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 mousedrag 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):

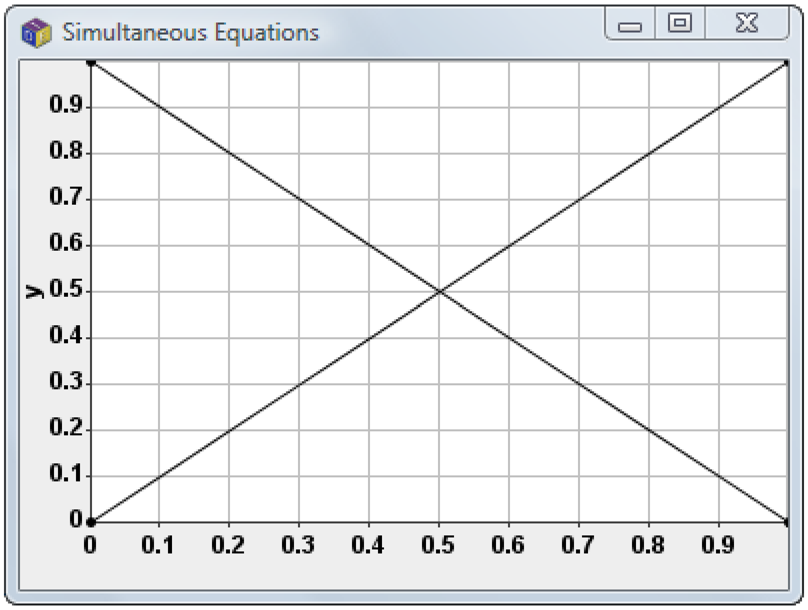
```
modelB.addPoint(0, 1);
modelB.addPoint(1, 0);
```

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);
```

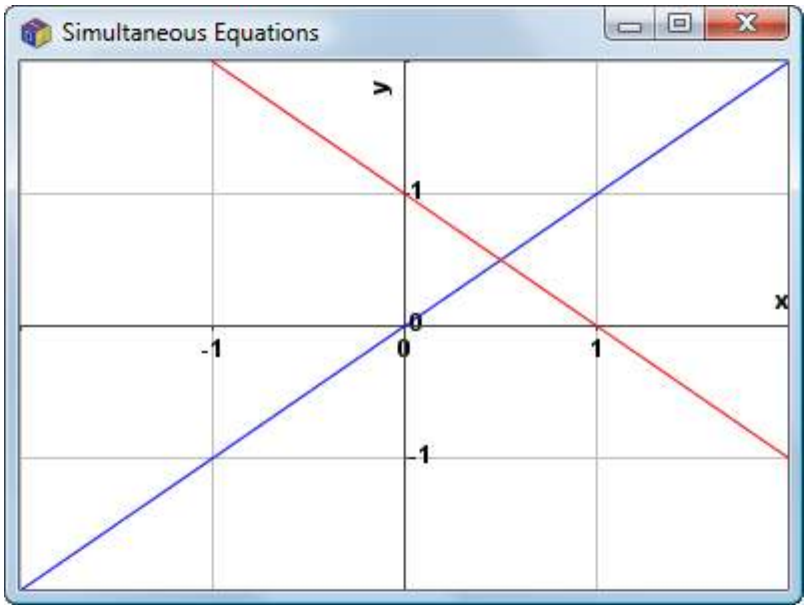
Finally, we are interested in seeing only the lines and not the points that make up the lines. We can specify this by using a ChartStyle. For example, to specify a ChartStyle in which you would like to see blue lines but no points, you create a ChartStyle and use it when adding the ChartModel to the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

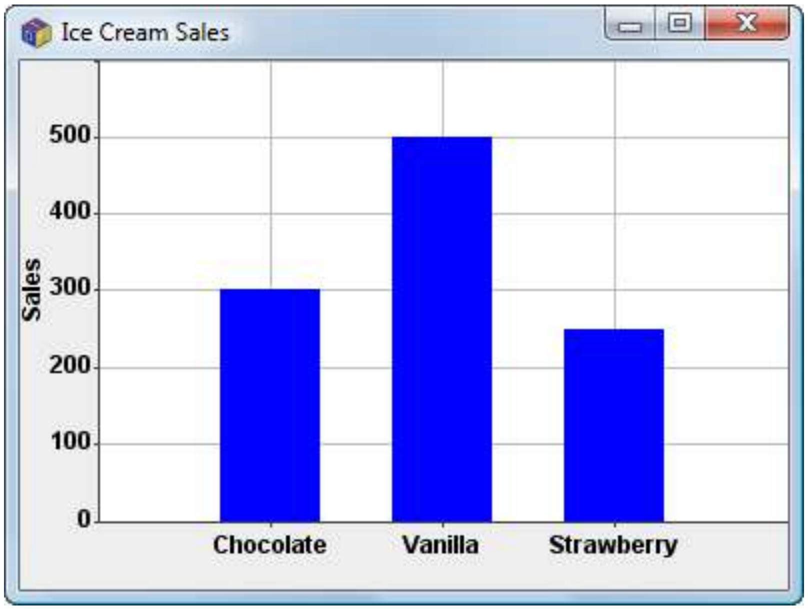
```
Chart chart = new Chart();
chart.setXAxis(new CategoryAxis(flavors, "Flavors"));
chart.setYAxis(new Axis(new NumericRange(0, 600), "Sales"));
```

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.setBarVisible(true);
chart.addModel(salesModel, style);
```

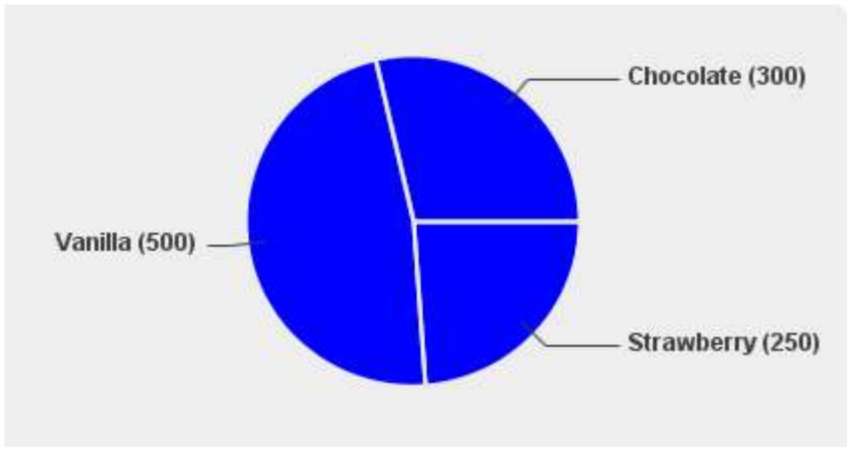
This generates the following bar chart:



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);
```

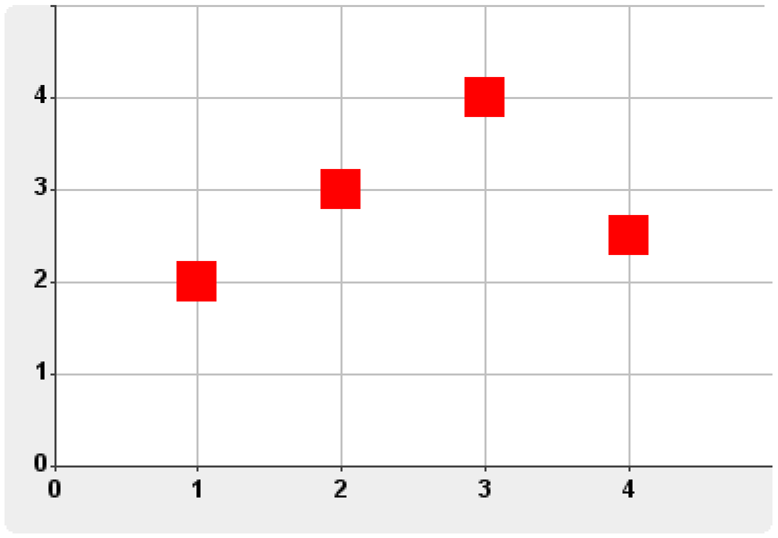
Point Plots

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:

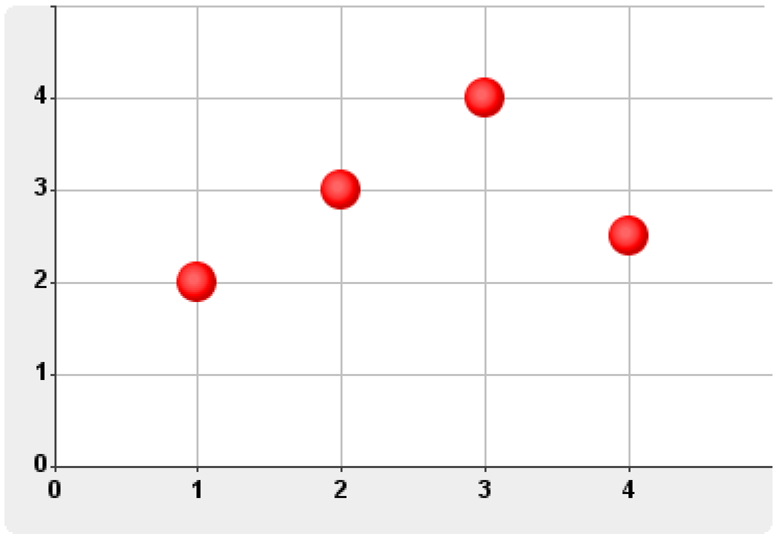


The predefined point shapes are CIRCLE, DISC, SQUARE, BOX, DIAMOND, DOWN_TRIANGLE, UP_TRIANGLE, HORIZONTAL_LINE, VERTICAL_LINE and UPRIGHT_CROSS.

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:

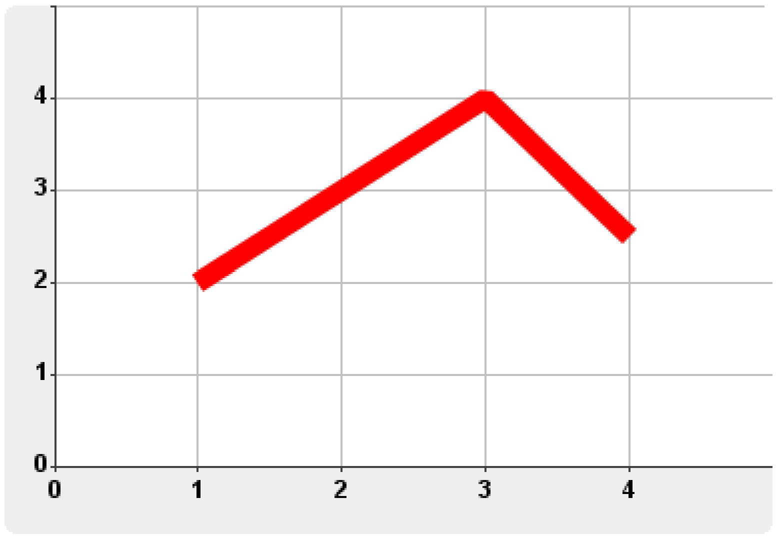


Line Plots

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:

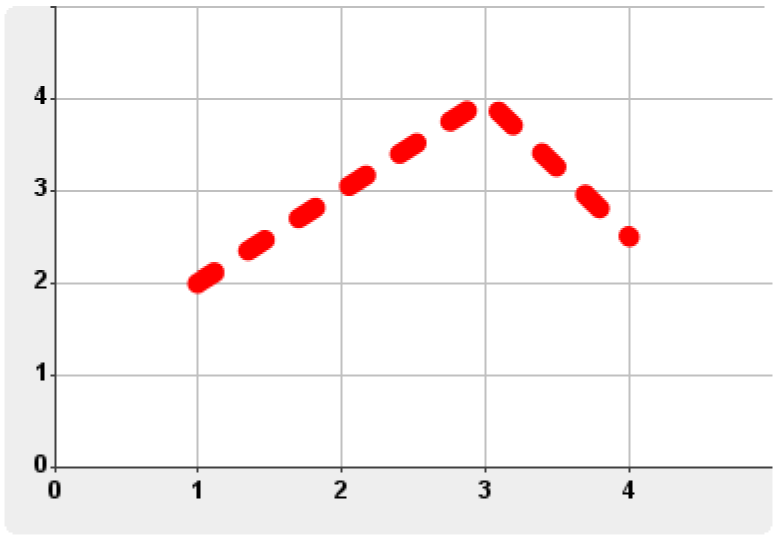


Note that you can produce much more detailed charts by using a thinner line width and a model containing many more points. For example, the same technique has been used for producing Electrocardiogram (ECG) plots from using data collected from a heart sensor.

As 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));
```

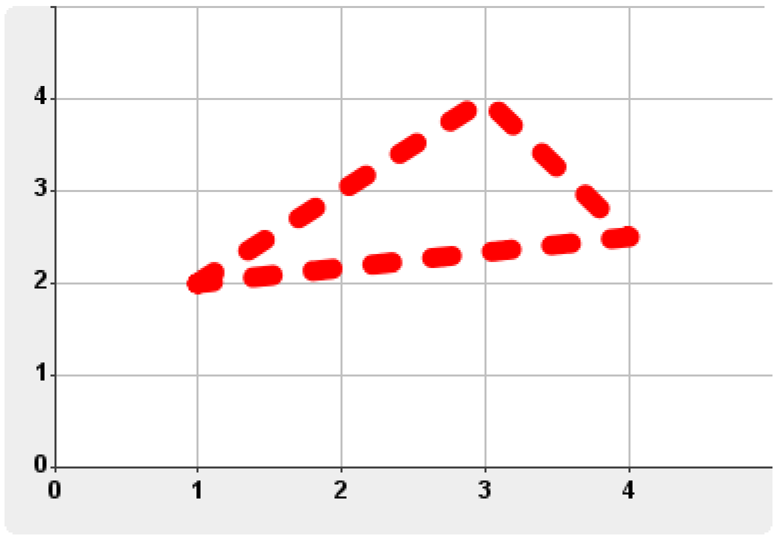
This generates the following chart:



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:



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());
```

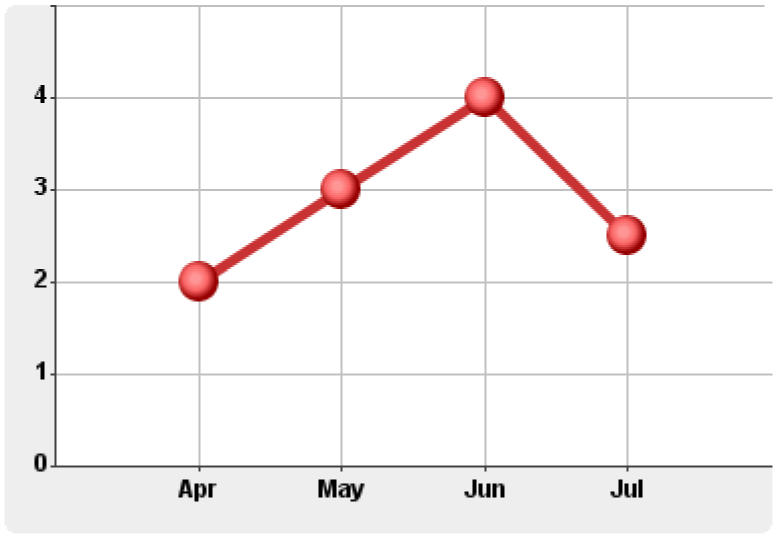
Lastly, we create and set a time range for the x axis of the chart:

```
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);
```

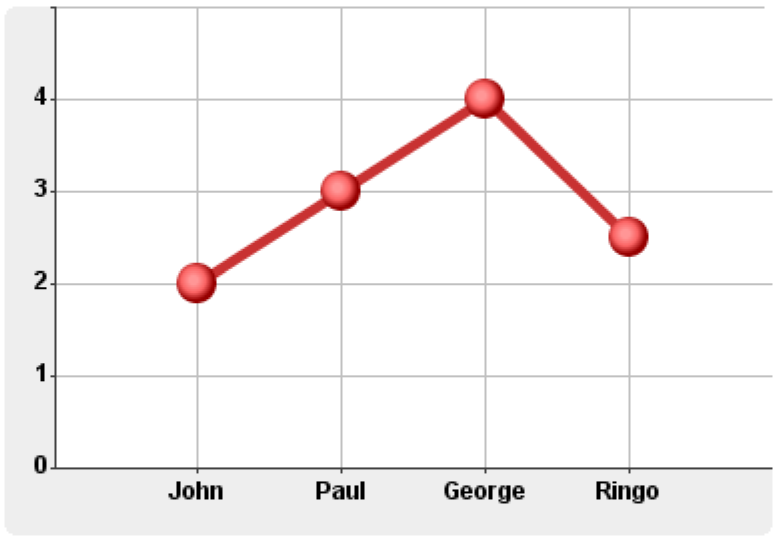
We also need to create a range made from these category values to set on the axis:

```
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:

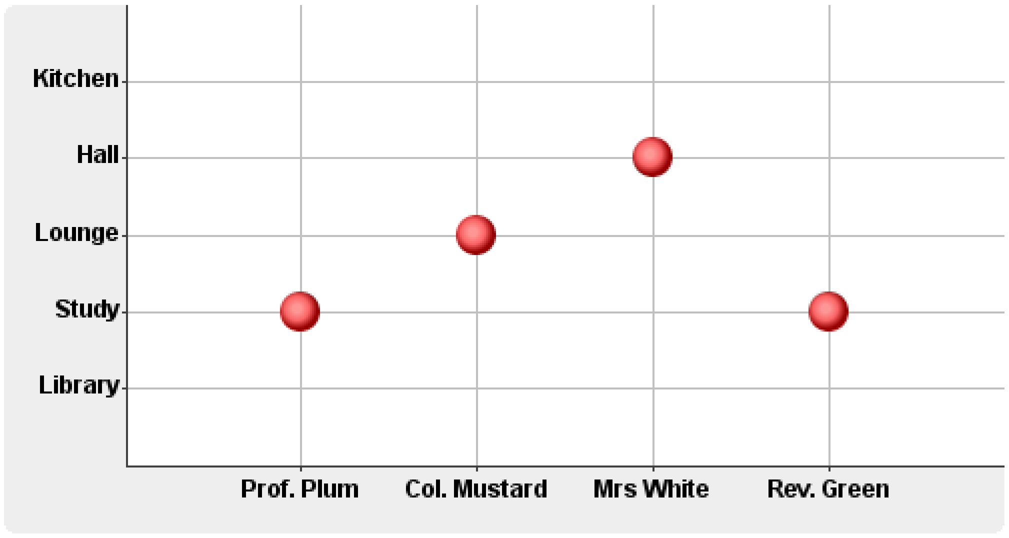
```
DefaultChartModel model = new DefaultChartModel();
model.addPoint(cJohn,   2);
model.addPoint(cPaul,   3);
model.addPoint(cGeorge, 4);
model.addPoint(cRingo,  2.5);
```

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

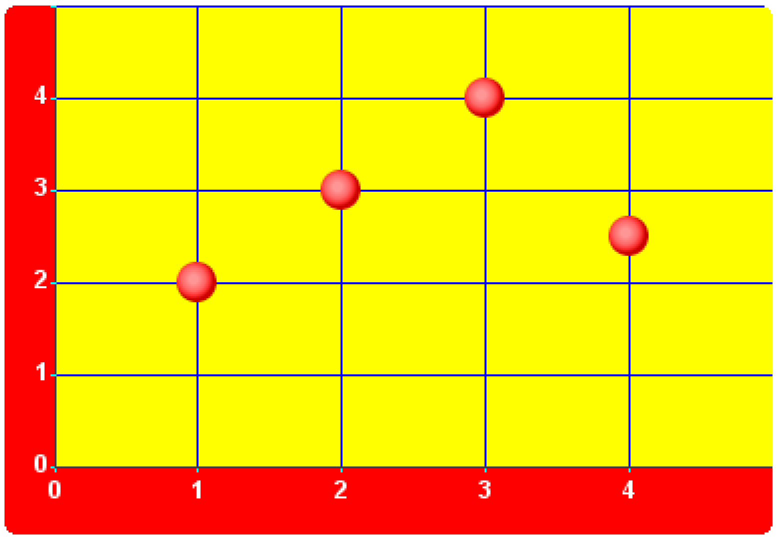
Colors

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:

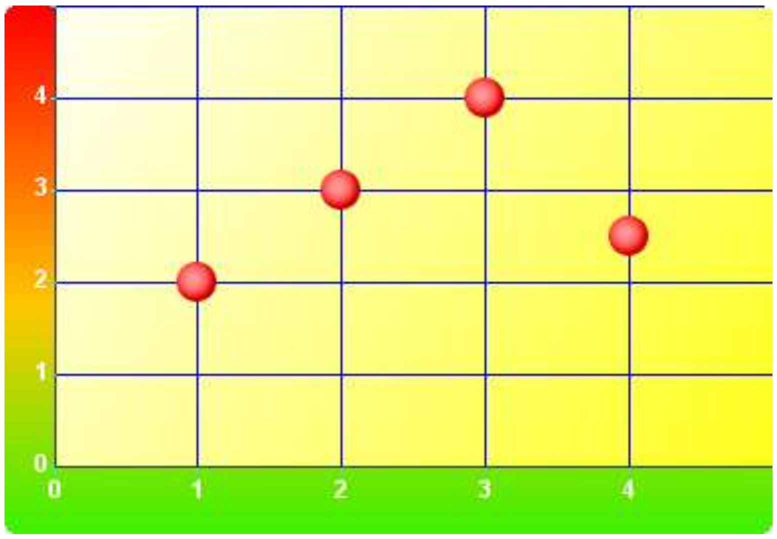


If you look at the API, you will notice that the chartBackground and panelBackground properties are actually an instance of a Paint, which means that you can supply a Color as in the example above, or you can supply a Paint with a colour gradient effect.

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:

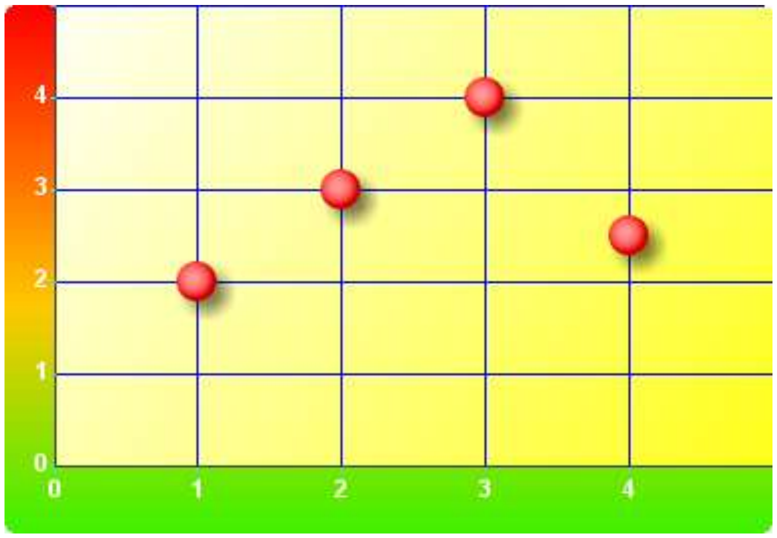


Shadow Effect

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

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

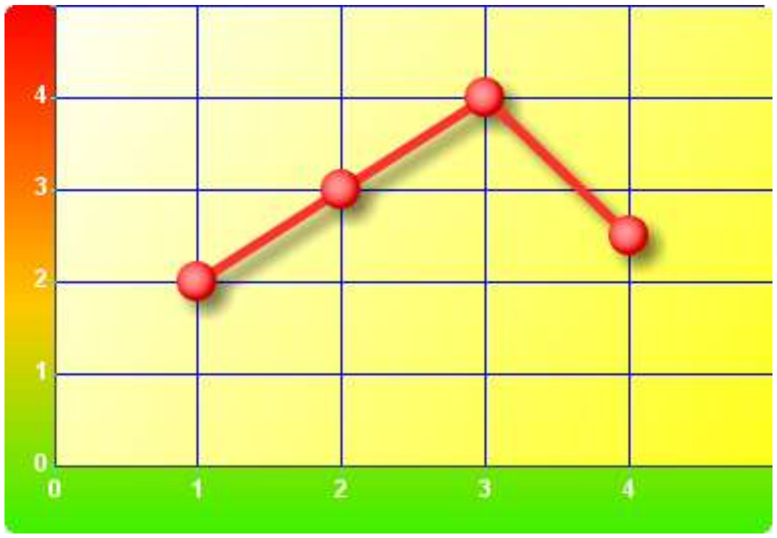
The same chart then looks as follows:



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.

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, (for example when using larger datasets) use chart.setAnimateOnShow(false). To re-start the animation, use chart.startAnimation().

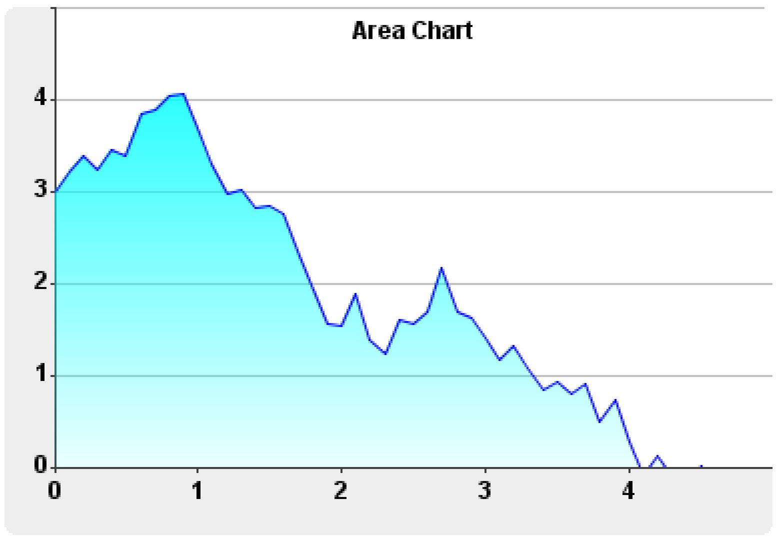
Area Charts

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):

```
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. 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).

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);
```

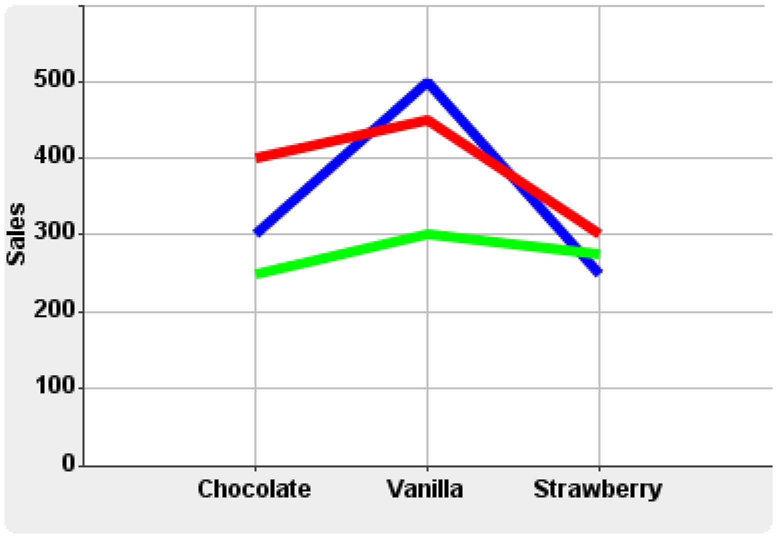
This will always perform background rendering, regardless of the number of points to plot. For more control, you can set a threshold of, say, 10000 so that background rendering is activated only when there are more than 10000 points to plot.

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) {...}
}
```

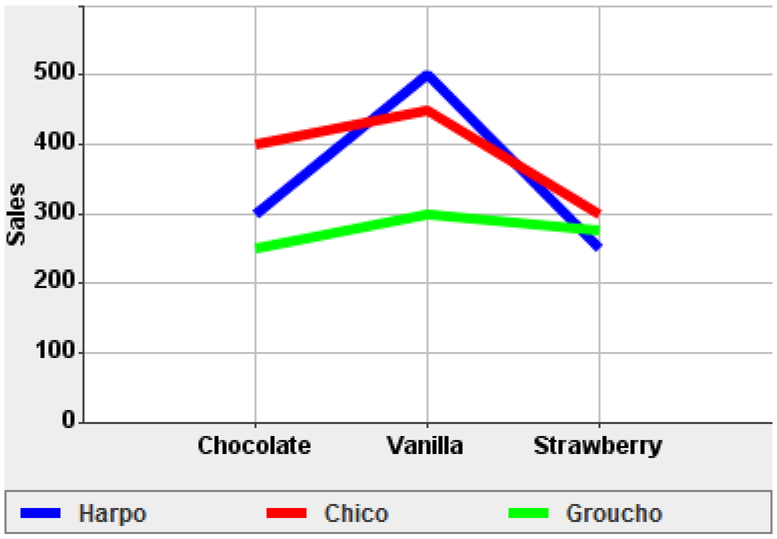
This creates the following chart:



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);
```

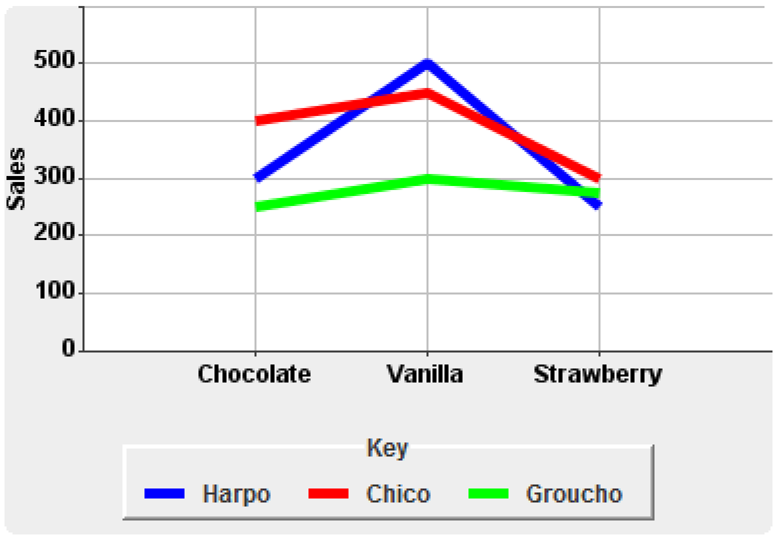
Then we can add the legend to the panel. In this case we add it to the south of the BorderLayout to create the following output:



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.

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

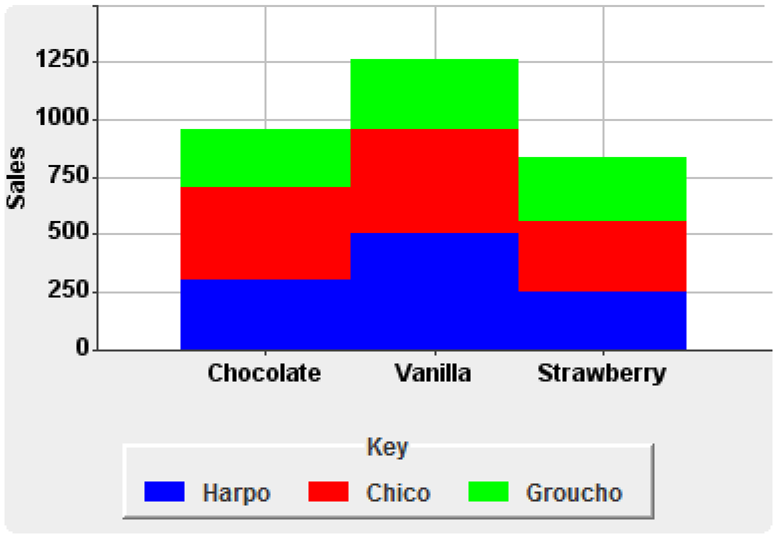
```
ChartStyle style1 = new ChartStyle(Color.blue, false, true, false);
```

we create a bar style with

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

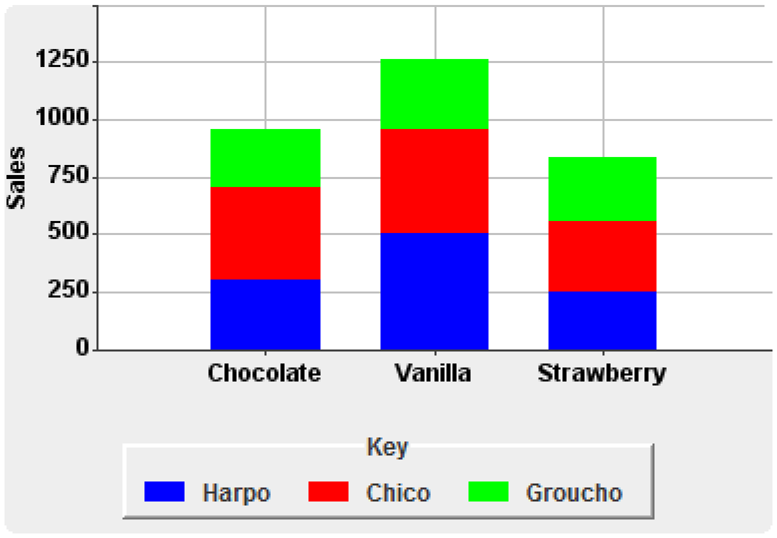
Stacked Bar Charts

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:

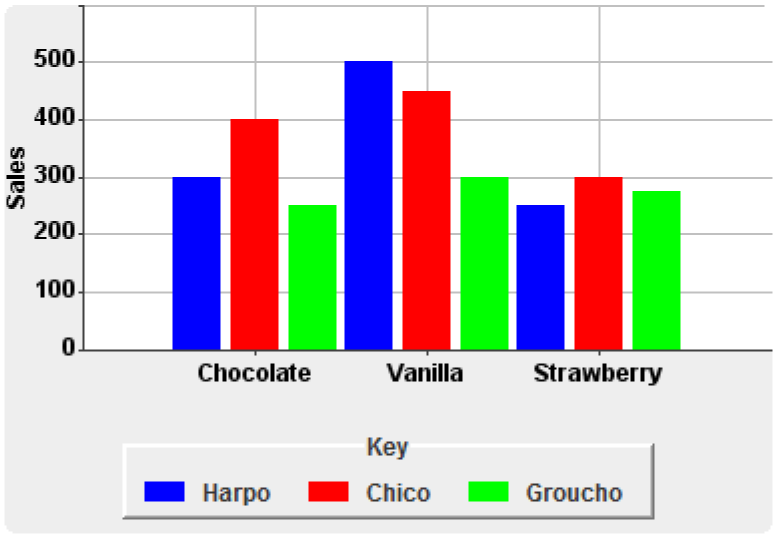


Grouped Bar Charts

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:

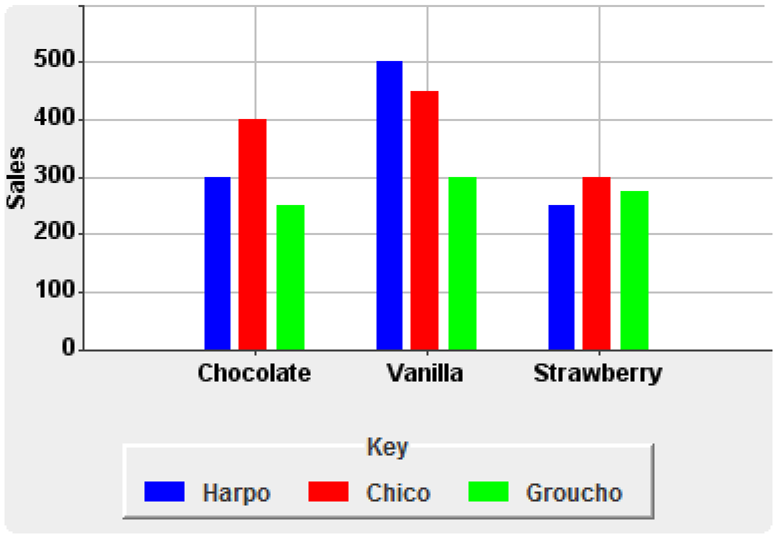


The bar gap is applied between all bars. To make the grouping into flavours a little more obvious you can also set a gap between the groups of bars.

For example,

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

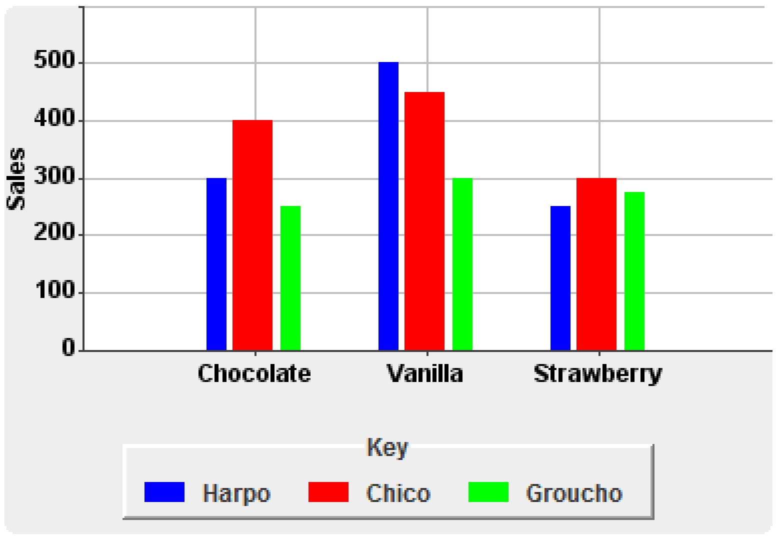
yields:



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:

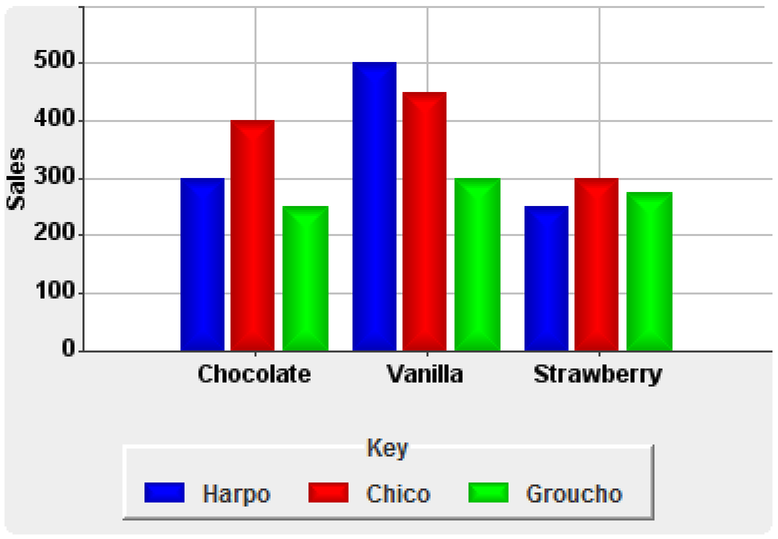


Changing the Renderer

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());
```

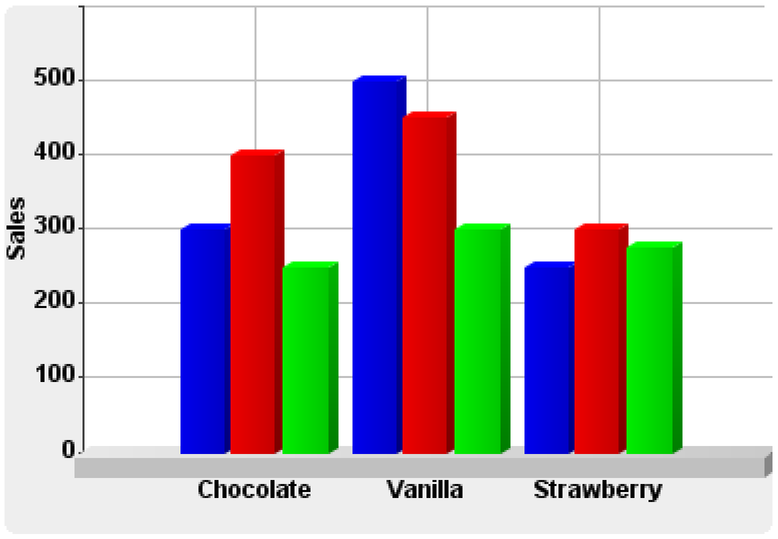
This gives the following:



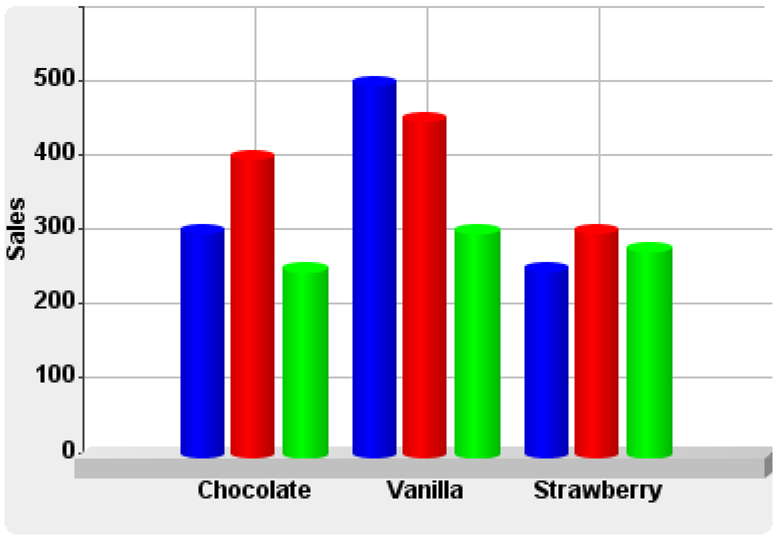
If we use the Bar3DRenderer or the CylinderRenderer, we should change the appearance of the x axis to give a true 3D effect.

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

Here is the Bar3DRenderer:

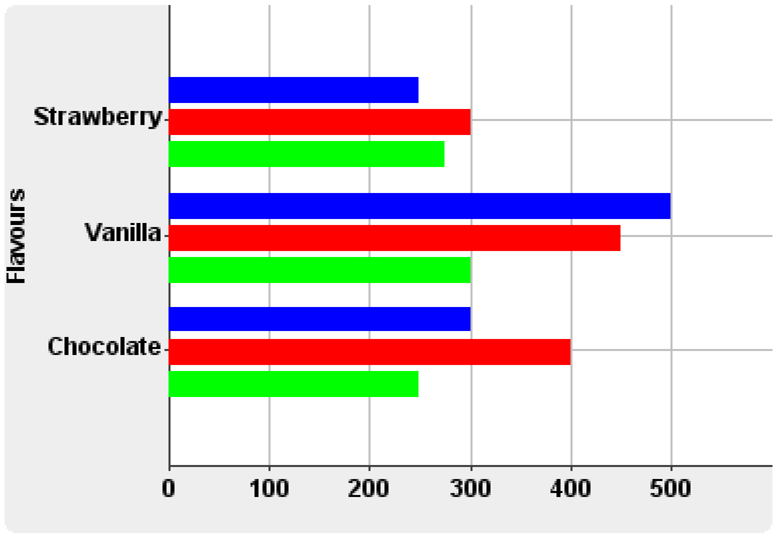


And here is the CylinderBarRenderer:



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)`.

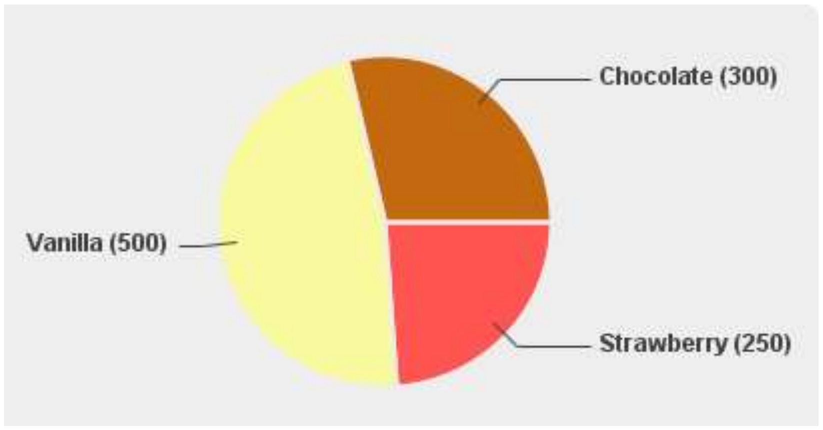
This approach makes it very easy to switch between bar chart and pie chart representations of the same data.

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'. A highlight is a semantic tag that can be attached to an individual data point and used to change its styling. (In general it could be used for other purposes too, but for the moment it is used only for styling.) Each segment of a pie chart corresponds to a single data point, so we can change the colouring of the segment by setting the highlight and associating it with a PointStyle.

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:



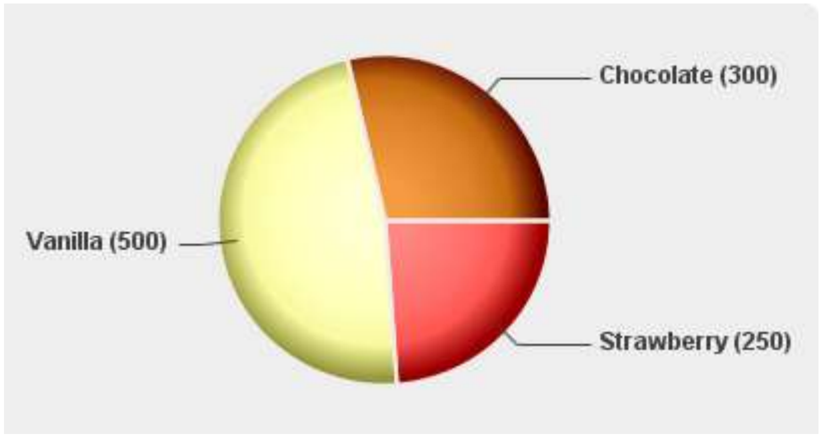
The reason we separate the highlight from its styling is so that we can change the appearance easily if the same highlight is used by many points, which is common for XY charts. There is no need to iterate through the points and change the colour of some of them – simply change the associated style on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

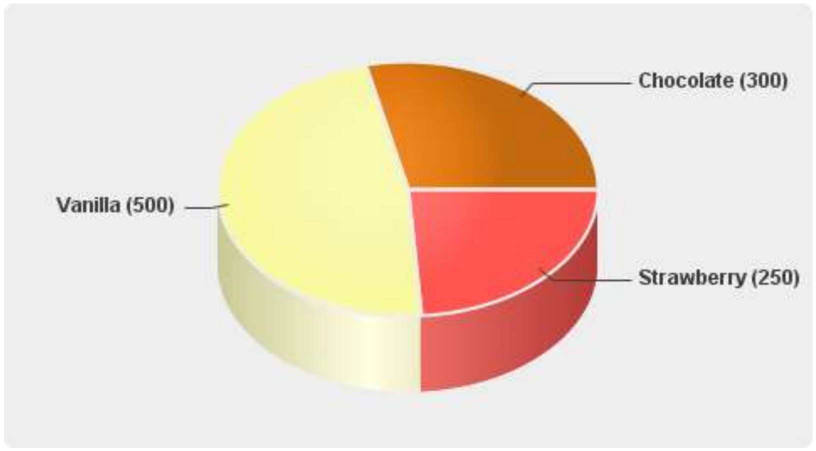
we get the following:



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).