String toString() {
    return name;
  }
}
```

and then created some instances:

```
Person john   = new Person("John");
Person paul   = new Person("Paul");
Person george = new Person("George");
Person ringo  = new Person("Ringo");
```

We can make these instances plottable by creating categories from them:

```
ChartCategory<Person> cJohn   = new ChartCategory<Person>(john);
ChartCategory<Person> cPaul   = new ChartCategory<Person>(paul);
ChartCategory<Person> cGeorge = new ChartCategory<Person>(george);
ChartCategory<Person> cRingo  = new ChartCategory<Person>(ringo);
```

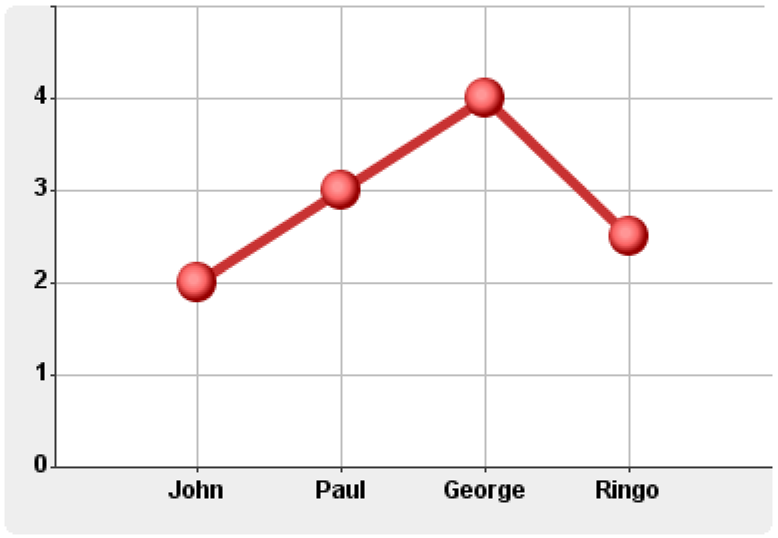
We also need to create a range made from these category values to set on the axis:

```
CategoryRange<Person> beatles = new CategoryRange<Person>();
beatles.add(cJohn).add(cPaul).add(cGeorge).add(cRingo);
Axis xAxis = new CategoryAxis<Person>(beatles);
chart.setXAxis(xAxis);
```

Lastly, we create a Chart Model using our category values as x coordinates:

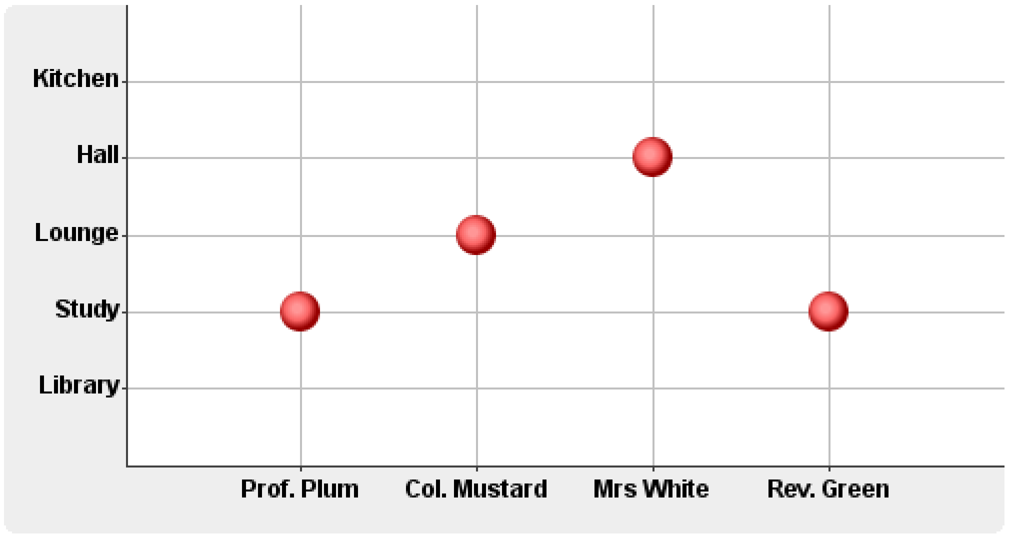
```
DefaultChartModel model = new DefaultChartModel();
model.addPoint(cJohn,   2);
model.addPoint(cPaul,   3);
model.addPoint(cGeorge, 4);
model.addPoint(cRingo,  2.5);
```

When we add this model to a chart and apply a style as for the other charts you have seen, we produce the following:



Category ranges do not have to be limited to the x axis, or to just one axis.

The following chart uses categorical ranges on both axes to indicate the locations of four people:



Customization and Effects

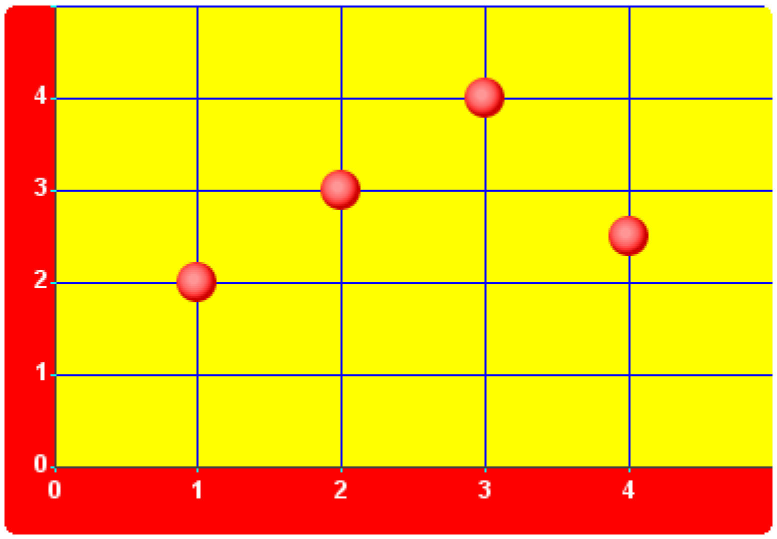
Colors

We can customize the appearance of the chart we saw earlier by changing some colours.

By adding the following lines:

```
chart.setChartBackground(Color.yellow);
chart.setPanelBackground(Color.red);
chart.setGridColor(Color.blue);
chart.setTickColor(Color.cyan);
chart.setLabelColor(Color.white);
```

we produce the following chart:

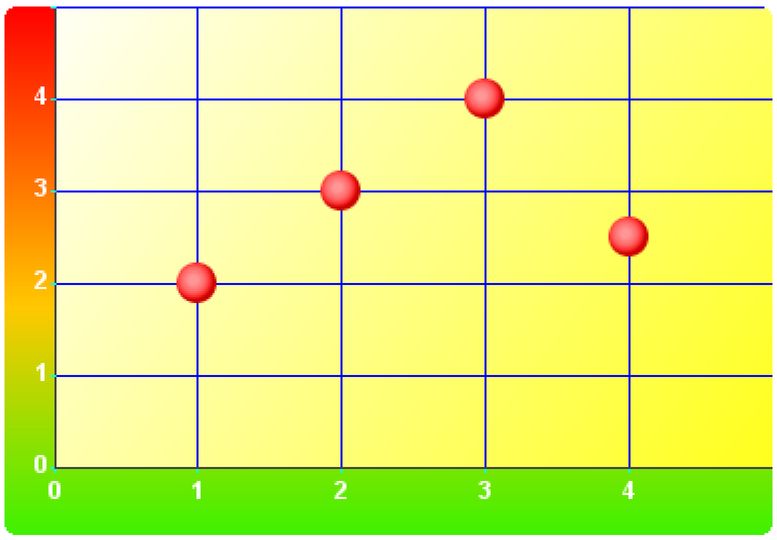


If you look at the API, you will notice that the chartBackground and panelBackground properties are actually an instance of a Paint, which means that you can supply a Color as in the example above, or you can supply a Paint with a colour gradient effect.

For example, if, instead of a yellow chart background and a red panel background we used the following linear gradient effects:

```
chart.setChartBackground(
  new LinearGradientPaint(
      0f, 0f, 400f, 300f,
      new float[] {0.0f, 1.0f},
      new Color[] {Color.white, Color.yellow}));

chart.setPanelBackground(
  new LinearGradientPaint(
      0f, 0f, 0f, 300f,
      new float[] {0.0f, 0.5f, 1.0f},
      new Color[] {Color.red, Color.orange, Color.green}));
```

then we produce the following chart:

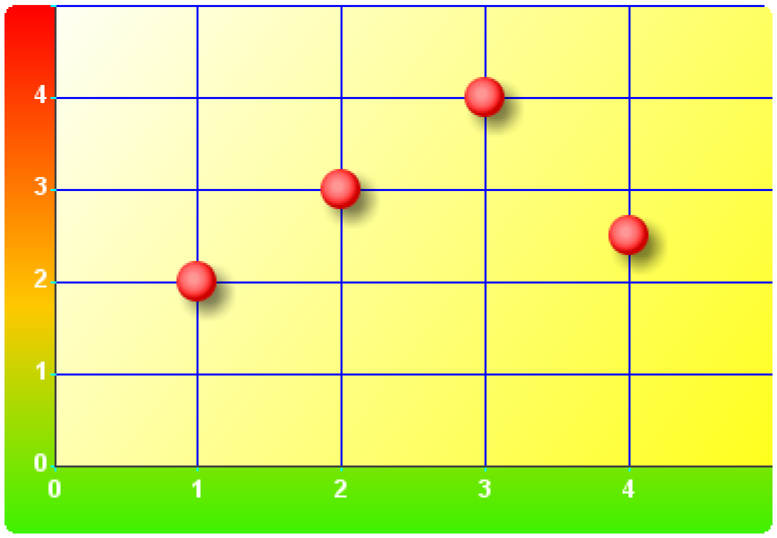


Shadow Effect

To make the chart look impressive you can add a shadow effect by calling the following method:

```
chart.setShadowVisible(true);
```

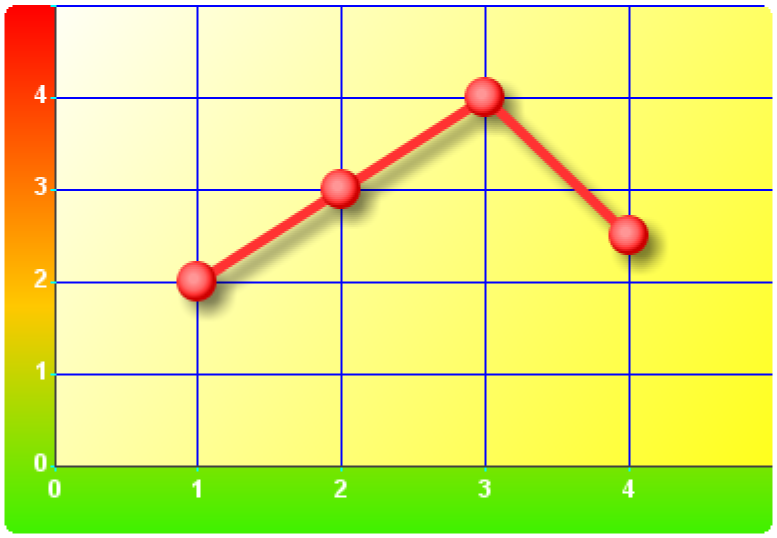
The same chart then looks as follows:



The shadow effect also works with lines, so if we switch on lines and points:

```
style.setLinesVisible(true);
style.setLineWidth(5);
```

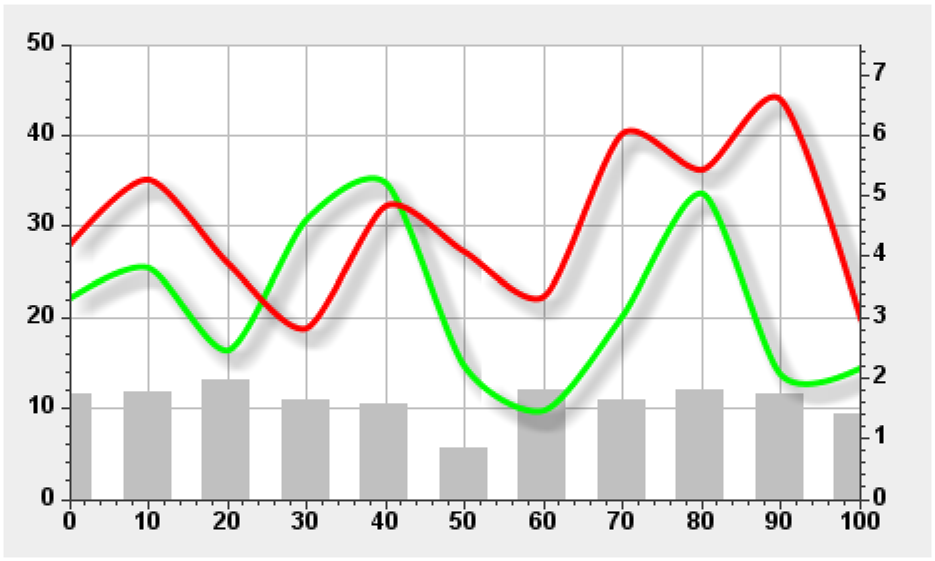
we get the following:



Note that computation of the shadow significantly lengthens the time for rendering of a chart so is not suitable for charts that need to be updated frequently.

NEW: It is now possible to specify that only some of the models should be displayed with a shadow. To do this, instead of using `chart.setShadowVisible(true)`, you first call `chart.setShadowVisibility(ShadowVisibility.SOME)` and then for each model that should be drawn with a shadow you call chart.setShadowVisible(model, true). Note that per-model shadows are not supported under lazy rendering: when using lazy rendering, you can only generate shadows for all or none of the models.

Here is an example (randomly generated) chart where the shadows have been applied to the lines, but not to the bars:



This approach could be used, for example, when the lines are the primary focus of the chart and the bars provides supporting information.

Rollover Effect

Sometimes it is useful to visually confirm to the user that the mouse is over a particular point by highlighting it — a so-called 'rollover effect'. The Chart component supports a rollover effect, but it is switched off by default. To use the effect, call chart.setRolloverEnabled(true). Once enabled, the colour intensity of the point underneath the mouse cursor (if any) is increased to help show the location of the mouse and to facilitate point selection.

Animation

When a chart is first shown, it will, by default, use animation for a fraction of a second to move the points (or bars or segments) into position. To switch this effect off, use chart.setAnimateOnShow(false). You may want to switch the effect off if you are dealing with large datasets, and you will need to switch it off if you are using the Chart component as a renderer such as a TableCellRenderer. To re-start the animation, use chart.startAnimation().

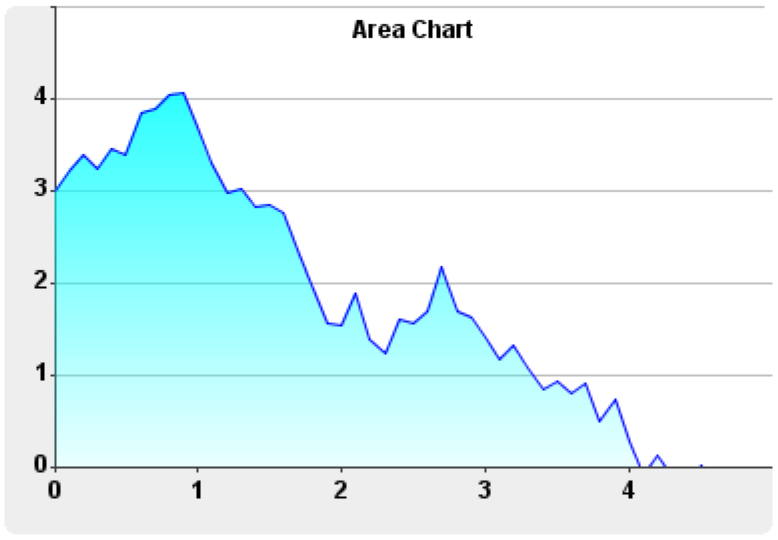
Area Charts

A variation of XY charts is one where the area from a line to the axis is filled in with a color — or a gradient paint.

The following code generates a ChartModel with some random points and then plots an area chart from the data (the static main() method is boiler-plate code to create the AreaChart instance, so has been omitted):

```java
public class AreaChart extends JPanel {
    private Chart chart = new Chart();

    public AreaChart() {
        setLayout(new BorderLayout());
        add(chart, BorderLayout.CENTER);
        ChartStyle style = new ChartStyle(Color.blue, false, true); // Lines only
        style.setLineFill(new LinearGradientPaint(0f, 0f, 0f, 250f,
                                new float[] {0.0f, 1.0f},
                                new Color[] {Color.cyan, new Color(255,255,255,0)}));
        chart.addModel(createModel("my model"), style);
        chart.setTitle("Area Chart");
        chart.setVerticalGridLinesVisible(false);
        chart.setXAxis(new Axis(new NumericRange(0, 5)));
        chart.setYAxis(new Axis(new NumericRange(0, 5)));
    }

    private ChartModel createModel(String name) {
        DefaultChartModel model = new DefaultChartModel(name);
        double y = 3.0;
        for (double x = 0; x <= 5; x += 0.1) {
            model.addPoint(x, y);
            y += Math.random() - 0.5;
        }
        return model;
    }

    public static void main(String[] args) {...}

}
```

Here is an example chart generated:



Notice that we added a title for the chart using setTitle() and switched off the vertical grid lines using the method setVerticalGridLinesVisible(). For the fill gradient, we progress from cyan at the top of the chart to a completely transparent white at the bottom, which allows us to see the horizontal grid lines through the chart.

Panning and Zooming

One of the best-liked features is the ability to easily navigate around an XY chart by panning and zooming using the mouse. *Panning* is the ability to click on a chart and move it by dragging it – this enables you to explore parts of the chart that are not initially shown. *Zooming* is the ability to rescale the axes chart to give the impression of moving closer or further away. You can add mouse panning and zooming to an XY chart with the following:

```
chart.addMousePanner().addMouseZoomer();
```

The chart component takes care of the rescaling of the axes and redrawing the chart. In some cases you might want to allow zooming in one axis but not the other. For example, if you have time series data then you are usually much more interested in zooming the time (x) axis and you may not want to rescale the y axis at the same time. You can do this as follows:

```
chart.addMousePanner().addMouseZoomer(true, false);
```

You can specify to allow panning in one axis only in a similar way. For example, you can specify horizontal panning only with chart.addMousePanner(true, false).

By default, zooming uses the point at the centre of the chart as the origin of the re-scaling operation, but if you prefer, you can configure the point under the mouse cursor to be the centre for the re-scaling. To do this, you need to create a MouseWheelZoomer and configure its ZoomLocation something like the following:

```
MouseWheelZoomer zoomer = new MouseWheelZoomer(chart, true, false);
zoomer.setZoomLocation(ZoomLocation.MOUSE_CURSOR);
chart.addMouseWheelListener(zoomer);
```

NEW: A new feature for the MouseWheelZoomer is that you can now set limits for the zooming. You can specify minimum and maximum range sizes to prevent users from zooming in or out too far from the region of interest, and you can also set specific range limits to prevent users from zooming to a region that is out of bounds. For example, when displaying a chart of weights you might want to prevent users from zooming out to a region that shows negative values for the weights, since negative values are not allowed. To limit zooming, use a combination of the XLimits/YLimits properties together with MaxXRangeSize/MinXRangeSize and MaxYRangeSize/MinYRangeSize.

NEW: Similarly, you can also limit panning to the region of interest by setting the XLimits/YLimits properties on the MouseDragPanner.

A recently-added feature to the MouseDragPanner is the ability to have a ‘continuous’ drag; that is, one in which the dragged area accumulates momentum and continues to move after you have let go of the drag, albeit slowing down and eventually stopping. The speed of the drag movement prior to letting go defines how much the chart will move before it stops, thus enabling a SmartPhone-like 'flick' interaction gesture to quickly navigate a chart. To use this feature, set the *continuous* property on MouseDragPanner to be true. You can also customise this feature by specifying the friction co-efficient, which is used to slow the chart down after the drag release. The higher the friction co-efficient, the faster it will slow down.

Large Data Sets

We have made special provision for dealing with large data sets in XY charts. Rendering large numbers of points can be time consuming and if this all occurs on Swing's Event Dispatch Thread it can affect the responsiveness of the user interface. In our tests with a conventional approach to rendering, we found the GUI to be less responsive than we would have liked when we were plotting more than around 20000 points. This 'pain threshold' will vary from machine to machine and from application to application, but we have been able to move the pain threshold out of range for many applications by performing most of the hard work of rendering as a background task.

The advantage of background rendering is that you can use panning and zooming on data sets containing hundreds of thousands of peoples without the users having to wait while their machine locks up to redrawing chart in response to a mouse event. The slight disadvantage is that the user interface must still work with an outdated chart until the new chart has been generated in the background. In practise this disadvantage is often not even noticed by users, as the user experience is similar to other well-known applications.

To move the rendering into the background, you simply add the following to your program:

```
chart.setLazyRenderingThreshold(0);
```

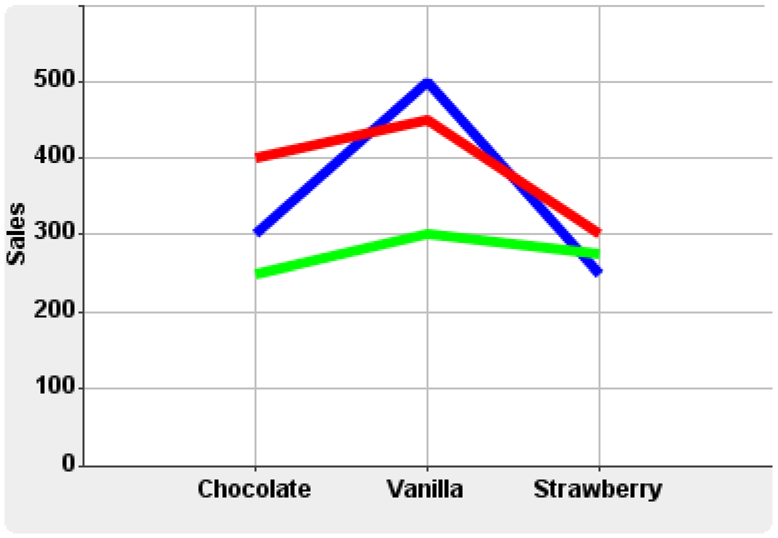
This will always perform background rendering, regardless of the number of points to plot. For more control, you can set a threshold of, say, 10000 so that background rendering is activated only when there are more than 10000 points to plot.

Creating a Legend

Once you have created a chart, it is easy to create a corresponding legend component. For example consider the ice cream sales example and now suppose that we have three salesmen Harpo, Chico and Groucho. We create a ChartModel for each of the three salesmen.

```
public class LegendExample extends JPanel {
  private Chart chart = new Chart();
  private ChartCategory<String> chocolate  = new ChartCategory<String>("Chocolate");
  private ChartCategory<String> vanilla    = new ChartCategory<String>("Vanilla");
  private ChartCategory<String> strawberry = new ChartCategory<String>("Strawberry");
  private DefaultChartModel model1 = new DefaultChartModel("Harpo");
  private DefaultChartModel model2 = new DefaultChartModel("Chico");
  private DefaultChartModel model3 = new DefaultChartModel("Groucho");
  public LegendExample() {
    setLayout(new BorderLayout());
    add(chart, BorderLayout.CENTER);

    CategoryRange<String> flavours = new CategoryRange<String>();
    flavours.add(chocolate).add(vanilla).add(strawberry);
    model1.addPoint(chocolate, 300).addPoint(vanilla, 500).addPoint(strawberry, 250);
    model2.addPoint(chocolate, 400).addPoint(vanilla, 450).addPoint(strawberry, 300);
    model3.addPoint(chocolate, 250).addPoint(vanilla, 300).addPoint(strawberry, 275);

    ChartStyle style1 = new ChartStyle(Color.blue,  false, true, false);
    ChartStyle style2 = new ChartStyle(Color.red,   false, true, false);
    ChartStyle style3 = new ChartStyle(Color.green, false, true, false);

    style1.setLineWidth(5);
    style2.setLineWidth(5);
    style3.setLineWidth(5);

    chart.setXAxis(new CategoryAxis<String>(flavours, "Flavours"));
    chart.setYAxis(new Axis(new NumericRange(0, 600), "Sales"));

    chart.addModel(model1, style1);
    chart.addModel(model2, style2);
    chart.addModel(model3, style3);

    }

    public static void main(String[] args) {...}

}
```

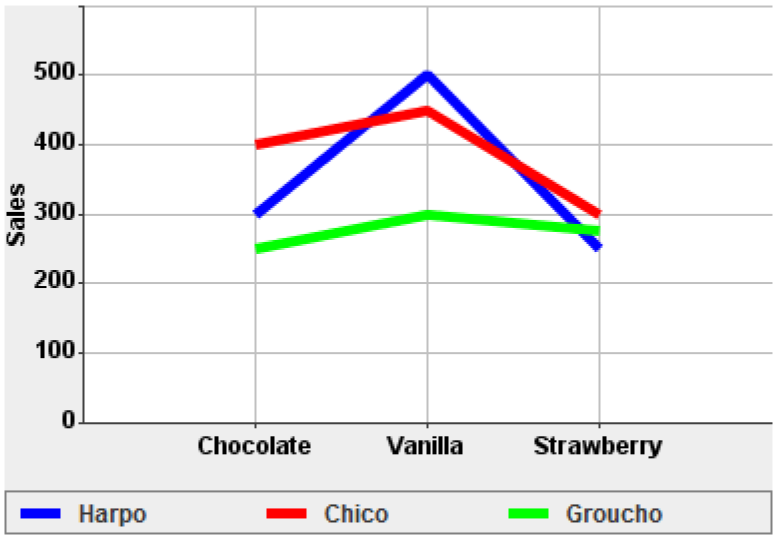
This creates the following chart:



We can create a legend for this chart simply by passing the chart as an argument to the constructor of the Legend component, along with the number of columns to use in the legend. In this case, we would like a single row of three columns, so we pass the argument 3.

```
Legend legend = new Legend(chart, 3);
```

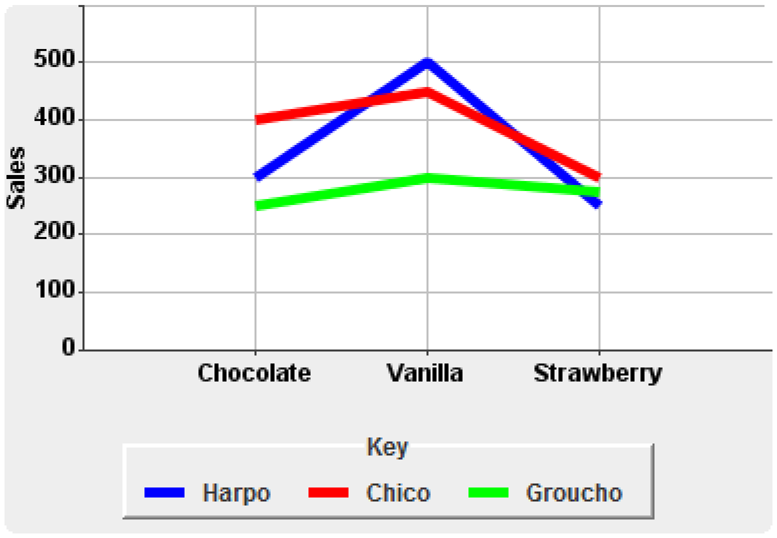
Then we can add the legend to the panel. In this case we add it to the south of the BorderLayout to create the following output:



Note that the BorderLayout has stretched the legend to the full width of the panel. If you prefer for it to take its natural width, you can first add the legend to a JPanel with a FlowLayout and then add that panel to the south of the BorderLayout. In the following code we have also changed the border and added a title using a TitledBorder.

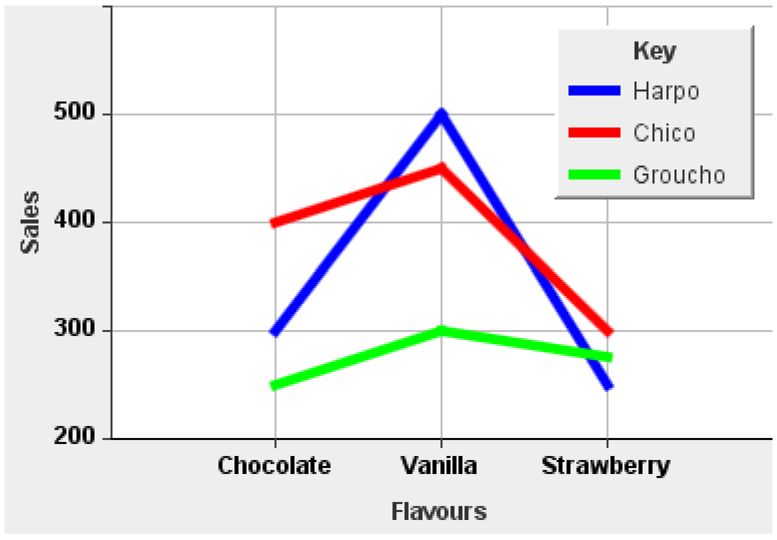
```
JPanel legendPanel = new JPanel();
Legend legend = new Legend(chart, 3);
Border border = new BevelBorder(BevelBorder.RAISED);
TitledBorder titledBorder = new TitledBorder(border,
                                             "Key",
                                             TitledBorder.CENTER,
                                             TitledBorder.TOP);
legend.setBorder(titledBorder);
legendPanel.add(legend);
add(legendPanel, BorderLayout.SOUTH);
```

This gives the following output:



Tip: If you prefer the title inside the border you can call setTitle() or setTitleLabel() on the legend component itself.

NEW: It is now also possible to add the Legend to the chart area. In the screenshot below, the y axis has been adjusted and instead of using an extra JPanel below the Chart component to display the Legend, it has been added to the Chart using chart.addDrawable(legend). Once you have done this, you need to set the location of the Legend component in pixel coordinates using legend.setLocation().



Bar Charts

Suppose that we would prefer to see our ice cream sales as a bar chart. This can be done with minor modifications to the code from the previous section.

First, the ChartStyle applied to each of the models needs to have the barVisible property set to true and the linesVisible and pointsVisible properties set to false. We could set these properties individually, but it is perhaps easiest when we construct the ChartStyle. So instead of the line style created with

```
ChartStyle style1 = new ChartStyle(Color.blue, false, true, false);
```

we create a bar style with

```
ChartStyle style1 = new ChartStyle(Color.blue, false, false, true);
```

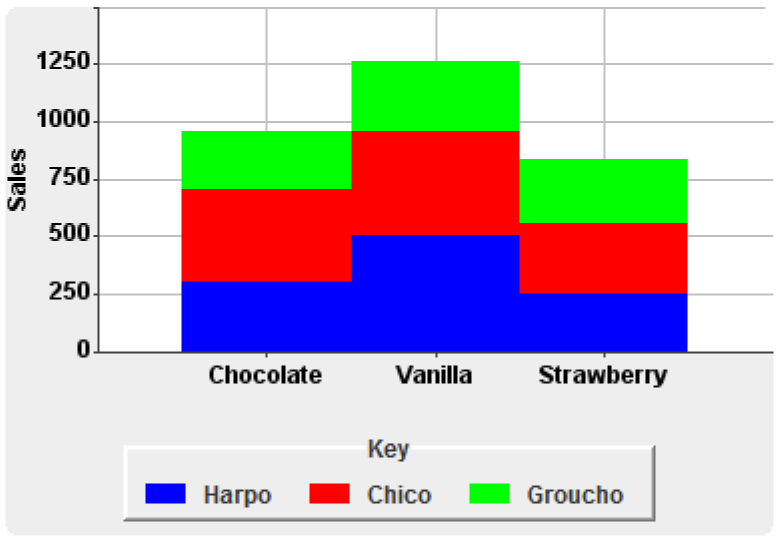
As it is not easy to remember the order of the Boolean arguments, we also provide for an alternative form that is more readable:

```
ChartStyle style1 = new ChartStyle(Color.blue).withBars();
```

There are equivalent methods called withPoints(), withLines(), and withPointsAndLines() for easily constructing or modifying chart styles to be of the indicated type. There is also a method called withNothing(), should you wish to create a model that is initially not visible.

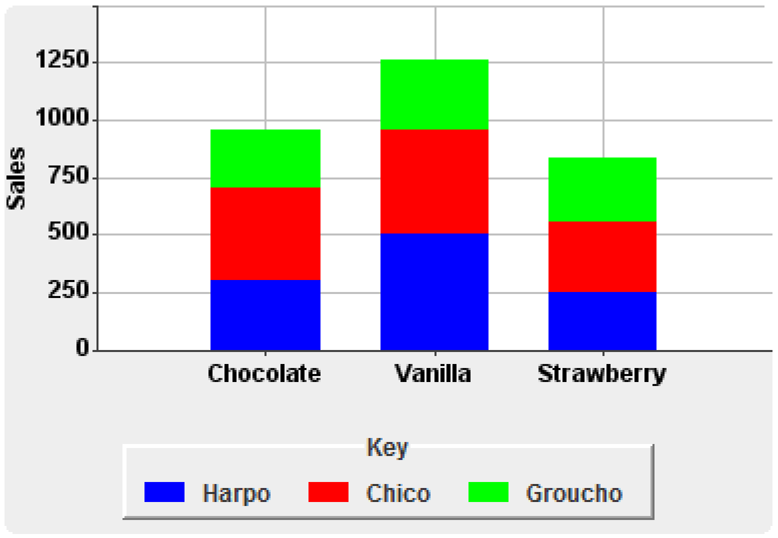
Stacked Bar Charts

We need to do this for each of the three chart styles used. By default, a stacked bar chart is created, so bars from different models with the same x value are placed on top of one another. To see all the bars we therefore need to increase the maximum value of the y axis. When we do so, we generate the following chart:



The bars would look better if they were separated, so we set a 30 pixel gap between the bars by calling chart.setBarGap(30);

Then we get the following chart:

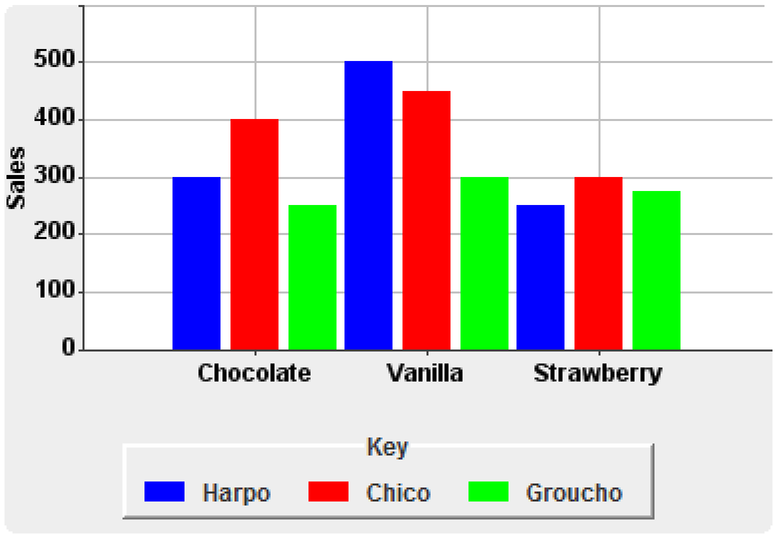


Grouped Bar Charts

Another way of displaying the bars is to use a separate bar for each ChartModel with the same x axis value. You can do this by using the barsGrouped property of Chart. When using grouped bars we have many more bars to display, so we also need a smaller gap size. The final change from the previous example is to set the y axis back to a smaller scale to reflect the range of values for the individual sales:

```
chart.setYAxis(new Axis(new NumericRange(0, 600), "Sales"));
chart.setBarGap(5);
chart.setBarsGrouped(true);
```

This is the resulting bar chart:

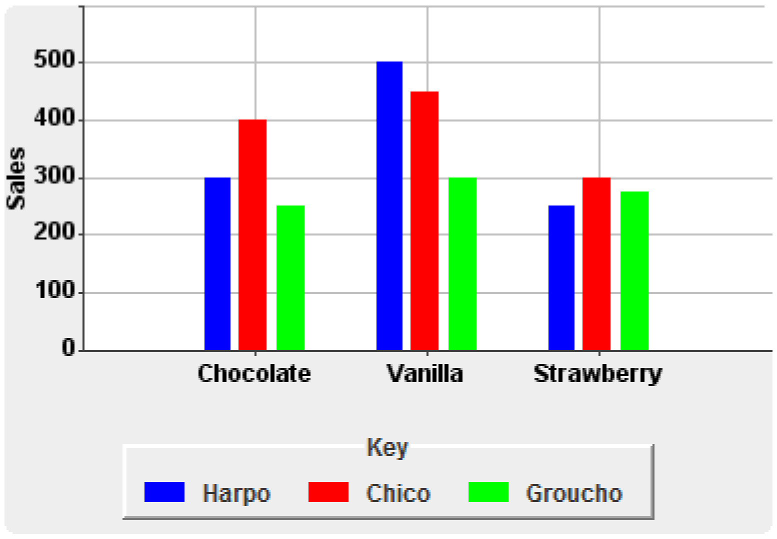


The bar gap is applied between all bars. To make the grouping into flavours a little more obvious you can also set a gap between the groups of bars.

For example,

```
chart.setBarGap(5);
chart.setBarGroupGap(30);
```

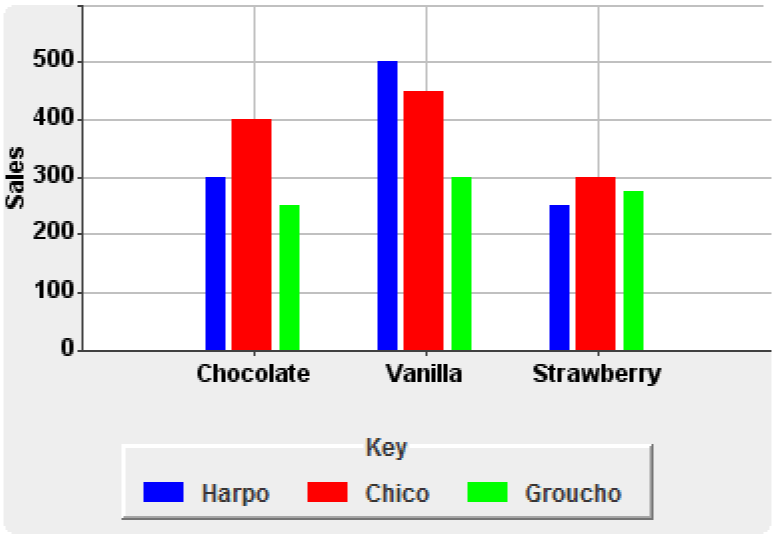
yields:



If you use the barGap and barGroupGap properties, the bars will expand in width to fill the size available. If you wish, you can explicitly set the width of the bars as part of the ChartStyle. The widths are set on a per model basis, so, for example, we could use the following to set the bar widths:

```
style1.setBarWidth(10);
style2.setBarWidth(20);
style3.setBarWidth(10);
```

The effect is that Chico's red bars become twice the width of the other two bars:



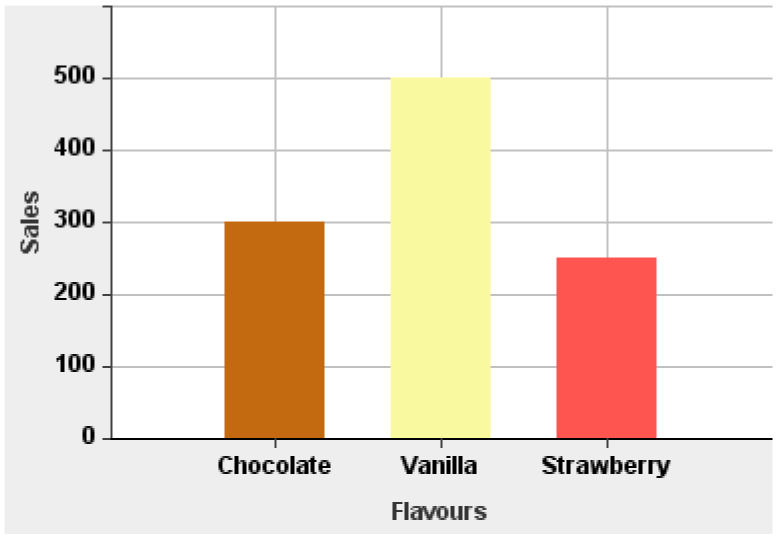
Changing the Colour of Individual Bars

We have already seen how to use a Chart Style to specify the colour of bars belonging to a particular Chart Model. This works well for stacked and grouped bar charts, but what if we want the bars of a single Chart Model to be displayed using different colours? For this, we use the concept of a 'highlight'. A highlight is a semantic tag that can be attached to one or more data points and used to change the styling. (In general it could be used for other purposes too, but for the moment it is used only for styling.) The highlight tag is associated with a ChartStyle, whose properties will override those that have been inherited from the Chart Model's style. In this way we can attach styling to multiple, arbitrary points and make it very easy to change their appearance without having to revisit each point in the model.

For the simple ice cream sales example for which the categories and axes have already been described, the following code fragment shows how to create the model containing highlighted values and customise the colours for individual 'points' of the model:

```
Highlight chocolateHighlight = new Highlight("Chocolate");
Highlight vanillaHighlight = new Highlight("Vanilla");
Highlight strawberryHighlight = new Highlight("Strawberry");

DefaultChartModel salesModel = new DefaultChartModel("Sales");
ChartPoint p1 = new ChartPoint(chocolate, 300, chocolateHighlight);
ChartPoint p2 = new ChartPoint(vanilla, 500, vanillaHighlight);
ChartPoint p3 = new ChartPoint(strawberry, 250, strawberryHighlight);

salesModel.addPoint(p1);
salesModel.addPoint(p2);
salesModel.addPoint(p3);

p1.setHighlight(chocolateHighlight);
p2.setHighlight(vanillaHighlight);
p3.setHighlight(strawberryHighlight);

chart.setHighlightStyle(chocolateHighlight, new ChartStyle(new Color(195, 105, 15)));
chart.setHighlightStyle(vanillaHighlight, new ChartStyle(new Color(249, 249, 159)));
chart.setHighlightStyle(strawberryHighlight, new ChartStyle(new Color(255, 85, 80)));
```

When this highlighted model is added to the chart and displayed as a bar chart, we get the following:



Changing the Outline

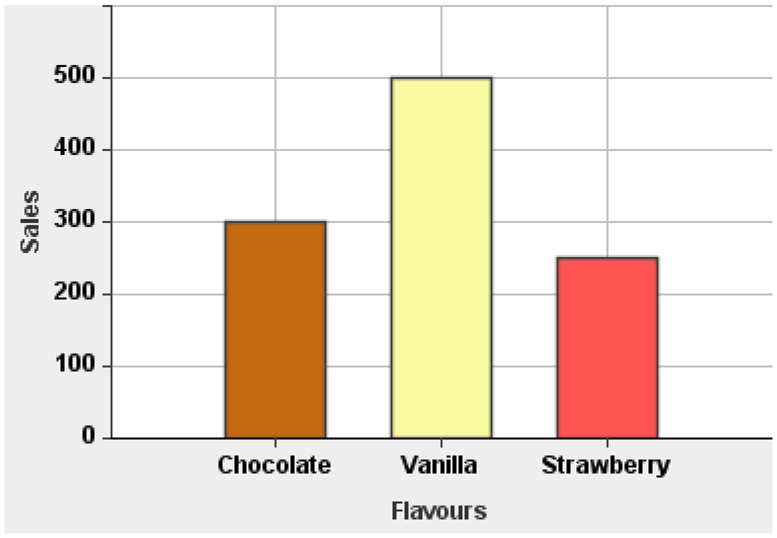
In the bar chart above, the vanilla coloured bar might be more distinguishable if the bars were to have a dark well-defined outline. The outline can be added by setting an option on the bar renderer that is used for drawing the bars. Different bar renderers offer different visual styles, but all of them provided by JIDE support the configuration of outline width and colour.

So to modify the example above, we could do the following:

```
DefaultBarRenderer renderer = new DefaultBarRenderer();
renderer.setOutlineColor(Color.darkGray);
renderer.setOutlineWidth(1.5f);
renderer.setAlwaysShowOutlines(true);
chart.setBarRenderer(renderer);
```

We need to tell the renderer that we wish to *always show outlines* because outlines are not shown by default, and there is an alternative setting for which an outline is shown only when the data point is selected.

Here is the result:



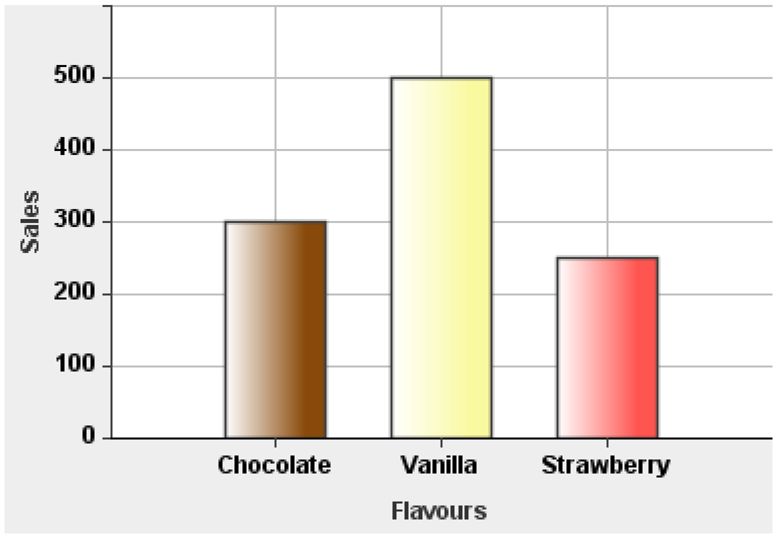
Changing the Fill

The examples above showed how to create bar charts in which the fill style of the bars is a uniform colour. It is also possible to create bars that use an implementation of the Paint interface to fill the bars. To make use of this, use the setBarPaint() method on the ChartStyle class instead of setBarColor(). (Note that a Color is also a Paint, so you could decide, as a matter of policy, to always use setBarPaint(), even when filling with a Color.)

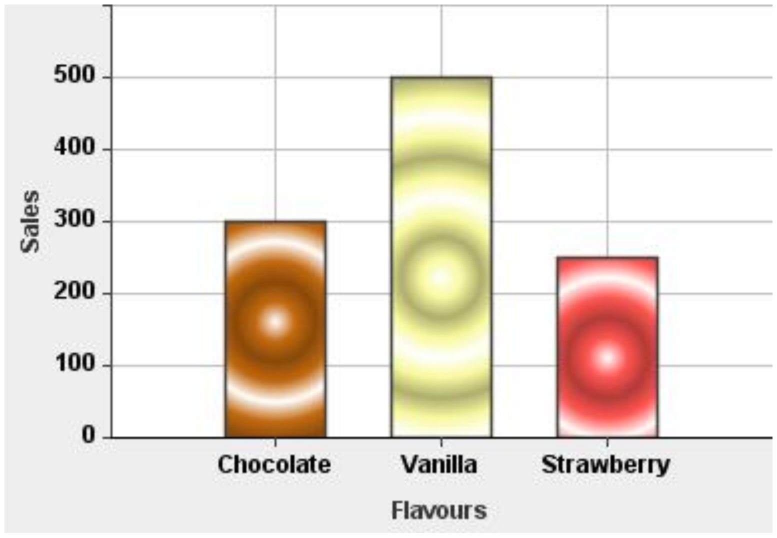
Here is how we might configure a LinearGradientPaint for the bar representing chocolate sales:

```
Color chocolateColor = new Color(195, 105, 15);
Paint chocolateFill = new LinearGradientPaint(0f, 0f, 50f, 0f, new float[] {0f,
0.8f}, new Color[] {Color.white, chocolateColor});
ChartStyle chocolateStyle = new ChartStyle();
chocolateStyle.setBarPaint(chocolateFill);
chart.setHighlightStyle(chocolateHighlight, chocolateStyle);
```

If we follow the same pattern for the vanilla and strawberry bars, we generate the following bar chart, in which the colours for each bar become progressively darker from left to right:

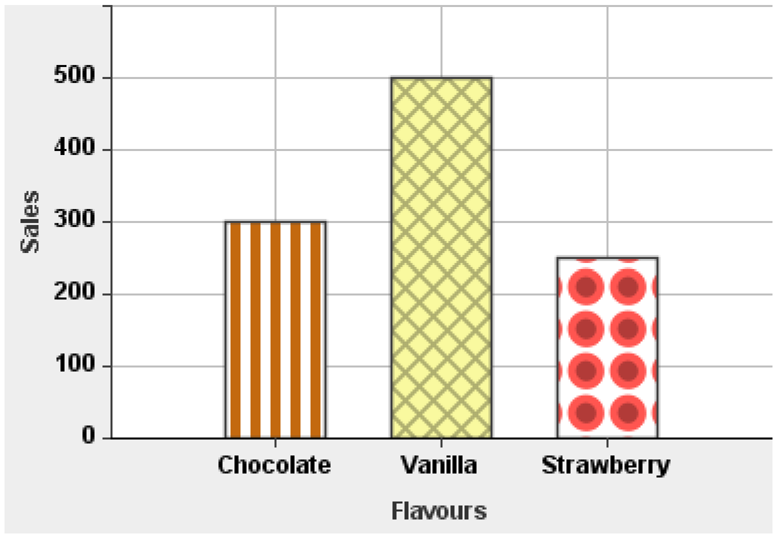


Instead of a LinearGradientPaint, we could use a RadialGradialPaint to generate the following:



Or you could use a TexturePaint to create a fill style based on an image loaded from a file.

We have also introduced a new Paint class called StripePaint that we believe, among other things, will prove very useful as the fill style for bars. As the name suggests, the basic idea behind a StripePaint is to build a fill pattern consisting of stripes. So, for example, if I construct an instance as new StripePaint(90, 10, 5f), it will create stripes with a rotation of 90 degrees – that is, vertical stripes; with a distance between the stripes of 10 pixels and the stripes themselves being 5 pixels wide. By setting the foreground colour to be a chocolatey brown and leaving the background colour white, we create the fill style used in the bar for chocolate ice cream sales in the following screen shot:



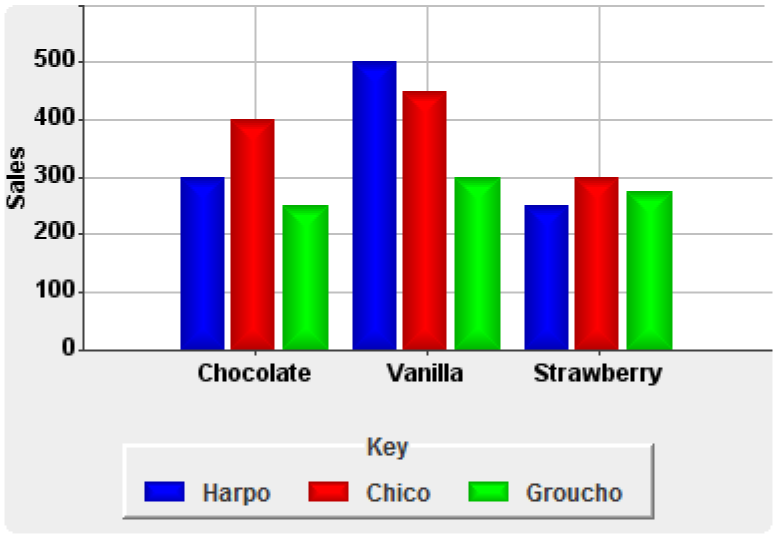
The clever thing about the StripePaint class is that the background need not be a Color instance – it can be any kind of Paint. The most obvious application of this is to use a StripePaint as the background for another StripePaint, and this is what we have done for the fill of the bar for Vanilla ice cream sales. The effect here is to create a check pattern, although many other effects are possible. Another feature of the StripePaint class is that you can set the Stroke that is used for the stripe – so it becomes easy to generate dotted stripes. In the bar for strawberry ice cream sales, we have used this technique to create dotted stripes that have circular end caps and a length of 0, together with the same layered approach used for the check effect, but this time to create dots inside larger dots. There are lots of possibilities with this class – further examples can be found in the JavaDoc.

Changing the Renderer

Until now, we have used a DefaultBarRenderer, which draws rectangular bars filled with a single colour. Three other renderers are currently available: a RaisedBarRenderer, a Bar3DRenderer and a CylinderBarRenderer. In the following, we set the barGap to 3, the barGroupGap to 10 and the bar renderer to the RaisedBarRenderer :

```
chart.setBarGap(3);
chart.setBarGroupGap(10);
chart.setBarRenderer(new RaisedBarRenderer());
```

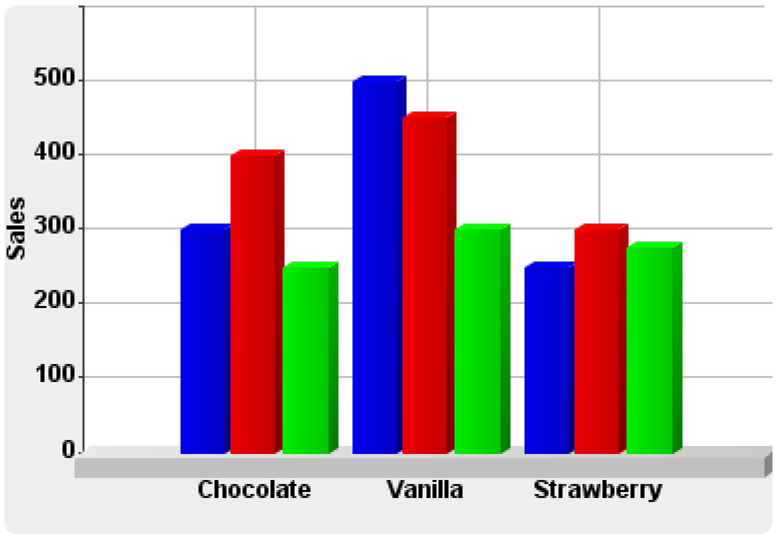
This gives the following:



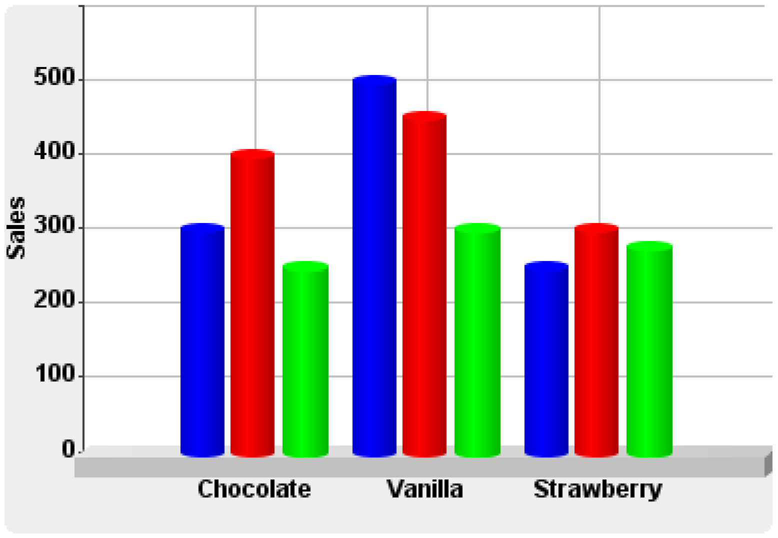
If we use the Bar3DRenderer or the CylinderRenderer, we should change the appearance of the x axis to give a true 3D effect.

```
chart.setBarRenderer(new Bar3DRenderer());
//chart.setBarRenderer(new CylinderBarRenderer());
chart.getXAxis().setAxisRenderer(new Axis3DRenderer());
```

Here is the Bar3DRenderer:

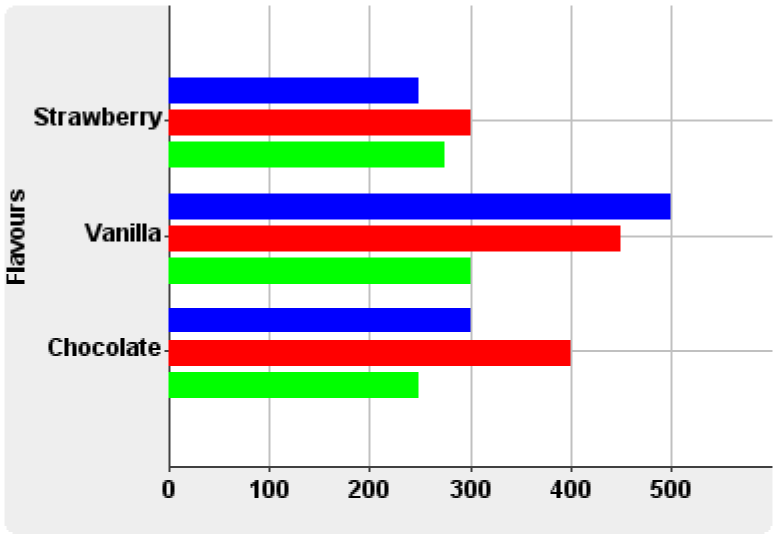


And here is the CylinderBarRenderer:



Bar Chart Orientation

The bars in your bar chart do not have to be vertical as shown in these examples – JIDE Charts also supports horizontal bars. Here is the same chart shown as a horizontal bar chart (and using the DefaultBarRenderer):



The differences in the code are:

- you need to reverse the axes, so that the ice cream flavours are used as the y axis and the numeric values are used as the x axis
- when preparing the models, you also need to swap the x and y coordinates so that flavours are used as the y coordinates.

- You need to set the Orientation on each of the three ChartStyles; as in `style1.setBarOrientation(Orientation.horizontal);`

Pie Charts

Pie charts are dealt with in a similar way to most bar charts. You need to prepare a chart model from a categorical axis and a numeric axis. However, the categorical axis needs to be used as the x axis on the Chart instance. Think of a pie chart as a special kind of bar chart in which the x axis has been wrapped around into the circumference of a circle. You tell the chart component that you would like a pie chart by calling `chart.setChartType(ChartType.PIE)`.

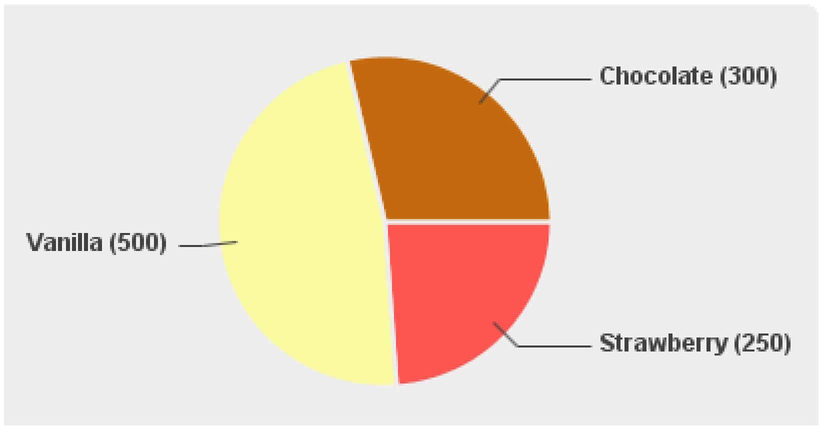
This approach makes it very easy to switch between bar chart and pie chart representations of the same data.

At this stage all the segments of the pie chart would be coloured the same, according to the style set up for the ChartModel. Usually we would want to colour each segment differently. For this, we use the concept of a 'highlight' to tag points in the same way that has already been described for bar charts. This highlight tag is then associated with a ChartStyle, so that it becomes easy for multiple points to share the same style.

For example, suppose in our original ice cream sales example, we wanted the segments of the bar chart to correspond to the colours of the ice creams they represent. We can do this as follows:

```
Highlight chocolateHighlight  = new Highlight("Chocolate");
Highlight vanillaHighlight    = new Highlight("Vanilla");
Highlight strawberryHighlight = new Highlight("Strawberry");

ChartPoint p1 = new ChartPoint(chocolate, 300);
ChartPoint p2 = new ChartPoint(vanilla, 500);
ChartPoint p3 = new ChartPoint(strawberry, 250);

DefaultChartModel salesModel = new DefaultChartModel("Sales");
salesModel.addPoint(p1).addPoint(p2).addPoint(p3);

p1.setHighlight(chocolateHighlight);
p2.setHighlight(vanillaHighlight);
p3.setHighlight(strawberryHighlight);

Chart chart = new Chart();
chart.setChartType(ChartType.PIE);

chart.setHighlightStyle(chocolateHighlight, new ChartStyle(new Color(195, 105, 15)));
chart.setHighlightStyle(strawberryHighlight, new ChartStyle(new Color(255, 85, 80)));
chart.setHighlightStyle(vanillaHighlight, new ChartStyle(new Color(249, 249, 159)));
```

The pie chart looks as follows:



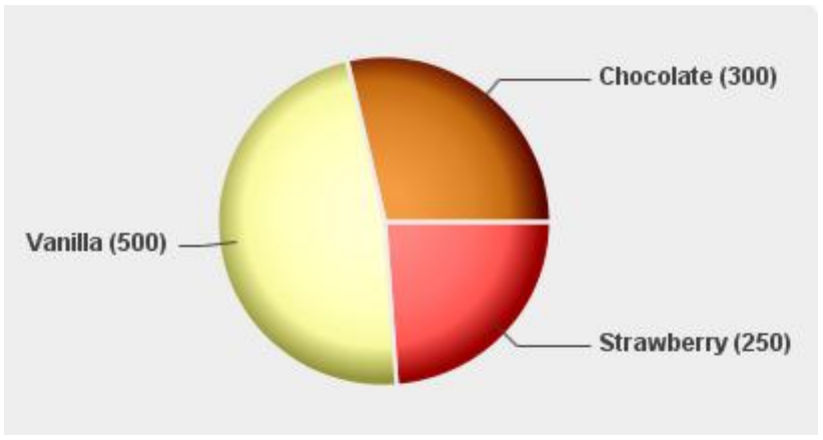
The reason we separate the highlight from its styling is so that we can change the appearance easily if the same highlight is used by many points, which is common for XY charts. There is no need to iterate through the points and change the colour of some of them – simply change the associated style on the chart instance.

Changing the Renderer

As with bar charts, we have a choice of renderers. The pie chart shown above uses the DefaultPieSegmentRenderer. If we call

```
chart.setPieSegmentRenderer(new RaisedPieSegmentRenderer());
```

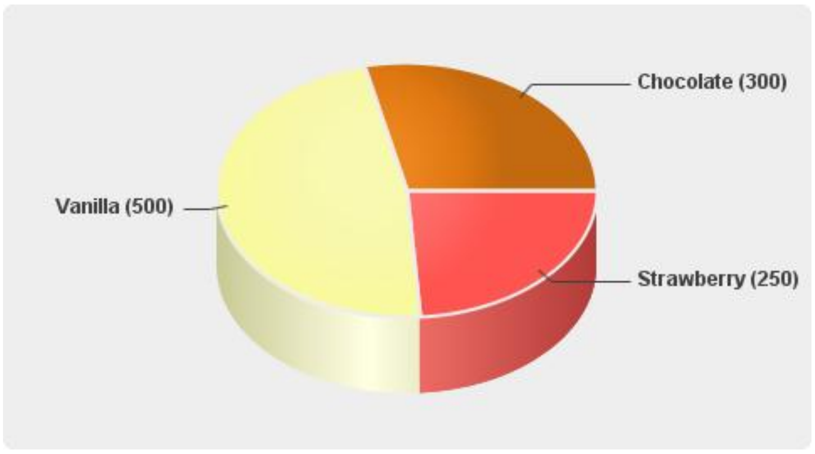
we get the following:



And if we call

```
chart.setPieSegmentRenderer(new Pie3DRenderer());
```

we get this 3D rendering:



By default, there is an outline surrounding the segments of the pie chart, whose default color is defined by the Swing UIManager's Chart.background property. However, you can easily override this default by calling setOutlineColor() on the PieSegmentRenderer. You can also specify the width of the outline with the setOutlineWidth() method. If you prefer not to show the outline, call chart.setAlwaysShowOutlines(false). The property is called alwaysShowOutlines because you still have the option of displaying outlines when a segment is selected.

Segment Selection

Pie Charts have been designed to be selectable, so that users can indicate segment(s) of interest. This could be used, for example, as part of a 'drill-down' mechanism in which the user selects the segments of the pie chart for which more detailed information should be displayed. The implementation uses a ListSelectionModel to store segment selections, so the selection model can be shared with other components, such as a JList or a JTable, which also use list selection models. (Another advantage is that you can use the same listener to listen to selection changes coming from a Chart, a JTable or a JList.)

Specify whether a chart is selectable by using the chart.setSelectionEnabled() method.

When a ChartModel is added to a chart, a corresponding ListSelectionModel is created and stored internally. You can retrieve the ListSelectionModel for a ChartModel by calling chart.getSelectionsForModel(ChartModel), or replace it by calling chart.setSelectionsForModel(ChartModel, ListSelectionModel).

A ListSelectionModel maintains a selectionMode, which specifies whether to allow single or multiple selection. Multiple selection is the default; if you need single selection of pie segments, call the following:

```
ListSelectionModel lsm = pieChart.getSelectionsForModel(pieChartModel);
lsm.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

Selections are indicated in one of two ways in a pie chart: either by displaying an outline around the selected segment or by 'exploding' the segment and moving it away from the centre of the pie. To switch on the outline, call chart.setSelectionShowsOutline(true); to switch on the 'explosion' effect, call chart.setSelectionShowsExplodedSegments(true). If you wish, you can have both effects switched on simultaneously. For the outline selection effect, you can specify the colour of the outline by calling pieSegmentRenderer.setSelectionColor(mySelectionColor).

Axes

There are three kinds of axis, which can be used in any kind of XY Chart: Numeric Axis, Time Axis and Category Axis. You should use the axis type that corresponds to the type of data that will be provided for positioning points along it.

Numeric Axes

By default, the Chart class creates numeric axes for x and y, both in the range [0, 1]. If you need a numeric axis, but for a different range of values – say 0 to 100 – you can do the following:

```
chart.getXAxis().setRange(0, 100);
```

Alternatively, you can create a new Axis object:

```
NumericAxis xAxis = new NumericAxis(0, 100);
chart.setXAxis(xAxis);
```

The code shown configures the x axis; you would take the same approach to configure the y axis.

To label an axis, the simplest approach is to call the setLabel() method on the axis with a string containing the text of the label:

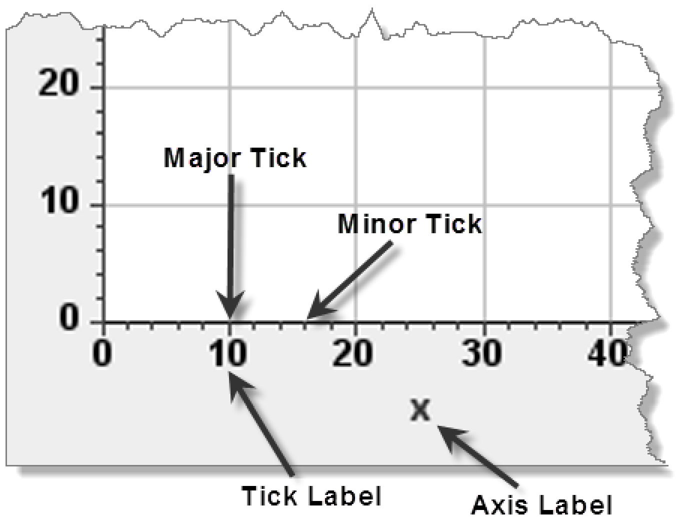
```
xAxis.setLabel("X Axis");
```

However, if you would like to configure the font and/or colour of the label then you will need to create an instance of an AutoPositionedLabel object. The following creates a bold blue label of point size 14, using the same font face as a JLabel:

```
Font labelFont = UIManager.getFont("Label.font").deriveFont(Font.BOLD, 14f);
xAxis.setLabel(new AutoPositionedLabel("X Axis", Color.blue, labelFont));
```

Custom Ticks

The ticks and labels that appear on an axis automatically are designed to suit most purposes. By default, a numeric axis displays a small number of major ticks, each with a tick label, and four minor ticks (five intervals) to divide up the region between major ticks:



You may find that you wish to change the position or type of ticks and their corresponding labels. You can customise the ticks on a numeric axis by changing the *tick calculator*. When you first create a NumericAxis, it uses a DefaultNumericTickCalculator to calculate the ticks and labels. The two most common ways of customising the ticks are to use a SimpleNumericTickCalculator or to subclass DefaultNumericTickCalculator and override the calculateTicks() method.

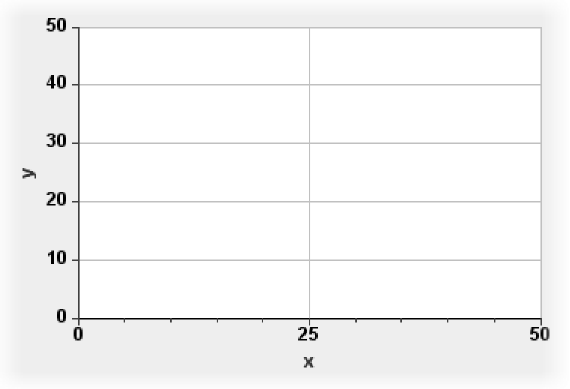
Using a SimpleNumericTickCalculator

The SimpleNumericTickCalculator class implements the TickCalculator interface and provides constructors and methods that allow you to specify where the ticks should be. For example, when constructing an instance of a SimpleNumericTickCalculator, you can supply the start value, the interval between major ticks and (optionally) the interval between minor ticks.

This code creates major ticks at a spacing of 10 on the y axis and on the x axis creates major ticks at a spacing of 25, with minor ticks every 5 units:

```
NumericAxis xAxis = new NumericAxis(0, 50, "x");
xAxis.setTickCalculator(new SimpleNumericTickCalculator(0, 25, 5));
NumericAxis yAxis = new NumericAxis(0, 50, "y");
yAxis.setTickCalculator(new SimpleNumericTickCalculator(0, 10));
chart.setXAxis(xAxis);
chart.setYAxis(yAxis);
```

Here is a screenshot of the tick layout generated by the code:



Using a DefaultNumericTickCalculator

The SimpleNumericTickCalculator is easy to use when a chart is being generated as a one-off event or the numeric ranges of the chart do not change. Tick generation becomes more involved if you display a chart in a graphical user interface and allow your users to zoom or pan. In this case you are best advised to use a DefaultNumericTickCalculator, but you can still customise the ticks if you wish by overriding the calculateTicks() method. For example, the following code generates ticks at an interval of 10, regardless of the range of the axis:

```
xAxis.setTickCalculator(new DefaultNumericTickCalculator() {
     public Tick[] calculateTicks(Range<Double> r) {
        Integer n = 10 * ((int) (r.minimum()/10));
        List<Tick> ticks = new ArrayList<Tick>();
        while (n < r.maximum()) {
          ticks.add(new Tick(n, n.toString()));
          n += 10;
        }
        return ticks.toArray(new Tick[ticks.size()]);
      }
  });
```

Specifying a Number Format for the Tick Labels

Both the SimpleNumericTickCalculator and the DefaultNumericTickCalculator maintain a *numberFormat* property, whose value is used when generating tick labels. This property could be used to good effect if your tick labels:

- are particularly large numbers;
- need to be labelled with more precision than is given by default; or
- need to be displayed in a particular notation, such as engineering notation.

For example, suppose instead of the default formatting of numbers as 10000 or 100000 you would like to add commas to help users easily see the magnitude of the number without having to count the zeros, giving a number such as 10,000 or 100,000.

You can do this as follows:

```
NumericAxis yAxis = new NumericAxis(0, 100000);
yAxis.setNumberFormat(new DecimalFormat("#,###"));
chart.setYAxis(yAxis);
```

Alternatively, you can set the number format directly on the tick calculator:

```
DefaultNumericTickCalculator c = new DefaultNumericTickCalculator();
c.setNumberFormat(new DecimalFormat("#,###"));
yAxis.setTickCalculator(c);
```

Time Axes

To create a time axis, you create an instance of the TimeAxis class and supply it with a TimeRange instance that covers the time period of interest. For example, the following code creates an x axis to show values over the last hour:

```
long now = System.currentTimeMillis();
long oneHourAgo = now - 1000 * 60 * 60;
TimeRange timeRange = new TimeRange(oneHourAgo, now);
TimeAxis xAxis = new TimeAxis(timeRange);
chart.setXAxis(xAxis);
```

Category Axes

A category axis is used to show discrete values taken from a category range. The following code creates a category range and then creates an axis to show those values:

```
Category<String> football = new Category<String>("Football");
Category<String> cricket = new Category<String>("Cricket");
Category<String> wellyWanging = new Category<String>("Welly Wanging");
CategoryRange<String> sports = new CategoryRange<String>();
sports.add(football).add(cricket).add(wellyWanging);
CategoryAxis<String> xAxis = new CategoryAxis(sports, "Popular Sports");
chart.setXAxis(xAxis);
```

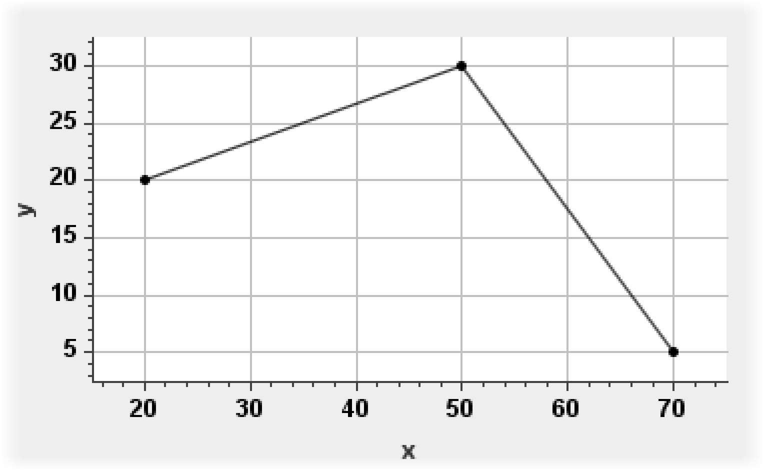
As categories are discrete values and it is not meaningful to assign a value that is between two categories, minor ticks are not normally displayed on a category axis. (If you wanted to do this you could set the tick calculator to be a DefaultCategoryTickCalculator and override the calculateTicks() method, as with a NumericAxis.)

Auto-Ranging of Axes

NEW: It is now possible to "auto-range" the axes; that is, to ask the charts package to derive good x and y ranges for your chart, based on the data being plotted. The simplest way to do this is to call chart.setAutoRanging(true) on a numeric or time-based chart. For instance, the following code:

```
        Chart chart = new Chart();
        chart.setAutoRanging(true);
        DefaultChartModel model = new DefaultChartModel();
        model.addPoint(20, 20).addPoint(50, 30).addPoint(70, 5);
        chart.addModel(model);
```

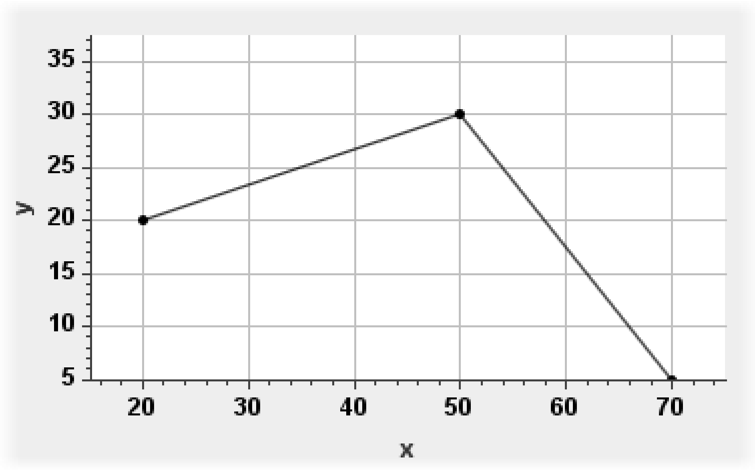
generates this chart, without having to explicitly set the axis ranges:



When auto-ranging is switched on, an instance of an AutoRanger class is used to calculate the appropriate axis ranges. If you don't explicitly set the AutoRanger, an instance of the class DefaultAutoRanger is used, and by default this allows a 10% margin of space around your data. If you prefer different amounts of space, you can configure this by calling setMarginProportions() on the instance of DefaultAutoRanger that you use. This method takes four parameters, each being a number in the range 0..1 to represent the amount of space for the margins on the top, left, bottom, and right of the chart, respectively. For example, in the following code we allow a 30% margin at the top of the chart, but 0% at the bottom, and leave the left and right margins at 10% as before:

```
Chart chart = new Chart();
chart.setAutoRanging(true);
DefaultAutoRanger ranger = new DefaultAutoRanger();
ranger.setMarginProportions(0.3, 0.1, 0.0, 0.1);
chart.setAutoRanger(ranger);
```

With this code, the chart looks as follows – notice how the range of the y axis has changed compared to the previous screenshot:

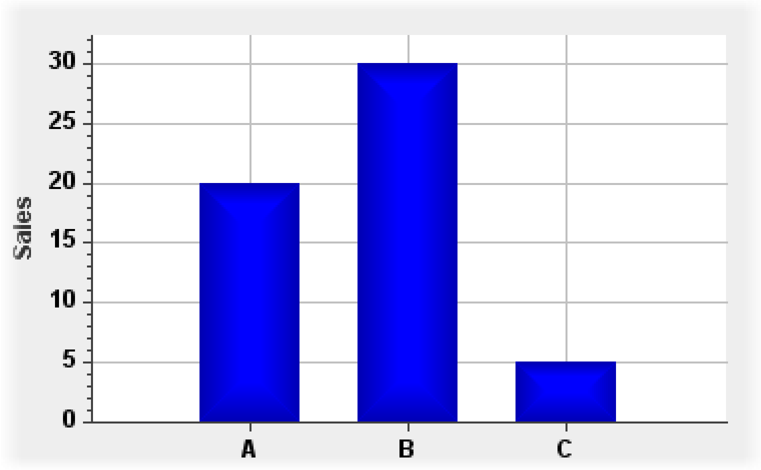


The example above showed how auto-ranging can be applied to all four sides of an XY chart, but sometimes this is not what you want. In fact, quite often, you need to be sure that an axis is showing zero at the low end of the scale, but you would like the higher values to be auto-ranged. For this kind of situation, the DefaultAutoRanger allows you to fix some of the axis range maxima or minima, while leaving the others to be calculated. You do this by passing in the fixed values to the constructor of the DefaultAutoRanger, using nulls for those that are to be calculated automatically. The fixed values are supplied as minX, minY, maxX, maxY; so if we create our instance with `new DefaultAutoRanger(null, 0.0, null, null)` then we are fixing the minimum value on the y axis to be 0. This approach can be used in many different situations, but is particularly well-suited for bar charts.

For example, this code:

```
Chart chart = new Chart();
chart.getYAxis().setLabel("Sales");
chart.setAutoRanging(true);
chart.setAutoRanger(new DefaultAutoRanger(null, 0.0, null, null));
ChartCategory<String> cat1 = new ChartCategory<String>("A");
ChartCategory<String> cat2 = new ChartCategory<String>("B");
ChartCategory<String> cat3 = new ChartCategory<String>("C");
CategoryRange<String> categoryRange
   = new CategoryRange<String>().add(cat1).add(cat2).add(cat3);

CategoryAxis<String> xAxis = new CategoryAxis<String>(categoryRange);
chart.setXAxis(xAxis);

DefaultChartModel model = new DefaultChartModel();
model.addPoint(cat1, 20).addPoint(cat2, 30).addPoint(cat3, 5);
ChartStyle style = new ChartStyle(Color.blue).withBars();
style.setBarWidth(50);
chart.addModel(model, style);
chart.setBarRenderer(new RaisedBarRenderer());
```

generates this chart:



Note that for performance reasons, it is not advisable to use auto-ranging when dealing with large data sets. In general, whenever *any* of the data changes, *all* of the points of the chart have to be re-visited to calculate the minima and maxima. However, we have taken steps to optimise this behaviour when you use a DefaultChartModel. The DefaultChartModel class maintains its own x and y ranges as data is added, so auto-ranging for this common case does not require a visit to all the points of the chart.

Cross Hairs and Value Reporters

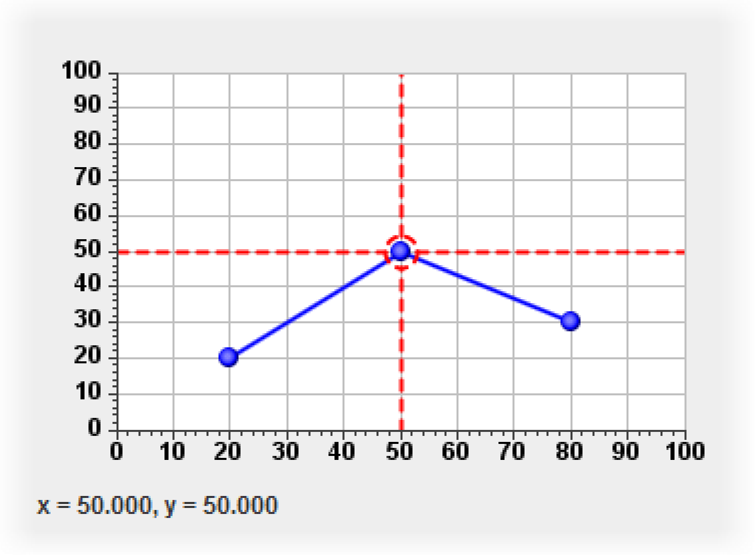
It is often important to give visual feedback to your users as they move their mouse over an XY chart. One useful way of doing this is to display a cross hair at a point (or connecting line) on a chart. The cross hair helps to draw the eye to the point of interest and makes it easier to read off the corresponding values on the x and y axis. Another way of providing feedback is to display a value reporter: a text label that provides the x and y values of the point nearest to the mouse cursor. These two features can be used independently, but in practise they are often used together on the same chart.

The easiest way to add a cross hair to a chart is to pass true as the second parameter to the ChartCrossHair constructor:

```
ChartCrossHair crossHair = new ChartCrossHair(chart, true);
```

Using this approach, the created object is automatically registered as a MouseListener and MouseMotionListener on the chart, and is also added as a Drawable to the chart. You can customise the appearance of the cross hair by setting properties such as the *color*, *stroke* and *circleDiameter*.

By default, the cross hair indicates a point on the nearest model to the mouse cursor, but if you wish, you can constrain the cross hair to points from one model only by calling setModel(). The cross hair assumes that points are joined by straight lines and will show values along those straight lines. However, if you set the snapToPoints property to true, it will only highlight the points of the model and will not indicate intermediate values.



x = 50.000, y = 50.000

The following code was used to generate the cross hair in the screenshot above:

```
ChartCrossHair crossHair = new ChartCrossHair(chart, true);
crossHair.setSnapToPoints(true);
crossHair.setColor(Color.red);
crossHair.setCircleDiameter(16);
crossHair.setStroke(new BasicStroke(2f,
                                    BasicStroke.CAP_ROUND,
                                    BasicStroke.JOIN_ROUND,
                                    5f, new float[] {5f, 5f}, 0f));

ChartValueReporter reporter = new ChartValueReporter(crossHair);
add(reporter, BorderLayout.SOUTH);
```

By passing the crosshair instance to the constructor of the ChartValueReporter, we are telling it that the ChartCrossHair and ChartValueReporter instances are being used together. When this is the case, the ChartValueReporter registers itself as a property listener with the ChartCrossHair instance and does not need to be added as a MouseListener or MouseMotionListener with the Chart. If you use the ChartValueReporter independently of ChartCrossHair, you pass the Chart instance into the constructor and you must add the ChartValueReporter as a MouseListener and MouseMotionListener to the Chart.

You can customise the label that reports x and y values in the ChartValueReporter by calling setFormatString(). The format string is applied to the x and y values of the corresponding point on the chart. So for numeric x and y values you could set the format string as follows:

```
reporter.setFormatString("Value is [%.2f, %.2f]");
```

Note that there is a special treatment if you are using a TimeAxis in your chart: the class automatically casts the time value to which the format string is applied from a double to a long. This enables you to use date formatting in the format string. For example, for a time series chart with time on the x axis, you might do the following:

```
reporter.setFormatString("x = %1$tH:%1$tM; y = %2$.3f");
```

Loading Data from a CSV or Tab-Separated File

JIDE Charts provides classes to read data from a comma-separated values (CSV) or tab-separated values (TSV) file into Java data structures. By using these classes, it becomes straightforward to generate charts from data files. In particular, spreadsheet applications such as Excel provide the facility to save data in CSV or TSV format.

A CSV file typically looks something like this:

```
Ice Cream Sales 2011
Salesman,Chocolate,Vanilla,Strawberry
Harpo,300,500,250
Chico,400,450,300
Groucho,250,300,275
```

In this example, the first line is a header line that is not translated into the ChartModel but could be used as the title of the chart. The second line contains a comma-separated list of column headings, and the other lines each contain the name of the salesman and the volume of chocolate, vanilla and strawberry ice creams sold. The data shown here is simple to deal with, and instead of using the CSV reader you might be tempted to parse the data using the split() method of java.lang.String. That approach would work with this data, but we advise you to use a CSV Reader class because problems would arise with the String.split() approach if one of the data values were to contain a comma. The CSV format allows values to contain commas if they are enclosed in quotes. For example, we could add another row of data to our example above with the salesman's name as "Chaplin, Charlie". A further complication is that quoted values may run over two or more lines. Our CSV Reader will take care of these complications for you.

To convert the data to a chart, we need to make some assumptions about the data that is being provided. The assumptions that need to be made will vary from one application to another and, as the developer, you will need to make some judgements about what is appropriate for your application. For this example, we shall assume that:

- there is exactly one header line, which we should use as a title for the chart;
- the second line has an entry for the salesman column, and some further entries, one per ice-cream flavour;

- the subsequent rows of the file contain a salesman's name, followed by numeric values — one for each of the flavours listed in line 2.

To read the file and create a chart, we first create the Chart object:

```
Chart chart = new Chart();
chart.setBorder(new EmptyBorder(5,5,5,30));
chart.setBarsGrouped(false);
chart.setAutoRanging(true);
chart.setBarRenderer(new RaisedBarRenderer(10));
```

Then we create a CSVReader object and use it to parse the input file. In the following code, the CSV file is found and read from the class path, so as not to hard-code the file's location. Alternatively, you can pass a File or a Reader object to the class as the input source. The result of the parsing is that a List of Strings is returned for each row of the input, the lists themselves presented as a List. Each string is one of the comma-separated values in the file. If the file you wish to parse is a tab-separated file, you pass the tab character (or other separator, if required) into the constructor of the CsvReader class.

```
CsvReader reader = new CsvReader();
InputStream is = getClass().getResourceAsStream("/resources/Ice-Cream Sales.csv");
List<List<String>> values = reader.parse(is);
```

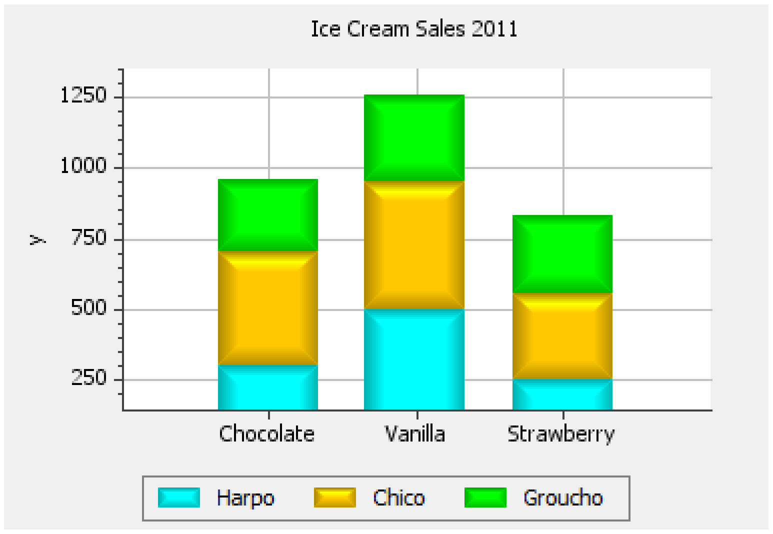
Now we can start processing the input and preparing the chart. Here, we strip and apply the title row from the input, create a category range from the column headers and use it to set the x axis to be a category axis:

```
List<String> titleRow = values.remove(0);
chart.setTitle(titleRow.get(0));
List<String> headerRow = values.remove(0);
CategoryRange<String> flavours = new CategoryRange<String>();
for (int i = 1; i < headerRow.size(); i++) {
    ChartCategory<String> flavour = new ChartCategory<String>(headerRow.get(i));
    flavours.add(flavour);
}
chart.setXAxis(new CategoryAxis<String>(flavours));
```

The following section creates a chart model for each row of the input data, with the model named after the salesman named on that row. A ChartStyle is also created for each model using the ColorFactory class to generate the colors. (It can be used to generate random colors, but here we use it to generate three preselected colours in a predefined order.)

```
ColorFactory colorFactory = new ColorFactory(Color.cyan, Color.orange, Color.green);

for (List<String> row : values) {
    String modelName = row.remove(0);
    DefaultChartModel model = new DefaultChartModel(modelName);
    int column = 1;
    for (String value : row) {
        Double v = Double.parseDouble(value);
        model.addPoint(flavours.getCategory(column), v);
        column++;
    }
    ChartStyle style = new ChartStyle(colorFactory.create()).withBars();
    style.setBarWidth(50);
    chart.addModel(model, style);
}
```

By adding a Legend to the South of the panel, we have produced the following chart:



Note that when a string in a CSV file contains spaces (or other white space) on either side of the value, there is no concensus over whether the space should be removed or retained. We therefore provide this as an option: see CsvReader.setTrimmingValues(). By default, trimming is switched on.

The approach described here reads the whole of the input data into memory, which works well for small files. For larger files, consider using using CsvReader.parseForEffects(), where you register a listener class that receives events as the input source is parsed. With this approach you need not hold the whole of the input data in memory at one time — you can write code that decides which parts to use and which parts to discard.

Frequently Asked Questions

How Do I Show Tooltips for Data Points?

The demo classes show examples of how to implements custom tooltip behaviour, but the simplest approach is to listen for a property change on the chart instance and call setTooltipText accordingly. This works with points and also with the bars of a bar chart.

```
chart.addPropertyChangeListener(new PropertyChangeListener() {
  public void propertyChange(PropertyChangeEvent evt) {
    if (Chart.PROPERTY_CURRENT_CHART_POINT.equals(evt.getPropertyName())) {
      Chartable p = chart.getCurrentChartPoint();
      if (p == null) {
        chart.setToolTipText(null);
      } else {
        String text = String.format("x = %.2f, y = %.2f", p.getX().position(),
                                                          p.getY().position());
        chart.setToolTipText(text);
      }
    }
  }
});
```

How Do I Save a Chart as an Image File?

There are methods in the class com.jidesoft.chart.util.ChartUtils to support this:

```
public static void writeGifToFile(Component, File);
public static void writeJpegToFile(Component, File);
public static void writePngToFile(Component, File);
```

These methods can be used for any Swing component, but are particularly useful for saving a Chart object as an image file.

How Do I Copy a Chart to the Clipboard?

There is a method, also in the ChartUtils class, which copies a PNG image of the supplied Swing component to the clipboard:

```
public static void copyImageToClipboard(Component);
```

Again, this can be used for any Swing component, but is particularly useful for providing the facility to copy Chart objects, so that they can easily be pasted into a PowerPoint presentation, for example.

What If I Have an Unanswered Question?

Contact us for technical support on the discussion forum! If you are not yet a JIDE customer, you can post a message on the pre-sales forum; if you are already a customer, please post your question to the JIDE Charts forum. See www.jidesoft.com/forum.