RogueWave SOFTWARE

Accelerating Great Code

JMSL™ CHART PROGRAMMER'S GUIDE

Version 7.3

Contents

Preface

What's in this Manual

The JMSL Numerical Library Chart Programmer's Guide is divided into Part 1: Chart 2D, and Part 2: Chart 3D, followed by some appendices.

Part 1: Chart 2D

Introduction
: Provides an overview of the scope of the most commonly used 2D charting classes.

Charting 2D Types
: Describes the JMSL 2D Charts.

Quality Control Charts
: Describes Quality Control and Improvement Charts.

2D Drawing Elements
: Discusses the JMSL 2D drawing elements.

XML
: Explains JMSL XML features.

Actions
: Discusses actions available in 2D charting.

Part 2: Chart 3D

Introduction
: Overview of 3D charting capabilities

Charting 3D Types
: Describes the JMSL 3D Charts. Charting 3D Types

3D Drawing Elements
: Discusses the JMSL 3D drawing elements.

Part 3: Appendices

Web Server Application
: Describes JMSL and web servlets.

Writing a Chart as a Bitmap Image File
: Describes the ways to save bitmap images.

Picture-in-Picture

Describes the picture-in-picture effects.

3D: Drawing to the Canvas
Describes how to use JMSL to draw to a canvas.

Java 3D
Describes the JMSL Chart 3D package.

Contacting Rogue Wave Software Support

Users within support warranty may contact Rogue Wave Software regarding the use of the JMSL Numerical Library. IMSL Support can consult on the following topics:

- Clarity of documentation
- Possible IMSL-related programming problems
- Choice of IMSL Libraries functions or procedures for a particular problem

Not included in these topics are mathematical/statistical consulting and debugging of your program.

Refer to the following for IMSL Product Support contact information:

`http://www.roguewave.com/support/contact-support.aspx`

The following describes the procedure for consultation with Rogue Wave Software:

1. Include your IMSL license number
2. Include the product name and version number: JMSL Numerical Library Version 7.0
3. Include compiler and operating system version numbers
4. Include the name of the routine for which assistance is needed and a description of the problem

PART I

2D Charting

Chapter 1 Introduction to 2D

Charting

Chart 2D

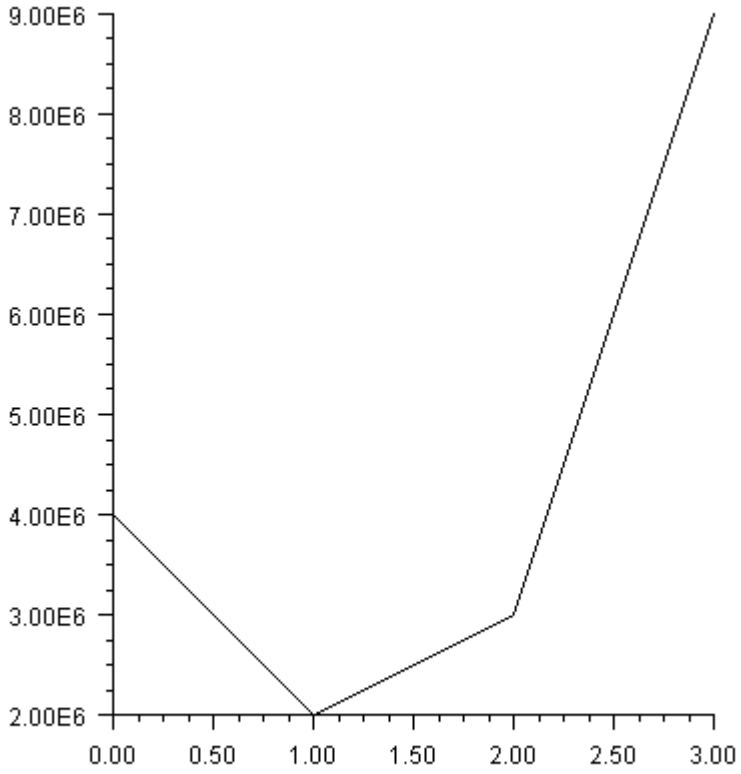
The *JMSL Numerical Library Chart Programmer's Guide* is an overview and discussion of the JMSL charting package. It is intended to be used in conjunction with the online Javadoc-generated ***Reference Manual***. This guide contains links into the Reference Manual, as appropriate. References to standard Java classes link to the "Java 2 Platform API Specification" reference manual on Oracle's web site.

This guide is both an introductory tutorial on the charting package and a "cookbook" for creating common chart types. The Reference Manual contains the complete details of the classes.

Overview

The JMSL chart package is designed to allow the creation of highly customizable charts using Java. A JMSL chart is created by assembling ***ChartNode***`s` into a tree. This chart tree is then rendered to the screen or printer.

The following class is a simple example of the use of the JMSL chart package. The chart is displayed in a `Swing` ***JFrame***. The code to create the chart is all in the constructor. The JMSL class ***JFrameChart*** extends the `Swing` class `JFrame` and creates a ***Chart*** object.

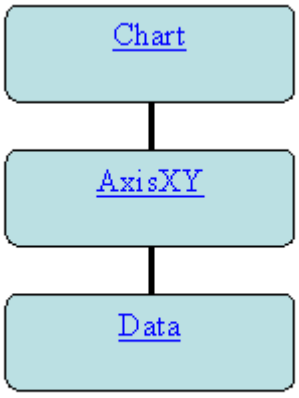


View code file

```
import com.imsl.chart.*;

public class Intro1 extends JFrameChart {

    public Intro1() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        double y[] = {4, 2, 3, 9};
        new Data(axis, y);
```

```
    }

    public static void main(String argv[]) {
        new Intro1().setVisible(true);
    }
}
```

The above example created and assembled three nodes into the following tree. The root `Chart` node is created by the `JFrameChart` class.



The general pattern of the ***ChartNode*** class, and classes derived from it, is to have constructors whose first argument is the parent `ChartNode` of the `ChartNode` being created.

The root node of the tree is always a ***Chart*** object. It is usually constructed within ***JFrameChart*** or ***JPanelChart***, which handles the repainting of the chart within `Swing`. If a `Chart` object is explicitly created, and used within a `Swing` GUI, then its `paint(Graphics)` method must be called from the container's `paint(Graphics)` or `paintComponent(Graphics)` method.

`Chart` nodes can contain attributes, such as `FillColor` and `LineWidth`. Attributes are inherited via the chart tree. For example, when a ***Data*** node is being painted, its `LineWidth` attribute determines the thickness of the line. If this attribute is set in the data node being drawn, that is the value used. If it is not set, then its parent node (an ***AxisXY*** node in the above example) is queried. If it is not set there, then ***its*** parent is queried. This continues until the root node is reached after which a default value is used. Note that this inheritance is not the same as Java class inheritance.

Attributes are set using setter methods, such as `setLineWidth(double)` and retrieved using getter methods, such as `getLineWidth()`.

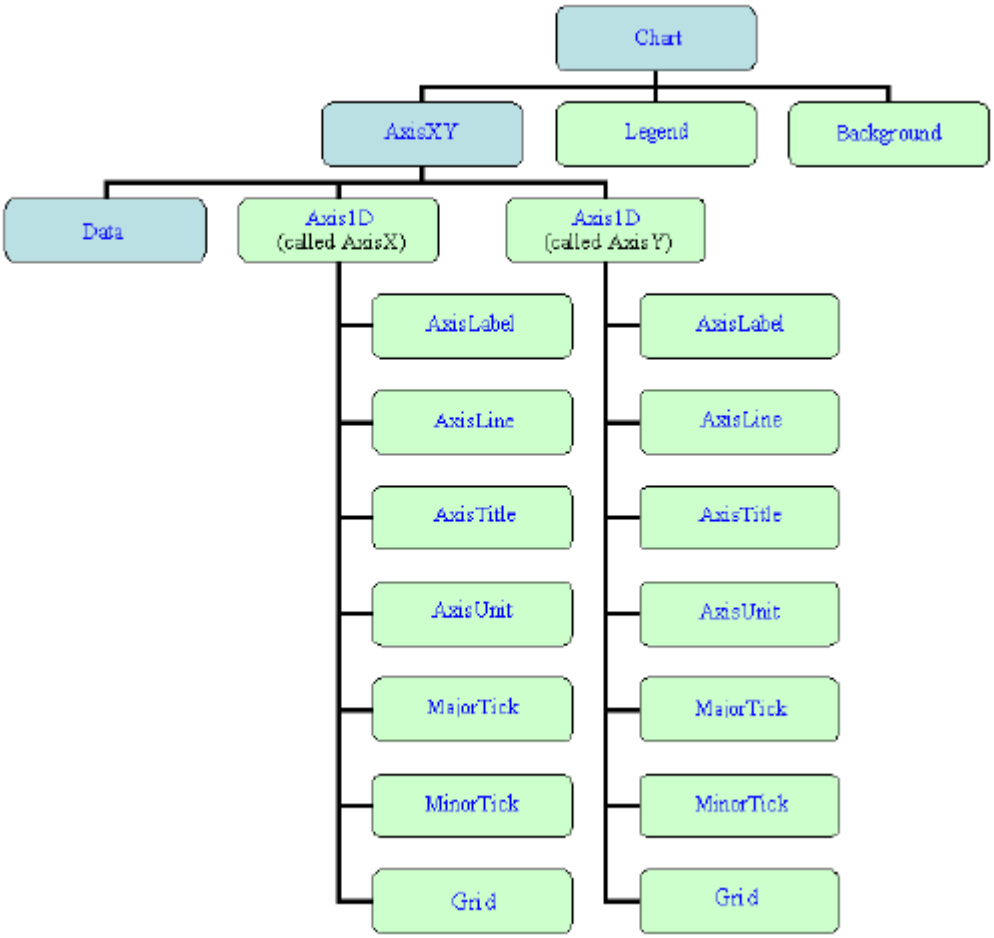
Implicitly Created Nodes

Some chart nodes automatically create additional nodes, as their children, when they are created. These nodes can be accessed via `get` methods. For example, the code

```
chart.getBackground().setFillColor(java.awt.Color.green);
```

changes the background color. The method `getBackground` retrieves the ***Background*** node from the chart object and the method `setFillColor` sets the `Background` object's `FillColor` attribute to `Color.green`.

In the following diagram, the nodes automatically created are shown in light green (to view in color see the online documentation).

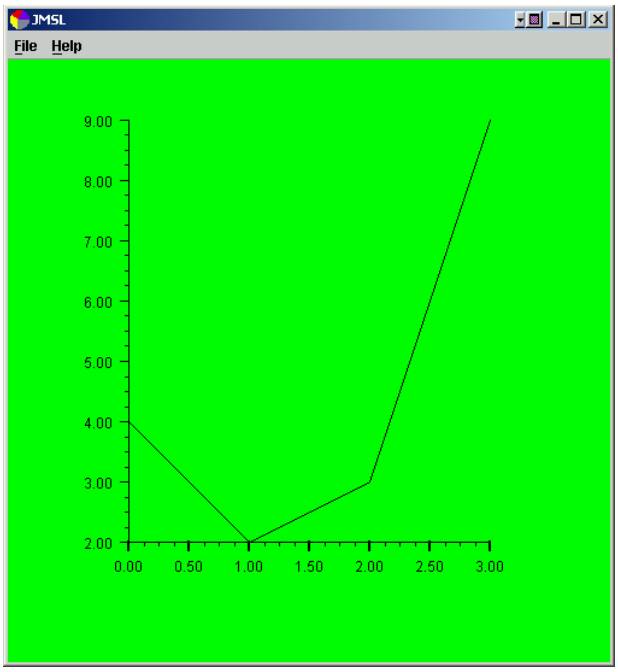


Method calls can be chained together. For example, the following sets the thickness of the major tick marks on the *x*-axis.

```
axis.getAxisX().getMajorTick().setLineWidth(2.0);
```

where `axis` is an ***AxisXY*** object.

A customized version of the above chart can be obtained by changing its constructor as in the following.



View code file

```
import com.imsl.chart.*;

public class Intro2 extends JFrameChart {

    public Intro2() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        double y[] = {4, 2, 3, 9};
        new Data(axis, y);
        chart.getBackground().setFillColor(java.awt.Color.green);
        axis.getAxisX().getMajorTick().setLineWidth(2.0);
    }

   public static void main(String argv[]) {
        new Intro2().setVisible(true);
    }
}
```

Adding a Chart to an Application

For simplicity, most of the examples in this manual use the ***JFrameChart*** class. JFrameChart is useful for quickly building an application that is a chart. The class ***JPanelChart*** is used to build a chart into a larger application. It extends Swing's ***JPanel*** class and can be used wherever a JPanel can be used. The following code shows a JPanel-Chart being created, added to a ***JFrame***, and having a chart tree added to the JPanelChart. (The generated chart is very similar to that shown at the beginning of this chapter.) The generated chart is the same as the first one on this page and so is not shown here.

View code file

```
import com.imsl.chart.*;

public class SampleJPanel extends javax.swing.JFrame {
    private JPanelChart jPanelChart;

    public SampleJPanel() {
        this.setSize(500, 500);
        jPanelChart = new JPanelChart();
        getContentPane().add(jPanelChart);

        Chart chart = jPanelChart.getChart();
        AxisXY axis = new AxisXY(chart);
        double y[] = {4, 2, 3, 9};
        new Data(axis, y);
    }

    public static void main(String argv[]) {
        new SampleJPanel().setVisible(true);
    }
}
```

Chapter 2 Charting 2D Types

Chart Types

JMSL provides these charting types:

Scatter Plot

This section describes the construction of scatter plots. The markers can be formatted using the Marker Attributes.

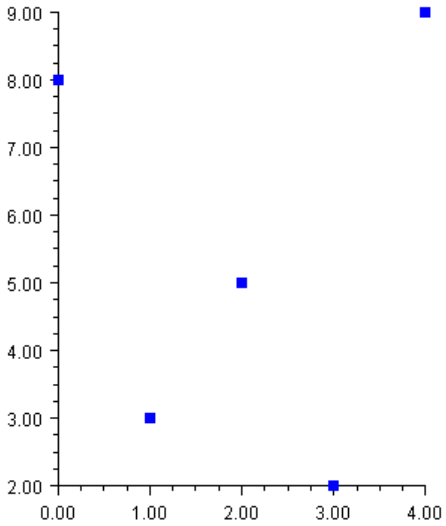
It is also possible to mix lines and markers (see the section Mixed Line and Marker Plot).

Simple Scatter Plot

The ***JFrameChart*** class is used to create a frame containing a ***Chart*** node. The Chart node is the root of the tree to which an ***AxisXY*** node is added. A ***Data*** node is then created as the child of the axis node. The Data node is created using an array of *y*-values. The *x*-values default to 0, 1, ...

The `DataType` attribute is set to `DATA_TYPE_MARKER` to make this a scatter plot.

The look of the markers is controlled by the marker attributes. In this example the `MarkerType` attribute is set to `MARKER_TYPE_FILLED_SQUARE`. The `MarkerColor` attribute is set to blue.



View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SampleSimpleScatter extends JFrameChart {

    public SampleSimpleScatter() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);

        double y[] = {8, 3, 5, 2, 9};
        Data data1 = new Data(axis, y);
```

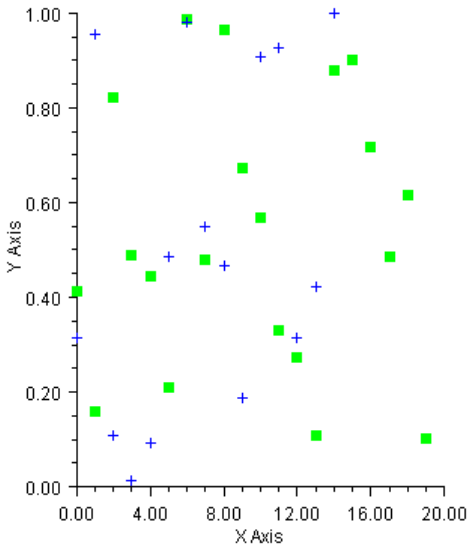
```
        data1.setDataType(Data.DATA_TYPE_MARKER);
        data1.setMarkerType(Data.MARKER_TYPE_FILLED_SQUARE);
        data1.setMarkerColor(Color.blue);
    }

    public static void main(String argv[]) {
        new SampleSimpleScatter().setVisible(true);
    }
}
```

Complex Scatter Plot

This example shows a scatter plot with two data sets. The `Data` nodes are created using random *y*-values generated by ***Random***. The *x*-values default to 0, 1, ...

The axes are labeled by setting the `AxisTitle` attribute for the *x*- and *y*-axes.

The `DataType` attribute is set in the axis node. The axis node does not itself use this attribute, but from there it is inherited by the child `Data` nodes.



View code file

```
import com.imsl.chart.*;
import java.awt.Color;
import com.imsl.stat.Random;

public class SampleScatter extends JFrameChart {

    public SampleScatter() {
        Random r = new Random(123457);

        Chart chart = getChart();
```

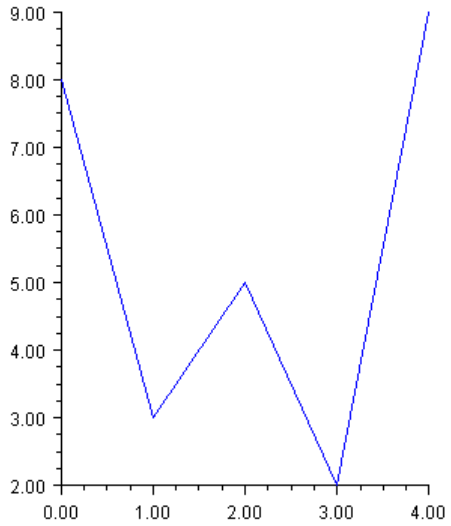
```
        AxisXY axis = new AxisXY(chart);
        axis.getAxisX().getAxisTitle().setTitle("X Axis");
        axis.getAxisY().getAxisTitle().setTitle("Y Axis");
        axis.setDataType(Data.DATA_TYPE_MARKER);

        // Data set 1
        double y1[] = new double[20];
        for (int k = 0;  k < y1.length;  k++)  y1[k] = r.nextDouble();
        Data data1 = new Data(axis, y1);
        data1.setDataType(Data.DATA_TYPE_MARKER);
        data1.setMarkerType(Data.MARKER_TYPE_FILLED_SQUARE);
        data1.setMarkerColor(Color.green);

        // Data set 2
        double y2[] = new double[15];
        for (int k = 0;  k < y2.length;  k++)  y2[k] = r.nextDouble();
        Data data2 = new Data(axis, y2);
        data2.setMarkerType(Data.MARKER_TYPE_PLUS);
        data2.setMarkerColor(Color.blue);
    }

    public static void main(String argv[]) {
        new SampleScatter().setVisible(true);
    }
}
```

Line Plot

A line plot consists of points connected by lines. The lines can be formatted using the Line Attributes.

Simple Line Plot

This example shows a simple line plot. The ***Data*** node is created using an array of *y*-values. The *x*-values default to 0, 1, ... The `DataType` attribute is set to `DATA_TYPE_LINE` to make this a line chart. The look of the line is controlled by the line attributes. Here the `LineColor` attribute is set to blue.

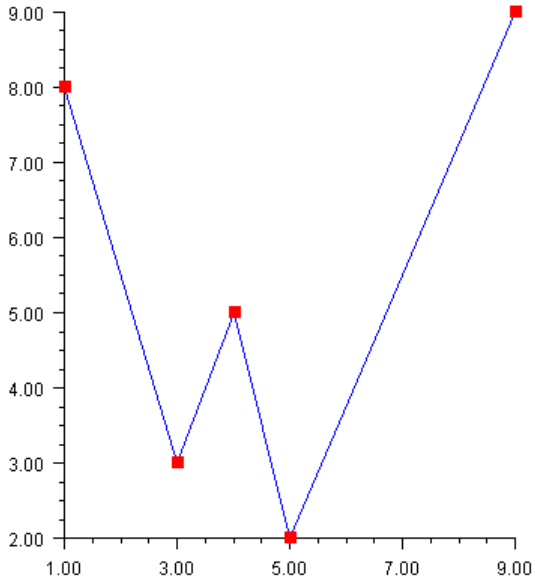


View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SampleSimpleLine extends JFrameChart {

    public SampleSimpleLine() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);

        double y[] = {8, 3, 5, 2, 9};
        Data data1 = new Data(axis, y);
        data1.setDataType(Data.DATA_TYPE_LINE);
        data1.setLineColor(Color.blue);
    }

    public static void main(String argv[]) {
        new SampleSimpleLine().setVisible(true);
    }
}
```

Mixed Line and Marker Plot

The `DataType` attribute can be set using "or" syntax to combine types. In this example, it is set to `DATA_TYPE_LINE | DATA_TYPE_MARKER`. This example also explicitly sets both the *x*-value and the *y*-value of the data points. Note that the *x*-values do not have to be uniformly spaced.



View code file

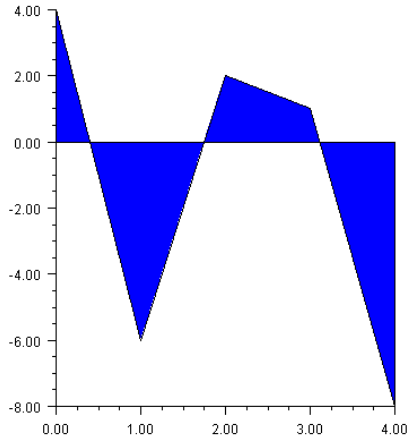
```
import com.imsl.chart.*;

public class SampleLineMarker extends JFrameChart {

    public SampleLineMarker() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);

        double x[] = {1, 3, 4, 5, 9};
        double y[] = {8, 3, 5, 2, 9};
        Data data = new Data(axis, x, y);
        data.setDataType(Data.DATA_TYPE_LINE | Data.DATA_TYPE_MARKER);
        data.setLineColor(java.awt.Color.blue);
        data.setMarkerColor(java.awt.Color.red);
        data.setMarkerType(Data.MARKER_TYPE_FILLED_SQUARE);
    }

    public static void main(String argv[]) {
        new SampleLineMarker().setVisible(true);
    }
}
```

Area Plot

Area plots are similar to line plots, but with the area between the line and a reference line filled in. An area plot is created if the `DATA_TYPE_FILL` bit is on in the value of the DataType attribute. The default reference line is $y=0$. The location of the reference line can be changed from 0 by using `setReference(double)`. The Fill Area Attributes determine how the area is filled.

Simple Area Plot

This example draws a simple area plot. The default value of the `FillType` attribute is `FILL_TYPE_SOLID`. The example sets the `FillColor` attribute to blue. So the area between the line and $y = 0$ is solid blue.



View code file

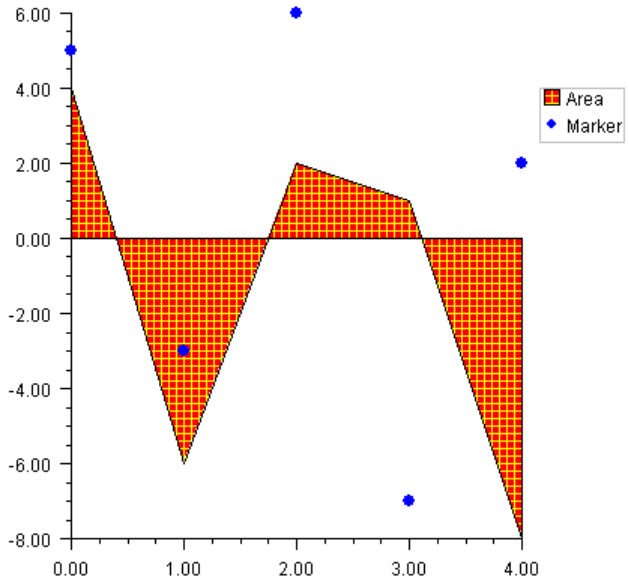
```
import com.imsl.chart.*;
import java.awt.Color;

public class SampleArea extends JFrameChart {

    public SampleArea() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);

        double y[] = {4, -6, 2, 1, -8};
        Data data = new Data(axis, y);
        data.setDataType(Data.DATA_TYPE_FILL);
        data.setFillColor(Color.blue);
    }

    public static void main(String argv[]) {
        new SampleArea().setVisible(true);
    }
}
```

Painted Area Example

This example shows an area chart filled in with a painted texture. The texture is created by a static method `FillPaint.crosshatch`.

A second data set is plotted as a set of markers.

The ***Legend*** node is painted in this example and has entries for both the filled area data set and the marker data set.



View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SampleAreaPaint extends JFrameChart {

    public SampleAreaPaint() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        chart.getLegend().setPaint(true);

        double y1[] = {4, -6, 2, 1, -8};
        Data data1 = new Data(axis, y1);
        data1.setTitle("Area");
        data1.setDataType(Data.DATA_TYPE_FILL);
        data1.setFillType(Data.FILL_TYPE_PAINT);
        data1.setFillPaint(FillPaint.crosshatch
        (10,5,Color.red,Color.yellow));

        double y2[] = {5, -3, 6, -7, 2};
        Data data2 = new Data(axis, y2);
```

```
        data2.setTitle("Marker");
        data2.setDataType(Data.DATA_TYPE_MARKER);
        data2.setMarkerColor(Color.blue);
        data2.setMarkerType(Data.MARKER_TYPE_FILLED_CIRCLE);
    }

    public static void main(String argv[]) {
        new SampleAreaPaint().setVisible(true);
    }
}
```

Attribute Reference

The attribute `Reference` defines the reference line. If its value is a, then the reference line is $y = a$. Its default value is 0.

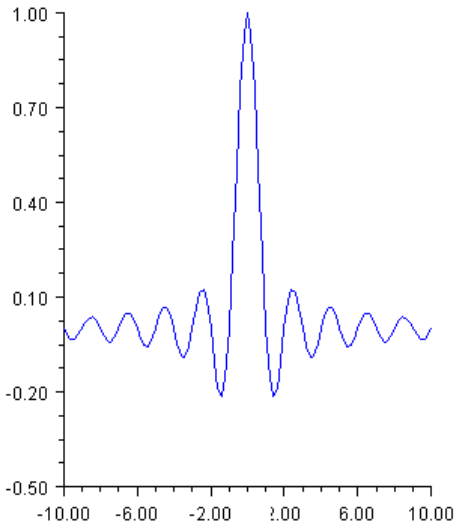
Function Plot

A function plot shows the value of a function *f*(*x*) over an interval [*a*,*b*]. The function must be defined as an implementation of the ***ChartFunction*** interface. A ***Data*** node constructor creates a line chart from the function. The look of the function is controlled by the line attributes.

The `ChartFunction` interface requires that the function name be "`f`", that the function have a single `double` argument and that it return a `double`.

Example

This example plots the sinc function on the interval [-10,10]. The sinc function is defined to be sin(*x*)/*x*. In this example, sinc is an anonymous inner class that implements `ChartFunction`. This is required by the function `Data` constructor. In the code, the case *x* =0 is handled specially to avoid returning `NaN`.



View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SampleFunction extends JFrameChart {

    public SampleFunction() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);

        ChartFunction sinc = new ChartFunction() {
            public double f(double x) {
                if (x == 0.0) return 1.0;
                return Math.sin(Math.PI*x) / (Math.PI*x);
```

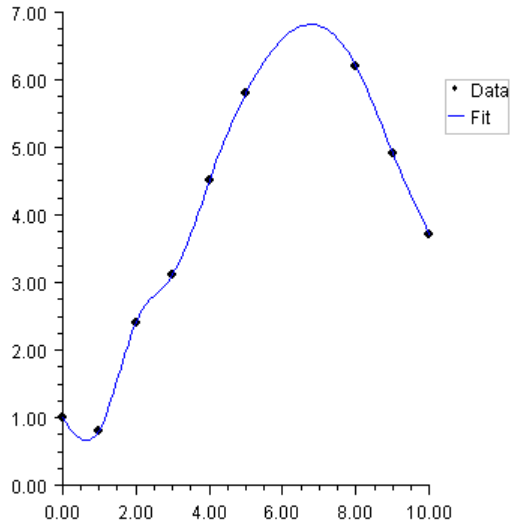
```
            }
        };
        Data data = new Data(axis, sinc, -10.0, 10.0);
        data.setLineColor(Color.blue);
    }

    public static void main(String argv[]) {
        new SampleFunction().setVisible(true);
    }
}
```

Histogram Example

For another example of a ***ChartFunction***, see the section Histogram.

Spline Chart

This example shows raw data points, as markers, and their fit to a shape preserving spline. The spline is computed using ***CsShape*** found in the JMSL Math package (which extends ***Spline***). The ***ChartSpline*** class wraps the `Spline` into a `ChartFunction`. This example also enables the ***Legend***.



View code file

```
import com.imsl.chart.*;
import com.imsl.math.CsShape;
import java.awt.Color;

public class SampleSpline extends JFrameChart {

    public SampleSpline() {
        try {
            Chart chart = getChart();
```

```
            AxisXY axis = new AxisXY(chart);
            chart.getLegend().setPaint(true);

            double x[] = {0, 1, 2, 3, 4, 5, 8, 9, 10};
            double y[] = {1.0, 0.8, 2.4, 3.1, 4.5, 5.8, 6.2, 4.9, 3.7};
            Data dataMarker = new Data(axis, x, y);
            dataMarker.setTitle("Data");
            dataMarker.setDataType(Data.DATA_TYPE_MARKER);
            dataMarker.setMarkerType(Data.MARKER_TYPE_FILLED_CIRCLE);

            CsShape spline = new CsShape(x, y);
            Data dataSpline =
                    new Data(axis, new ChartSpline(spline), 0., 10.);
            dataSpline.setTitle("Fit");
            dataSpline.setLineColor(Color.blue);
        } catch (com.imsl.IMSLException e) {
            e.printStackTrace();
        }
    }

    public static void main(String argv[]) {
        new SampleSpline().setVisible(true);
    }
}
```

Log and SemiLog Plot

In a semilog plot the *y*-axis is logarithmically scaled, while the *x*-axis is linearly scaled. In a log-log plot both axes are logarithmically scaled.

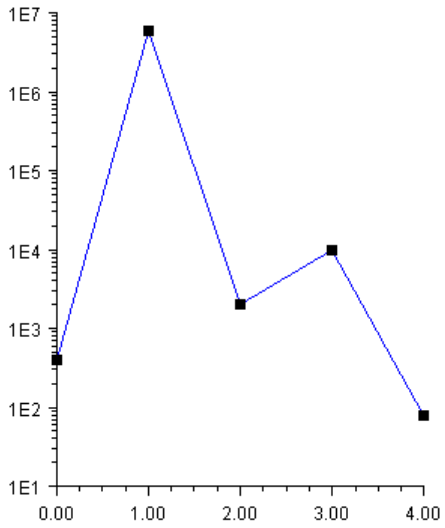
SemiLog Plot

In this example, data is plotted as lines and markers on a semilog axis.

To set up the *y*-axis as a logarithmic axis:

- the `Transform` attribute is set to `TRANSFORM_LOG`.
- the `Density` attribute is set to 9. Density is the number of minor ticks between major ticks. It is 9 because base 10 is used and the major tick marks are not counted.

The `TextFormat`, used by ***AxisLabel***, is changed to use scientific notation (see Text Attributes). The default decimal format would result in large numbers written with many zeros.



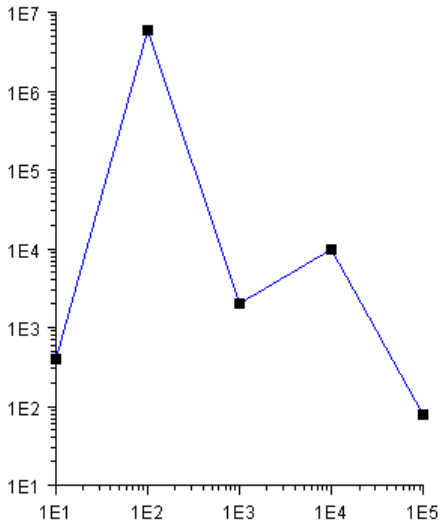
View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SampleSemiLog extends JFrameChart {

    public SampleSemiLog() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        axis.getAxisY().setTransform(axis.TRANSFORM_LOG);
        axis.getAxisY().setDensity(9);
```

```
        axis.getAxisY().setTextFormat(new java.text.DecimalFormat("0.E0"));

        double y[] = {4e2, 6e6, 2e3, 1e4, 8e1};
        Data data = new Data(axis, y);
        data.setDataType(Data.DATA_TYPE_LINE | Data.DATA_TYPE_MARKER);
        data.setLineColor(Color.blue);
        data.setMarkerType(Data.MARKER_TYPE_FILLED_SQUARE);
    }

    public static void main(String argv[]) {
        new SampleSemiLog().setVisible(true);
    }
}
```

Log-Log Plot

A log-log plot is set up in much the same way as a semilog plot, except that the changes must be made to both the *x*-axis and the *y*-axis. In this example, the changes are made to the axis node and inherited by both the *x* and *y*-axis nodes.



View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SampleLogLog extends JFrameChart {

    public SampleLogLog() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        axis.setTransform(axis.TRANSFORM_LOG);
        axis.setDensity(9);
        axis.setTextFormat(new java.text.DecimalFormat("0.E0"));
```

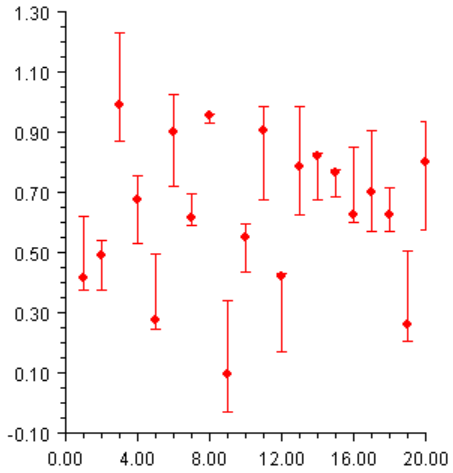
```
        double x[] = {1e1, 1e2, 1e3, 1e4, 1e5};
        double y[] = {4e2, 6e6, 2e3, 1e4, 8e1};
        Data data = new Data(axis, x, y);
        data.setDataType(Data.DATA_TYPE_LINE | Data.DATA_TYPE_MARKER);
        data.setLineColor(Color.blue);
        data.setMarkerType(Data.MARKER_TYPE_FILLED_SQUARE);
    }

    public static void main(String argv[]) {
        new SampleLogLog().setVisible(true);
    }
}
```

Error Bar Plot

Error bars are used to indicate the estimated error in a measurement. Errors bars indicate the uncertainty in the *x* and/or *y*-values.

Vertical Error Bars

The most common error bar plot is one in which the errors are in the *y*-values. This example shows such an error bar plot. Note that the values of the low and high arguments are absolute *y*-values, not relative or percentage values.



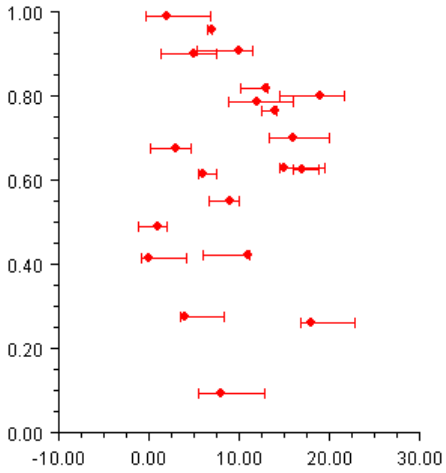
View code file

```
import com.imsl.chart.*;
import java.awt.Color;
import com.imsl.stat.Random;

public class SampleErrorBar extends JFrameChart {

    public SampleErrorBar() {
        Random r = new Random(123457);

        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);

        // Generate a random data set, with random errors
        int n = 20;
        double x[] = new double[n];
        double y[] = new double[n];
        double low[] = new double[n];
        double high[] = new double[n];
        for (int k = 0;  k < n;  k++) {
            x[k] = k + 1;
```

```
            y[k] = r.nextDouble();
            low[k] = y[k] - 0.25*r.nextDouble();
            high[k] = y[k] + 0.25*r.nextDouble();
        }

        ErrorBar data = new ErrorBar(axis, x, y, low, high);
        data.setDataType(data.DATA_TYPE_ERROR_Y | data.DATA_TYPE_MARKER);
        data.setMarkerType(Data.MARKER_TYPE_FILLED_CIRCLE);
        data.setMarkerColor(Color.red);
    }

    public static void main(String argv[]) {
        new SampleErrorBar().setVisible(true);
    }
}
```

Horizontal Error Bars

It is also possible to have horizontal error bars, indicating errors in *x*, as shown in this example.



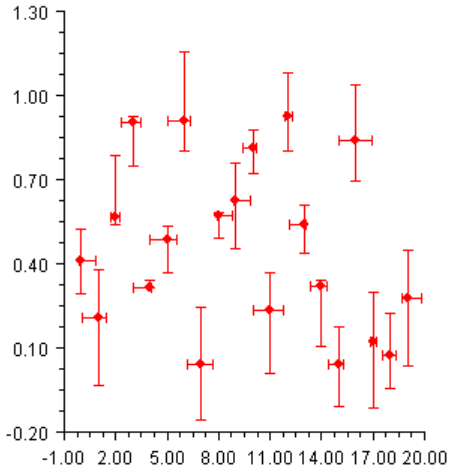
View code file

```
import com.imsl.chart.*;
import java.awt.Color;
import com.imsl.stat.Random;

public class SampleHorizontalErrorBar extends JFrameChart {

    public SampleHorizontalErrorBar() {
        Random r = new Random(123457);

        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
```

```
        // Generate a random data set, with random errors in x
        int n = 20;
        double x[] = new double[n];
        double y[] = new double[n];
        double low[] = new double[n];
        double high[] = new double[n];
        for (int k = 0;  k < n;  k++) {
            x[k] = k;
            y[k] = r.nextDouble();
            low[k] = x[k] - 5.*r.nextDouble();
            high[k] = x[k] + 5.*r.nextDouble();
        }

        ErrorBar data = new ErrorBar(axis, x, y, low, high);
        data.setDataType(
                ErrorBar.DATA_TYPE_ERROR_X | ChartNode.DATA_TYPE_MARKER);
        data.setMarkerType(Data.MARKER_TYPE_FILLED_CIRCLE);
        data.setMarkerColor(Color.red);
    }

    public static void main(String argv[]) {
        new SampleHorizontalErrorBar().setVisible(true);
    }
}
```

Mixed Error Bars

To show errors in both *x* and *y*, it is necessary to create both vertical and horizontal error bar objects. This example shows such a chart.



View code file

```
import com.imsl.chart.*;
import java.awt.Color;
```

```
import com.imsl.stat.Random;

public class SampleMixedErrorBar extends JFrameChart {

    public SampleMixedErrorBar() {
        Random r = new Random(123457);

        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);

        // Generate a random data set, with random errors in x
        int n = 20;
        double x[] = new double[n];
        double y[] = new double[n];
        double xlow[] = new double[n];
        double xhigh[] = new double[n];
        double ylow[] = new double[n];
        double yhigh[] = new double[n];
        for (int k = 0;  k < n;  k++) {
            x[k] = k;
            y[k] = r.nextDouble();
            xlow[k] = x[k] - r.nextDouble();
            xhigh[k] = x[k] + r.nextDouble();
            ylow[k] = y[k] - 0.25*r.nextDouble();
            yhigh[k] = y[k] + 0.25*r.nextDouble();
        }

        ErrorBar dataY = new ErrorBar(axis, x, y, ylow, yhigh);
        dataY.setDataType(
                ErrorBar.DATA_TYPE_ERROR_Y | ChartNode.DATA_TYPE_MARKER);
        dataY.setMarkerType(Data.MARKER_TYPE_FILLED_CIRCLE);
        dataY.setMarkerColor(Color.red);

        ErrorBar dataX = new ErrorBar(axis, x, y, xlow, xhigh);
        dataX.setDataType(ErrorBar.DATA_TYPE_ERROR_X);
        dataX.setMarkerColor(Color.red);
    }

    public static void main(String argv[]) {
        new SampleMixedErrorBar().setVisible(true);
    }
}
```

High-Low-Close Plot

`High-Low-Close` plots are used to show stock prices. They are created using the ***HighLowClose*** class.

The markers can be formatted using the attribute `MarkerColor` (`setMarkerColor`)

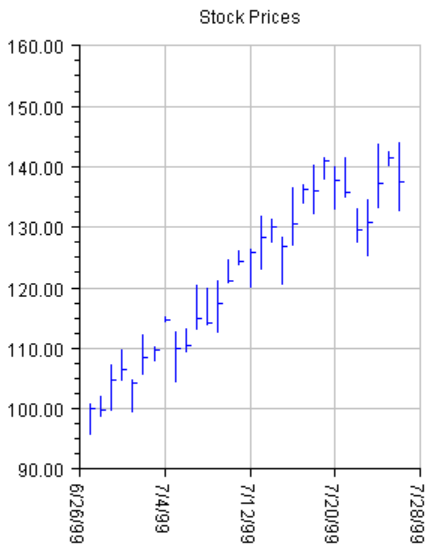
In a high-low-close plot the vertical line represents the high and low values. The close value is represented by a "tick" to the right. The open value, if present, is represented by a tick to the left.

The `setDateAxis` method will configure the *x*-axis for dates. This turns off autoscaling of the axis.

In a high-low-close chart the *x*-values are the number of milliseconds since the epoch (January 1, 1970). This is the standard Java convention for representing dates as numbers.

Example

In this example, random security prices are computed in the `createData` method. The time axis is prepared by calling `setDateAxis`.



View code file

```
import com.imsl.chart.*;
import java.text.DateFormat;
import java.util.*;

public class SampleHighLowClose extends JFrameChart {
    private double high[], low[], close[];

    public SampleHighLowClose() {
        Chart chart = getChart();
```

```
    AxisXY axis = new AxisXY(chart);

    // Date is June 27, 1999
    Date date = new GregorianCalendar(1999,
    GregorianCalendar.JUNE, 27).getTime();

    int n = 30;
    createData(n);

    // Create an instance of a HighLowClose Chart
    HighLowClose hilo = new HighLowClose(axis, date, high, low, close);
    hilo.setMarkerColor(java.awt.Color.blue);

    // Set the HighLowClose Chart Title
    chart.getChartTitle().setTitle("Stock Prices");

    // Setup the x axis
    Axis1D axisX = axis.getAxisX();

    // Set the text angle for the X axis labels
    axisX.getAxisLabel().setTextAngle(270);
    // Change the text format to the local date format
    DateFormat df = DateFormat.getDateInstance(DateFormat.SHORT);
    axisX.getAxisLabel().setTextFormat(df);

    // Setup the time axis
    hilo.setDateAxis("Date(SHORT)");

    // Turn on grid and make it light gray
    axisX.getGrid().setPaint(true);
    axisX.getGrid().setLineColor(java.awt.Color.lightGray);
    axis.getAxisY().getGrid().setPaint(true);
    axis.getAxisY().getGrid().setLineColor(java.awt.Color.lightGray);
}


private void createData(int n) {
    high = new double[n];
    low = new double[n];
    close = new double[n];
    Random r = new Random(123457);
    for (int k = 0;  k < n;  k++) {
        double f = r.nextDouble();
        if (k == 0) {
            close[0] = 100;
        } else {
            close[k] = (0.95+0.10*f)*close[k-1];
        }
        high[k] = close[k]*(1.+0.05*r.nextDouble());
        low[k] = close[k]*(1.-0.05*r.nextDouble());
    }
```

```
    }

    public static void main(String argv[]) {
        new SampleHighLowClose().setVisible(true);
    }
}
```

Candlestick Chart

A ***Candlestick*** chart is used to show stock price. Each candlestick shows the stock's high, low, opening and closing prices.

The `Candlestick` constructors create two child ***CandlestickItem*** nodes. One is for the up days and one is for the down days. A day is an up day if the closing price is greater than the opening price. The `getUp` and `getDown` accessor methods can be used to retrieve these nodes.

The line ("whisker") part of a candlestick shows the stock's high and low prices. The whiskers can be formatted using the line attributes (see Line Attributes).

The body of a candlestick shows the stock's opening and closing prices. The body color is used to flag if the top of the body is the closing price (up day) or the opening price (down day). The fill area attributes determine how the body is drawn (see Fill Area Attributes). By default, up days are white and down days are black.

The width of a candlestick is controlled by the `MarkerSize` attribute (see Attribute MarkerSize).

Example

In this example, random security prices are computed in the `createData` method. The time axis is prepared by calling `setDateAxis`.

The up days are colored green and the down days are colored red.

View code file

```
import com.imsl.chart.*;
```

```
import java.util.*;

public class SampleCandlestick extends JFrameChart {

    private double high[],  low[],  open[],  close[];

    public SampleCandlestick() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);

        // Date is June 27, 1999
        Date date = new GregorianCalendar(1999,
                GregorianCalendar.JUNE, 27).getTime();

        int n = 30;
        createData(n);

        // Create an instance of a HighLowClose Chart
        Candlestick stick =
                new Candlestick(axis, date, high, low, close, open);

        // Show up days in green and down days in red
        stick.getUp().setFillColor(java.awt.Color.green);
        stick.getDown().setFillColor(java.awt.Color.red);

        // Set the HighLowClose Chart Title
        chart.getChartTitle().setTitle("Stock Prices");

        // Setup the x axis
        Axis1D axisX = axis.getAxisX();

        // Setup the time axis
        stick.setDateAxis("Date(SHORT)");

        // Turn on grid and make it light gray
        axisX.getGrid().setPaint(true);
        axisX.getGrid().setLineColor(java.awt.Color.lightGray);
        axis.getAxisY().getGrid().setPaint(true);
        axis.getAxisY().getGrid().setLineColor(java.awt.Color.lightGray);
    }

    private void createData(int n) {
        high = new double[n];
        low = new double[n];
        close = new double[n];
        open = new double[n];
        Random r = new Random(123457);
        for (int k = 0; k < n; k++) {
            double f = r.nextDouble();
            if (k == 0) {
                close[0] = 100;
```

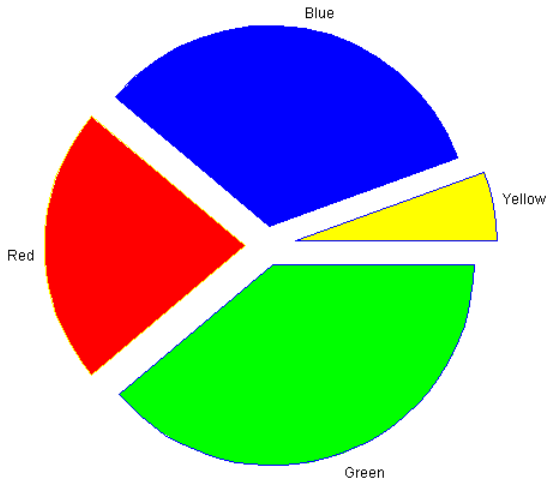
```
            } else {
                close[k] = (0.95 + 0.10 * f) * close[k - 1];
            }
            high[k] = close[k] * (1. + 0.05 * r.nextDouble());
            low[k] = close[k] * (1. - 0.05 * r.nextDouble());
            open[k] = low[k] + r.nextDouble() * (high[k] - low[k]);
        }
    }

    public static void main(String argv[]) {
        new SampleCandlestick().setVisible(true);
    }
}
```

Pie Chart

A pie chart is a graphical way to organize data. This section describes the construction of a pie chart.

Example

The ***JFrameChart*** class is used to create a frame containing a ***Chart*** node. A ***Pie*** node is then created as a child of the `Chart` node. The `Pie` node constructor creates `PieSlice` nodes as its children. The number of ***PieSlice*** nodes created is `y.length`, here equal to 4. After the `PieSlice` nodes are created they are retrieved from the object pie and customized by setting attributes.



The `LabelType` (`setLabelType`) attribute is set in the pie node. The pie node itself does not use this attribute, but from there it is inherited by all of the `PieSlice` nodes.

The `Title` attribute is set in each `PieSlice` node. This is the slice label. It is used to label the slice only if the slice's `LabelType` attribute is `LABEL_TYPE_TITLE`.

The `FillColor` (`setFillColor`) attribute is also set in each slice. This determines the color of the slice. Since the default value of `FillColor` is black, it is generally recommended that `FillColor` be set in each slice.

The `FillOutlineColor` (`setFillOutlineColor`) attribute sets the border color of each slice. In this example it is set in the pie node to be blue and set in the `slice[1]` node to be yellow. All except `slice[1]` are outlined in blue, and `slice[1]` is outlined in yellow.

The `Explode` (`setExplode`) attribute moves a pie slice out from the center. Its default value is 0, which puts the slice vertex in the center. A value of 1.0 would put the slice vertex on the circumference.

`View code file`

```
import com.imsl.chart.*;
import java.awt.Color;

public class SamplePieChart extends JFrameChart {

    public SamplePieChart() {
        Chart chart = getChart();

        double y[] = {35., 20., 30., 5.};
        Pie pie = new Pie(chart, y);
        pie.setLabelType(Pie.LABEL_TYPE_TITLE);
        pie.setFillOutlineColor(Color.blue);
        pie.setViewport(0.0, 1.0, 0.0, 1.0);

        PieSlice[] slice = pie.getPieSlice();

        slice[0].setFillColor(Color.green);
        slice[0].setTitle("Green");
        slice[0].setExplode(0.1);

        slice[1].setFillColor(Color.red);
        slice[1].setTitle("Red");
        slice[1].setFillOutlineColor(Color.yellow);
        slice[1].setExplode(0.1);

        slice[2].setFillColor(Color.blue);
        slice[2].setTitle("Blue");
        slice[2].setExplode(0.1);

        slice[3].setFillColor(Color.yellow);
        slice[3].setTitle("Yellow");
        slice[3].setExplode(0.15);
    }

    public static void main(String argv[]) {
        new SamplePieChart().setVisible(true);
    }
}
```

Box Plot

BoxPlots are used to show statistics about multiple groups of observations.

For each group of observations, the box limits represent the lower quartile (25-th percentile) and upper quartile (75-th percentile). The median is displayed as a line across the box. Whiskers are drawn from the upper quartile to the upper adjacent value, and from the lower quartile to the lower adjacent value.

Optional notches may be displayed to show a 95 percent confidence interval about the median, at

$$\pm 1.58\ IRQ / \sqrt{n}$$

where *IRQ* is the interquartile range and ***n*** is the number of observations. Outside and far outside values may be displayed as symbols. Outside values are outside the inner fence. Far out values are outside the outer fence.

The `BoxPlot` has several child nodes. Any of these nodes can be disabled by setting their `Paint` attribute to `false`.

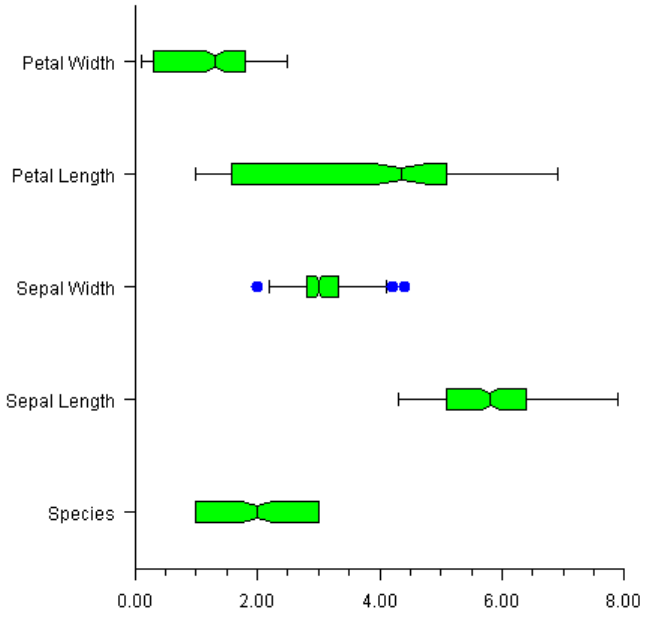
- The `Bodies` node has the main body of the box plot elements. Its fill attributes determine the drawing of (notched) rectangle (see Fill Area Attributes). Its line attributes determine the drawing of the median line. The width of the box is controlled by the `MarkerSize` attribute (see Attribute MarkerSize).
- The `Whiskers` node draws the lines to the upper and lower quartile. Its drawing is affected by the marker attributes.
- The `FarMarkers` node holds the far markers. Its drawing is affected by the marker attributes.
- The `OutsideMarkers` node holds the outside markers. Its drawing is affected by the marker attributes.

Example

In this example, the Fisher iris data set is read from a file and a Box plot is created from data. The data is in a file called `FisherIris.csv` in the same directory as this class.

The *y*-axis labels are taken from the column names.

The boxes are colored green, the markers are all filled circles. The outside markers are blue and the far outside markers would be red, if there were any.



View code file

```
import com.imsl.chart.*;
import com.imsl.io.FlatFile;
import java.io.*;
import java.util.StringTokenizer;

public class SampleBoxPlot extends JFrameChart {

    public SampleBoxPlot() throws IOException, java.sql.SQLException {
        // Read the data
        InputStream is =
                SampleBoxPlot.class.getResourceAsStream("FisherIris.csv");
        DataReader reader = new DataReader(is);
        int nColumns = 5;
        int nObs = 150;
        String labels[] = new String[nColumns];
        for (int i = 0; i < nColumns; i++) {
            labels[i] = reader.getMetaData().getColumnName(i + 1);
        }
        double irisData[][] = new double[nColumns][nObs];
        for (int j = 0; reader.next(); j++) {
            for (int i = 0; i < nColumns; i++) {
                irisData[i][j] = reader.getDouble(i + 1);
            }
        }
```

```
        // Setup the chart
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        BoxPlot boxPlot = new BoxPlot(axis, irisData);
        boxPlot.setBoxPlotType(BoxPlot.BOXPLOT_TYPE_HORIZONTAL);
        boxPlot.setLabels(labels);
        boxPlot.getBodies().setFillColor(java.awt.Color.green);
        boxPlot.setMarkerType(BoxPlot.MARKER_TYPE_FILLED_CIRCLE);
        boxPlot.getOutsideMarkers().setMarkerColor(java.awt.Color.blue);
        boxPlot.getFarMarkers().setMarkerColor(java.awt.Color.red);
        boxPlot.setNotch(true);
    }

    public static void main(String argv[])
            throws IOException, java.sql.SQLException {
        new SampleBoxPlot().setVisible(true);
    }

    /**
     *  Read the Fisher Iris data
     */
    static private class DataReader extends FlatFile {

        public DataReader(InputStream is) throws IOException {
            super(new BufferedReader(new InputStreamReader(is)));
            String line = readLine();
            StringTokenizer st = new StringTokenizer(line, ",");
            for (int j = 0; st.hasMoreTokens(); j++) {
                setColumnName(j + 1, st.nextToken().trim());
                setColumnClass(j, Double.class);
            }
        }
    }
}
```

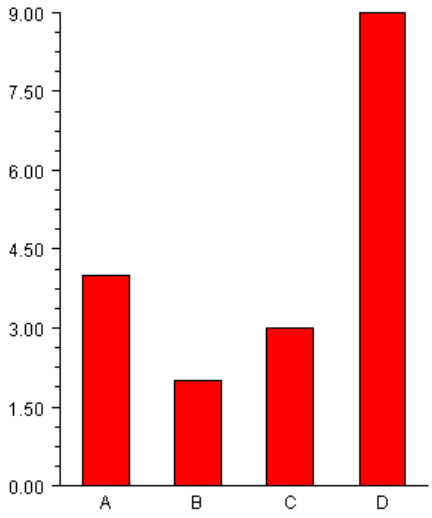
Bar Chart

The class `Bar` is used to create bar charts and histograms. This pagesection describes the construction of labeled bar charts. For a discussion of histograms, see Histogram.

Simple Bar Chart

The following code creates this labeled bar chart. The `BarType` attribute can be either `BAR_TYPE_VERTICAL` or `BAR_TYPE_HORIZONTAL`. The method `setLabels` sets the bar labels and adjusts the attributes of the axis to be appropriate for bar labels. The `setLabels` method must be called after the `setBarType` method, so that the correct axis has its attributes adjusted.

The drawing of the bars is controlled by the FillType and FillOutlineType attributes. By default `FillType` has the value `FILL_TYPE_SOLID`, so setting the associated attribute FillColor to red causes solid red bars to be drawn.



View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SampleBar extends JFrameChart {

    public SampleBar() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        double y[] = {4, 2, 3, 9};
        Bar bar = new Bar(axis, y);
        bar.setBarType(Bar.BAR_TYPE_VERTICAL);
        bar.setLabels(new String[]{"A","B","C","D"});
        bar.setFillColor(Color.red);
    }
```

```
    public static void main(String argv[]) {
        new SampleBar().setVisible(true);
    }
}
```

Attribute BarGap

The `BarGap` attribute sets the gap between bars in a group. A gap of 1.0 means that space between bars is the same as the width of an individual bar in the group. Its default value is 0.0, meaning there is no space between groups.

Attribute BarWidth

The `BarWidth` attribute sets the width of the groups of bars at each index. Its default value is 0.5. If the number of groups is increased, the width of each individual bar is reduced proportionately.

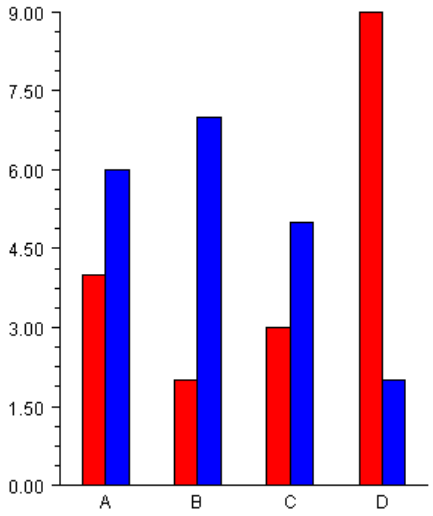
See Histogram for an example of the use of the `BarWidth` attribute.

Grouped Bar Chart

In a grouped bar chart multiple sets of data are displayed as side-by-side bars.

The data argument to the constructor for a grouped bar chart is an `nGroups` by `nItems` array of `doubles`. In this example there are two groups, each containing four items. All of the groups must contain the same number of items.

The `getBarSet(int)` method returns a ***BarSet*** object that is a collection of the ***BarItem*** objects that make up a given group. Here the bars in group 0 are set to red and those in group 1 are set to blue.



View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SampleBarGroup extends JFrameChart {

    public SampleBarGroup() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        double y[][] = {{4,2,3,9},{6,7,5,2}};
        Bar bar = new Bar(axis, y);
        bar.setBarType(Bar.BAR_TYPE_VERTICAL);
        bar.setLabels(new String[]{"A","B","C","D"});
        bar.getBarSet(0).setFillColor(Color.red);
        bar.getBarSet(1).setFillColor(Color.blue);
    }

    public static void main(String argv[]) {
        new SampleBarGroup().setVisible(true);
    }
}
```

In the above grouped bar chart example, the `Bar` constructor creates a collection of chart nodes. For each group, it creates a `BarSet` node as its direct child. Each `BarSet` node has `BarItem` nodes as children, one for each bar in the set.

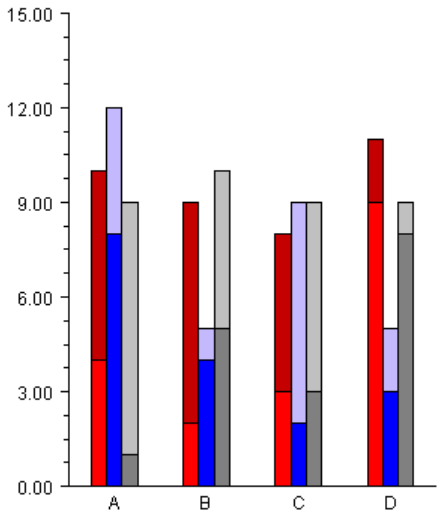
Stacked Grouped Bar Chart

The most general form of the bar chart is a stacked, grouped bar chart.

The data argument to the constructor for a stacked, grouped bar chart is an `nStacks` by `nGroups` by `nItems` array of `doubles`. In this example there are two stacks in three groups each containing four items. All of the stacks must contain the same number of groups and all of the groups must contain the same number of items.

The `getBarSet(int,int)` method returns a ***BarSet*** object that is a collection of the ***BarItem*** objects that make up a given stack/group. Here within each group the stacks are set to shades of the same color.

A stacked bar chart, without groups, can be constructed as a stacked-grouped bar chart with one group.



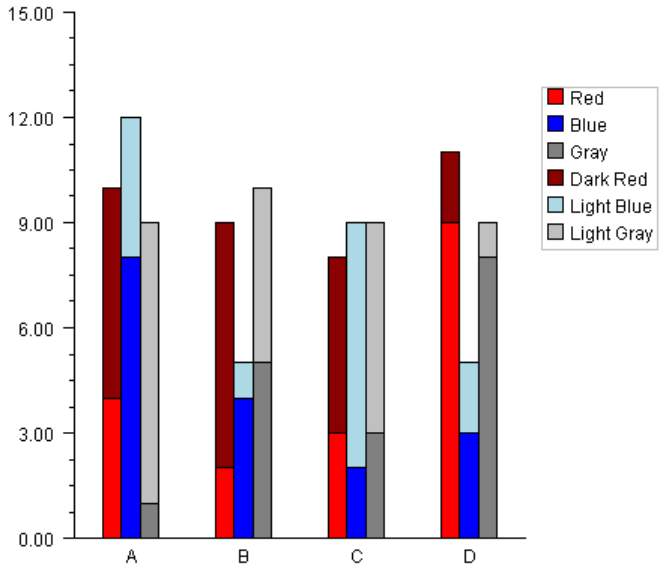
View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SampleBarGroupStack extends JFrameChart {
    static final Color darkRed = new Color(196,0,0);
    static final Color lightBlue = new Color(196,185,253);

    public SampleBarGroupStack() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        // y is a 2 by 3 by 4 array
        double y[][][] = {
            {{4,2,3,9},{8,4,2,3},{1,5,3,8}},
            {{6,7,5,2},{4,1,7,2},{8,5,6,1}}
        };
        Bar bar = new Bar(axis, y);
        bar.setBarType(Bar.BAR_TYPE_VERTICAL);
        bar.setLabels(new String[]{"A","B","C","D"});
```

```
        // group 0 - shades of red
        bar.getBarSet(0,0).setFillColor(Color.red);
        bar.getBarSet(1,0).setFillColor(darkRed);
        // group 1 - shades of blue
        bar.getBarSet(0,1).setFillColor(Color.blue);
        bar.getBarSet(1,1).setFillColor(lightBlue);
        // group 2 - shades of gray
        bar.getBarSet(0,2).setFillColor(Color.gray);
        bar.getBarSet(1,2).setFillColor(Color.lightGray);
    }

    public static void main(String argv[]) {
        new SampleBarGroupStack().setVisible(true);
    }
}
```

Legend

The ***Legend*** for a bar chart is turned on by setting the `Legend`'s `Paint` attribute to `true` and defining the `Title` attributes for the legend entries. The legend entries are the ***BarSet*** objects. The following example is the stacked, grouped bar example with the legend enabled.



View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SampleBarLegend extends JFrameChart {
    public SampleBarLegend() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        chart.getLegend().setPaint(true);
        // y is a 2 by 3 by 4 array
        double y[][][] = {
            {{4,2,3,9},{8,4,2,3},{1,5,3,8}},
            {{6,7,5,2},{4,1,7,2},{8,5,6,1}}
        };
        Bar bar = new Bar(axis, y);
        bar.setBarType(Bar.BAR_TYPE_VERTICAL);
        bar.setLabels(new String[]{"A","B","C","D"});
        // group 0 - shades of red
        bar.getBarSet(0,0).setTitle("Red");
        bar.getBarSet(0,0).setFillColor(Color.red);
        bar.getBarSet(1,0).setTitle("Dark Red");
        bar.getBarSet(1,0).setFillColor("DarkRed");
        // group 1 - shades of blue
        bar.getBarSet(0,1).setTitle("Blue");
```

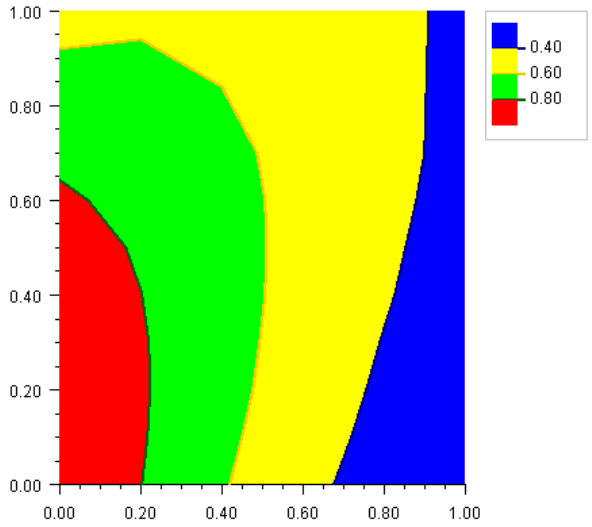
```
        bar.getBarSet(0,1).setFillColor(Color.blue);
        bar.getBarSet(1,1).setTitle("Light Blue");
        bar.getBarSet(1,1).setFillColor("LightBlue");
        // group 2 - shades of gray
        bar.getBarSet(0,2).setTitle("Gray");
        bar.getBarSet(0,2).setFillColor(Color.gray);
        bar.getBarSet(1,2).setTitle("Light Gray");
        bar.getBarSet(1,2).setFillColor(Color.lightGray);
    }

    public static void main(String argv[]) {
        new SampleBarLegend().setVisible(true);
    }
}
```

Contour Chart

A Contour chart shows level curves of a two-dimensional function.

Example

The ***JFrameChart*** class is used to create a frame containing a ***Chart*** node. A ***Contour*** node is then created as a child of the `Chart` node. The `Contour` node constructor creates ***ContourLevel*** nodes as its children. The number of `ContourLevel` nodes created is equal to the number of contour levels plue one, here equal to 4. After the `ContourLevel` nodes are created they are retrieved from the object pie and customized by setting attributes.



View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SampleContour extends JFrameChart {

    public SampleContour() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);

        double xGrid[] = {0, .2, .4, .6, .8, 1.0};
        double yGrid[] = {0, .1, .2, .3, .4, .5, .6, .7, 1.};
        double zData[][] = new double[xGrid.length][yGrid.length];
        for (int i = 0;  i < xGrid.length;  i++) {
```

```
            for (int j = 0;  j < yGrid.length;  j++) {
                double x = xGrid[i];
                double y = yGrid[j];
                zData[i][j] = Math.exp(-x)*Math.cos(x-y);
            }
        }
        double level[] = {.4, .6, .8};
        Contour contour = new Contour(axis, xGrid, yGrid, zData, level);
        contour.getContourLegend().setPaint(true);
        contour.getContourLevel(0).setFillColor(Color.blue);
        contour.getContourLevel(1).setFillColor(Color.yellow);
        contour.getContourLevel(2).setFillColor(Color.green);
        contour.getContourLevel(3).setFillColor(Color.red);

        contour.getContourLevel(0).setLineColor("darkblue");
        contour.getContourLevel(1).setLineColor(Color.orange);
        contour.getContourLevel(2).setLineColor("darkgreen");
        contour.getContourLevel(3).setLineColor("darkred");
    }

    public static void main(String argv[]) {
        new SampleContour().setVisible(true);
    }
}
```

The `FillColor` and `LineColor` attributes are set in each level. This determines the color of the fill area for the level and the color of the level curves. Since the default value of `FillColor` is black, it is generally recommended that `FillColor` be set in each level.

Each level corresponds to the area less than or equal to the contour level value and greater than the previous level, if any. So in this example, since the 0-th level value is 0.4, the area where the function is less than 0.4 is filled with blue (the level-0 fill color) and the level curve equal to 0.4 is drawn with dark blue, the level-0 line color.

Heatmap

A ***Heatmap*** divides a rectangle into subrectangles. The color of each subrectangle is determined by the value of a data array and a colormap.

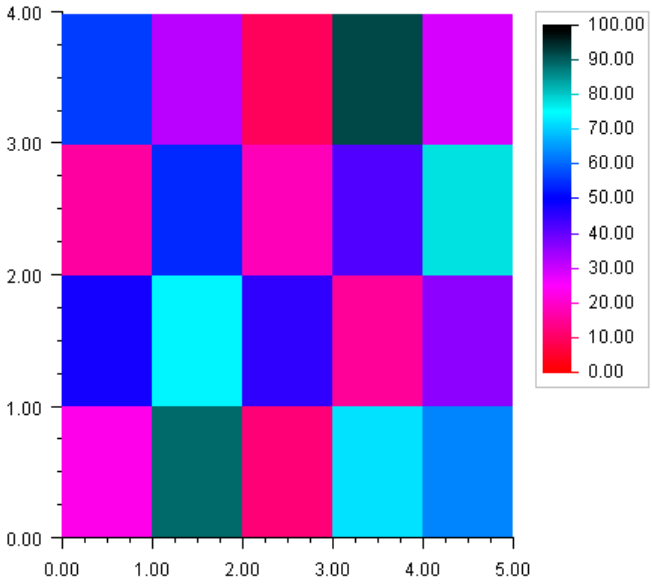
If the data array is *m* by *n* then there are *m* divisions along the *x*-axis and *n* divisions along the *y*-axis.

A ***Colormap*** is a mapping from [0,1] to color values. The blue-red colormap, used in the example below, maps 0 to red and 1 to dark blue and interpolates between these endpoints values. The heatmap maps the minimum data value to the color corresponding to 0 and the highest data value to the color corresponding to 1.

The `Heatmap` class has a special legend for `Colormap`s. It displays the color values as a gradient labeled with corresponding data values.

Example

In this example a two-dimensional array of data is plotted as a heatmap. The "red-blue" `Colormap` is used. The `Heatmap` `Legend` is enabled by settings its `Paint` attribute to `true`.



View code file

```
import com.imsl.chart.*;

public class SampleHeatmap extends JFrameChart {

    public SampleHeatmap() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        double xmin = 0.0;
        double xmax = 5.0;
```

```
        double ymin = 0.0;
        double ymax = 4.0;
        double zmin = 0.0;
        double zmax = 100.0;
        double data[][] = {
            {23, 48, 16, 56},
            {89, 74, 54, 32},
            {12, 45, 18,  9},
            {72, 15, 42, 92},
            {63, 36, 78, 29}
        };
        Heatmap heatmap = new Heatmap(axis, xmin, xmax, ymin, ymax,
            zmin, zmax, data, Colormap.BLUE_RED);
        heatmap.getHeatmapLegend().setPaint(true);
    }

    public static void main(String argv[]) {
        new SampleHeatmap().setVisible(true);
    }
}
```

Treemap

A ***Treemap*** divides a rectangle into subrectangles. The size of each rectangle is determined by an array of data values. The color of each subrectangle is determined by the value of a second data array and a ***Colormap***.

The primary data array is a sorted set of values to map to the subrectangle areas. The placement algorithm is adapted from Bruls, Mark and Huizing, Kees and Wijk, Jarke J. van (2000) *Squarified Treemaps*. In Proceedings of the Joint Eurographics and IEEE TCVG Symposium on Visualization.

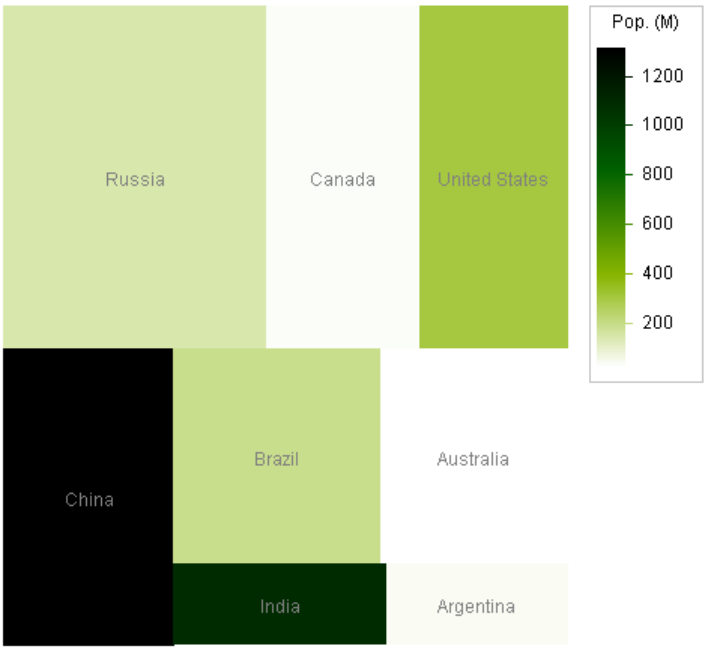
By default, the algorithm fills either rows first or columns first determined by the aspect ratio of the resulting sub-rectangles. The user may force drawing orientation by using the `setOrientation()` method with `Treemap.COLUMNFIRST` or `Treemap.ROWFIRST` fields.

A `Colormap` is a mapping from [0,1] to color values. The `Treemap` maps the minimum value of the second data array to the color corresponding to 0 and the highest value to the color corresponding to 1.

The `Treemap` class has a special `legend` for `Colormap`s. It displays the color values as a gradient labeled with corresponding data values.

Example

In this example an array of country data is plotted as a treemap. The area is proportional to the country's land area while the shading maps to the population. The "green-white-linear" `Colormap` is used. The `Treemap legend` is enabled by setting its `Paint` attribute to `true`.



View code file

```
import com.imsl.chart.*;
```

```
public class SampleTreemap extends JFrameChart {

    public SampleTreemap() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);

        double[] areas = {
            6592735, 3855081, 3718691, 3705386,
            3286470, 2967893, 1269338, 1068296};

        double[] population = {
            142.893540, 33.098932, 298.444215, 1313.973713,
            188.078227, 20.264082, 1095.351995, 39.930091};

        String[] names = {
            "Russia", "Canada", "United States", "China",
            "Brazil", "Australia", "India", "Argentina"};

        Treemap treemap = new Treemap(axis, areas, population,
                Colormap.GREEN_WHITE_LINEAR);
        treemap.setLabels(names);
        treemap.setTextColor(java.awt.Color.gray);
        treemap.getTreemapLegend().setPaint(true);
        treemap.getTreemapLegend().setTitle("Pop. (M)");
        treemap.getTreemapLegend().setTextFormat("0");
        axis.setViewport(0.05, 0.8, 0.1, 0.95);
    }

    public static void main(String argv[]) {
        new SampleTreemap().setVisible(true);
    }
}
```

Histogram

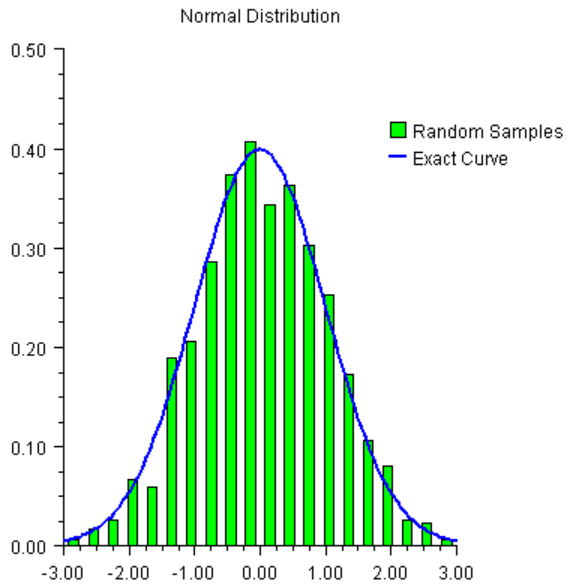
A histogram is a bar chart in which the height of the bars is proportional to the frequencies. A histogram generally uses the same axis style as a scatter plot (i.e. the bars are numbered not labeled.)

In JMSL, histograms are drawn using the ***Bar*** class, but its `setLabels` method is not used.

Example

In this example `normally` distributed random numbers are generated and placed into 20 uniformly sized bins in the interval [-3,3]. Points outside of this interval are ignored. The bin counts are scaled by the number of samples and the bin width. The scaled bin counts are charted using `Bar` chart. The exact normal distribution is implemented as a ***ChartFunction*** and plotted.

The ***Legend*** is displayed by setting the legend node's `Paint` attribute to true and defining the bar chart's `Title` attribute. The `Legend` is positioned on the chart by setting its `Viewport` attribute.



View code file

```
import com.imsl.math.Sfun;
import com.imsl.stat.Random;
import com.imsl.chart.*;
import java.awt.*;
import javax.swing.*;

public class SampleHistogram extends JFrameChart {

    public SampleHistogram() {
        int nSamples = 1000;
        int nBins = 20;
```

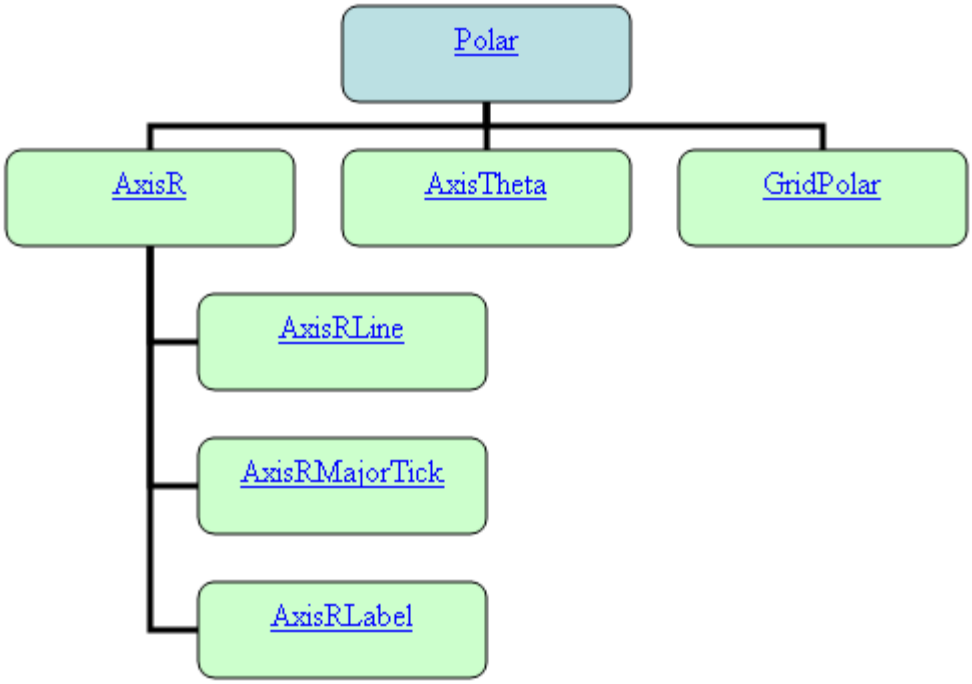
```
        // Setup the bins
        double bins[] = new double[nBins];
        double dx = 6.0/nBins;
        double x[] = new double[nBins];
        for (int k = 0;  k < nBins;  k++) {
            x[k] = -3.0 + (k+0.5)*dx;
        }

        Random r = new Random(123457);
        for (int k = 0;  k < nSamples;  k++) {
            double t = r.nextNormal();
            int j = (int)Math.round((t+3.0-0.5*dx)/dx);
            if (j >= 0  &&  j < nBins)  bins[j]++;
        }

        // Scale the bins
        for (int k = 0;  k < nBins;  k++) {
            bins[k] /= nSamples*dx;
        }

        // create the chart
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);

        chart.getChartTitle().setTitle("Normal Distribution");
        chart.getLegend().setPaint(true);
        chart.getLegend().setViewport(0.7, 1.0, 0.2, 0.3);
        chart.getLegend().setFillOutlineType(chart.FILL_TYPE_NONE);

        Bar bar = new Bar(axis, x, bins);
        bar.setBarType(bar.BAR_TYPE_VERTICAL);
        bar.setFillColor(Color.green);
        bar.setBarWidth(0.5*dx);
        bar.setTitle("Random Samples");

        // plot the expected curve
        ChartFunction f = new ChartFunction() {
            public double f(double x) {
                return Math.exp(-0.5*x*x)/Math.sqrt(2.0*Math.PI);
            }
        };
        Data data = new Data(axis, f, -3, 3.0);
        data.setLineColor(Color.blue);
        data.setTitle("Exact Curve");
        data.setLineWidth(2.0);
    }

    public static void main(String argv[]) {
        new SampleHistogram().setVisible(true);
    }
```

```
}
```

Polar Plot

The class ***Polar*** is used to plot (*r*, *theta*) data, where *r* is the distance from the origin and *theta* is the angle from the positive *x*-axis. ***Data*** node (*x*, *y*) values are treated as if they were (*r*, *theta*) values.

When a `Polar` node is created, a subtree of other nodes is created. These automatically created nodes are shown in the following diagram.



AxisR is the *r*-axis node. This is drawn along the positive *x*-axis. This node has three subnodes: `AxisRLine` for the actual line, ***AxisRMajorTick*** for the tick marks along the line and `AxisRLabel` for the tick mark labels.

The drawing of the `AxisRLine` and the `AxisRMajorTick` is controlled by the Line Attributes. The drawing of the `AxisRLabel` is controlled by the Text Attributes.

The `Window` attribute in `AxisR` specifies the length of the *r*-axis. Since the *r*-axis range is [0,*rmax*], the `Window` attribute for this node is the *rmax* value. Autoscaling is used to automatically determine *rmax* from the data. Autoscaling can be turned off. See `Autoscale` for more details.

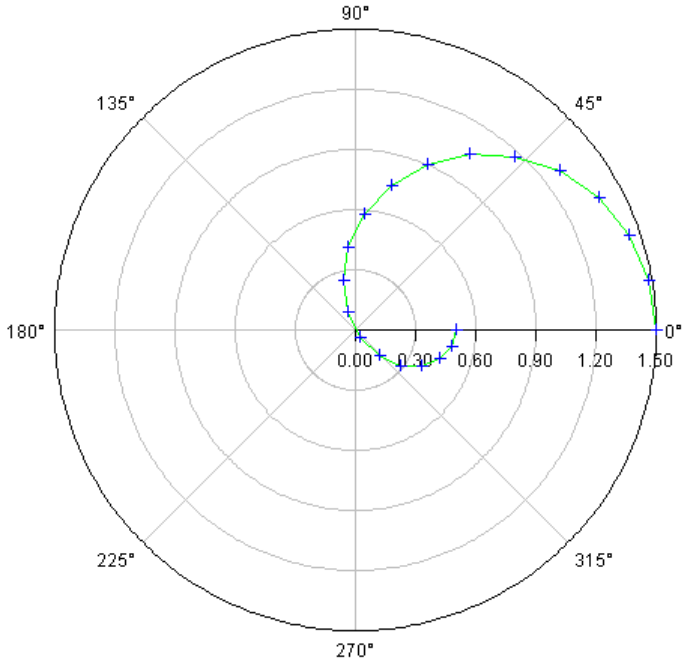
AxisTheta is the *theta*-axis node. It is drawn as a circle around the plot. The drawing of the circle itself is controlled by the line attributes. The drawing of the axis labels is controlled by the text attributes.

The `Window` attribute in the `AxisTheta` node contains the angular range, in radians. Its default value is [0,π]. The `Window` attribute can be explicitly set to change the range, but autoscaling is not available to determine the angular range.

GridPolar is the plot grid. It consists of both radial and circular lines. The drawing of these lines is controlled by the line attributes. By default, these lines are drawn and are light gray.

Example

The function *r* = 0.5 + *cos*(*q*), for 0 <= *q* <= *p* is plotted.



View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SamplePolar extends JFrameChart {

    public SamplePolar() {
        Chart chart = getChart();
        Polar axis = new Polar(chart);

        double r[] = new double[20];
        double theta[] = new double[r.length];
        for (int k = 0;  k < r.length;  k++) {
            theta[k] = Math.PI*k/(r.length-1);
            r[k] = 0.5 + Math.cos(theta[k]);
        }
        Data data = new Data(axis, r, theta);
        data.setDataType(Data.DATA_TYPE_MARKER | Data.DATA_TYPE_LINE);
        data.setLineColor(Color.green);
        data.setMarkerColor(Color.blue);
    }

    public static void main(String argv[]) {
        new SamplePolar().setVisible(true);
    }
```

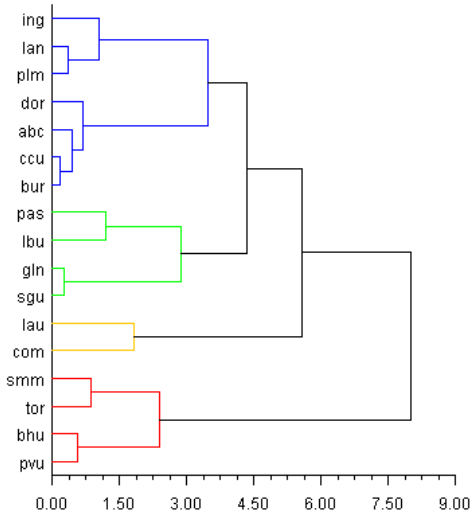
```
}
```

Dendrogram Chart

A ***Dendrogram*** chart is a graphical way to display results from hierarchical cluster analysis. This section describes the construction of a `Dendrogram` chart.

Example

The data for this example is grouped into clusters using the ***Dissimilarities*** and ***ClusterHierarchical*** classes. A `Dendrogram` node is then created as a child of an axis node. The `Dendrogram` constructor requires input values from the `ClusterHierarchical` object.



The `setLables` and `setLineColor` methods are used to customize the look of the chart. `Labels` are provided in a `String` array in the order of the input data and sorted by the `Dendrogram` object to match the output order. Clusters are grouped by color based on the number of elements in the array passed to the `setLineColor` method.

View code file

```
import com.imsl.stat.*;
import com.imsl.chart.*;

public class SampleDendrogram extends javax.swing.JApplet {

    private JPanelChart panel;

    public void init() {
```

```
    Chart chart = new Chart(this);
    panel = new JPanelChart(chart);
    getContentPane().add(panel, java.awt.BorderLayout.CENTER);
    setup(chart);
}

static private void setup(Chart chart) {

    /*
    1998 test data from 17 school districts in Los Angeles County.

    The variables were:
    lep - Proportion of LEP students to total tested
    read - The Reading Scaled Score for 5th Grade
    math - The Math Scaled Score for 5th Grade
    lang - The Language Scaled Score for 5th Grade

    The districts were:
    lau - Los Angeles
    ccu - Culver City
    bhu - Beverly Hills
    ing - Inglewood
    com - Compton
    smm - Santa Monica Malibu
    bur - Burbank
    gln - Glendale
    pvu - Palos Verdes
    sgu - San Gabriel
    abc - Artesia, Bloomfield, and Carmenita
    pas - Pasadena
    lan - Lancaster
    plm - Palmdale
    tor - Torrance
    dow - Downey
    lbu - Long Beach

    input lep read math lang str3 district
    .38 626.5 601.3 605.3 lau
    .18 654.0 647.1 641.8 ccu
    .07 677.2 676.5 670.5 bhu
    .09 639.9 640.3 636.0 ing
    .19 614.7 617.3 606.2 com
    .12 670.2 666.0 659.3 smm
    .20 651.1 645.2 643.4 bur
    .41 645.4 645.8 644.8 gln
    .07 683.5 682.9 674.3 pvu
    .39 648.6 647.8 643.1 sgu
    .21 650.4 650.8 643.9 abc
    .24 637.0 636.9 626.5 pas
    .09 641.1 628.8 629.4 lan
    .12 638.0 627.7 628.6 plm
```

```
        .11 661.4 659.0 651.8 tor
        .22 646.4 646.2 647.0 dow
        .33 634.1 632.0 627.8 lbu
         */

        double[][] data = {
            {.38, 626.5, 601.3, 605.3},
            {.18, 654.0, 647.1, 641.8},
            {.07, 677.2, 676.5, 670.5},
            {.09, 639.9, 640.3, 636.0},
            {.19, 614.7, 617.3, 606.2},
            {.12, 670.2, 666.0, 659.3},
            {.20, 651.1, 645.2, 643.4},
            {.41, 645.4, 645.8, 644.8},
            {.07, 683.5, 682.9, 674.3},
            {.39, 648.6, 647.8, 643.1},
            {.21, 650.4, 650.8, 643.9},
            {.24, 637.0, 636.9, 626.5},
            {.09, 641.1, 628.8, 629.4},
            {.12, 638.0, 627.7, 628.6},
            {.11, 661.4, 659.0, 651.8},
            {.22, 646.4, 646.2, 647.0},
            {.33, 634.1, 632.0, 627.8}};

        String[] lab = {
            "lau", "ccu", "bhu", "ing", "com", "smm",
            "bur", "gln", "pvu", "sgu", "abc", "pas",
            "lan", "plm", "tor", "dor", "lbu"};


        // 3rd arg in Dissimilarities gives different results for 0,1,2
        try {
            Dissimilarities dist = new Dissimilarities(data);
            dist.setScalingOption(Dissimilarities.STD_DEV);
            dist.compute();
            ClusterHierarchical clink =
                    new ClusterHierarchical(dist.getDistanceMatrix());
            clink.setMethod(ClusterHierarchical.LINKAGE_WARDS);
            clink.compute();

            AxisXY axis = new AxisXY(chart);

            // use either method below to create the chart
            Dendrogram dc = new Dendrogram(axis, clink,
                    Data.DENDROGRAM_TYPE_HORIZONTAL);

//              Dendrogram dc = new Dendrogram(
//                      axis,
//                      clink.getClusterLevel(),
//                      clink.getClusterLeftSons(),
//                      clink.getClusterRightSons(),
```

```
//                    Data.DENDROGRAM_TYPE_HORIZONTAL);

            dc.setLabels(lab);
            dc.setLineColor(new java.awt.Color[]{
                        java.awt.Color.blue,
                        java.awt.Color.green,
                        java.awt.Color.red,
                        java.awt.Color.orange});
        } catch (com.imsl.IMSLException e) {
            System.out.println(e.getStackTrace());
        }
    }

    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        SampleDendrogram.setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Chapter 3 Quality Control and Improvement Charts

Introduction

Quality improvement charts have a variety of uses. In this library the charts are organized into three broad groups: Shewhart control charts, other control charts and process improvement charts. The Shewhart control charts were originally described by the statistician Dr. Walter A. Shewhart (1931). Since this early work, other charts have been developed for engineering and management analysis of processes. In the 1980s customized charts were developed for other retrospective analysis of quality management data.

JMSL provides these Quality Control Charts:

Shewhart Charts

While working for Western Electric in the 1920s, Dr. Shewhart developed a general, practical approach to statistical monitoring of manufacturing processes. He advised managers on implementing these within Western Electric and later published his work in Shewhart (Montgomery, 1931). All Shewhart control charts share several characteristics in common. First, the horizontal axis represents time or lot sequence, but they all have different vertical axes, depending upon the chart time.

Next, all Shewhart control charts have a center line that is drawn parallel to the time axis. This represents the mean of the process, but the value of the process mean can vary depending upon which data are first used to design the chart. In some cases it will be the mean of the data plotted, in others it could be the mean calculated from a much larger number of measurements on the historical operation of the process.

Lastly, all Shewhart control charts have lines drawn to represent either the upper or lower control limits. In most cases both control lines are present, in others where the data have a natural bound, such as zero, only one of these control limits might be drawn.

Shewhart control charts are also broadly classified into two groups: ***variable*** and ***attribute*** data charts. ***Variable control charts*** are used when the quality of interest is a continuous variable, such as the diameter of a valve. If ***w*** is a continuous measure of a quality of interest, with mean μ_W and within-sample standard deviation σ_W, then the center line is at μ_W and the upper and lower controls limits are at $\mu_W \pm k\sigma_W$. Typically k =3 and the charts are called 3-sigma control charts.

Attribute control charts are used when qualities, not quantities are measured. For example, items may be characterized as conforming or nonconforming to a specification. Items may also be characterized as defective or nondefective. Examples of attributes include the number of failures in a manufacturing run or the number of defects on a computer chip wafer.

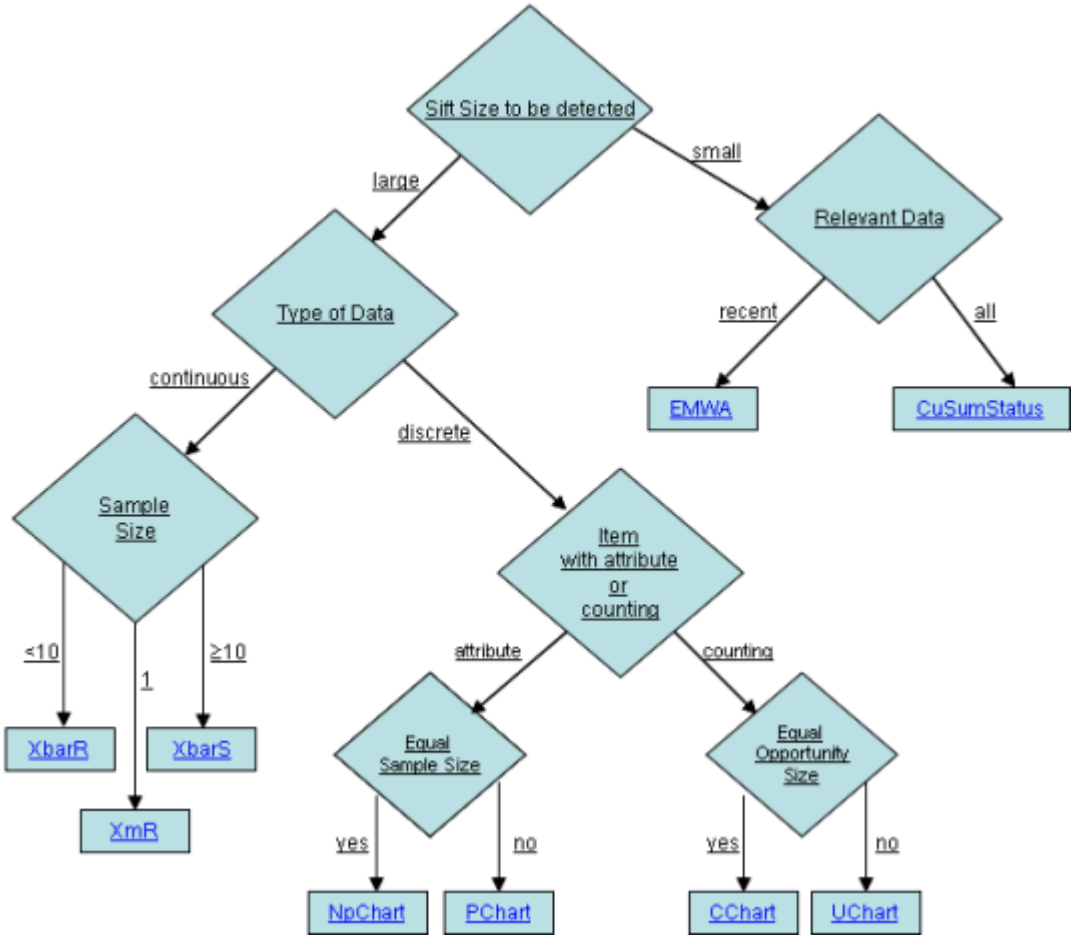
P-charts and np-charts are used for plotting the percentage or number of defective items in a sample, respectively. Inspectors, for example, might inspect 100 items in a subgroup and record the number of items with one or more defects. These can be plotted using either the number of defects or the percentage of defects found in each subgroup. In JMSL, class ***PChart*** produces p-charts p-charts for plotting the percentage of nonconforming items, and class ***NpChart*** produces an np-chart np-chart for the number of nonconforming items. In general, when the subgroup sample sizes are unequal a p-chart is used to adjust for unequal samples; otherwise an np-chart is used to display the number of defective items.

In some cases, instead of recording the number of defective items, the total number of defects of all types might be recorded. For example, the packaging for a semiconductor might be inspected for solder defects such as bridging, insufficient solder, bent leads, etc. The total number of all such defects might be recorded for a sample. Theoretically, there is no upper bound on the total number of defects found in a sample. These are typically plotted using a c-chart or u-chart, depending upon whether the sample size or area is constant from one subgroup to another. In JMSL, c-charts are implemented in class ***CChart*** and u-charts are implemented in class ***UChart***.

If a single item can have multiple defects then a `CChart` or `UChart` is used, depending upon also, whether the area inspected for defect(s) is consistent or varying. An example of multiple defects per item would be the count of the number of scratches on mirrors. If all samples have an equal opportunity for defects use `CChart`, otherwise use `UChart`. So to monitor the number of scratches on mirrors use `CChart` when all mirrors being made are the same size and use `UChart` for mirrors being made that are sized differently.

In JMSL the ***ShewhartControlChart*** class is the base of a number of classes; it is not usually used by itself. Most of the charts in this chapter extend `ShewhartControlChart`.

The following diagram can be used to determine the appropriate control chart to be used in a given situation.



Variables Control Charts:

- `XbarR` estimates μ_W and σ_W using the ranges of the samples. It is best used when the sample size of a continuous variable is between 2 and 10.
- `RChart` plots the sample ranges. It is typically used in conjunction with `XbarR`.
- `XbarS` estimates μ_W and σ_W using the means and standard deviations of the samples. It is best used when the sample size of a continuous variable is at least 10.
- `SChart` plots the sample standard deviations. It is typically used in conjunction with `XbarS`.
- `XmR` is a moving range chart. It is used when the sample size of a continuous variable is one.
- `EWMA` (Exponentially Weighted Moving-Average) plots weighted moving average values. It is used when the sample size of a continuous variable is one.

Attribute Control Charts:

- `NpChart` plots the number of defects. It is used when defects are not rare.
- `PChart` plots the rate of defects. It is used when defects are not rare.
- `CChart` plots the defect count. It is used when defects are rare.
- `UChart` plots the rate of defects. It is used when defects are rare.

Other Control Charts

CuSum and CuSumStatus

The ***CuSum*** and ***CuSumStatus*** charts are more efficient than ***Shewhart*** charts at detecting small shifts in the process mean because the plot represents the moving average of the cumulative differences between the process and its target (centerline).

Cumulative Probability

Cumulative probability charts are used if the defect rate is so small that there will be long runs when the number of defects is zero.

Process Improvement Charts

Pareto Chart

Pareto charts are used to analyze the root cause of process defects. They are based on the "Pareto Principal" which claims that 80% of the defects are caused by only 20% of the defect categories

Pareto charts resemble bar charts but are different. In a Pareto chart, the defect category with the largest number of defects always appears as the first and tallest bar on the chart. The other bars represent the remaining defect categories in descending order of magnitude.

ShewhartControlChart and ControlLimit

The ***ShewhartControlChart*** class is primarily used as the base class of other control chart classes. It provides the common functionality for the control charts.

For example, the default JMSL control charts include a center line and an upper and lower control limit at plus or minus three standard deviations from the center line. The Western Electric Company Rules (or WECO) are control limits at plus or minus one, two, and three standard deviations from the mean. They can be added using the method `addWecoLimits` in the `ShewhartControlChart` class.

The `ShewhartControlChart` uses the ***ControlLimit*** class for the upper and lower control limits and for the center line. Additional control limits, such as the WECO limits, can be added to a `ShewhartControlChart`. The line attributes can be used with `ControlLimit` to modify the drawing of each control limit.

The `ShewhartControlChart` class can be used directly when the statistics are computed by the user. In this example, data from Montgomery, Douglas C., Introduction to Statistical Quality Control, 4th Ed., 2001, New York: John Wiley and Sons, p. 406 is plotted. The code explicitly sets the lower control limit to 7 and the upper control limit to 13.

Further citations throughout this chapter for data plotted from Montgomery appear as parenthetical citations, e.g. (Montgomery 406).

Control Limit Customization

Control Limits and center lines can be added using the `ControlLimit.setValue` methods; `addCenterLine().setValue(int)` (see `addCenterLine()`), `addUpperControlLimit().setValue(int)` (see `addUpperControlLimit()`) and `addLowerControlLimit()`.`setValue(int)` (see `addLowerControlLimit()`). It is foreseeable that the default appearance might not be optimal for every application. In particular moving components or removing the `ControlLimit` objects can accomplished.

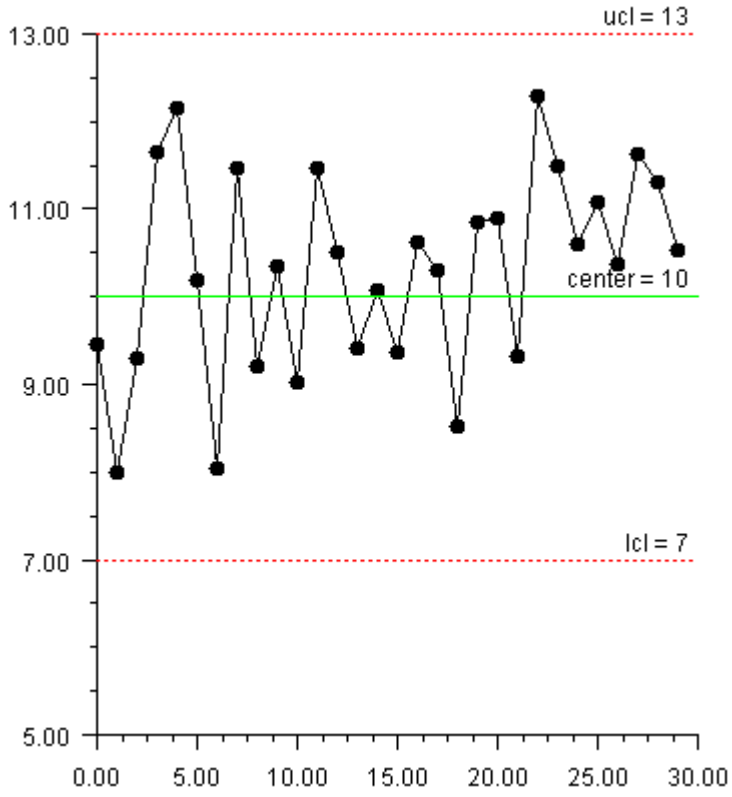
Before modifying a `ControlLimit` title it is important to realize the `ControlLimit` title text is justified in the lower right corner of a ***Text*** object (`TEXT_X_RIGHT` | `TEXT_Y_BOTTOM`). With this in mind, formatted strings may be used to alter their appearance. For example, to retain the `ControlLimit` line but eliminate the title, get the `ControlLimit` and set the title to an empty string (i.e. `getUpperControlLimit().setTitle("");`). In addition, the following can be used to move the title closer to the *y*-axis (note that repeated, leading, and trailing new lines `"\n"` are ignored):

```
Formatter fmt = new Formatter();
xbars.getUpperControlLimit().setTitle(fmt.format("ucl=15%90s",
    " ").toString());
```

After being added, it may be the case that a `ControlLimit` should be removed. This is accomplished with the `AbstractChartNode remove()` method. In the case of an upper control limit this would be `AbstractChartNode.remove()`.

View code file

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class SampleShewhart extends JFrameChart {

    static final double data[] = {
        9.45, 7.99, 9.29, 11.66, 12.16, 10.18,
        8.04, 11.46, 9.20, 10.34, 9.03, 11.47,
        10.51, 9.40, 10.08, 9.37, 10.62, 10.31,
        8.52, 10.84, 10.90, 9.33, 12.29, 11.50,
        10.60, 11.08, 10.38, 11.62, 11.31, 10.52
    };

    public SampleShewhart() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        ShewhartControlChart xbars = new ShewhartControlChart(axis);
        xbars.setData(data);
        xbars.addLowerControlLimit().setValue(7);
        xbars.addCenterLine().setValue(10);
        xbars.addUpperControlLimit().setValue(13);
    }

    public static void main(String argv[]) {
        new SampleShewhart().setVisible(true);
    }
}
```


ucl = 13
13.00
11.00
center = 10
9.00
lcl = 7
7.00
5.00
0.00
5.00
10.00
15.00
20.00
25.00
30.00

XbarS and SChart

The ***XbarS*** class plots the mean of each sample as well as control limits computed using the mean of the in-sample standard deviations. The positions of the control limits are determined by the equations

$$UCL = \bar{\bar{x}} + 3\frac{\bar{s}}{c_4\sqrt{n}}$$

$$Centerline = \bar{\bar{x}}$$

$$LCL = \bar{\bar{x}} - 3\frac{\bar{s}}{c_4\sqrt{n}}$$

where $\bar{\bar{x}}$ is the grand mean, the average of all the observations.

The factor of three in the above equations can be changed by setting the chart attribute ***ControlLimit***. The attribute applies similarly to all of the control charts.

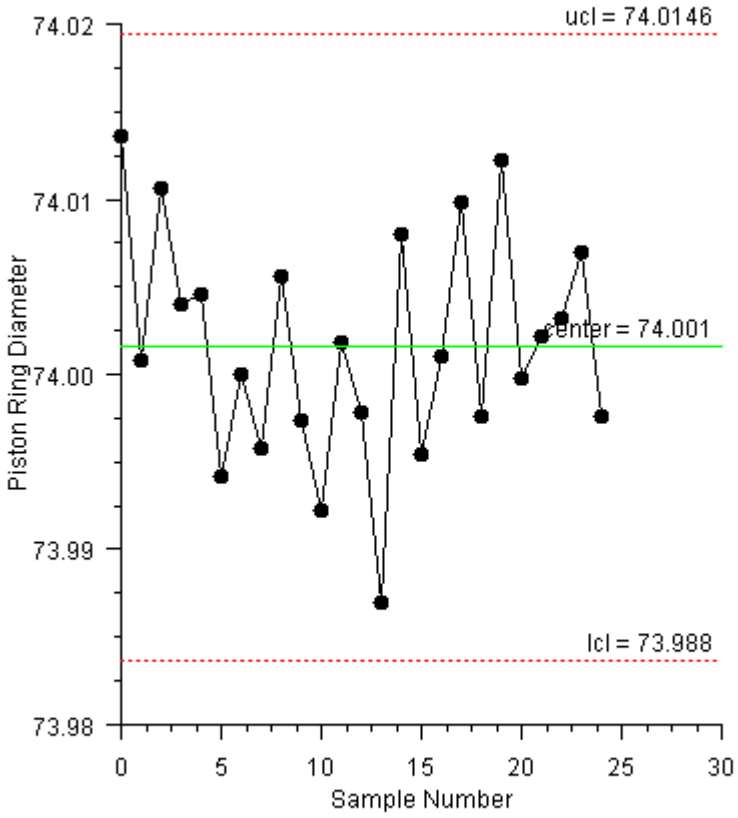
XbarS Example

The process of forging piston rings for automobile engines was monitored. The inside diameters of 25 samples, each containing 5 piston rings, were measured. The center line is at 74.001, the average diameter of all of the measured piston rings. The upper and lower control limits are determined by the average of the standard deviations of the 25 samples of 5 rings (Montgomery 215).

View code file

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class SampleXbarS extends JFrameChart {
    static final double diameter[][] = {
        {74.03, 74.002, 74.019, 73.992, 74.008},
        {73.995, 73.992, 74.001, 74.011, 74.004},
        {73.988, 74.024, 74.021, 74.005, 74.002},
        {74.002, 73.996, 73.993, 74.015, 74.009},
        {73.992, 74.007, 74.015, 73.989, 74.014},
        {74.009, 73.994, 73.997, 73.985, 73.993},
        {73.995, 74.006, 73.994, 74, 74.005},
        {73.985, 74.003, 73.993, 74.015, 73.988},
        {74.008, 73.995, 74.009, 74.005, 74.004},
        {73.998, 74, 73.99, 74.007, 73.995},
        {73.994, 73.998, 73.994, 73.995, 73.99},
        {74.004, 74, 74.007, 74, 73.996},
        {73.983, 74.002, 73.998, 73.997, 74.012},
        {74.006, 73.967, 73.994, 74, 73.984},
        {74.012, 74.014, 73.998, 73.999, 74.007},
        {74, 73.984, 74.005, 73.998, 73.996},
```

```
        {73.994, 74.012, 73.986, 74.005, 74.007},
        {74.006, 74.01, 74.018, 74.003, 74},
        {73.984, 74.002, 74.003, 74.005, 73.997},
        {74, 74.01, 74.013, 74.02, 74.003},
        {73.982, 74.001, 74.015, 74.005, 73.996},
        {74.004, 73.999, 73.99, 74.006, 74.009},
        {74.01, 73.989, 73.99, 74.009, 74.014},
        {74.015, 74.008, 73.993, 74, 74.01},
        {73.982, 73.984, 73.995, 74.017, 74.013}
    };

    public SampleXbarS() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        XbarS xbars = new XbarS(axis, diameter);

        xbars.getUpperControlLimit().setTitle("ucl = {0,number,0.0000}");

        axis.getAxisX().getAxisTitle().setTitle("Sample Number");
        axis.getAxisX().getAxisLabel().setTextFormat("0");
        axis.getAxisY().getAxisTitle().setTitle("Piston Ring Diameter");
        axis.getAxisY().setAutoscaleInput(axis.AUTOSCALE_OFF);
        axis.getAxisY().setWindow(73.985, 74.015);
    }


    public static void main(String argv[]) {
        new SampleXbarS().setVisible(true);
    }
}
```



The ***SChart*** plots the in-sample standard deviations of the observations as well as control limits computed using

$$UCL = \overline{s} + 3\frac{\overline{s}}{c_4}\sqrt{1 - c_4^2}$$

$$Centerline = \overline{s}$$

$$LCL = \overline{s} - 3\frac{\overline{s}}{c_4}\sqrt{1 - c_4^2}$$

The factor c_4 is such that $\overline{s} / c_4$ is an unbiased estimator of standard deviation.

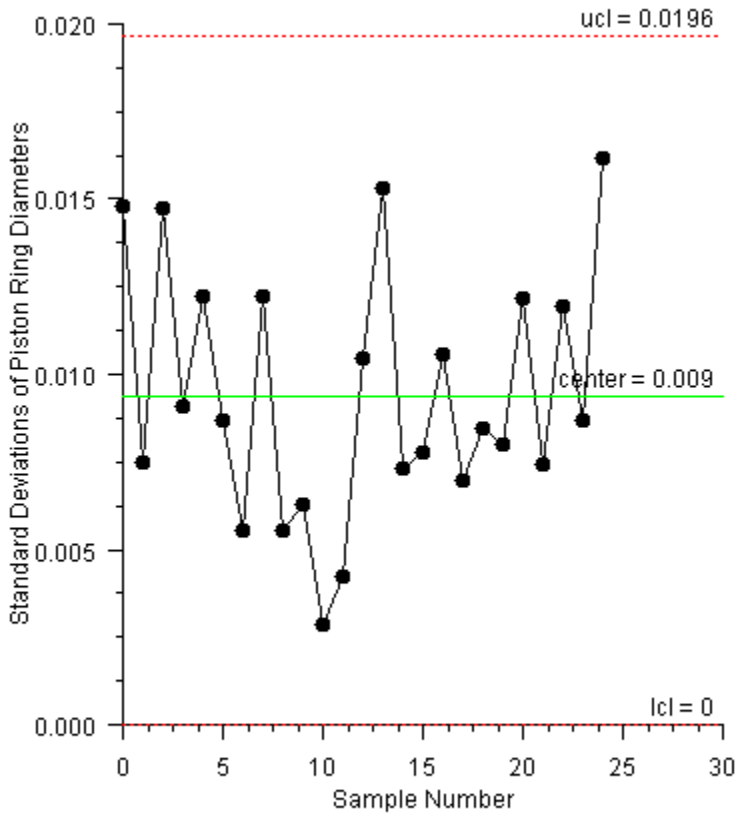
SChart Example

This example uses the same piston ring data as was used in `XbarS`, but now the standard deviations of the samples are plotted. The center line is at 0.009, the average of the sample standard deviations.

View code file

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;
```

```
import javax.swing.JFrame;

public class SampleSChart extends JFrameChart {
    public SampleSChart() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        SChart schart = new SChart(axis, SampleXbarS.diameter);

        schart.getUpperControlLimit().setTitle("ucl = {0,number,0.0000}");

        axis.getAxisX().getAxisTitle().setTitle("Sample Number");
        axis.getAxisX().getAxisLabel().setTextFormat("0");
        String title = "Standard Deviations of Piston Ring Diameters";
        axis.getAxisY().getAxisTitle().setTitle(title);
        axis.getAxisY().getAxisLabel().setTextFormat("0.000");
        axis.getAxisY().setAutoscaleInput(axis.AUTOSCALE_OFF);
        axis.getAxisY().setWindow(0.0, 0.02);
    }

    public static void main(String argv[]) {
        new SampleSChart().setVisible(true);
    }
}
```



Often the `XbarS` and `SChart` are plotted together. The `XbarS` class contains the static method `createCharts()`, which creates this pair of plots on a single chart.

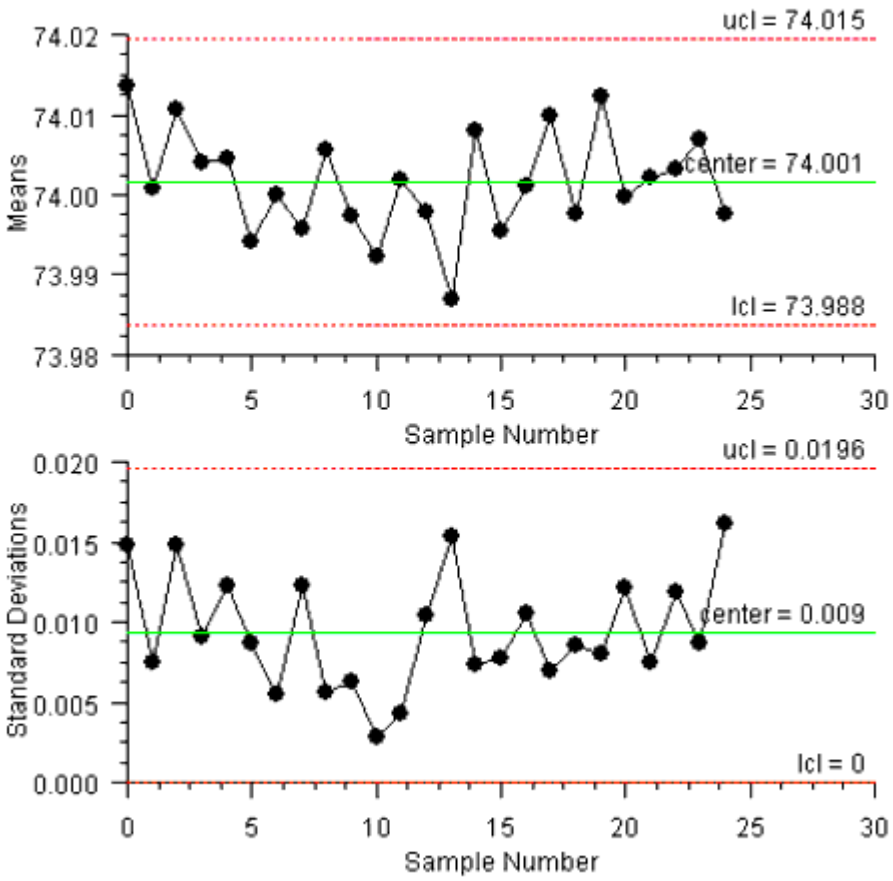
XbarSCombo Example

This example combines the charts in `XbarS` and `SChart`. The `Viewport` attribute of each is set so that they can appear on the same chart.

View code file

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class SampleXbarSCombo extends JFrameChart {
    public SampleXbarSCombo() {
        Chart chart = getChart();
        ShewhartControlChart charts[] =
            XbarS.createCharts(chart, SampleXbarS.diameter);
```

```
        AxisXY axis = (AxisXY)(charts[0].getAxis());
        axis.getAxisY().setAutoscaleInput(axis.AUTOSCALE_OFF);
        axis.getAxisY().setWindow(73.985, 74.015);

        axis = (AxisXY)(charts[1].getAxis());
        axis.getAxisY().getAxisLabel().setTextFormat("0.000");
        axis.getAxisY().setAutoscaleInput(axis.AUTOSCALE_OFF);
        axis.getAxisY().setWindow(0.0, 0.02);
        String title = "ucl = {0,number,0.0000}";
        ((SChart)charts[1]).getUpperControlLimit().setTitle(title);
    }


    public static void main(String argv[]) {
        new SampleXbarSCombo().setVisible(true);
    }
}
```



XbarSUnequal Example

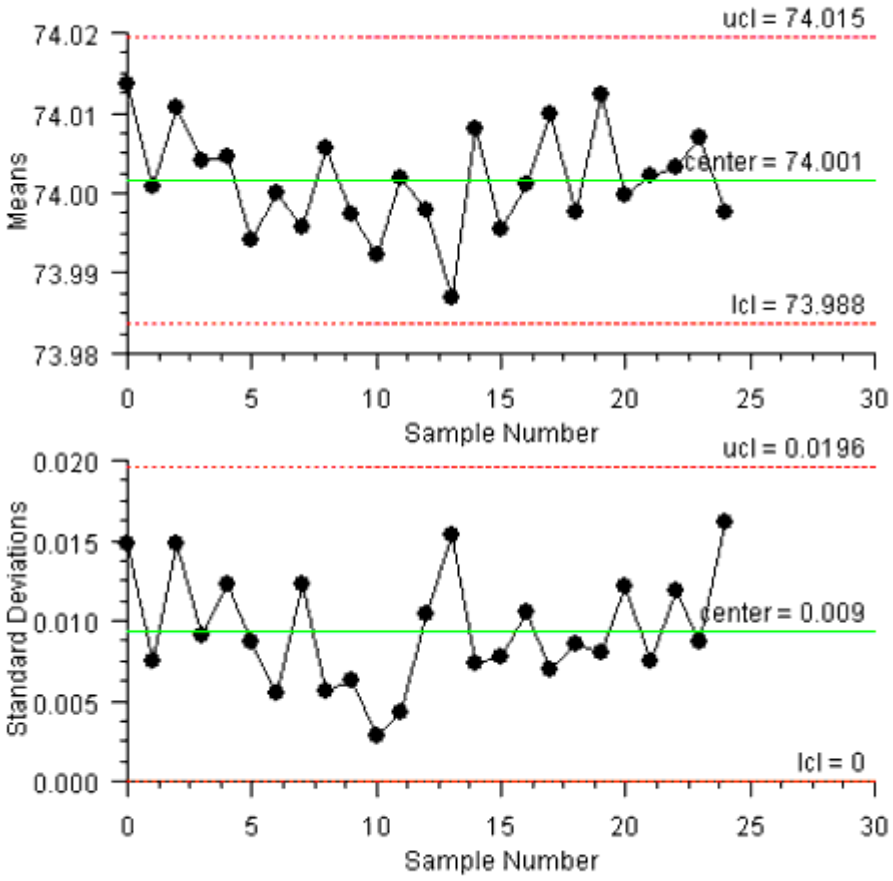
The example again uses the piston ring data, but now the sample size is not uniform. The upper and lower control limits are now stair step lines. The control limits are farther apart for smaller samples.

It is also possible to plot just the ***XbarS*** or ***SChart*** with unequal sample sizes.

View code file

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class SampleXbarSUnequal extends JFrameChart {

    static final double diameter[][] = {
        {74.03, 74.002, 74.019, 73.992, 74.008},
        {73.995, 73.992, 74.001},
        {73.988, 74.024, 74.021, 74.005, 74.002},
        {74.002, 73.996, 73.993, 74.015, 74.009},
        {73.992, 74.007, 74.015, 73.989, 74.014},
```

```
        {74.009, 73.994, 73.997, 73.985},
        {73.995, 74.006, 73.994, 74},
        {73.985, 74.003, 73.993, 74.015, 73.988},
        {74.008, 73.995, 74.009, 74.005},
        {73.998, 74, 73.99, 74.007, 73.995},
        {73.994, 73.998, 73.994, 73.995, 73.99},
        {74.004, 74, 74.007, 74, 73.996},
        {73.983, 74.002, 73.998},
        {74.006, 73.967, 73.994, 74, 73.984},
        {74.012, 74.014, 73.998},
        {74, 73.984, 74.005, 73.998, 73.996},
        {73.994, 74.012, 73.986, 74.005},
        {74.006, 74.01, 74.018, 74.003, 74},
        {73.984, 74.002, 74.003, 74.005, 73.997},
        {74, 74.01, 74.013},
        {73.982, 74.001, 74.015, 74.005, 73.996},
        {74.004, 73.999, 73.99, 74.006, 74.009},
        {74.01, 73.989, 73.99, 74.009, 74.014},
        {74.015, 74.008, 73.993, 74, 74.01},
        {73.982, 73.984, 73.995, 74.017, 74.013}
    };

    public SampleXbarSUnequal() {
        Chart chart = getChart();
        ShewhartControlChart charts[] = XbarS.createCharts(chart, diameter);
        AxisXY axis = (AxisXY) (charts[0].getAxis());
        axis.getAxisY().setAutoscaleInput(Axis.AUTOSCALE_OFF);
        axis.getAxisY().setWindow(73.980, 74.020);
        String uclTitle = "ucl = {0,number,0.0000}";
        ((SChart) charts[1]).getUpperControlLimit().setTitle(uclTitle);
    }

    public static void main(String argv[]) {
        new SampleXbarSUnequal().setVisible(true);
    }
}
```


ucl = 74.015
74.02
74.01
74.00
73.99
73.98
Means
center = 74.001
lcl = 73.988
0
5
10
15
20
25
30
Sample Number
ucl = 0.0196
0.020
0.015
0.010
0.005
0.000
Standard Deviations
center = 0.009
lcl = 0
0
5
10
15
20
25
30
Sample Number

XbarR and RChart

If the sample sizes are small, say less than 10, then the in-sample ranges can be used instead of the in-sample standard deviations. The ***XbarR*** class plots the mean of each sample as well as control limits computed using the mean of the in-sample ranges. The positions of the control limits are determined by the equations

$$UCL = \bar{\bar{x}} + \frac{3}{d_2\sqrt{n}}\bar{R}$$

$$Centerline = \bar{\bar{x}}$$

$$LCL = \bar{\bar{x}} - \frac{3}{d_2\sqrt{n}}\bar{R}$$

where $\bar{\bar{x}}$ is the grand mean (the average of all observations), and d_2 is the mean of the distribution of the ranges of n samples from the normal distribution with mean of zero and standard deviation of one. The standard deviation of this distribution is d_3. Therefore

$$\frac{d_3}{d_2}\bar{R}$$

is an estimator of the standard deviation of the ranges.

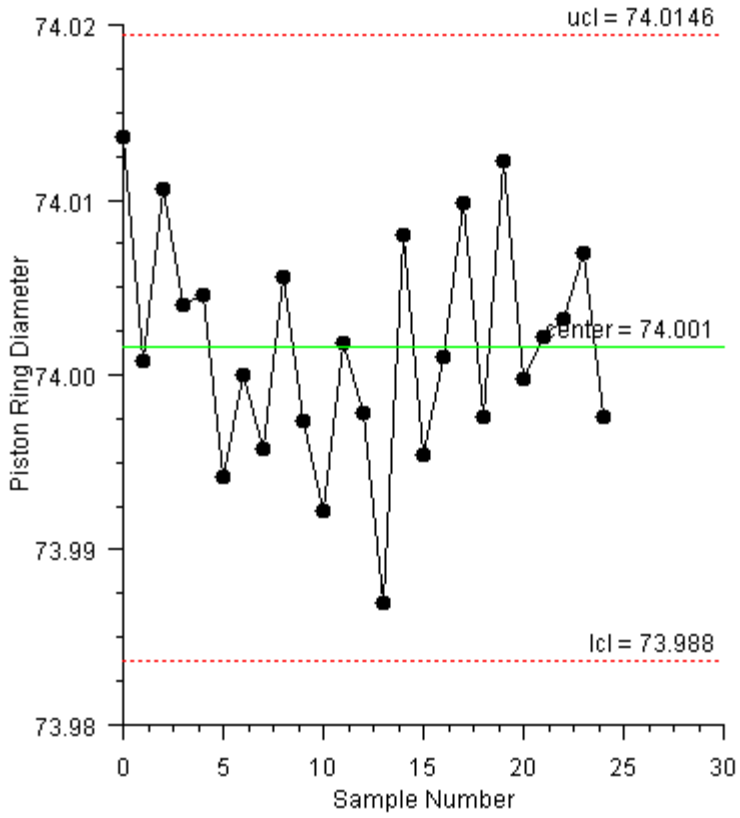
XbarR Example

This example creates an ***XbarR*** chart using the same piston ring data previously used for the ***XbarS*** chart. Here the in-sample ranges are used to compute the control limits, rather than the in-sample standard deviations (Montgomery 215).

View code file

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class SampleXbarR extends JFrameChart {
    public SampleXbarR() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        XbarR xbarr = new XbarR(axis, SampleXbarS.diameter);

        xbarr.getUpperControlLimit().setTitle("ucl = {0,number,0.0000}");

        axis.getAxisX().getAxisTitle().setTitle("Sample Number");
        axis.getAxisX().getAxisLabel().setTextFormat("0");
        axis.getAxisY().getAxisTitle().setTitle("Piston Ring Diameter");
        axis.getAxisY().setAutoscaleInput(axis.AUTOSCALE_OFF);
        axis.getAxisY().setWindow(73.985, 74.015);
    }
```

```
    public static void main(String argv[]) {
        new SampleXbarR().setVisible(true);
    }
}
```



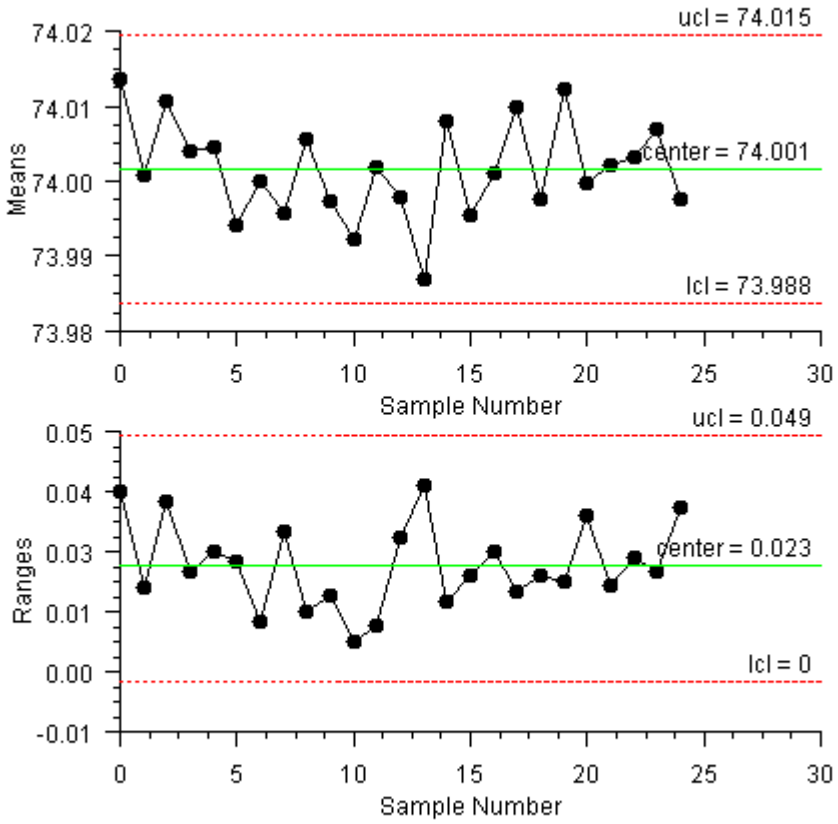
XbarRCombo Example

This example combines the ***XbarR*** chart with the corresponding ***RChart*** into a single chart. This is done by adjusting the `Viewport` attribute values for the two subcharts (Montgomery 215).

View code file

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class SampleXbarRCombo extends JFrameChart {

    public SampleXbarRCombo() {
        Chart chart = getChart();
```

```
        ShewhartControlChart charts[] =
            XbarR.createCharts(chart, SampleXbarS.diameter);
        AxisXY axis = (AxisXY)(charts[0].getAxis());
        axis.getAxisY().setAutoscaleInput(axis.AUTOSCALE_OFF);
        axis.getAxisY().setWindow(73.985, 74.015);
    }

    public static void main(String argv[]) {
        new SampleXbarRCombo().setVisible(true);
    }
}
```



XmR

XmR is a moving range chart. It is used when only samples of size one are available. The moving range statistic is

$$MR_i = |x_i - x_{i-1}|$$

The control limits are computed using

$$UCL = \bar{x} + 3\frac{\overline{MR}}{d_{2,2}}$$

$$Centerline = \bar{x}$$

$$LCL = \bar{x} - 3\frac{\overline{MR}}{d_{2,2}}$$

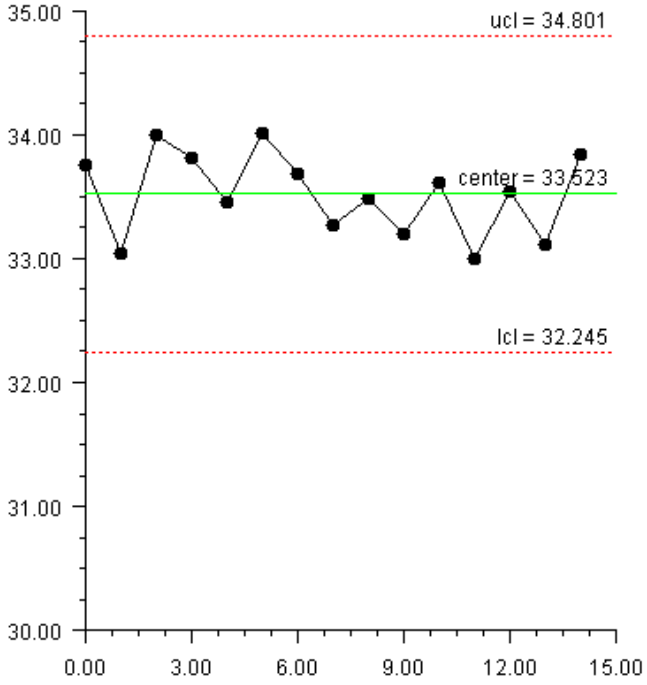
Where $\bar{x}$ is the mean of the observations, $\overline{MR}$ is the mean of the moving ranges and $d_{2,n}$ is the mean of the distribution of the ranges of n samples from the normal distribution with mean of zero and standard deviation of one.

XmR Example

The viscosity of aircraft primer paint was measured. Since the paint is produced in batches, only single samples are available (Montgomery 251).

View code file

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class SampleXmR extends JFrameChart {
    static final double viscosity[] = {
        33.75, 33.05, 34.00, 33.81, 33.46, 34.02, 33.68,
        33.27, 33.49, 33.20, 33.62, 33.00, 33.54, 33.12, 33.84
    };

    public SampleXmR() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        XmR xmr = new XmR(axis, viscosity);
    }

    public static void main(String argv[]) {
        new SampleXmR().setVisible(true);
    }
}
```


35.00
34.00
33.00
32.00
31.00
30.00
ucl = 34.801
center = 33.523
lcl = 32.245
0.00
3.00
6.00
9.00
12.00
15.00

PChart

The ***PChart*** plots the proportion or fraction of defective products.

The production process is assumed to be stable and successive units are assumed independent. So each unit produced is a realization of a Bernoulli random variable with parameter ***p***, the proportion defective.

The control limits are at

$$UCL = p + 3\sqrt{\frac{p(1-p)}{n}}$$

$$Centerline = p$$

$$LCL = p - 3\sqrt{\frac{p(1-p)}{n}}$$

By default, ***p*** is the proportion of defects observed in the data. To use a known ***p***, set the attribute `Center` to ***p***.

PChart Example

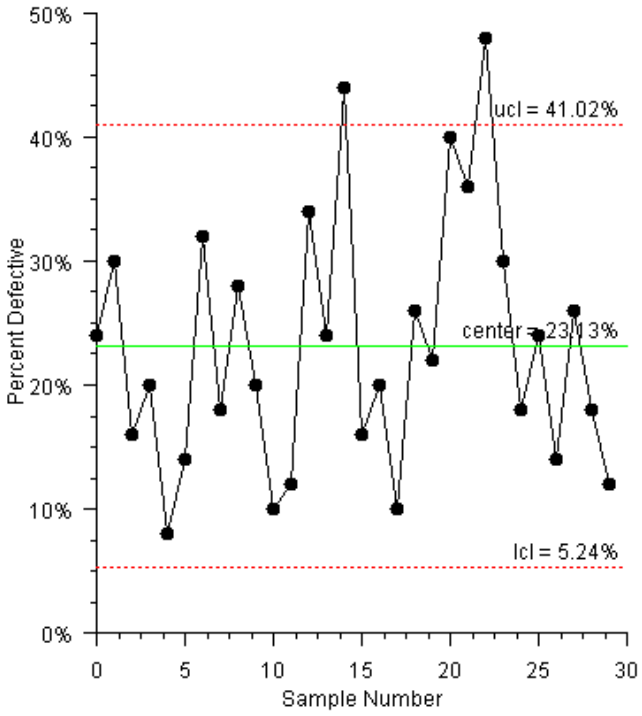
Defects in the manufacturing of orange juice cans are measured. The number of defects in samples of 50 cans is counted (Montgomery 290).

View code file

The *y*-axis labels and the title on the control limits have been changed to use a percentage format.

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class SamplePChart extends JFrameChart {
    static final int sampleSize = 50;
    static final int numberDefects[] = {
        12, 15, 8, 10, 4, 7, 16, 9, 14, 10, 5, 6, 17, 12, 22, 8,
        10, 5, 13, 11, 20, 18, 24, 15, 9, 12, 7, 13, 9, 6
    };

    public SamplePChart() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        PChart pchart = new PChart(axis, sampleSize, numberDefects);

        pchart.getLowerControlLimit().setTitle("lcl = {0,number,0.00%}");
        pchart.getCenterLine().setTitle("center = {0,number,0.00%}");
        pchart.getUpperControlLimit().setTitle("ucl = {0,number,0.00%}");

        axis.getAxisX().getAxisTitle().setTitle("Sample Number");
        axis.getAxisX().getAxisLabel().setTextFormat("0");
```

```
        axis.getAxisY().getAxisTitle().setTitle("Percent Defective");
        axis.getAxisY().getAxisLabel().setTextFormat("percent");
    }

    public static void main(String argv[]) {
        new SamplePChart().setVisible(true);
    }
}
```



PChartUnequal Example

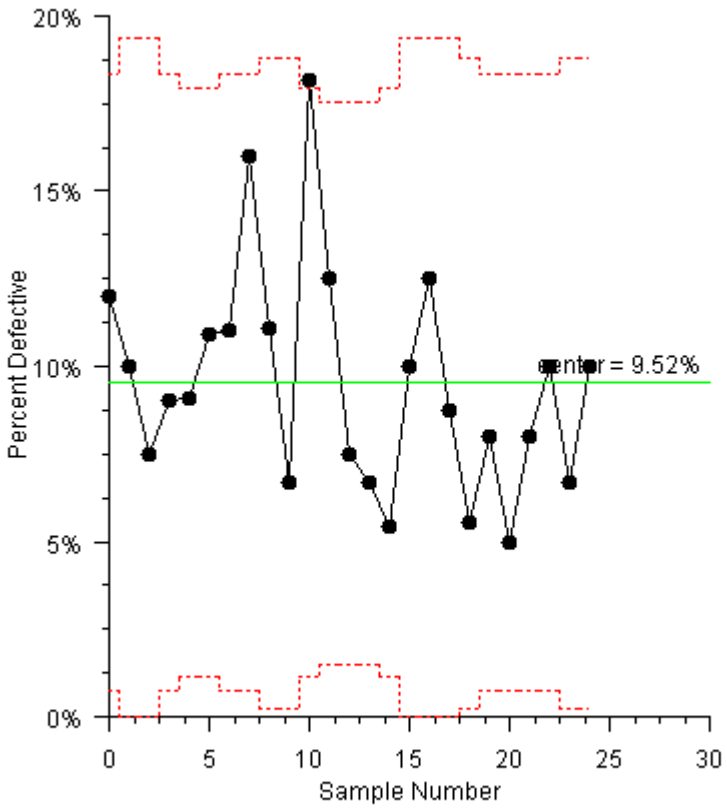
In this example a ***PChart*** is computed with unequal sample sizes. The upper and lower control limits are now stair step lines (Montgomery 300).

The *y*-axis labels and the title on the center line have been changed to use a percentage format.

View code file

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class SamplePChartUnequal extends JFrameChart {

    static final int sampleSize[] = {
```

```
        100, 80, 80, 100, 110, 110, 100, 100, 90, 90,
        110, 120, 120, 120, 110, 80, 80, 80, 90, 100,
        100, 100, 100, 90, 90
    };
    static final int numberDefects[] = {
        12, 8, 6, 9, 10, 12, 11, 16, 10, 6,
        20, 15, 9, 8, 6, 8, 10, 7, 5, 8, 5,
        8, 10, 6, 9
    };

    public SamplePChartUnequal() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        PChart pchart = new PChart(axis, sampleSize, numberDefects);

        pchart.getCenterLine().setTitle("center = {0,number,0.00%}");

        axis.getAxisX().getAxisTitle().setTitle("Sample Number");
        axis.getAxisX().getAxisLabel().setTextFormat("0");
        axis.getAxisY().getAxisTitle().setTitle("Percent Defective");
        axis.getAxisY().getAxisLabel().setTextFormat("percent");
        axis.getAxisY().setWindow(0.0, 0.2);
        axis.getAxisY().setAutoscaleInput(0);
    }

    public static void main(String argv[]) {
        new SamplePChartUnequal().setVisible(true);
    }
}
```


20%
15%
10%
5%
0%
Percent Defective
center = 9.52%
0
5
10
15
20
25
30
Sample Number

NpChart

NpChart is similar to ***PChart***, except that the number of defects per sample is plotted, not the sample rate. The position of the control limits are given by

$$UCL = np + 3\sqrt{np(1-p)}$$

$$Centerline = np$$

$$LCL = np - 3\sqrt{np(1-p)}$$

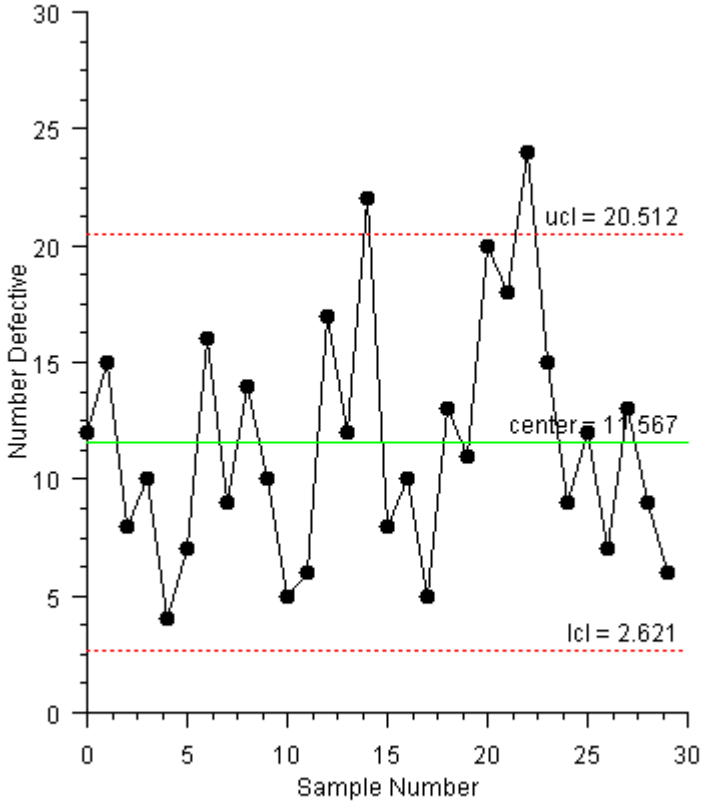
where ***n*** is the number of items and ***p*** is the proportion of defective items.

NpChart Example

View code file

This example uses the orange juice can manufacturing data used earlier in the `PChart` example. In this example, the number of defects, rather than the defect rate is plotted.

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class SampleNpChart extends JFrameChart {
    public SampleNpChart() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        NpChart npchart =
            new NpChart(axis, SamplePChart.sampleSize,
                              SamplePChart.numberDefects);

        axis.getAxisX().getAxisTitle().setTitle("Sample Number");
        axis.getAxisX().getAxisLabel().setTextFormat("0");
        axis.getAxisY().getAxisTitle().setTitle("Number Defective");
    }

    public static void main(String argv[]) {
        new SampleNpChart().setVisible(true);
    }
}
```


30
25
20
15
10
5
0
Number Defective
ucl = 20.512
center = 11.567
lcl = 2.621
0
5
10
15
20
25
30
Sample Number

CChart

CChart plots the number of defects or nonconformities.

$$UCL = \bar{c} + k\sqrt{\bar{c}}$$

$$Centerline = \bar{c}$$

$$LCL = \bar{c} - k\sqrt{\bar{c}}$$

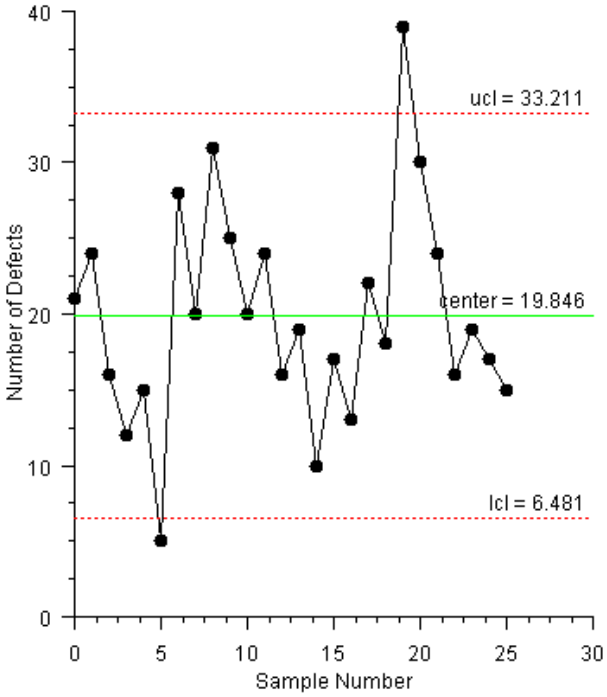
Where $\bar{c}$ is the mean number of defects per sample and ***k*** is the value of the `ControlLimit` attribute for the line.

CChart Example

The number of defects in samples of 100 printed circuit boards was measured (Montgomery 321).

View code file

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class SampleCChart extends JFrameChart {
    static final int numberDefects[] = {
        21, 24, 16, 12, 15, 5, 28, 20, 31, 25, 20, 24,
        16, 19, 10, 17, 13, 22, 18, 39, 30, 24, 16, 19,
        17, 15
    };

    public SampleCChart() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        CChart cchart = new CChart(axis, numberDefects);

        axis.getAxisX().getAxisTitle().setTitle("Sample Number");
        axis.getAxisX().getAxisLabel().setTextFormat("0");
        axis.getAxisY().getAxisTitle().setTitle("Number of Defects");
    }

    public static void main(String argv[]) {
        new SampleCChart().setVisible(true);
    }
}
```



CChartOmit Example

This is similar to the previous chart, but two points are omitted from the statistical computations.

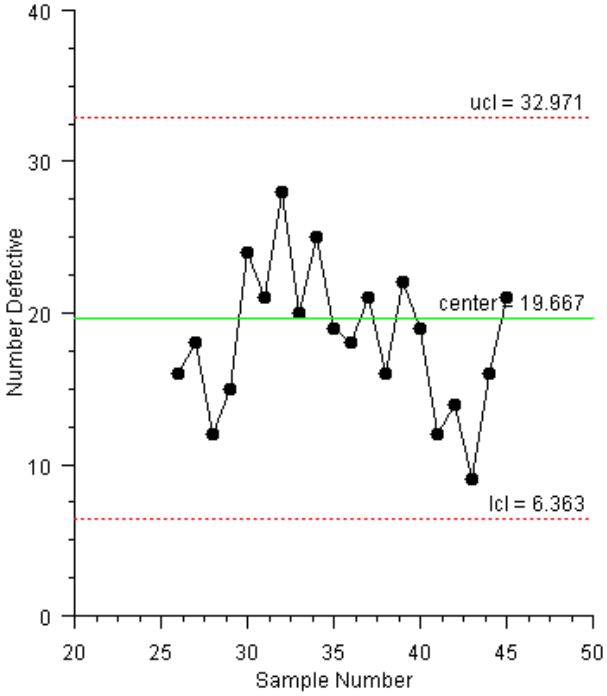
To do this, first the censored chart is created with the bad points omitted. The positions of the control limits in this chart are saved. This censored chart is then removed from the chart.

The second step is to create a chart using later data points, but with the position of the control limit limits set to the values computed using the censored data.

View code file

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class SampleCChartOmit extends JFrameChart {
    static final int censoredNumberDefects[] = {
        21, 24, 16, 12, 15, /*5,*/ 28, 20, 31, 25, 20, 24,
        16, 19, 10, 17, 13, 22, 18, /*39,*/ 30, 24, 16, 19,
        17, 15
    };

    static final int furtherNumberDefects[] = {
        16, 18, 12, 15, 24, 21, 28, 20, 25,
```

```
        19, 18, 21, 16, 22, 19, 12, 14, 9, 16, 21
    };

    public SampleCChartOmit() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);

        CChart censoredCChart = new CChart(axis, censoredNumberDefects);
        double lcl = censoredCChart.getLowerControlLimit().getValue()[0];
        double center = censoredCChart.getCenter();
        double ucl = censoredCChart.getUpperControlLimit().getValue()[0];
        censoredCChart.remove();

        double x[] = new double[furtherNumberDefects.length];
        for (int i = 0;  i < x.length;  i++) {
            x[i] = SampleCChart.numberDefects.length + i;
        }
        CChart fullCChart = new CChart(axis, furtherNumberDefects);
        fullCChart.getControlData().setX(x);
        fullCChart.getLowerControlLimit().setValue(lcl);
        fullCChart.getCenterLine().setValue(center);
        fullCChart.getUpperControlLimit().setValue(ucl);

        axis.getAxisX().getAxisTitle().setTitle("Sample Number");
        axis.getAxisX().getAxisLabel().setTextFormat("0");
        axis.getAxisY().getAxisTitle().setTitle("Number Defective");
    }


    public static void main(String argv[]) {
        new SampleCChartOmit().setVisible(true);
    }
}
```


ucl = 32.971
center = 19.667
lcl = 6.363
Number Defective
Sample Number
40
30
20
10
0
20
25
30
35
40
45
50

UChart

UChart is a chart for monitoring the defect rate when defects are rare.

$$UCL = \bar{u} + 3\sqrt{\frac{\bar{u}}{n}}$$

$$Centerline = \bar{u}$$

$$LCL = \bar{u} - 3\sqrt{\frac{\bar{u}}{n}}$$

Where $\bar{u}$ is the observed average number of defects per unit and n is the number of inspection units.

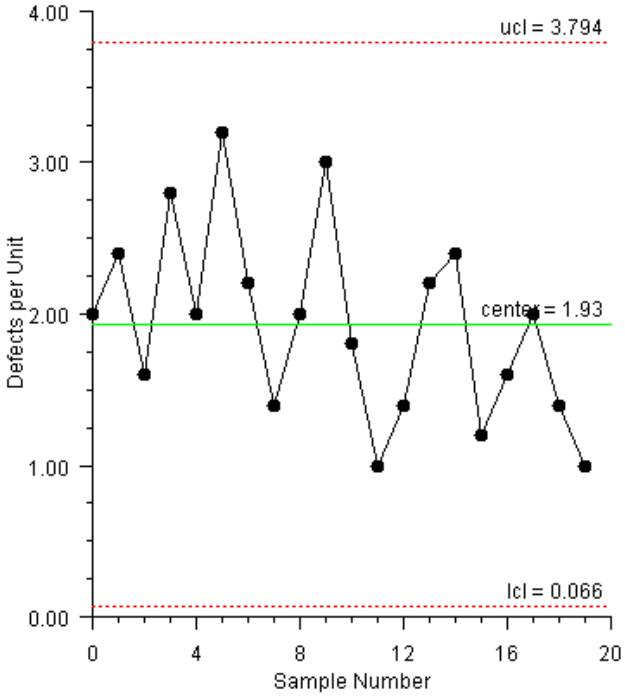
UChart Example

Samples Sizes are Equal

The number of defects in the manufacturing of computers was measured. There were 20 samples, each containing 5 computers. The center line is at 1.93, the average number of defects per computer (Montgomery 318).

View code file

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class SampleUChart extends JFrameChart {
    static final int sampleSize = 5;
    static final int numberDefects[] = {
        10, 12, 8, 14, 10, 16, 11, 7, 10, 15, 9, 5,
        7, 11, 12, 6, 8, 10, 7, 5
    };

    public SampleUChart() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        new UChart(axis, sampleSize, numberDefects);

        axis.getAxisX().getAxisTitle().setTitle("Sample Number");
        axis.getAxisX().getAxisLabel().setTextFormat("0");
        axis.getAxisY().getAxisTitle().setTitle("Defects per Unit");
    }


    public static void main(String argv[]) {
        new SampleUChart().setVisible(true);
    }
}
```



UChartUnequal Example

Unequal Sample Sizes

Defects in dying rolls of cloth were measured. The number of defects per 50 square meters was counted. Since the sizes of the rolls differed, the number of inspection units per roll varied. The center line is at 1.423, the average number of defects per 50 square meters (Montgomery, 321).

View code file

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class SampleUChartUnequal extends JFrameChart {
    static final double inspectionUnitsPerRoll[] = {
        10, 8, 13, 10, 9.5, 10, 12, 10.5, 12, 12.5
    };
    static final int numberDefects[] = {
        14, 12, 20, 11, 7, 10, 21, 16, 19, 23
    };

    public SampleUChartUnequal() {
        Chart chart = getChart();
```

```
        AxisXY axis = new AxisXY(chart);
        new UChart(axis, inspectionUnitsPerRoll, numberDefects);

        axis.getAxisX().getAxisTitle().setTitle("Sample Number");
        axis.getAxisX().getAxisLabel().setTextFormat("0");
        axis.getAxisY().getAxisTitle().setTitle("Defects per Unit");
    }


    public static void main(String argv[]) {
        new SampleUChartUnequal().setVisible(true);
    }
}
```

EWMA

EWMA is the exponentially weighted moving average control chart. It is very effective in detecting small process shifts, but is slower than Shewhart charts at detecting large process shifts.

The exponentially weighted moving average is defined to be

$$z_i = \lambda x_i + \left(1 - \lambda\right) z_{i-1}$$

where $0 < \lambda \leq 1$ is a constant and the starting value is μ_0, the expected mean.

The control limits in EWMA are defined by the equations:

$$UCL = \mu_0 + 3\sigma\sqrt{\frac{\lambda}{2-\lambda}}\left[1 - \left(1-\lambda\right)^{2i}\right]$$

$$Centerline = \mu_0$$

$$LCL = \mu_0 - 3\sigma\sqrt{\frac{\lambda}{2-\lambda}}\left[1 - \left(1-\lambda\right)^{2i}\right]$$

Where μ_0 is the historical mean, σ is the historical standard deviation, and *i* is the sample number. Because of the presence of *i* the control limits are stair steps, not constants.

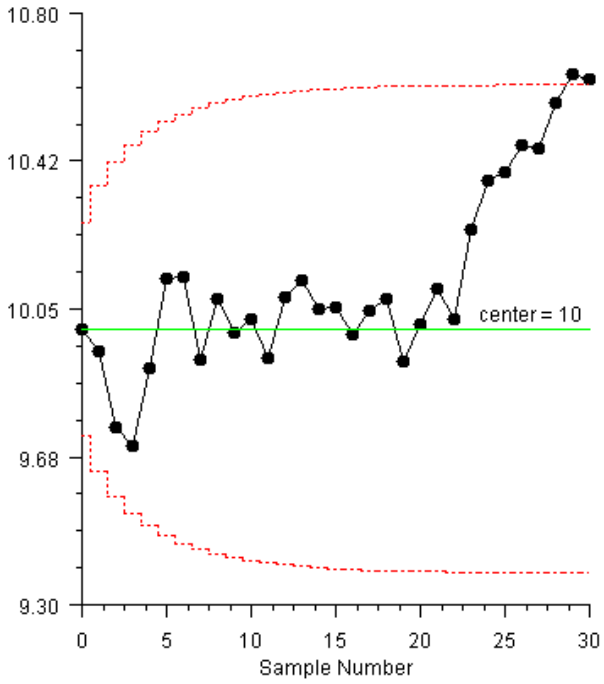
EWMA Example

The expected mean is 10., the expected standard deviation is 1.0 and λ is 0.10. Also, the control limit values are 2.7, not the default value of 3 (Montgomery 428).

View code file

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class SampleEWMA extends JFrameChart {

    static final double data[] = {
        9.45, 7.99, 9.29, 11.66, 12.16, 10.18, 8.04, 11.46,
        9.20, 10.34, 9.03, 11.47, 10.51, 9.40, 10.08, 9.37,
        10.62, 10.31, 8.52, 10.84, 10.90, 9.33, 12.29, 11.50,
        10.60, 11.08, 10.38, 11.62, 11.31, 10.52
    };

    public SampleEWMA() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        double lambda = 0.10;
        double mean = 10.0;
        double stdev = 1.0;
```

```
        EWMA ewma = new EWMA(axis, data, lambda, mean, stdev);
        ewma.getLowerControlLimit().setControlLimit(-2.7);
        ewma.getUpperControlLimit().setControlLimit(2.7);

        axis.getAxisX().getAxisTitle().setTitle("Sample Number");
        axis.getAxisX().getAxisLabel().setTextFormat("0");
        axis.getAxisY().setWindow(9.3, 10.8);
        axis.getAxisY().setAutoscaleInput(0);
    }

    public static void main(String argv[]) {
        new SampleEWMA().setVisible(true);
    }
}
```



CuSum

CuSum is the cumulative sum control chart. It plots the cumulative sum of the deviations of the expected value. If μ_0 is the expected mean for a process and $\bar{x}_i$ are the sample means then the cumulative sum is

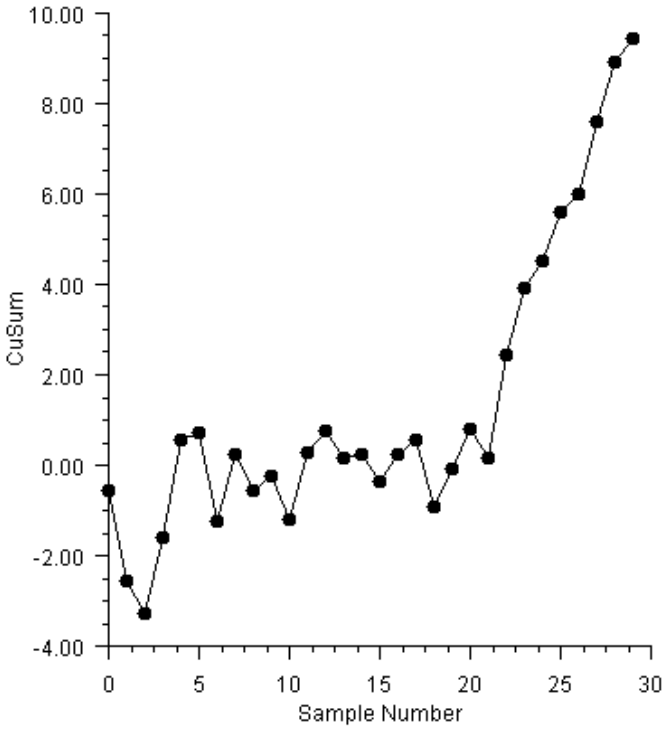
$$C_i = C_{i-1} + \left(\bar{x}_i - \mu_0\right)$$

CuSum Example

The data used is the same as for the EWMA example.

View code file

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class SampleCuSum extends JFrameChart {
    public SampleCuSum() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        CuSum cusum = new CuSum(axis, SampleEWMA.data);
        cusum.setExpectedMean(10.0);

        axis.getAxisX().getAxisTitle().setTitle("Sample Number");
        axis.getAxisX().getAxisLabel().setTextFormat("0");
        axis.getAxisY().getAxisTitle().setTitle("CuSum");
    }

    public static void main(String argv[]) {
        new SampleCuSum().setVisible(true);
    }
}
```


10.00
8.00
6.00
4.00
2.00
0.00
-2.00
-4.00
CuSum
0
5
10
15
20
25
30
Sample Number

CuSumStatus

CuSumStatus is a tabular or status `CuSum` chart. The tabular ***CuSum*** statistics are

$$C_i^+ = \max\left(0, x_i - \left(\mu_0 + K\right) + C_{i-1}^+\right)$$

$$C_i^- = \max\left(0, \left(\mu_0 + K\right) - x_i + C_{i-1}^-\right)$$

By default, both statistics have initial value zero. The parameter *K* is the slack value (or allowance or reference value) and μ_0 is the expected mean.

The `CuSumStatus` chart contains two bar charts: a bar chart of C_i^+ above the *x*-axis and a bar chart of C_i^- below the *x*-axis. There are also control limits at plus and minus *H*. The value of *H* can be set either as an absolute value or as a relative value *h*. They are related by $H = h\sigma$, where σ is the standard deviation. By default, bars which are out-of-control are filled red while in-control bars are green. The data is also plotted on the chart.

The `CuSumStatus` has a print method to print the C_i^+ and C_i^- values as well as N_i^+ and N_i^-, where N_i^+ is the number of consecutive periods since C_i^+ rose above zero.

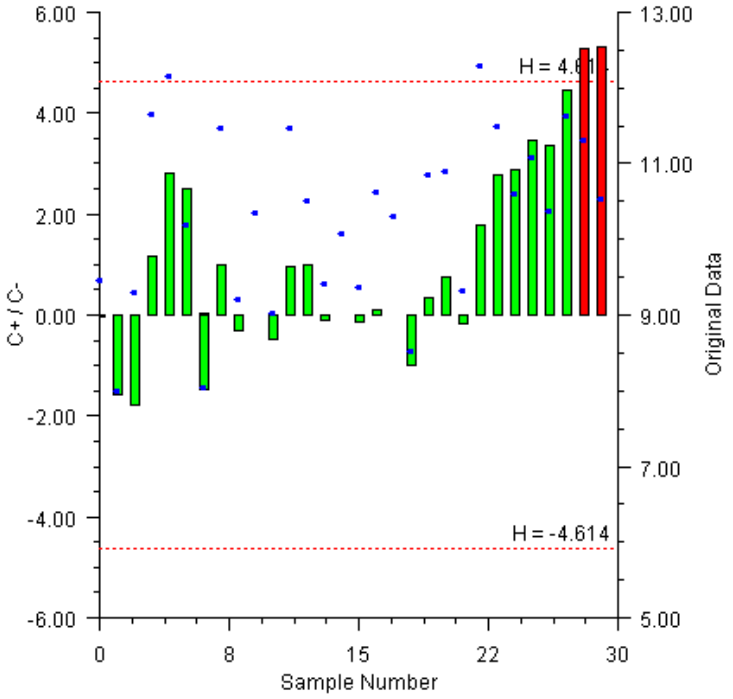
CuSumStatus Example

This example uses the same data as used for the `CuSum` and `EWMA` examples. In this example $H = 4\sigma$:

View code file

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class SampleCuSumStatus extends JFrameChart {

    public SampleCuSumStatus() {
        double expectedMean = 10;
        double slackValue = 0.5;

        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        CuSumStatus cusumStatus = new CuSumStatus(axis,
                SampleEWMA.data, expectedMean, slackValue);
        cusumStatus.setRelativeH(4);
        cusumStatus.print();

        axis.getAxisX().getAxisTitle().setTitle("Sample Number");
        axis.getAxisX().getAxisLabel().setTextFormat("0");
        axis.getAxisY().getAxisTitle().setTitle("C+ / C-");

        axis.getAxisX().setWindow(0, 30);
```

```
        axis.getAxisX().setAutoscaleInput(0);

        cusumStatus.addDataMarkers();
        cusumStatus.getDataMarkers().setMarkerSize(0.5);
        cusumStatus.getDataMarkers().setMarkerColor(java.awt.Color.blue);
        Axis1D axisY = cusumStatus.getDataMarkersAxis().getAxisY();
        axisY.getAxisTitle().setTitle("Original Data");
    }

    public static void main(String argv[]) {
        new SampleCuSumStatus().setVisible(true);
    }
}
```



Pareto Chart

A ***ParetoChart*** is a frequency distribution of attribute data. The bars are ordered by the number of defects, with the attribute category having the largest number of defects appearing first.

There is an option to add a cumulative percentage line to the chart. This is the percent of defects accounted for by the current item and all items to its left. If the cumulative percentage line is added, a second axis is created on the right representing the cumulative percentage from 0% to 100%. The units of the original axis, which always appear on the left, represent the number of defects, 36.

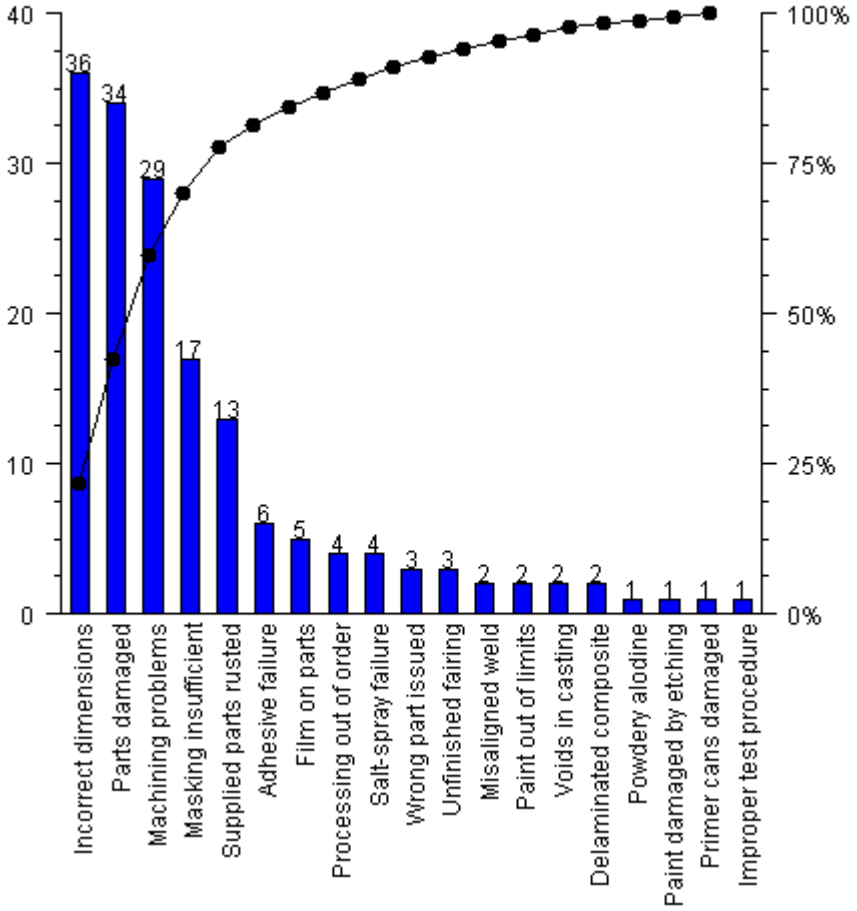
ParetoChart Example

The number of defects attributable to 1 of 19 of a variety of causes was collected (Montgomery 179). In this example, the "Incorrect dimensions" defect category has the largest number of defects, 36.

View code file

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class SampleParetoChart extends JFrameChart {
    static final String labels[] = {
        "Parts damaged",
        "Machining problems",
        "Supplied parts rusted",
        "Masking insufficient",
        "Misaligned weld",
        "Processing out of order",
        "Wrong part issued",
        "Unfinished fairing",
        "Adhesive failure",
        "Powdery alodine",
        "Paint out of limits",
        "Paint damaged by etching",
        "Film on parts",
        "Primer cans damaged",
        "Voids in casting",
        "Delaminated composite",
        "Incorrect dimensions",
        "Improper test procedure",
        "Salt-spray failure"
    };
    static final int numberDefects[] = {
        34, 29, 13, 17, 2, 4, 3, 3, 6,
        1, 2, 1, 5, 1, 2, 2, 36, 1, 4
    };

    public SampleParetoChart() {
```

```
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        ParetoChart pareto = new ParetoChart(axis, labels, numberDefects);

        Data cumulativeLine = pareto.addCumulativeLine();
        cumulativeLine.setMarkerType(Data.MARKER_TYPE_FILLED_CIRCLE);

        pareto.setLabelType(pareto.LABEL_TYPE_Y);
        pareto.setTextFormat("0");
        pareto.setFillColor(java.awt.Color.blue);
        axis.getAxisX().getAxisLabel().setTextAngle(90);

        double vp[] = axis.getViewport();
        vp[0] = 0.1;
        vp[3] = 0.7;
        axis.setViewport(vp);
        cumulativeLine.getAxis().setViewport(vp);
    }


    public static void main(String argv[]) {
        new SampleParetoChart().setVisible(true);
    }
}
```


40
30
20
10
0
100%
75%
50%
25%
0%
36
34
29
17
13
6
5
4
4
3
3
2
2
2
2
1
1
1
1
Incorrect dimensions
Parts damaged
Machining problems
Masking insufficient
Supplied parts rusted
Adhesive failure
Film on parts
Processing out of order
Salt-spray failure
Wrong part issued
Unfinished fairing
Misaligned weld
Paint out of limits
Voids in casting
Delaminated composite
Powdery alodine
Paint damaged by etching
Primer cans damaged
Improper test procedure

Cumulative Probability

If the defect rate is very small there will be long runs when the number of defects is zero. In this situation the ***CChart*** and ***UChart*** are ineffective. An alternative to defect counts is to measure the time between defects.

If the distribution of defect occurrences is Poisson, then the distribution of times between defects is exponential. Unfortunately the exponential distribution is highly skewed. Nelson suggested transforming to Weibull random variables using the transformation $x = y^{1/3.6}$.

CumulativeProbability Example

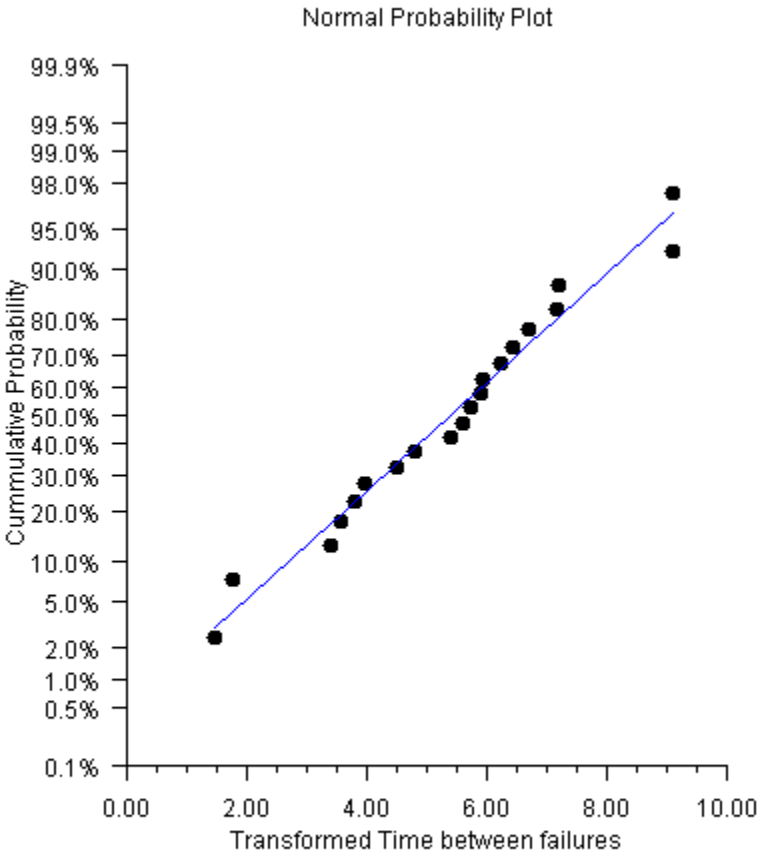
The number of hours between failures is measured. A normal probability plot is constructed from the transformed data. A linear regression to the transformed data is also computed. To fit a regression with the normal probability axis, the regression variable needs to be transformed using the inverse normal cumulative distribution function (Montgomery, Example 6-21, 327).

View code file

```
import com.imsl.chart.*;
import com.imsl.stat.Cdf;
import com.imsl.stat.InvCdf;
import com.imsl.stat.Sort;
import com.imsl.stat.LinearRegression;

public class SampleCumulativeProbability extends JFrameChart {

    static final double timeBetweenFailures[] = {
        286, 948, 536, 124, 816, 729, 4, 143, 431, 8, 2837,
        596, 81, 227, 603, 492, 1199, 1214, 2831, 96
    };

    static final double ticks[] = {
        0.001, 0.005, 0.01, 0.02, 0.05,
        0.10, 0.20, 0.30, 0.40, 0.50, 0.60, 0.70, 0.80, 0.90,
        0.95, 0.98, 0.99, 0.995, 0.999
    };

    public SampleCumulativeProbability() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        double a = ticks[0];
        double b = ticks[ticks.length - 1];
        chart.getChartTitle().setTitle("Normal Probability Plot");
        Axis1D axisX = axis.getAxisX();
        axisX.getAxisTitle().setTitle("Transformed Time between failures");
        Axis1D axisY = axis.getAxisY();
        axisY.getAxisTitle().setTitle("Cummulative Probability");
        axisY.setTransform(axis.TRANSFORM_CUSTOM);
```

```
        Transform transform = new NormalTransform();
        axis.getAxisY().setCustomTransform(transform);
        axis.getAxisY().setAutoscaleInput(axis.AUTOSCALE_OFF);
        axis.getAxisY().setWindow(a, b);
        axis.getAxisY().setTextFormat("0.0%");
        axis.getAxisY().setTicks(ticks);
        axis.getAxisY().getMinorTick().setPaint(false);

        int n = timeBetweenFailures.length;
        double x[] = new double[n];
        double y[] = new double[n];
        for (int i = 0; i < x.length; i++) {
            x[i] = Math.pow(timeBetweenFailures[i], 1.0 / 3.6);
        }
        Sort.ascending(x);
        for (int i = 0; i < x.length; i++) {
            y[i] = (double) (i + 0.5) / (double) n;
        }
        Data data = new Data(axis, x, y);
        data.setDataType(Data.DATA_TYPE_MARKER);
        data.setMarkerType(Data.MARKER_TYPE_FILLED_CIRCLE);

        /*
         * Compute an plot a regresssion line
         */
        LinearRegression lr = new LinearRegression(1, true);
        for (int i = 0; i < n; i++) {
            lr.update(new double[]{x[i]}, InvCdf.normal(y[i]));
        }
        double coef[] = lr.getCoefficients();
        double lry[] = new double[x.length];
        for (int i = 0; i < x.length; i++) {
            lry[i] = Cdf.normal(coef[0] + coef[1] * x[i]);
        }
        Data lrData = new Data(axis, x, lry);
        lrData.setDataType(Data.DATA_TYPE_LINE);
        lrData.setLineColor(java.awt.Color.blue);
    }

    static class NormalTransform implements Transform {

        private double scaleA, scaleB;

        /**
         * Initializes the mappings between user and coordinate space.
         */
        public void setupMapping(Axis1D axis1d) {
            double w[] = axis1d.getWindow();
            double t = InvCdf.normal(w[0]);
            scaleB = 1.0 / (InvCdf.normal(w[1]) - t);
            scaleA = -scaleB * t;
```

```
        }

        /**
         * Maps a point in [0,1] to a probability.
         */
        public double mapUnitToUser(double unit) {
            return Cdf.normal((unit - scaleA) / scaleB);
        }

        /**
         * Maps a probability to the interval [0,1].
         */
        public double mapUserToUnit(double p) {
            return scaleA + scaleB * InvCdf.normal(p);
        }
    }

    public static void main(String argv[]) {
        new SampleCumulativeProbability().setVisible(true);
    }
}
```



Chapter 4 2D Drawing Elements

JMSL provides these Chart2D drawing elements:

Line Attributes

All lines, except for those used to draw markers or outline filled regions, are affected by the attributes described in this section. These include axis lines and lines drawn when a ***Data*** node is rendered with its `DataType` attribute having its `DATA_TYPE_LINE` bit set.

Attribute LineColor

`LineColor` is a ***Color***-valued attribute that determines the color of the line. Its default value is `Color.black`.

Attribute LineWidth

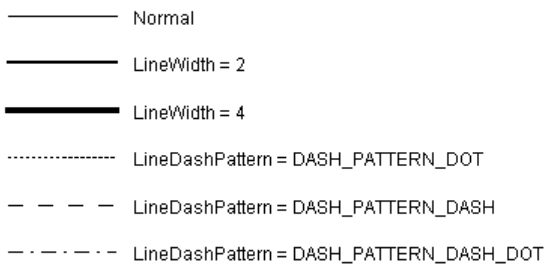
`LineWidth` is a double-valued attribute that determines the thickness of the lines. Its default value is 1.0.

Attribute LineDashPattern

`LineDashPattern` is a double-array-valued attribute that determines the line pattern used to draw the line. It defaults to a solid line. Alternate entries in the array represent the lengths of the opaque and transparent segments of the dashes.

Some dash patterns are defined. They are `DASH_PATTERN_DOT`, `DASH_PATTERN_DASH` and `DASH_PATTERN_DASH_DOT`.

Samples



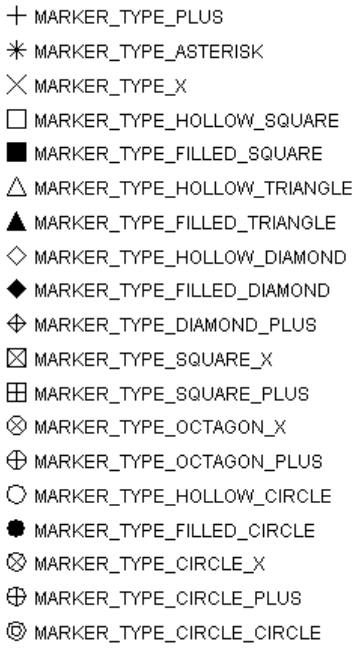
Marker Attributes

Markers are drawn when a ***Data*** node is rendered with its `DataType` attribute having its `DATA_TYPE_MARKER` bit set. Drawing of markers is affected by the attributes described in this section. Note that even though some markers are drawn using lines, the line attributes do not apply to markers.

An alternative to markers are images, which can be used to draw arbitrary symbols instead of markers.

MarkerType

`MarkerType` is an integer-valued attribute that determines which marker will be drawn. There are constants defined in ***ChartNode*** for the marker types. The default value is `MARKER_TYPE_PLUS`. The following are all of the defined marker types. For clarity, these are drawn larger than normal.



Attribute MarkerColor

`MarkerColor` is a ***Color***-valued attribute that determines the color of the marker. Its default value is `Color.black`.

Attribute MarkerSize

`MarkerSize` is a double-valued attribute that determines the size of the marker. Its default value is 1.0. The actual size of the drawn marker, in pixels, is 0.007**MarkerSize*width*, where *width* is the width of the ***Component*** containing the chart.

Attribute

`MarkerThickness` is a double-valued attribute that determines the thickness of the lines used to draw the marker. Its default value is 1.0.

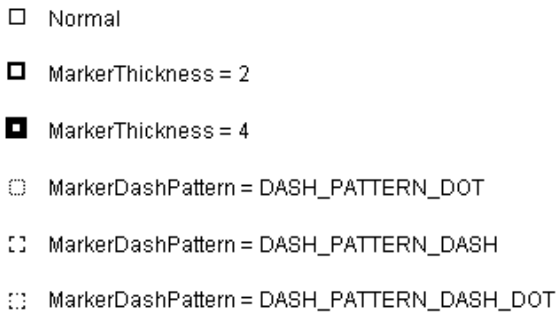
Attribute MarkerDashPattern

`MarkerDashPattern` is a double-array-valued attribute that determines the line pattern used to draw the marker. It defaults to a solid line. Alternate entries in the array represent the lengths of the opaque and transparent segments of the dashes.

Some dash patterns are defined. They are `DASH_PATTERN_DOT`, `DASH_PATTERN_DASH` and `DASH_PATTERN_DASH_DOT`.

> **NOTE**: `MarkerDashPattern` is a double-array-valued attribute that determines the line pattern used to draw the marker. It defaults to a solid line. Alternate entries in the array represent the lengths of the opaque and transparent segments of the dashes. These patterns will be more recognizable with larger marker sizes.

Samples



Fill Area Attributes

FillOutlineType

`FillOutlineType` is an integer-value attribute that turns on or off the drawing of an outline around filled areas. Its value should be `FILL_TYPE_NONE` (for outline off) or `FILL_TYPE_SOLID` (for outline on). The default is to draw solid lines.

FillOutlineColor

`FillOutlineColor` is a `Color`-valued attribute that determines the color used to outline the filled regions. The outline is drawn only if the attribute `FillOutlineType` has the value `FILL_TYPE_SOLID`. Its default value is `Color.black`.

FillType

FillType is an integer-value attribute that turns on or off the drawing of the interior of filled areas. Its value should be

- `FILL_TYPE_NONE` for fill off.
- `FILL_TYPE_SOLID` for fill by a single, solid color. This is the default.
- `FILL_TYPE_GRADIENT` for fill by a color gradient.
- `FILL_TYPE_PAINT` for fill by a `Paint` object. This is usually a `TexturePaint` object.

FillColor

`FillColor` is a `Color`-valued attribute that determines the color used to fill a region with a solid color. Its default value is `Color.black`.

Gradient

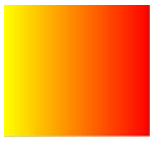
`Gradient` is a ***Color*** array-valued attribute that fills a region with a gradient color. It does not have a default value.

The value of `Gradient` should be an array of four colors. These are the colors at the four corners of a square. In order they are: lower-left, lower-right, upper-right and upper-left.

- If the lower-side colors are equal (`color[0]` equals `color[1]`) and the upper-side colors are equal (`color[2]` equals `color[3]`), then a vertical gradient is drawn.
- If the left-side colors are equal (`color[0]` equals `color[3]`) and the right-side colors are equal (`color[1]` equals `color[2]`), then a horizontal gradient is drawn.

- If the lower-right and upper-left colors (`color[1]` and `color[3]`) are null, then a diagonal gradient is drawn using the lower-left (`color[0]`) and upper-right (`color[2]`) colors.
- If the lower-left and upper-right colors (`color[0]` and `color[2]`) are null, then a diagonal gradient is drawn using the lower-right (`color[1]`) and upper-left (`color[3]`) colors.

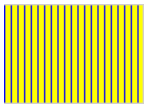
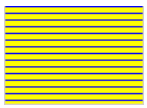
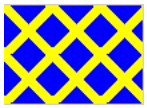
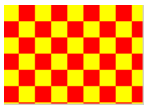
If none of the above patterns exist, then no gradient is drawn.

Vertical		setGradient(Color.red, Color.red, Color.yellow, Color.yellow)
Horizontal		setGradient(Color.yellow, Color.red, Color.red, Color.yellow)
Diagonal		setGradient(Color.yellow, null, Color.red, null)
Diagonal		setGradient(null, Color.yellow, null, Color.red)

Fill Paint

`FillPaint` is a ***Paint***-valued attribute that fills a region with a tiled pattern. It does not have a default value. The class ***FillPaint*** contains utilities to define some useful paint patterns.

Some Examples of FillPaint

	`FillPaint.verticalStripe(10, 5, Color.yellow,Color.blue)`
	`FillPaint.horizontalStripe(10, 5, Color.yellow,Color.blue)`
	`FillPaint.diamond(36, 5, Color.blue,Color.yellow)`
	`FillPaint.checkerboard(24, Color.red,Color.yellow)`

The `FillPaint` attribute can also be set using an ***Image***, which is used to tile filled regions.

Text Attributes

Attribute Font

Text is drawn using ***Font*** objects constructed using the `FontName`, `FontSize` and `FontStyle` attributes. The `AbstractChartNode.setFont(Font)` method does not save the `Font` object, but sets these three attributes.

This arrangement allows one to specify font size and style at lower nodes and change the font face at the root node.

Multiline text strings are allowed. They are created by newline characters in the string that creates the text item.

Attribute FontName

`FontName` is a string-valued attribute that specifies the logical font name or a font face name. Its default value is SansSerif. Java, always defines the font names Dialog, DialogInput, Monospaced, Serif, SansSerif, and Symbol. Depending on the system, additional font names may be defined.

Attribute FontSize

`FontSize` is an integer-valued attribute that specifies the point size of the font. Its default value is 12.

Attribute FontStyle

`FontStyle` is an integer-valued attribute that specifies the font style. This can be a bitwise combination of `Font.PLAIN`, `Font.BOLD` and/or `Font.ITALIC`. Its default value is `Font.PLAIN`.

Samples

Following are sample fonts in styles plain, italic and bold. These may look different on different platforms.

Dialog	*Dialog*	**Dialog**
DialogInput	*DialogInput*	**DialogInput**
Monospaced	*Monospaced*	**Monospaced**
Serif	*Serif*	**Serif**
SansSerif	*SansSerif*	**SansSerif**

Attribute TextColor

`TextColor` is a ***Color***-valued attribute that specifies the color in which the text is drawn. Its default value is `Color.black`.

Attribute TextFormat

`TextFormat` is a ***Format***-valued or a ***String***-valued attribute that specifies how to format string objects

`TextFormat` can also be set using a `String`, which is used to generate a `Format` object when the attribute is accessed. The `Format` object is created using the value of locale in the chart node.

Some strings have special meanings (case is ignored in these strings):

- `"Date(SHORT)"` means use:
  `DateFormat.getDateInstance`(`SHORT`, ***Locale***)
- `"Date(MEDIUM)"` means use
  `DateFormat.getDateInstance`(`MEDIUM`, ***Locale***)
- `"Date(LONG)"` means use:
  `DateFormat.getDateInstance`(`LONG`, ***Locale***)
- `"Currency"` means use
  `NumberFormat.getCurrencyInstance`(***Locale***)
- `"Percent"` means use:
  `NumberFormat.getPercentInstance`(***Locale***)

If `TextFormat` has a string value that is not one of the above, then

`new DecimalFormat` (*value*, `new DecimalFormatSymbols(`***Locale***))

is used. For example, if its value is a `String "0.00"`, then numbers will be formatted with exactly two digits after the decimal place. See ***DecimalFormat*** for a detailed description of these format patterns.

`The default value of TextFormat` is the value returned by the factory method `NumberFormat.getInstance`(***Locale***).

Attribute Title

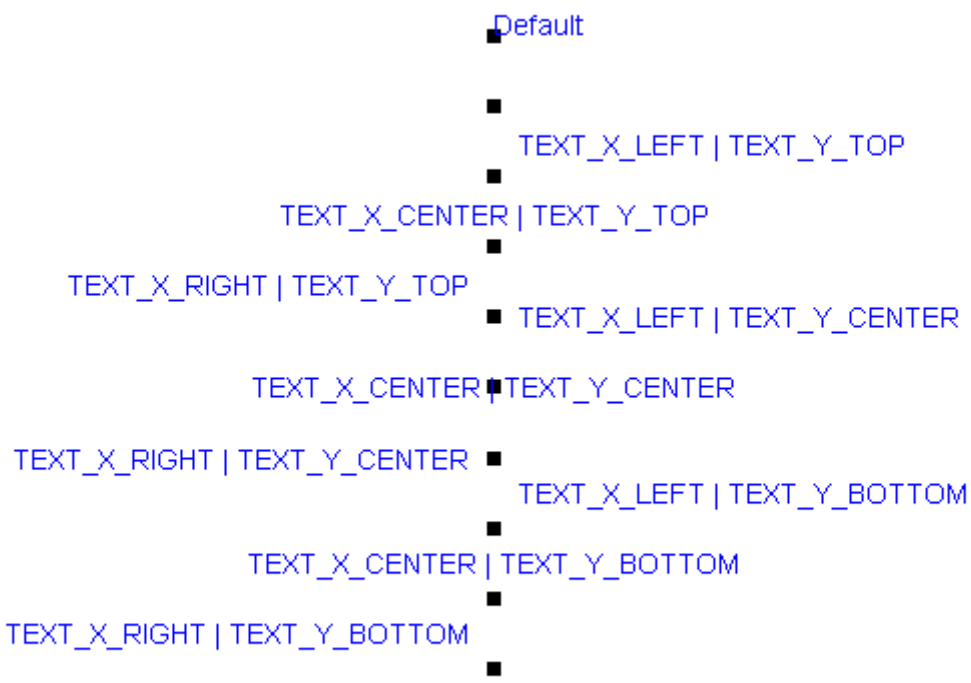
`Title` is a ***Text***-valued attribute that contains the title for a node. The class `Text` holds a string and its alignment and offset information.

The alignment of a `Text` object is the bitwise combination of one of

- `TEXT_X_LEFT`, `TEXT_X_CENTER`, `TEXT_X_RIGHT`, and one of
- `TEXT_Y_BOTTOM`, `TEXT_Y_CENTER`, `TEXT_Y_TOP`.

The offset moves the start of the text away from the reference point in the direction of the alignment. So if the alignment bit TEXT_X_LEFT is set and the offset is greater than zero then the text starts a distance further to the left than if the offset were zero. The distance moved is the value of offset times the default marker size. The offset is usually zero, but the ***Data*** node sets it to 2.0 for labeling data points.

A `Text` object is drawn relative to a reference point. The alignment specifies the position of the reference point on the box that contains the text. There are nine such possible positions. In the following samples, the reference point is marked with a square.



If the text is drawn at an angle, then the alignment is relative to the horizontally/vertically aligned bounding box of the text.

Labels

Labels annotate data points or pie slices. As a degenerate case they can be used to place text on a chart without drawing a point. Labels are controlled by the value of the attribute `LabelType` in a ***Data*** node.

Multiline labels are allowed. They are created by newline characters in the string that creates the label.

Attribute LabelType

The attribute `LabelType` takes on one of the following values:

- `LABEL_TYPE_NONE` for no label. This is the default.
- `LABEL_TYPE_X` label with the value of the ***x***-coordinate.
- `LABEL_TYPE_Y` label with the value of the y-coordinate.
- `LABEL_TYPE_TITLE` label with the value of the `Title` attribute.
- `LABEL_TYPE_PERCENT` label with the percentage value. This applies only to `PieSlice` objects.

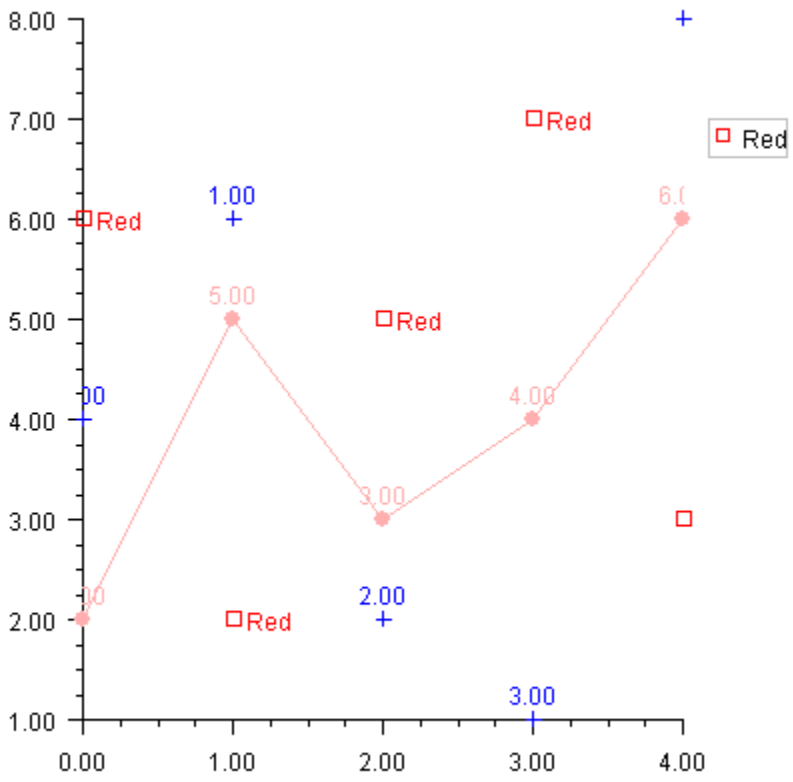
Data Point Labeling Example

This example shows three data sets, each labeled in a different fashion.

The first data set is blue and the points are labeled by the value of their ***x***-coordinates.

The second data set is pink and the points are labeled by the value of their ***y***-coordinates. The data set is drawn with both markers and lines.

The third data set is red and the points are labeled with the ***Data*** node's `Title` attribute.



View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SampleLabels extends JFrameChart {

    public SampleLabels() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        chart.getLegend().setPaint(true);

        // Data set 1 - labeled with X
        double y1[] = {4, 6, 2, 1, 8};
        Data data1 = new Data(axis, y1);
        data1.setDataType(Data.DATA_TYPE_MARKER);
        data1.setMarkerColor(Color.blue);
        data1.setTextColor(Color.blue);
        data1.setLabelType(Data.LABEL_TYPE_X);

        // Data set 2 - labeled with Y
        double y2[] = {2, 5, 3, 4, 6};
        Data data2 = new Data(axis, y2);
        data2.setDataType(Data.DATA_TYPE_MARKER | Data.DATA_TYPE_LINE);
```

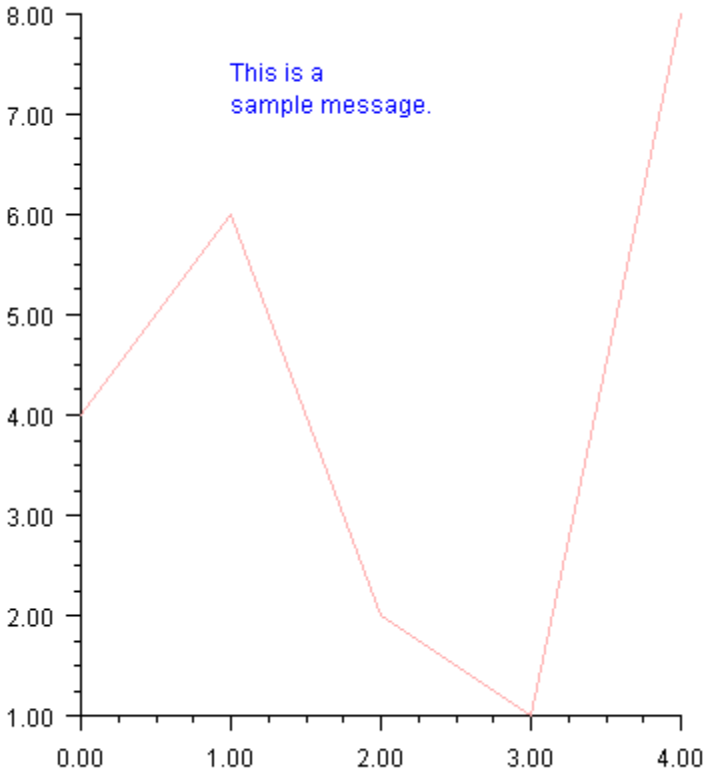
```
        data2.setMarkerColor(Color.pink);
        data2.setMarkerType(Data.MARKER_TYPE_FILLED_CIRCLE);
        data2.setLineColor(Color.pink);
        data2.setTextColor(Color.pink);
        data2.setLabelType(Data.LABEL_TYPE_Y);

        // Data set 3 - labeled with Title
        double y3[] = {6, 2, 5, 7, 3};
        Data data3 = new Data(axis, y3);
        data3.setDataType(Data.DATA_TYPE_MARKER);
        data3.setMarkerColor(Color.red);
        data3.setMarkerType(Data.MARKER_TYPE_HOLLOW_SQUARE);
        data3.setLineColor(Color.red);
        data3.setTextColor(Color.red);
        data3.setLabelType(Data.LABEL_TYPE_TITLE);
        data3.setTitle("Red");
        data3.getTitle().setAlignment(
                Data.TEXT_X_LEFT | Data.TEXT_Y_CENTER);
    }

    public static void main(String argv[]) {
        new SampleLabels().setVisible(true);
    }
}
```

Annotation

The *Annotation* class can be used to place text on a chart. In this example a text message is drawn at (1,7) as its bottom left corner.

Note that it is a two line message. New lines in the string are reflected in the chart.

The text alignment is set to `TEXT_X_LEFT` | `TEXT_Y_BOTTOM`, so (1,7) is the lower left corner of the text's bounding box (see the section Attribute Title).



View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SampleLabelMessage extends JFrameChart {

    public SampleLabelMessage() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);

        double y[] = {4, 6, 2, 1, 8};
        Data data = new Data(axis, y);
        data.setDataType(Data.DATA_TYPE_LINE);
        data.setLineColor(Color.pink);

        Text message = new Text("This is a\nsample message.");
        message.setAlignment(Data.TEXT_X_LEFT | Data.TEXT_Y_BOTTOM);
        Annotation messageAnnotation =
                new Annotation(axis, message, 1.0, 7.0);
        messageAnnotation.setTextColor(Color.blue);
    }
```

```
    public static void main(String argv[]) {
        new SampleLabelMessage().setVisible(true);
    }
}
```

AxisXY

The ***AxisXY*** node is the basis of many chart types, including scatter, line, area and error bar. Its parent node must be the root ***Chart*** node.

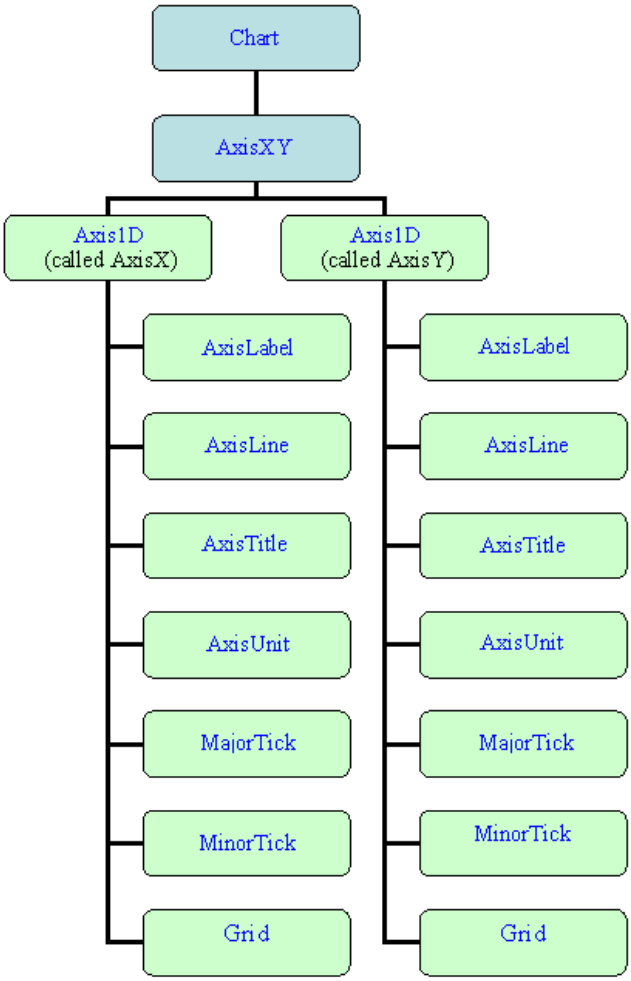
When an `AxisXY` node is created, it creates two ***Axis1D*** nodes. They can be obtained by using the methods `AxisXY.getAxisX()` and `AxisXY.getAxisY()`. Each of the `Axis1D` nodes in turn creates additional child nodes, as seen in the diagram below.

Accessor methods can be chained together, so the *x*-axis line can be retrieved using

```
axis.getAxisX().getAxisLine()
```

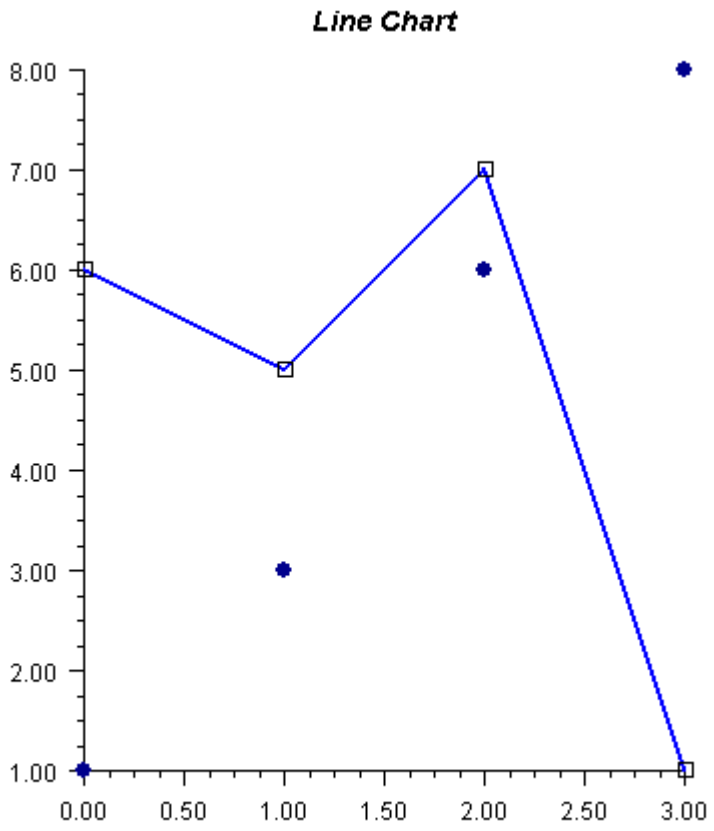
The code to set the *x*-axis line to blue is

```
axis.getAxisX().getAxisLine().setLineColor(Color.blue)
```



Axis Layout

The layout of an `AxisXY` chart, such as the one below, is controlled by the attributes `Window`, `Density`, and `Number`.



View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class Line extends JFrameChart {

    public Line() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        chart.getChartTitle().setFontSize(14);
        chart.getChartTitle().setFontStyle(3);
        chart.getChartTitle().setTitle("Line Chart");

        double y1[] = {6, 5, 7, 1};
        Data data1 = new Data(axis, y1);
        data1.setDataType(Data.DATA_TYPE_LINE|Data.DATA_TYPE_MARKER);
        data1.setLineColor(Color.blue);
```

```
        data1.setLineWidth(2.0);
        data1.setMarkerType(Data.MARKER_TYPE_HOLLOW_SQUARE);
        data1.setTitle("Blue Line");

        double y2[] = {1, 3, 6, 8};
        Data data2 = new Data(axis, y2);
        data2.setDataType(Data.DATA_TYPE_MARKER);
        data2.setMarkerColor(Data.parseColor("darkblue"));
        data2.setMarkerType(Data.MARKER_TYPE_FILLED_CIRCLE);
        data2.setTitle("Markers");
    }

    public static void main(String argv[]) {
        new Line().setVisible(true);
    }
}
```

Window

`Window` is a double-array-valued attribute that contains the axis limits. Its value is {*min, max*}. For the above chart, the value of the `Window` attribute for the *x*-axis is [0, 3] and for the *y*-axis it is [1, 8].

Number

`Number` is an integer-valued attribute that contains the number of major tick marks along an axis. In an `Axis1D` node its default value is 5, which is also the value in the above example.

Density

`Density` is the number of minor tick marks per major tick mark. Its default value is 4, which is also the value in the example.

Transform

The *x*- and *y*-axes may be linear, logarithmic, or customized, as specified by the `Transform` attribute. This attribute can have the values:

- `TRANSFORM_LINEAR` indicating a linear axis. This is the default.
- `TRANSFORM_LOG` indicating a logarithmic axis.
- `TRANSFORM_CUSTOM` indicating a custom transformation. The transformation specified by the `CustomTransform` attribute is used.

CustomTransform

A customized transformation is used when the `Transform` attribute has the value `TRANSFORM_CUSTOM`. A custom transform is an object implementing the ***Transform*** interface. It defines a mapping from the interval specified by the `Window` attribute to and from the interval `[0,1]`. See the section Custom Transform for more information.

The class `TransformDate` implements the `Transform` interface. It maps from a regular calendar to one without weekends, which is useful for charting stock prices that are defined only on weekdays.

Autoscale

Autoscaling is used to automatically determine the attribute `Window` (the range of numbers along an axis) and the attribute `Number` (the number of major tick marks). The goal is to adjust the attributes so that the data fits on the axes and the axes have "nice" numbers as labels.

Autoscaling is done in two phases. In the first ("input") phase the actual range is determined. In the second ("output") phase chart attributes are updated.

Attribute AutoscaleInput

The action of the input phase is controlled by the value of attribute `AutoscaleInput` in the axis node. It can have one of three values.

- `AUTOSCALE_OFF` turns off autoscaling.
- `AUTOSCALE_DATA` scans the values in the `Data` nodes that are attached to the axis to determine the data range. This is the default value.
- `AUTOSCALE_WINDOW` uses the value of the Window attribute to determine the data range.

Attribute AutoscaleOutput

The value of the `AutoScaleOutput` attribute can be the bitwise combination of the following values.

- `AUTOSCALE_OFF` no attributes updated.
- `AUTOSCALE_NUMBER` updates the value of the `Number` attribute. This is the number of major tick marks along the axis.
- `AUTOSCALE_WINDOW` updates the value of the `Window` attribute. This is the range of numbers displayed along the axis.

The default is `AUTOSCALE_NUMBER | AUTOSCALE_WINDOW`; both the attributes `Number` and `Window` are adjusted.

Axis Label

The ***AxisLabel*** node controls the labeling of an axis. The drawing of this node is controlled by Text Attributes.

Scientific Notation

If the values along an axis are large, scientific notation is more readable than a decimal format with many zeros. In this example the *y*-axis is labeled with scientific notation where each number has exactly two fractional digits displayed. This format pattern is `"0.00E0"`. See ***DecimalFormat*** for details on number formatting patterns.

Date Labels

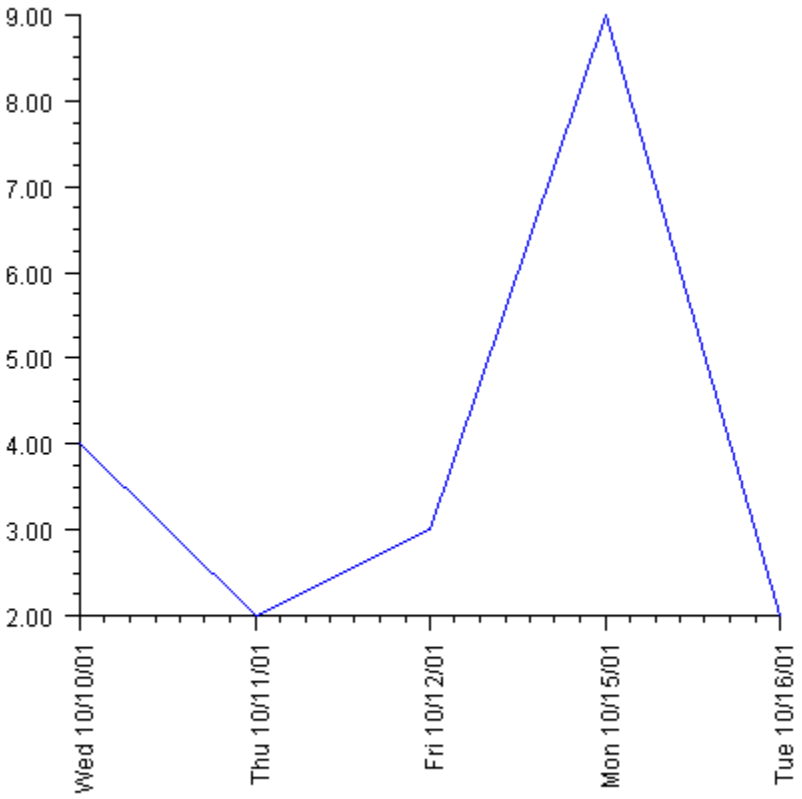
If the `TextFormat` attribute for an axis is an instance of ***DateFormat***, then the axis is scaled and labeled as a date/time axis, instead of as a real axis.

Date information passed to the `Date` constructor must be a `double` number representing the number of milliseconds since the standard base time known as "the epoch", namely January 1, 1970, 00:00:00 GMT. This is used by the constructor for `Date`.

Skipping Weekends

An additional feature of `Date` axes is the ability to skip weekends. This feature is often needed for charting stock price data.

To skip weekends it is necessary to adjust the autoscaling for weekdays-only. This is done by setting the attribute `SkipWeekends` to `true`. It is also necessary to set a custom transformation, ***TransformDate***, on the axis.



View code file

```
import com.imsl.chart.*;
import java.text.SimpleDateFormat;
import java.util.GregorianCalendar;

public class SampleWeekend extends JFrameChart {

    public SampleWeekend() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        Axis1D axisX = axis.getAxisX();
        axisX.getAxisLabel().setTextFormat(new SimpleDateFormat("EEE
MM/d/yy"));
        axisX.getAxisLabel().setTextAngle(90);

        // Skip weekends
        axisX.setSkipWeekends(true);
```

```
        axisX.setTransform(Axis1D.TRANSFORM_CUSTOM);
        axisX.setCustomTransform(new TransformDate());

        GregorianCalendar t[] = {
            new GregorianCalendar(2001, GregorianCalendar.OCTOBER, 10),
            new GregorianCalendar(2001, GregorianCalendar.OCTOBER, 11),
            new GregorianCalendar(2001, GregorianCalendar.OCTOBER, 12),
            new GregorianCalendar(2001, GregorianCalendar.OCTOBER, 15),
            new GregorianCalendar(2001, GregorianCalendar.OCTOBER, 16)
        };
        double x[] = new double[5];
        for (int k = 0;  k < x.length;  k++)  x[k] =
t[k].getTime().getTime();
        double y[] = {4, 2, 3, 9, 2};
        Data data = new Data(axis, x, y);
        data.setLineColor(java.awt.Color.blue);

        axis.setViewport(0.2, 0.9, 0.1, 0.7);
    }

    public static void main(String argv[]) {
        new SampleWeekend().setVisible(true);
    }
}
```

Skipping weekends is intended only for data sets where no weekend data exists. It will not give the expected results if the data set contains weekend data.

String Labels

Any array of strings can be used to label the tick marks using the `setLabels(String[])` method. The `setLabels(String[])` method sets the `Number` attribute to the number of strings.

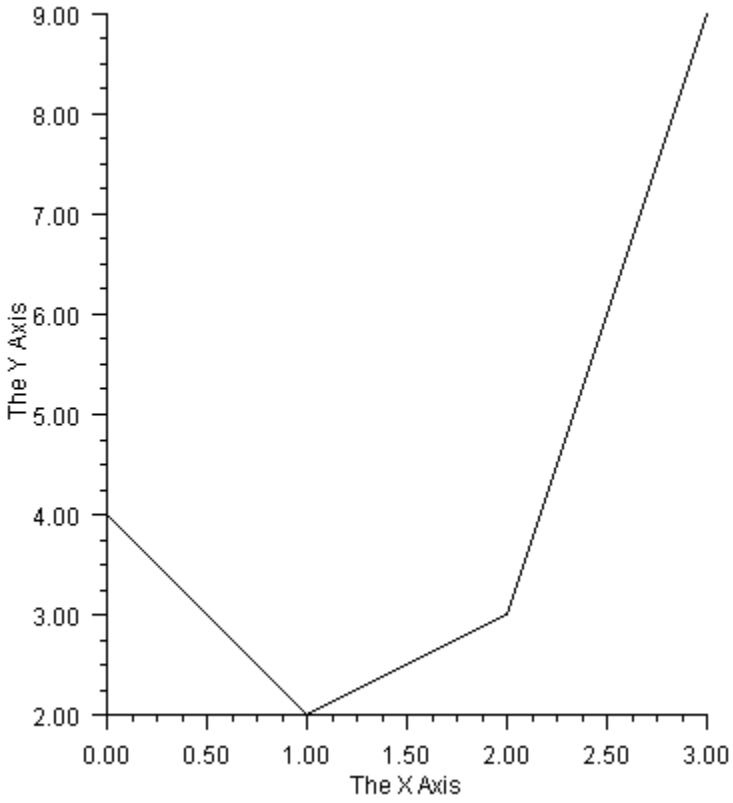
Axis Title

The ***AxisTitle*** node controls the titling of an axis. The drawing of this node is controlled by the Text Attributes.

Axes are titled with the value of the `Title` attribute in the `AxisTitle` node. By default, the title is empty.

Setting the Title

To set an axis title, set the `Title` attribute in the `AxisTitle` node. In this example both the *x*-axis and *y*-axis titles are set.



View code file

```
import com.imsl.chart.*;

public class SampleAxisTitle extends JFrameChart {

    public SampleAxisTitle() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        double y[] = {4, 2, 3, 9};
```

```
        new Data(axis, y);

        axis.getAxisX().getAxisTitle().setTitle("The X Axis");
        axis.getAxisY().getAxisTitle().setTitle("The Y Axis");
    }

    public static void main(String argv[]) {
        new SampleAxisTitle().setVisible(true);
    }
}
```

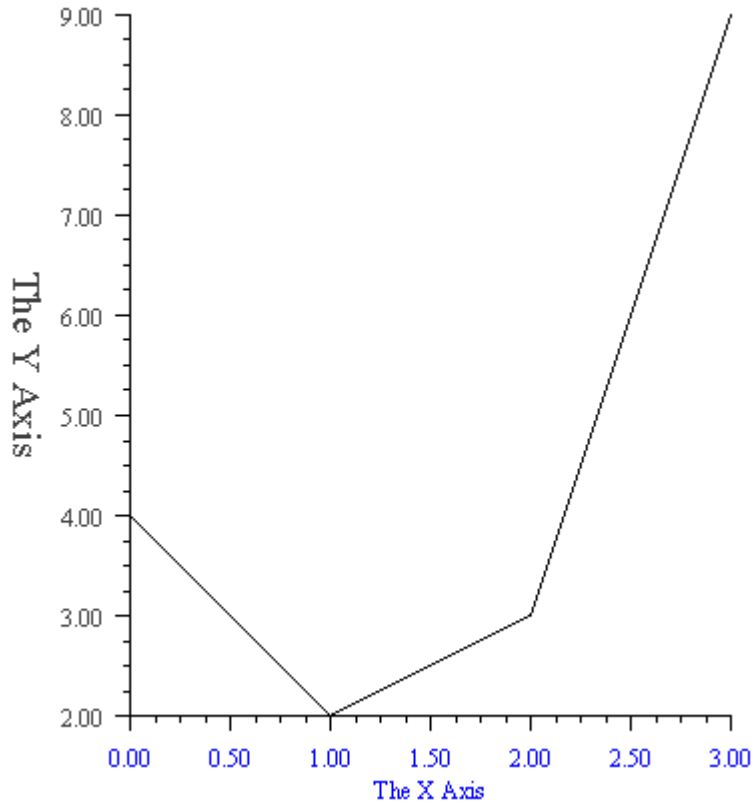
More Formatting

An axis title, like any other text string, can have further formatting applied.

The `FontName` attribute is set to "`Serif`" in the axis node. This value is then inherited by all of its ancestor nodes.

The `TextColor` attribute is set differently in the *x*-axis and *y*-axis nodes. Note that this setting is inherited by both the `AxisTitle` and nodes within the same axis.

The y-axis title node has its `FontSize` attribute set to 20, so it is larger. Also, its `TextAngle` attribute is set to `-`90. By default, the *y*-axis title is drawn at a 90 degree angle, so the `-`90 degree setting flips the title from its usual position.



View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SampleAxisTitleFormatted extends JFrameChart {

    public SampleAxisTitleFormatted() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        double y[] = {4, 2, 3, 9};
        new Data(axis, y);

        axis.setFontName("Serif");

        axis.getAxisX().setTextColor(Color.blue);
        axis.getAxisX().getAxisTitle().setTitle("The X Axis");

        axis.getAxisY().setTextColor(Color.darkGray);
        axis.getAxisY().getAxisTitle().setTitle("The Y Axis");
        axis.getAxisY().getAxisTitle().setTextAngle(-90);
        axis.getAxisY().getAxisTitle().setFontSize(20);
    }

    public static void main(String argv[]) {
        new SampleAxisTitleFormatted().setVisible(true);
    }
}
```

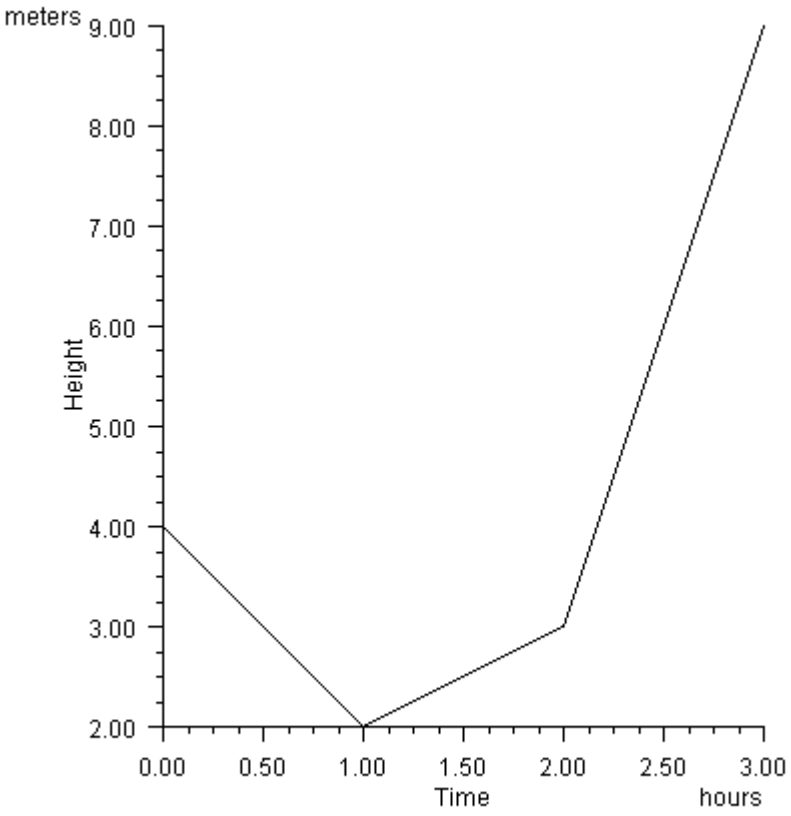
Axis Unit

The ***AxisUnit*** node controls the placing of a unit tag on an axis. The tag is the value of the `Title` attribute in the node. By default, it is empty.

The axis unit is used for labeling only; it has no other effect on the chart.

Setting the Axis Unit

In this example, the axis unit for the *x*-axis is set to "hours" and the axis unit for the *y*-axis is set to "meters". While not required, the axis titles are also set.



View code file

```
import com.imsl.chart.*;

public class SampleAxisUnit extends JFrameChart {

    public SampleAxisUnit() {
        Chart chart = getChart();
```

```
        AxisXY axis = new AxisXY(chart);
        double y[] = {4, 2, 3, 9};
        new Data(axis, y);

        axis.getAxisX().getAxisTitle().setTitle("Time");
        axis.getAxisY().getAxisTitle().setTitle("Height");

        axis.getAxisX().getAxisUnit().setTitle("hours");
        axis.getAxisY().getAxisUnit().setTitle("meters");
    }

    public static void main(String argv[]) {
        new SampleAxisUnit().setVisible(true);
    }
}
```

Major and Minor Tick Marks

The nodes ***MajorTick*** and ***MinorTick*** control the drawing of tick marks on an ***AxisXY***.

The location of the major tick marks can be set explicitly, using the `Axis1D.setTicks(double[])` method. However, it is usually easier to allow autoscaling to automatically set the tick locations (see the section Autoscale).

Attribute TickLength

The length of the tick marks is proportional to the screen size. They can be made relatively longer or shorter by setting the attribute `TickLength`. Its default value is 1.0. If its value is negative, the tick marks are drawn in the opposite direction: i.e., into the center of the plot region rather than away from it.

Attribute Number

Number is the number of major tick marks along an axis.

Attribute Density

`Density` is the number of minor tick marks in each interval between major tick marks. The minor ticks are equally spaced in user coordinates. If the Transform attribute in not `TRANSFORM_LINEAR`, then they will not be equally spaced on the screen.

Attribute FirstTick

The `FirstTick` attribute, in an ***Axis1D*** node, is the position of the first major tick mark. The default value is the 0-th element of the `Windows` attribute.

Attribute TickInterval

The `TickInterval` attribute, in an `Axis1D` node, is the interval between tick marks in the user coordinates. If this attribute is not explicitly set, its value is computed from the attributes `Number`, `Window`, and `Transform`.

Attribute Ticks

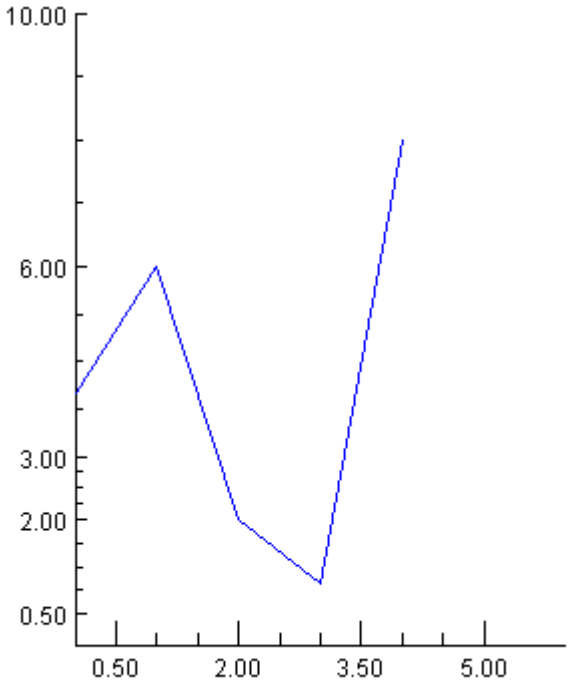
The `Ticks` attribute, in an `Axis1D` node, contains the position of the major tick marks. If this attribute is not explicitly set, its value is computed from the attributes FirstTick, TickInterval, `Number`, `Window`, and `Transform`.

Example

This example shows the effects of the tick mark attributes. Note that autoscaling is turned off so that the attribute values are not overridden (see the section Autoscale).

On the *x*-axis, there are four major tick marks (`Number`), starting with 0.5 (`FirstTick`) at an interval of 1.5 (`TickInterval`). There are three minor tick intervals (`Density`) between each major tick mark. The tick marks are twice as long and are drawn in the opposite direction as normal (`TickLength`).

On the *y*-axis, the tick mark locations are set explicitly by the attribute `Ticks`. This automatically sets the attribute `Number`. The `TickLength` is set to `-`1, so the tick marks are drawn inward (to the right) instead of outward (to the left).



View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SampleTicks extends JFrameChart {

    public SampleTicks() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        axis.setAutoscaleOutput(axis.AUTOSCALE_OFF);

        axis.getAxisX().setWindow(0.0, 6.0);
        axis.getAxisX().setDensity(3);
        axis.getAxisX().setNumber(4);
        axis.getAxisX().setFirstTick(0.5);
        axis.getAxisX().setTickInterval(1.5);
        axis.getAxisX().setTickLength(-2);

        axis.getAxisY().setWindow(0.0, 10.0);
```

```
        double ticksY[] = {0.5, 2.0, 3.0, 6.0, 10.0};
        axis.getAxisY().setTicks(ticksY);
        axis.getAxisY().setTickLength(-1);

        double y[] = {4, 6, 2, 1, 8};
        Data data = new Data(axis, y);
        data.setDataType(Data.DATA_TYPE_LINE);
        data.setLineColor(Color.blue);
    }

    public static void main(String argv[]) {
        new SampleTicks().setVisible(true);
    }
}
```

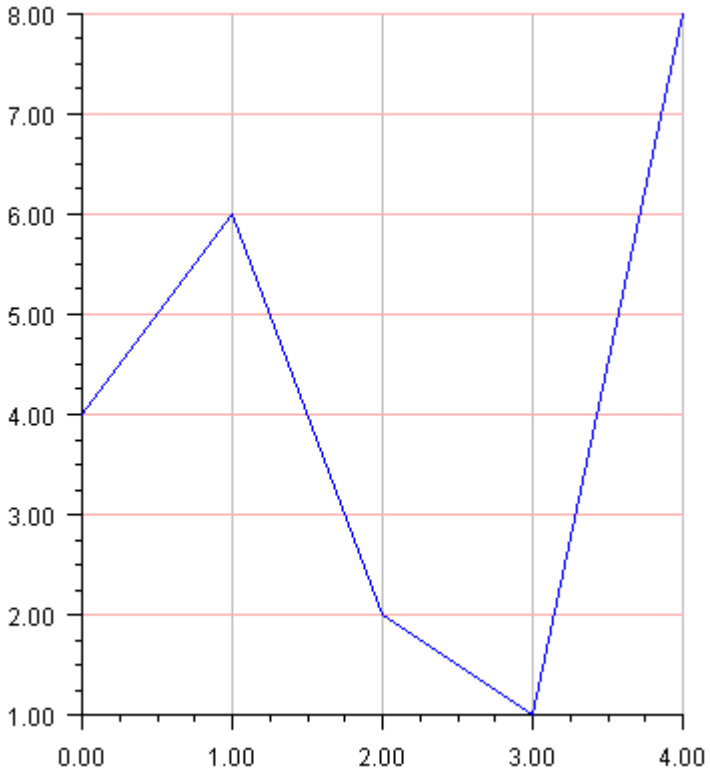
Grid

The ***Grid*** node controls the drawing of grid lines on a chart. The `Grid` is created by ***Axis1D*** as its child. It can be retrieved using the method `Axis1D.getGrid()`.

By default, `Grid` nodes are not drawn. To enable them, set their `Paint` attribute to `true`. `Grid` nodes control the drawing of the grid lines perpendicular to their parent axis. So the *x*-axis `Grid` node controls the drawing of the vertical grid lines.

Example

In this example, the *x*-axis grid lines are painted light gray and the *y*-axis grid lines are pink.



View code file

```
import com.imsl.chart.*;

public class SampleGrid extends JFrameChart {

    public SampleGrid() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
```

```
        axis.getAxisX().getGrid().setPaint(true);
        axis.getAxisY().getGrid().setPaint(true);
        axis.getAxisX().getGrid().setLineColor(java.awt.Color.lightGray);
        axis.getAxisY().getGrid().setLineColor(java.awt.Color.pink);

        double y[] = {4, 6, 2, 1, 8};
        Data data = new Data(axis, y);
        data.setDataType(Data.DATA_TYPE_LINE);
        data.setLineColor(java.awt.Color.blue);
    }

    public static void main(String argv[]) {
        new SampleGrid().setVisible(true);
    }
}
```

Custom Transform

A custom transformation allows a user-defined mapping of data to an axis. A custom transform is used when the `Transform` attribute has the value `TRANSFORM_CUSTOM`. The custom transform is the value of the `CustomTransform` attribute.

A custom transform implements the ***Transform*** interface. This interface has three methods. One, `setupMapping`, is called first to allow the transformation to initialize itself. The other two, `mapUserToUnit` and `mapUnitToUser`, specify a mapping from the window interval onto `[0,1]`. These methods must each be the inverse of the other.

NormalPlot

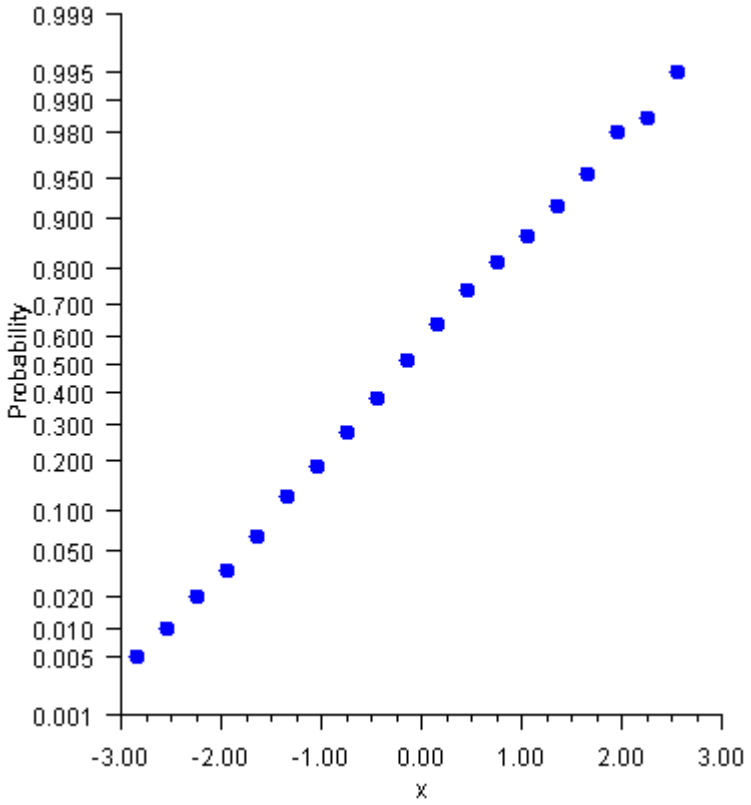
A normal probability plot maps normally distributed data into a straight line, just as a semilog plot maps exponential data into a straight line.

In this example, 400 normally distributed random numbers are grouped into 20 bins. The bin counts are normalized by dividing by the number of samples (400). Cumulative probabilities are computed by summing the probabilities in all of the bins to the left of the current bin.

The custom transformation is the normal cumulative distribution function, `Cdf.normal`, and its inverse, `InvCdf.normal`, scaled to map the probability range `[0.001,0.999]` onto `[0,1]`.

Autoscaling is turned off on the probability (*y*) axis and a fixed set of probability tick marks are specified.

The plot is of the bin center on the *x*-axis versus the cumulative probabilities on the *y*-axis. The points are not in an exactly straight line because with only a finite number of samples, the distribution does not exactly match the normal distribution.



View code file

```
import com.imsl.chart.*;
import com.imsl.stat.Cdf;
import com.imsl.stat.InvCdf;
import com.imsl.stat.Random;
import java.awt.Color;

public class SampleProbability extends JFrameChart {

    public SampleProbability() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);

        double ticks[] = {
            0.001, 0.005, 0.01, 0.02, 0.05, 0.10, 0.20, 0.30,
            0.40, 0.50, 0.60, 0.70, 0.80, 0.90, 0.95, 0.98,
            0.99, 0.995, 0.999};
        double a = ticks[0];
        double b = ticks[ticks.length - 1];

        axis.getAxisX().getAxisTitle().setTitle("x");
        axis.getAxisY().getAxisTitle().setTitle("Probability");

        axis.getAxisY().setTransform(Axis.TRANSFORM_CUSTOM);
```

```
        axis.getAxisY().setCustomTransform(new NormalTransform());
        axis.getAxisY().setAutoscaleInput(Axis.AUTOSCALE_OFF);
        axis.getAxisY().setWindow(a, b);
        axis.getAxisY().setTextFormat(new java.text.DecimalFormat("0.000"));
        axis.getAxisY().setTicks(ticks);
        axis.getAxisY().getMinorTick().setPaint(false);

        int nSamples = 400;
        int nBins = 20;

        // Setup the bins
        double bins[] = new double[nBins];
        double dx = 6.0 / nBins;
        double x[] = new double[nBins];
        for (int k = 0; k < nBins; k++) {
            x[k] = -3.0 + (k + 0.5) * dx;
        }

        // Generate random normal deviates and sort into bins
        Random r = new Random(123457);
        for (int k = 0; k < nSamples; k++) {
            double t = r.nextNormal();
            int j = (int) Math.round((t + 3.0 - 0.5 * dx) / dx);
            if (j <= 0) {
                bins[0]++;
            } else if (j >= nBins - 1) {
                bins[nBins - 1]++;
            } else {
                bins[j]++;
            }
        }

        // Compute the cumulative distribution
        // y[k] is the sum of bins[j] for j=0,...,k divided by the nSamples.
        double y[] = new double[nBins];
        y[0] = bins[0] / nSamples;
        for (int k = 1; k < nBins; k++) {
            y[k] = y[k - 1] + bins[k] / nSamples;
        }

        // Plot the data using markers
        Data data = new Data(axis, x, y);
        data.setDataType(Data.DATA_TYPE_MARKER);
        data.setMarkerType(Data.MARKER_TYPE_FILLED_CIRCLE);
        data.setMarkerColor(Color.blue);
    }

    public static void main(String argv[]) {
        new SampleProbability().setVisible(true);
    }
```

```
    static class NormalTransform implements Transform {

        private double scaleA,  scaleB;

        /**
         *    Initializes the mappings between user and coordinate space.
         */
        public void setupMapping(Axis1D axis1d) {
            double w[] = axis1d.getWindow();
            double t = InvCdf.normal(w[0]);
            scaleB = 1.0 / (InvCdf.normal(w[1]) - t);
            scaleA = -scaleB * t;
        }

        /**
         * Maps a point in [0,1] to a probability.
         */
        public double mapUnitToUser(double unit) {
            return Cdf.normal((unit - scaleA) / scaleB);
        }

        /**
         * Maps a probablity to the interval [0,1].
         */
        public double mapUserToUnit(double p) {
            return scaleA + scaleB * InvCdf.normal(p);
        }
    }
}
```

Multiple Axes

A JMSL chart can contain any number of axes. Each axis has its own mapping from the user coordinates to the device (screen) coordinates.

Normally, the *x*-axis is at the bottom of the chart and the *y*-axis is to the left. The attribute `Type` can be changed to move the *x*-axis to the top of the chart and/or the *y*-axis to the right of the chart.

Axis can be moved from the chart edge, either away from the chart or into the middle of the chart data area, by setting the attribute `Cross`.

Attribute Type

The attribute `Type` specifies the position of an *x* or *y*-axis. Applied to the *x*-axis it can have the values `AXIS_X` (the default) or `AXIS_X_TOP`. Applied to the *y*-axis, it can have the value `AXIS_Y` (the default) or `AXIS_Y_RIGHT`.

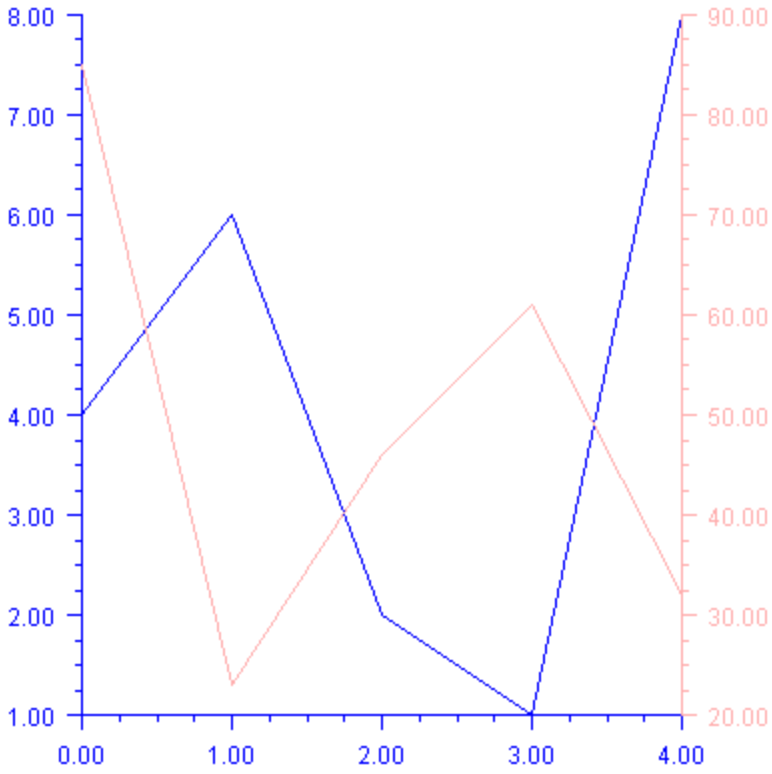
Attribute Cross

The `Cross` attribute specifies the coordinates of intersection of the *x*-axis and *y*-axis. This can be inside or outside of the chart body. `Cross` can be used to place multiple *y*-axes to the left of the chart or multiple *x*-axes below the chart.

Example

Two data sets are plotted, each on its own axis. The first (blue) axis is left in the default position with the *y*-axis on the left. The second (pink) axis has its *y*-axis moved to the right.

In this example the *x*-axis is shared between the two axes, so it is not painted as part of the second (pink) axis. This is done by setting its `Paint` attribute to `false` (see `setPaint`).



View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SampleAxesLeftRight extends JFrameChart {

    public SampleAxesLeftRight() {
        Chart chart = getChart();

        AxisXY axisLeft = new AxisXY(chart);
        double yLeft[] = {4, 6, 2, 1, 8};
        Data dataLeft = new Data(axisLeft, yLeft);
        dataLeft.setDataType(Data.DATA_TYPE_LINE);
        axisLeft.setLineColor(Color.blue);
        axisLeft.setTextColor(Color.blue);

        AxisXY axisRight = new AxisXY(chart);
        axisRight.getAxisX().setPaint(false);
        axisRight.getAxisY().setType(Axis1D.AXIS_Y_RIGHT);
        double yRight[] = {85, 23, 46, 61, 32};
        Data dataRight = new Data(axisRight, yRight);
```

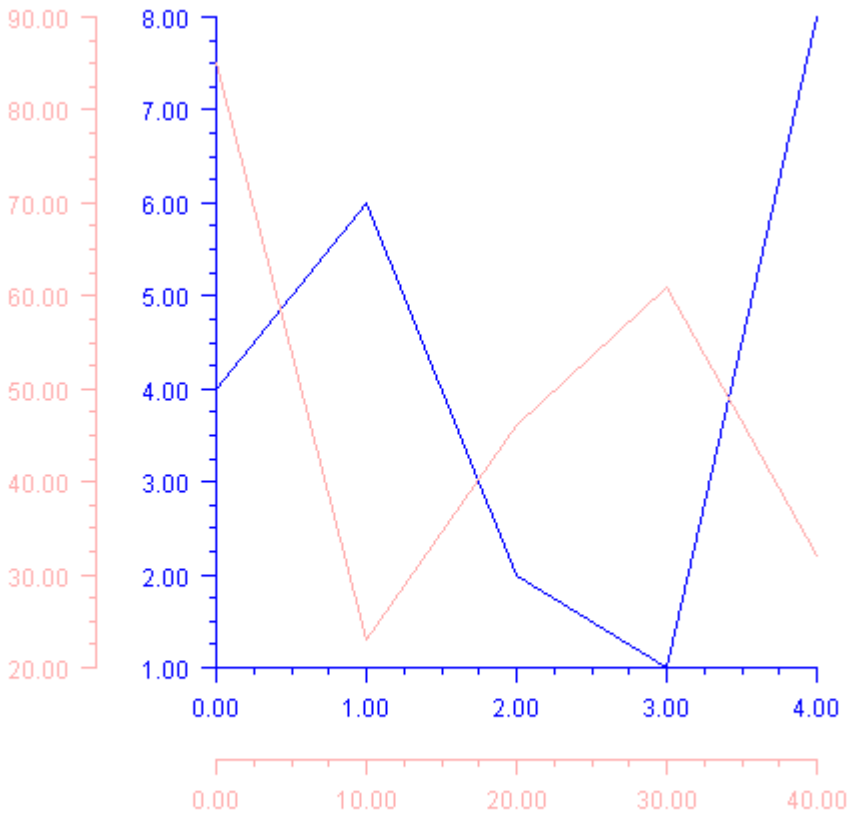
```
        dataRight.setDataType(Data.DATA_TYPE_LINE);
        axisRight.setLineColor(Color.pink);
        axisRight.setTextColor(Color.pink);
    }

    public static void main(String argv[]) {
        new SampleAxesLeftRight().setVisible(true);
    }
}
```

Cross Example

Multiple *x* and *y*-axes are shown.

The `Cross` attribute is used to specify that the second (pink) axes intersect at (-8,10), in the pink-axis coordinate system.

The `Viewport` attribute is changed in both sets of axes to shrink the size of the chart body and leave more room to the left and below the chart for the pink axes.



View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SampleAxesCross extends JFrameChart {

    public SampleAxesCross() {
        Chart chart = getChart();

        AxisXY axisLeft = new AxisXY(chart);
        double yLeft[] = {4, 6, 2, 1, 8};
        Data dataLeft = new Data(axisLeft, yLeft);
        dataLeft.setDataType(Data.DATA_TYPE_LINE);
        axisLeft.setLineColor(Color.blue);
        axisLeft.setTextColor(Color.blue);

        AxisXY axisRight = new AxisXY(chart);
        axisRight.setCross(-8,10);
        double xRight[] = {0, 10, 20, 30, 40};
        double yRight[] = {85, 23, 46, 61, 32};
        Data dataRight = new Data(axisRight, xRight, yRight);
        dataRight.setDataType(Data.DATA_TYPE_LINE);
        axisRight.setLineColor(Color.pink);
        axisRight.setTextColor(Color.pink);

        double viewport[] = {0.3, 0.9, 0.05, 0.7};
        axisLeft.setViewport(viewport);
        axisRight.setViewport(viewport);
    }

    public static void main(String argv[]) {
        new SampleAxesCross().setVisible(true);
    }
}
```

Background

Background controls the drawing of the chart's background. It is created by ***Chart*** as its child. It can be retrieved from a `Chart` object using the `Chart.getBackground()` method.

The fill area attributes in the `Background` node determine how the background is drawn (see the section Fill Area Attributes).

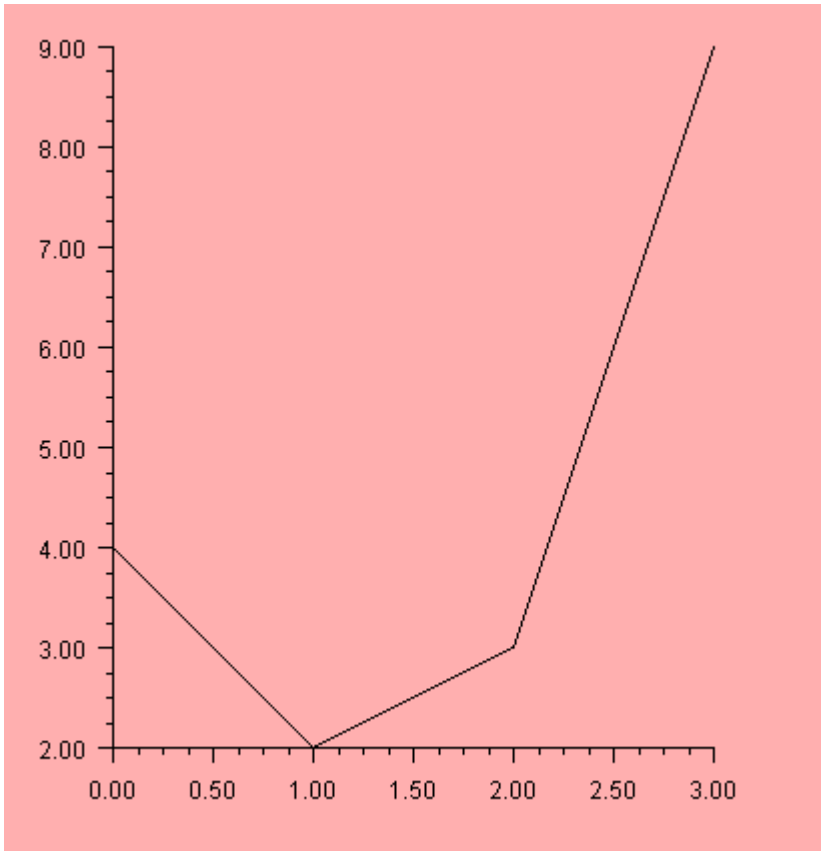
The attribute `FillType` has the global default value of `FILL_TYPE_SOLID`. The attribute `FillColor` attribute is set to `Color.white` in this node.

Solid Color Background

To set the background to a solid color:

- set the attribute `FillType` to `FILL_TYPE_SOLID`, and
- set the attribute `FillColor` to the desired color.

For example the following code sets the background to pink. To view chart in color please see the online documentation.



`View code file`

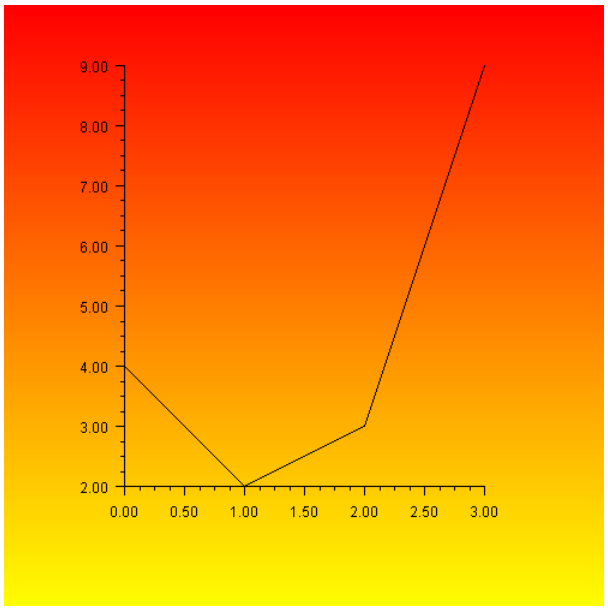
```
import com.imsl.chart.*;

public class SampleBackgroundSolid extends JFrameChart {

    public SampleBackgroundSolid() {
        Chart chart = getChart();
        chart.getBackground().setFillType(ChartNode.FILL_TYPE_SOLID);
        chart.getBackground().setFillColor(java.awt.Color.pink);
        AxisXY axis = new AxisXY(chart);
        double y[] = {4, 2, 3, 9};
        new Data(axis, y);
    }

    public static void main(String argv[]) {
        new SampleBackgroundSolid().setVisible(true);
    }
}
```

Gradient Color Background

To set the background to a color gradient:

- set the attribute `FillType` to `FILL_TYPE_GRADIENT`, and
- set the attribute `Gradient` to the desired color gradient specification.

For example the following code uses a yellow-to-red vertical gradient for the background setting. See the section Fill Area Attributes for more information on gradients.



View code file

```
import com.imsl.chart.*;
```

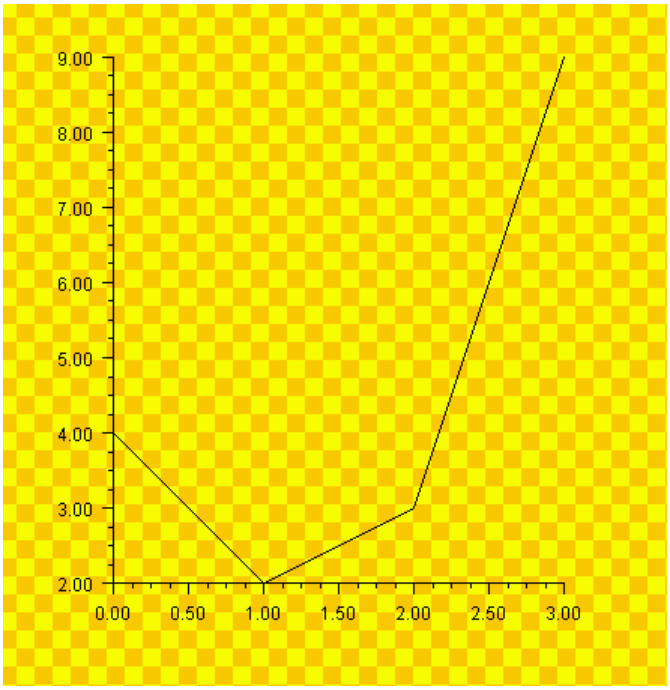
```
import java.awt.Color;

public class SampleBackgroundGradient extends JFrameChart {

    public SampleBackgroundGradient() {
        Chart chart = getChart();
        chart.getBackground().setFillType(ChartNode.FILL_TYPE_GRADIENT);
        chart.getBackground().setGradient(Color.yellow, Color.yellow,
            Color.red, Color.red);
        AxisXY axis = new AxisXY(chart);
        double y[] = {4, 2, 3, 9};
        new Data(axis, y);
    }

    public static void main(String argv[]) {
        new SampleBackgroundGradient().setVisible(true);
    }
}
```

Pattern Background

To set the background to a color pattern:

- set the attribute `FillType` to `FILL_TYPE_PAINT`, and
- set the attribute `FillPaint` to the desired pattern.

For example the following code sets the background to yellow/orange checkerboard pattern. See the section Fill Area Attributes for more information on patterns.



View code file

```
import com.imsl.chart.*;
import java.awt.Color;
import java.awt.Paint;

public class SampleBackgroundPaint extends JFrameChart {

    public SampleBackgroundPaint() {
        Chart chart = getChart();
        chart.getBackground().setFillType(ChartNode.FILL_TYPE_PAINT);
        Paint paint = FillPaint.checkerboard(24, Color.yellow,Color.orange);
        chart.getBackground().setFillPaint(paint);
        AxisXY axis = new AxisXY(chart);
        double y[] = {4, 2, 3, 9};
        new Data(axis, y);
    }

    public static void main(String argv[]) {
        new SampleBackgroundPaint().setVisible(true);
    }
}
```

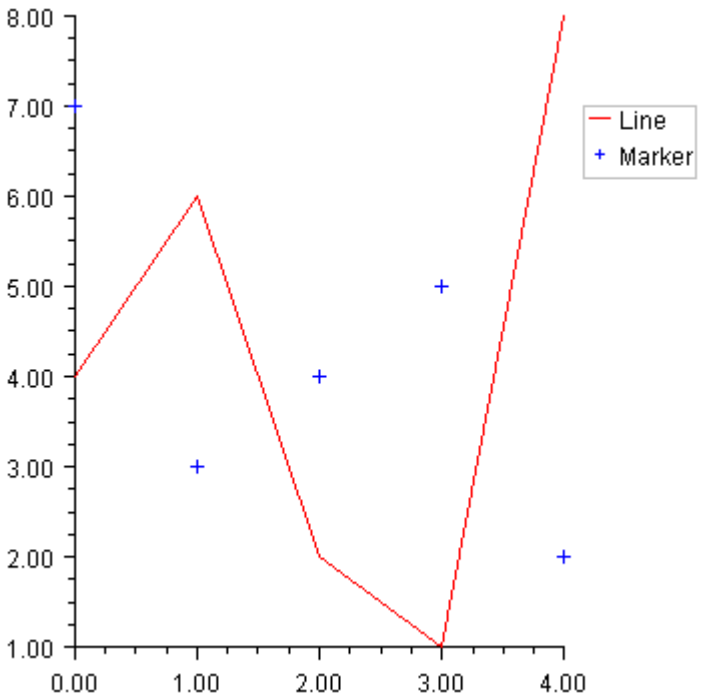
Legend

The legend is used to identify data sets. ***Data*** nodes that have their `Title` attribute defined are automatically included in the legend box.

The ***Legend*** node is automatically created by the ***Chart*** node as its child. By default, the legend is not drawn because its `Paint` attribute is set to `false`.

Simple Legend Example

At a minimum, adding a legend to a chart requires setting the legend's `Paint` attribute to `true` and setting the `Title` attribute in the `Data` nodes that are to appear in the legend box. This example shows such a minimal legend.



View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SampleSimpleLegend extends JFrameChart {

    public SampleSimpleLegend() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        chart.getLegend().setPaint(true);
```

```
        double y1[] = {4, 6, 2, 1, 8};
        Data data1 = new Data(axis, y1);
        data1.setDataType(Data.DATA_TYPE_LINE);
        data1.setLineColor(Color.red);
        data1.setTitle("Line");

        double y2[] = {7, 3, 4, 5, 2};
        Data data2 = new Data(axis, y2);
        data2.setDataType(Data.DATA_TYPE_MARKER);
        data2.setMarkerColor(Color.blue);
        data2.setTitle("Marker");
    }

    public static void main(String argv[]) {
        new SampleSimpleLegend().setVisible(true);
    }
}
```

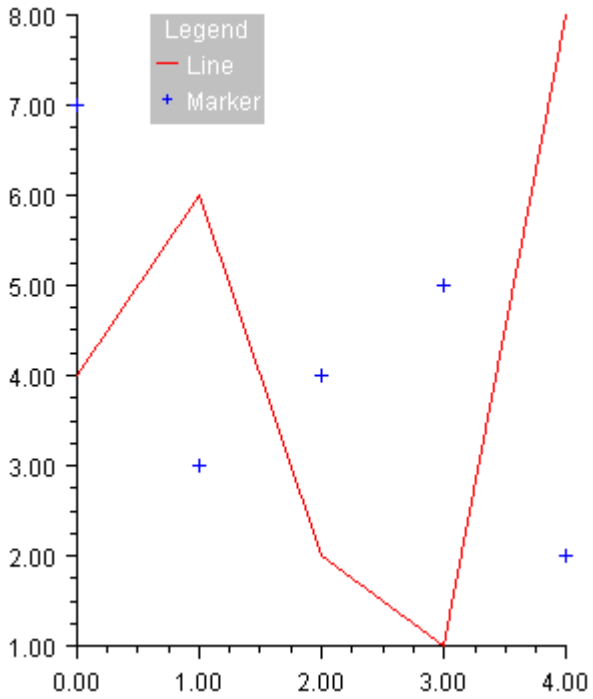
Legend Example

This example shows more of the attributes that affect a legend. If the legend's `Title` attribute is set, then it is used as a header in the legend box.

The text properties for all of the text strings in the legend box are obtained from the `Legend` node, not from the associated `Data` nodes (see the section Text Attributes). Here the `TextColor` is set to white.

The background of the legend box can be set by changing the fill attributes (see the section Fill Area Attributes). By default in the `Legend` node, `FillType` is set to `FILL_TYPE_NONE` and `FillColor` is set to `Color.lightGray`.

The position of the legend box is controlled by its `Viewport` attribute. The viewport is the region of the `Component`, in which the chart is being drawn, that the legend box occupies. The upper left corner is (0,0) and the lower right corner is (1,1). The default value of the legend viewport is [0.83, 0.0] by [0.2, 0.2]. The position of the legend can be controlled by the *xmin* and *ymin* parameters of method `setViewport` (note that the *xmax* and *ymax* parameters do not affect the legend viewport).



View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SampleLegend extends JFrameChart {

    public SampleLegend() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);

        Legend legend = chart.getLegend();
        legend.setPaint(true);
        legend.setTitle("Legend");
        legend.setTextColor(Color.white);
        legend.setFillType(legend.FILL_TYPE_SOLID);
        legend.setViewport(0.3, 0.4, 0.1, 0.4);

        double y1[] = {4, 6, 2, 1, 8};
        Data data1 = new Data(axis, y1);
        data1.setDataType(Data.DATA_TYPE_LINE);
        data1.setLineColor(Color.red);
        data1.setTitle("Line");

        double y2[] = {7, 3, 4, 5, 2};
        Data data2 = new Data(axis, y2);
        data2.setDataType(Data.DATA_TYPE_MARKER);
        data2.setMarkerColor(Color.blue);
```

```
        data2.setTitle("Marker");
    }

    public static void main(String argv[]) {
        new SampleLegend().setVisible(true);
    }
}
```

Color

Many color names are defined in JMSL. You can follow this link to a defined set of color names and RGB values beyond the fields already provided in the Java ***java.awt.Color*** class. The color name/ RGB Value associations are consistent with HTML4 color designations. All class attributes that have a method to set color (i.e. line, marker, fill, ...) and take a string argument use the `AbstractChartNode.parseColor(string)` method. Therefore the color associations in color names and RGB values are used.

Colormaps

Colormaps are mappings from [0,1] to colors. They are a one-dimensional parameterized path through the color cube. For an example of their use, see Heatmap.

A number of colormaps are predefined in the ***Colormap*** class. It is also possible to create new custom colormaps.

The predefined colormaps are listed in the table below.

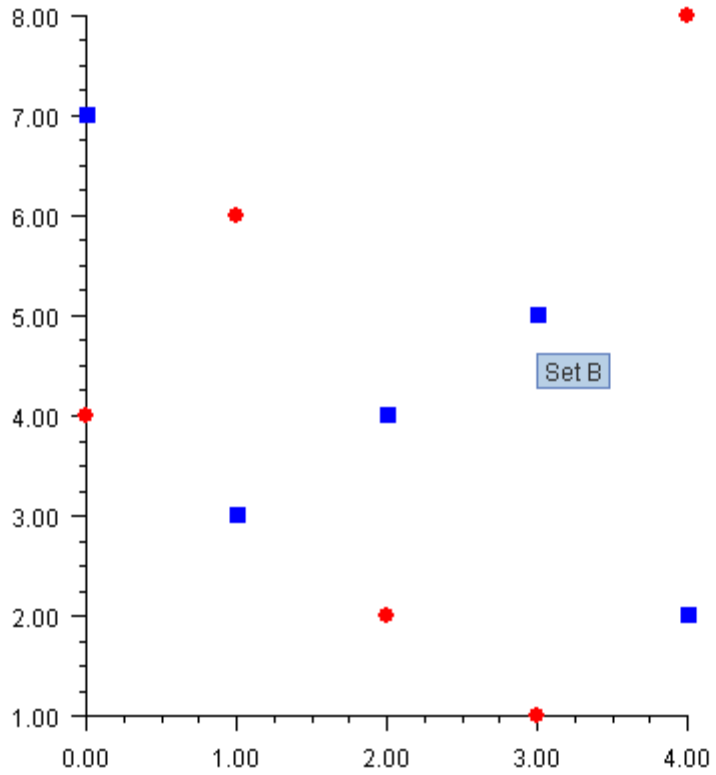
Fields	Description
`RED`	Light red to dark red
`GREEN`	Light green to dark green
`BLUE`	Light blue to dark blue
`BW_LINEAR`	Black and white linear
`BLUE_WHITE`	Blue/White
`GREEN_RED_BLUE_WHITE`	Green/Red/Blue/White
`RED_TEMPERATURE`	Red Temperature
`BLUE_GREEN_RED_YELLOW`	Blue/Green/Red/Yellow
`STANDARD_GAMMA`	Standard Gamma
`PRISM`	Prism
`RED_PURPLE`	Red/Purple
`GREEN_WHITE_LINEAR`	Green/White Linear
`GREEN_WHITE_EXPONENTIAL`	Green/White Exponential
`GREEN_PINK`	Green/Pink
`BLUE_RED`	Blue/Red
`SPECTRAL`	Spectral
`WB_LINEAR`	White/Black Linear

Tool Tips

`Swing` supports tool tips, small help boxes that pop up when the mouse hovers over some element. The ***ToolTip*** node allows tool tips to be associated with chart elements. The text displayed in the tool tip is the `Title` attribute of the `ToolTip`'s parent node.

Example

In this example two ***Data*** nodes are created. Each `Data` node has a `Title` defined and a `ToolTip` node added as a child. Note that the `ToolTip` node just has to be created, no methods using it are normally required.



View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SampleToolTip extends JFrameChart {

    public SampleToolTip() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
```

```
        double y1[] = {4, 6, 2, 1, 8};
        Data data1 = new Data(axis, y1);
        data1.setDataType(Data.DATA_TYPE_MARKER);
        data1.setMarkerColor(Color.red);
        data1.setMarkerType(Data.MARKER_TYPE_FILLED_CIRCLE);
        data1.setTitle("Set A");
        new ToolTip(data1);

        double y2[] = {7, 3, 4, 5, 2};
        Data data2 = new Data(axis, y2);
        data2.setDataType(Data.DATA_TYPE_MARKER);
        data2.setMarkerColor(Color.blue);
        data2.setMarkerType(Data.MARKER_TYPE_FILLED_SQUARE);
        data2.setTitle("Set B");
        new ToolTip(data2);
    }

    public static void main(String argv[]) {
        new SampleToolTip().setVisible(true);
    }
}
```

Chapter 5 XML

JMSL and XML

A JMSL chart tree can be created from an XML file. XML (Extensible Markup Language) is a universal format for structured data. XML looks similar to HTML but has application-dependent tag names instead of HTML tags (`<p>`, `<h1>`, etc.).

> **NOTE:** This chapter provides links — `View code file` — that display the actual code file of the XML code that follows the link. If you are viewing this in HTML, you should right-click the link and select "Open link in new tab" for best results. In PDF, the file opens in whatever the default application is for files with a `.xml` extension.

Chart XML Syntax

The Chart XML syntax is defined by a document type definition (DTD) file at `http://www.roguewave.com/products/jmsl/Chart.dtd`.

Valid XML files begin with a prolog that contains the XML declaration and the document type declaration. For example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE Chart PUBLIC "-//Rogue Wave Software//DTD JMSL Chart//EN"
"http://www.roguewave.com/products/jmsl/Chart.dtd">
```

Following the prolog is the body of the chart XML file. This is a tree of tags, with the `<Chart>` tag at the root. These tags mirror a JMSL chart tree. The following outline shows the hierarchy of tags.

```
<Chart >
   <Array> | <Array id=name>
   <Background>
   <ChartTitle>
   <Legend>
   <AxisXY>
      <AxisX> | <AxisY>
         <AxisTitle>
         <AxisLabel>
         <AxisLine>
         <AxisUnit>
         <Grid>
         <MajorTick>
         <MinorTick>
      <Data x=array y=array> | <Data y=array> | <Data>
      <Bar x=array y=array> | <Bar y=array>
         <BarSet> | <BarSet index=int>
         <BarItem> | <BarItem index=int>
      <ErrorBar x=array y=array low=array high=array>
      <HighLowClose start=array high=array low=array close=array>
      <HighLowClose high=array low=array close=array>
   <Polar>
   <Pie y=array>
      <PieSlice> | <PieSlice index=int>
      < Contour xGrid=array yGrid=array zData=array>
      < Heatmap xmin=double xmax=double ymin=double y max=double
            zmin=double  zmax=double data=array colormap=name>
```

Some of the tags have array-valued arguments. These are either a list of doubles of the form "{1.2,3.4,5}" or "#*id*", where *id* is an `<Array>` tag id.

Each of the chart tags can also have any number of <Attribute> tags as children. They can have either of two forms:

```
<Attribute name="attribute-name" value="attribute-value"/>
```

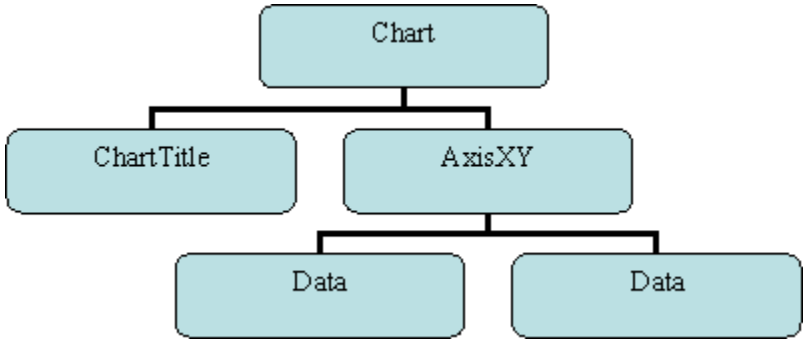
or

```
<Attribute name="attribute-name">attribute-value</ Attribute>
```

Also, while not shown in the above list, each chart node can have a `<ToolTip>` node as a child.

Syntax Example

The chart tree is created from the following XML file.



View code file

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE Chart PUBLIC "-//Rogue Wave Software//DTD JMSL Chart//EN"
"http://www.roguewave.com/products/jmsl/Chart.dtd">
<Chart>
    <ChartTitle>
        <Attribute name="Title">Line Chart</Attribute>
        <Attribute name="FontSize" value="14"/>
        <Attribute name="FontStyle" value="3"/>
    </ChartTitle>
    <AxisXY>
        <Data y="{6,5,7,1}">
            <Attribute name="LineColor" value="blue"/>
            <Attribute name="LineWidth" value="2.0"/>
            <Attribute name="DataType"
            value="DATA_TYPE_LINE|DATA_TYPE_MARKER"/>
            <Attribute name="MarkerType" value="MARKER_TYPE_HOLLOW_SQUARE"/>
            <Attribute name="Title" value="Blue Line"/>
            <ToolTip/>
        </Data>
        <Data y="{1,3,6,8}">
            <Attribute name="MarkerColor" value="darkblue"/>
            <Attribute name="DataType" value="DATA_TYPE_MARKER"/>
            <Attribute name="MarkerType" value="MARKER_TYPE_FILLED_CIRCLE"/>
            <Attribute name="Title" value="Markers"/>
```

```
            <ToolTip/>
        </Data>
    </AxisXY>
</Chart>
```

In the ***ChartTitle*** node there are three attributes defined: `Title`, `FontSize` and `FontStyle`. The argument "*y* ={6,5,7,1}" in the ***Data*** node is used to create an array passed to the `Data` constructor.

Note that the attribute name must be specified using `"name="`, but that the attribute's value can be either the value of a `"value="` or the body of the attribute tag.

Attribute Tags

Attribute tags can have either of two forms:

```
<Attribute name="attribute-name" value="attribute-value"/>
```

or

```
<Attribute name="attribute-name">attribute-value</Attribute>
```

The type of an attribute's value is determined by the return type of its getter method. For example, if the attribute type is `Foo`, its type is the return type of the 0-argument method `getFoo()` defined in the node being defined or in one of its ancestor nodes.

Symbolic names can be used for enumerated types. See the section Enumerated Types in Chart XML Files for details.

Attribute values of the following types can be handled: ***String***, ***Color***, `int` or ***Integer***, `double` or ***Double***, `Double[]`, `boolean` or ***Boolean***, ***Image***, ***Paint***, or ***Locale***.

Colors are either red, green, blue triples in the range [0,255] or color names. For example `"255,215,0"` or `"gold"`. See the `Color`s page for more details.

A ***Locale*** can be either the name of one of the static, public fields in Locale, such as `ENGLISH` or of the form `language_country` or `language_country_variant`. For example, `"nl_BE"` or `"nl_BE_EURO"`.

Attribute values of type ***Image*** and ***Paint*** are specified by giving the ***URL*** of an image file, for example

```
"http://www.google.com/images/logo.gif"
```

As a special case, the URL protocol can be a classpath, for example

```
"classpath:/com/imsl/example/chart/marker.gif"
```

The image is then loaded as a resource by the classloader using `getResource(String)`. The classloader used is the one used to load the ***ChartXML*** class. This feature is used to load images from JAR files.

Array Tags

The array tag is used to define data. The `id` argument is used to name the array. Array tags can be nested to form multidimensional arrays. At the inner-most level the following tags are allowed.

- <String value=*string*/>
- <Color value=*color*/>

The value of color can be either a red, green, blue triple in the range [0,255] or color names. For example `"255,215,0"` or `"gold"`. See this Color page for information on usable values.

- `<Date value=`*`date`* style=*style* locale=*locale*/>

If the argument locale is specified, it is parsed the same as the locale argument in the tag Attribute as described above. If the locale is not specified, then the default locale is used.

If the argument style is `SHORT`, `MEDIUM`, `LONG` or `FULL`, then the date is parsed using the object `DateFormat.getDateInstance(int, Locale)`. If style is not specified, then `SHORT` is assumed.

Style can also be a pattern used to construct a `SimpleDateFormat(int, Locale)`.

- <Number value=*number*/>

Numbers are parsed using `Double.valueOf(number)`.

- <NumberList>*number,...,number*</NumberList> | <NumberList value=*list*/>

The body of this tag is a comma separated list of numbers. Each number is parsed using `Double.valueOf(number)`.

The value arguments are optional. If omitted, the tag's body is used instead.

Array Tag Examples

The first example is of a three-dimensional array called `foo`.

```
<Array id="foo">
  <Array>
    <Array><NumberList value="1,2,3"/></Array>
    <Array><NumberList value="4,5,6"/></Array>
  </Array>
  <Array>
    <Array><NumberList value="11,12,13"/></Array>
    <Array><NumberList value="14,15,16"/></Array>
  </Array>
</Array>
```

This corresponds to the Java declaration

```
double foo[][][] = {{{1,2,3},{4,5,6}},{{11,12,13},{14,15,16}}}
```

The second example is an array of three strings called `states`.

```
<Array id="states">
  <String value="NJ"/>
  <String value="TX"/>
  <String value="CO"/>
</Array>
```

The final example is an array of four colors, called `bg`.

```
<Array id="bg">
  <Color value="yellow"/>
  <Color value="0,128,64"/>
  <Color value="lightblue"/>
  <Color value="yellow"/>
</Array>
```

Creating a Chart from XML

The class ***ChartXML*** creates a chart tree from an XML file. It can be used directly as an application to view an XML file. The following command displays the chart described by the XML file *filename*. (This assumes that the CLASS-PATH is correctly set.)

```
java com.imsl.chart.xml.ChartXML filename
```

Interacting with an XML Created Chart

The class `ChartXML` can also be used to create a chart tree within a larger Java program. The object's `getChart()` method is used to retrieve the created chart object. The `get(String)` method is used to obtain objects created by array tags.

Enumerated Types in Chart XML Files

In a chart XML file, the following symbolic names can be used for enumerated values. Their numerical equivalents can also be used.

These enumerated types can be "or-ed" together. For example,

```
<Attribute name="DataType" value=DATA_TYPE_LINE|DATA_TYPE_MARKER"/>
```

is allowed.

Marker Types

MARKER_TYPE_PLUS
MARKER_TYPE_ASTERISK
MARKER_TYPE_X
MARKER_TYPE_HOLLOW_SQUARE
MARKER_TYPE_FILLED_SQUARE
MARKER_TYPE_HOLLOW_TRIANGLE
MARKER_TYPE_FILLED_TRIANGLE
MARKER_TYPE_HOLLOW_DIAMOND
MARKER_TYPE_FILLED_DIAMOND
MARKER_TYPE_DIAMOND_PLUS
MARKER_TYPE_SQUARE_X
MARKER_TYPE_SQUARE_PLUS
MARKER_TYPE_OCTAGON_X
MARKER_TYPE_OCTAGON_PLUS
MARKER_TYPE_HOLLOW_CIRCLE
MARKER_TYPE_FILLED_CIRCLE
MARKER_TYPE_CIRCLE_X
MARKER_TYPE_CIRCLE_PLUS
MARKER_TYPE_CIRCLE_CIRCLE

Fill Types

FILL_TYPE_NONE
FILL_TYPE_SOLID

FILL_TYPE_GRADIENT
FILL_TYPE_PAINT

Text Alignments

TEXT_X_LEFT
TEXT_X_CENTER
TEXT_X_RIGHT
TEXT_Y_BOTTOM
TEXT_Y_CENTER
TEXT_Y_TOP

Font Styles

PLAIN
BOLD
ITALIC

Axis Parameters

AXIS_X
AXIS_X_TOP
AXIS_Y
AXIS_Y_RIGHT

Autoscale

AUTOSCALE_OFF
AUTOSCALE_DATA
AUTOSCALE_WINDOW
AUTOSCALE_NUMBER

Axis Transforms

TRANSFORM_LINEAR
TRANSFORM_LOG

Data Node Types

DATA_TYPE_LINE
DATA_TYPE_MARKER
DATA_TYPE_FILL
DATA_TYPE_PICTURE

Label Types

LABEL_TYPE_NONE
LABEL_TYPE_TITLE
LABEL_TYPE_X
LABEL_TYPE_Y
LABEL_TYPE_PERCENT

Bar Types

BAR_TYPE_VERTICAL
BAR_TYPE_HORIZONTAL

Creating Charts from General XML Files Using XSLT

JMSL can create charts from XML files that are written using a particular set of tags, but you can also generate charts from more general XML files. This is done by using `XSL Transformations` (XSLT) to transform other types of XML files into the type of XML file required by JMSL.

Example

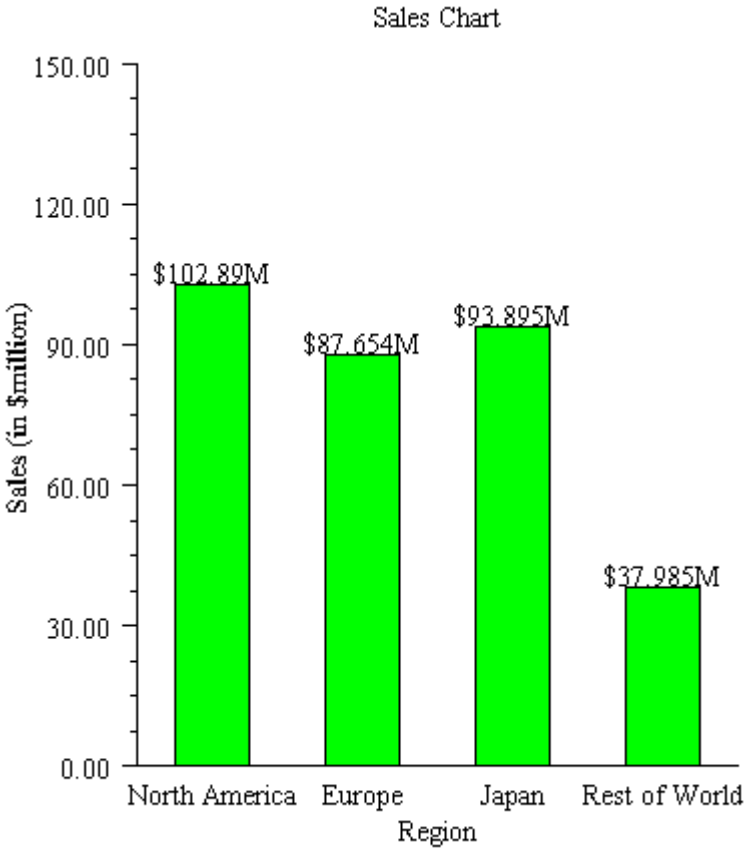
Consider the following sales report in XML that does not use the tag names and structure that JMSL requires.

```
<?xml version="1.0" encoding="UTF-8" ?>
<sales>
    <region>
        <name>North America</name>
        <amount>102.890</amount>
    </region>
    <region>
        <name>Europe</name>
        <amount>87.654</amount>
    </region>
    <region>
        <name>Japan</name>
        <amount>93.895</amount>
    </region>
    <region>
        <name>Rest of World</name>
        <amount>37.985</amount>
    </region>
</sales>
```

To create a bar chart from this XML, it must be transformed into an XML file similar to the following

```
<?xml version="1.0" encoding="UTF-8" ?>
<Chart>
    <Array id="amount">
        <Number>102.890</Number>
        <Number>87.654</Number>
        <Number>93.895</Number>
        <Number>37.985</Number>
    </Array>
    <Array id="region">
        <String value="North America"/>
        <String value="Europe"/>
        <String value="Japan"/>
        <String value="Rest of World"/>
    </Array>
```

```
    <Attribute name="FontName" value="Serif"/>
    <Attribute name="FontSize" value="14"/>
    <ChartTitle><Attribute name="Title" value="Sales Chart"/></ChartTitle>
    <AxisXY>
        <AxisY>
            <AxisTitle><Attribute name="Title" value="Sales (in $million)"/>
            </AxisTitle>
        </AxisY>
        <AxisX>
            <AxisTitle><Attribute name="Title" value="Region"/></AxisTitle>
        </AxisX>
        <Bar y="#amount">
           <Attribute name="BarType" value="BAR_TYPE_VERTICAL"/>
           <Attribute name="Labels" value="#region"/>
           <Attribute name="LabelType"  value="LABEL_TYPE_Y"/>
           <Attribute name="FillColor" value="lime"/>
           <Attribute name="TextFormat">$###.###M</Attribute>
        </Bar>
    </AxisXY>
</Chart>
```



The following XSLT file can be used to perform this transformation.

View code file

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
    <xsl:output method="xml" indent="yes"/>
    <xsl:template match="/">
        <Chart>
            <Array id="region">
                <xsl:for-each select="sales/region/name">
                    <String>
                        <xsl:apply-templates/>
                    </String>
                </xsl:for-each>
            </Array>
            <Array id="amount">
                <xsl:for-each select="sales/region/amount">
                    <Number>
                        <xsl:apply-templates/>
                    </Number>
                </xsl:for-each>
            </Array>
            <Attribute name="FontName" value="Serif"/>
            <Attribute name="FontSize" value="14"/>
            <ChartTitle>
                <Attribute name="Title" value="Sales Chart"/>
            </ChartTitle>
            <AxisXY>
                <AxisY>
                    <AxisTitle>
                        <Attribute name="Title" value="Sales (in
$million)"/>
                    </AxisTitle>
                </AxisY>
                <AxisX>
                    <AxisTitle>
                        <Attribute name="Title" value="Region"/>
                    </AxisTitle>
                </AxisX>
                <Bar y="#amount">
                    <Attribute name="BarType" value="BAR_TYPE_VERTICAL"/>
                    <Attribute name="Labels" value="#region"/>
                    <Attribute name="LabelType" value="LABEL_TYPE_Y"/>
                    <Attribute name="FillColor" value="lime"/>
                    <Attribute name="TextFormat">$###.###M</Attribute>
                </Bar>
            </AxisXY>
        </Chart>
    </xsl:template>
</xsl:stylesheet>
```

While a complete discussion of XSLT is beyond the scope of this manual, note that most of the file is the same as the target. The XSL tags are in the "xsl" namespace. In this example, the key XSL tags are the two `for` loops that reformat the data. For instance, the region names are formatted by the fragment

```
<xsl:for-each select="sales/region/name">
   <String><xsl:apply-templates/></String>
</xsl:for-each>
```

where the path "sales/region/name" selects tags in the original XML file and the body of the tag

```
<String><xsl:apply-templates/></String>
```

is copied into the new file.

The given XSL document will work for similar documents with different data, but other XML document formats may require different XSL files.

The Java code required to implement this transformation is:

```
import com.imsl.chart.JFrameChart;
import com.imsl.chart.xml.ChartXML;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.dom.DOMResult;
import org.w3c.dom.Document;

public class SampleSalesTransform {

    public static void main(String argv[]) throws Exception {
        String filenameXSL = argv[0];
        String filenameXML = argv[1];

        // Create an XML parser factory
        DocumentBuilderFactory factory =
                DocumentBuilderFactory.newInstance();
        factory.setNamespaceAware(true);
        DocumentBuilder builder = factory.newDocumentBuilder();

        // Parse the XSL file as an XML file
        Document docXSL = builder.parse(filenameXSL);
        DOMSource sourceXSL = new DOMSource(docXSL);
        sourceXSL.setSystemId(filenameXSL);

        // Create a transformation based on the XSL file
        TransformerFactory tFactory = TransformerFactory.newInstance();
        Transformer transformer = tFactory.newTransformer(sourceXSL);

        // Parse the input XML file
```

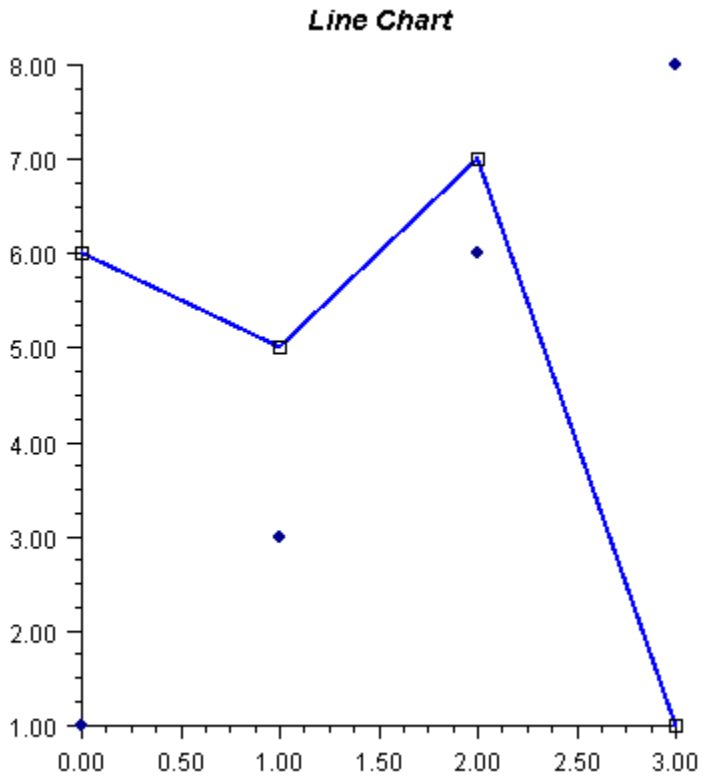
```
        Document docXML = builder.parse(filenameXML);
        DOMSource sourceXML = new DOMSource(docXML);
        sourceXML.setSystemId(filenameXML);

        // Transform the input XML file using the XSL transformer
        DOMResult result = new DOMResult();
        transformer.transform(sourceXML, result);

        // Create a chart from the transformed XML
        ChartXML chartXML = new ChartXML((Document) result.getNode());
        new JFrameChart(chartXML.getChart()).show();
    }
}
```

This program takes two arguments, the names of the XSL and XML file, and creates a JMSL chart. The transformed XML exists only in memory and is not written to disk. This class will work with any (correct) XSL and XML file; it is not dependent on the details of this example. In the interests of simplicity, error handling and validation are not implemented, as they should be for a production implementation.

Also, for simplicity, this example is a desktop application. XML and XSLT are generally more useful as part of a server-side Web application. The above code can be used with ***JspBean*** to create a server-side XML/XSLT solution.

XML Examples

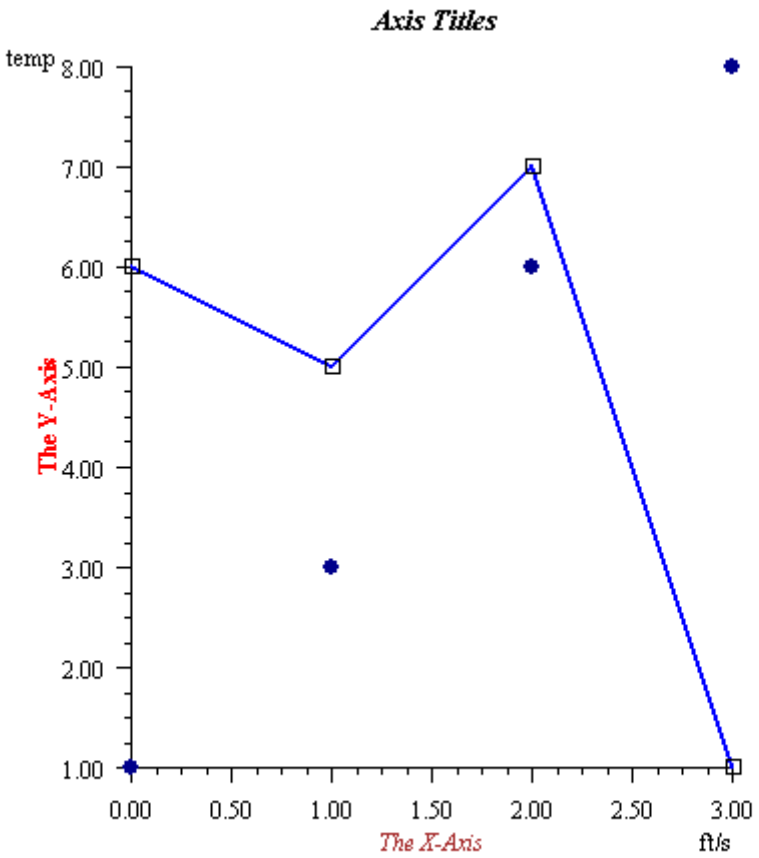
Line Chart



View code file

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE Chart PUBLIC "-//Rogue Wave Software//DTD JMSL Chart//EN"
"http://www.roguewave.com/products/jmsl/Chart.dtd">
<Chart>
    <ChartTitle>
        <Attribute name="Title">Line Chart</Attribute>
        <Attribute name="FontSize" value="14"/>
        <Attribute name="FontStyle" value="3"/>
    </ChartTitle>
    <AxisXY>
        <Data y="{6,5,7,1}">
            <Attribute name="LineColor" value="blue"/>
            <Attribute name="LineWidth" value="2.0"/>
            <Attribute name="DataType"
            value="DATA_TYPE_LINE|DATA_TYPE_MARKER"/>
            <Attribute name="MarkerType" value="MARKER_TYPE_HOLLOW_SQUARE"/>
            <Attribute name="Title" value="Blue Line"/>
            <ToolTip/>
        </Data>
        <Data y="{1,3,6,8}">
            <Attribute name="MarkerColor" value="darkblue"/>
            <Attribute name="DataType" value="DATA_TYPE_MARKER"/>
            <Attribute name="MarkerType" value="MARKER_TYPE_FILLED_CIRCLE"/>
            <Attribute name="Title" value="Markers"/>
            <ToolTip/>
        </Data>
    </AxisXY>
</Chart>
```

Axis Titles

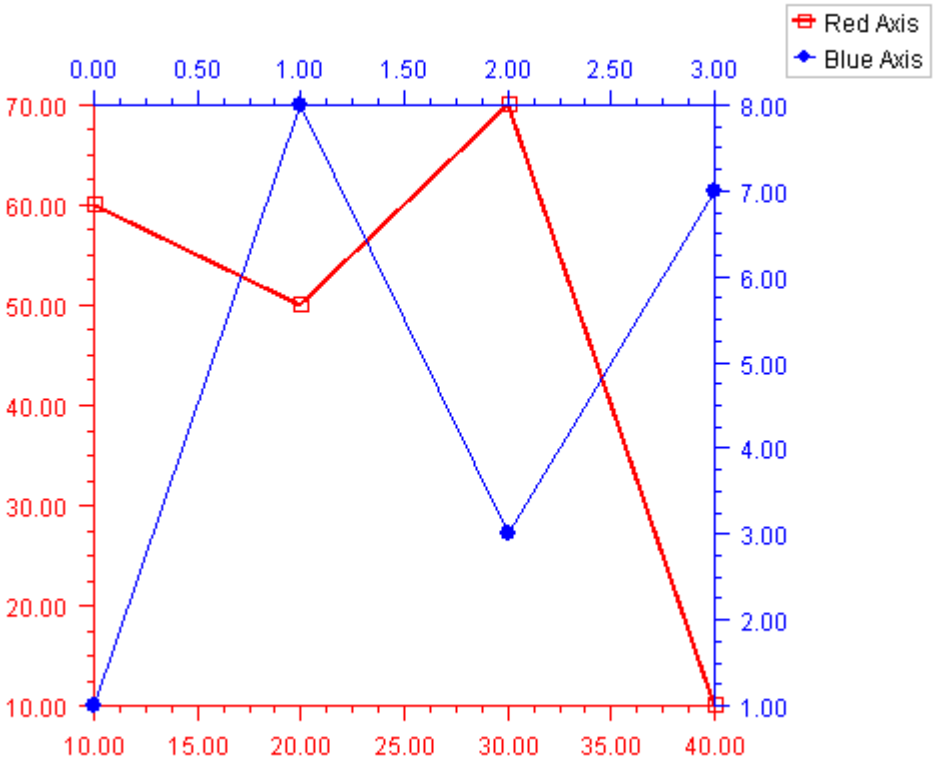


View code file

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE Chart PUBLIC "-//Rogue Wave Software//DTD JMSL Chart//EN"
"http://www.roguewave.com/products/jmsl/Chart.dtd">
<Chart>
    <Attribute name="FontName" value="Serif"/>
    <ChartTitle>
        <Attribute name="Title">Axis Titles</Attribute>
        <Attribute name="FontSize" value="14"/>
        <Attribute name="FontStyle" value="3"/>
    </ChartTitle>
    <AxisXY>
        <AxisX>
            <AxisTitle>
                <Attribute name="Title" value="The X-Axis"/>
                <Attribute name="TextColor" value="brown"/>
                <Attribute name="FontStyle" value="ITALIC"/>
            </AxisTitle>
            <AxisUnit>
```

```
                <Attribute name="Title" value="ft/s"/>
            </AxisUnit>
        </AxisX>
        <AxisY>
            <AxisTitle>
                <Attribute name="Title" value="The Y-Axis"/>
                <Attribute name="TextColor" value="red"/>
                <Attribute name="FontStyle" value="BOLD"/>
            </AxisTitle>
            <AxisUnit>
                <Attribute name="Title" value="temp"/>
            </AxisUnit>
        </AxisY>
        <Data y="{6,5,7,1}">
            <Attribute name="LineColor" value="blue"/>
            <Attribute name="LineWidth" value="2.0"/>
            <Attribute name="DataType"
            value="DATA_TYPE_LINE|DATA_TYPE_MARKER"/>
            <Attribute name="MarkerType" value="MARKER_TYPE_HOLLOW_SQUARE"/>
        </Data>
        <Data y="{1,3,6,8}">
            <Attribute name="MarkerColor" value="darkblue"/>
            <Attribute name="DataType" value="DATA_TYPE_MARKER"/>
            <Attribute name="MarkerType" value="MARKER_TYPE_FILLED_CIRCLE"/>
        </Data>
    </AxisXY>
</Chart>
```

Multiple Axes

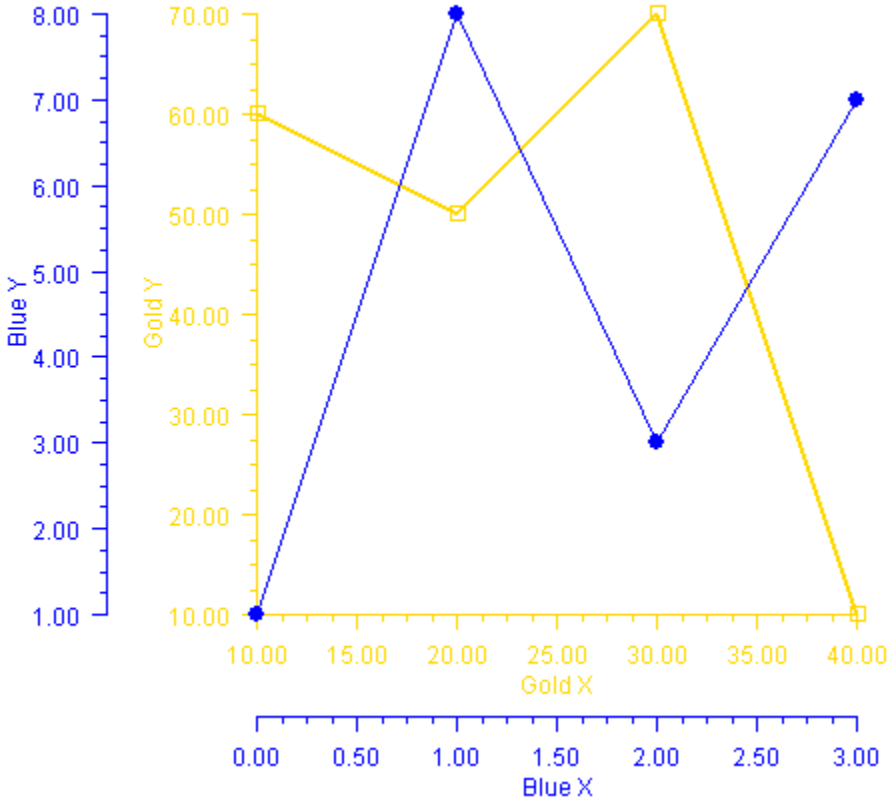


View code file

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE Chart PUBLIC "-//Rogue Wave Software//DTD JMSL Chart//EN"
"http://www.roguewave.com/products/jmsl/Chart.dtd">
<Chart>
    <Legend>
        <Attribute name="Paint" value="true"/>
        <Attribute name="Viewport" value="0.83,1.0,0.1,0.3"/>
    </Legend>
    <AxisXY>
        <Attribute name="Viewport" value="0.13,0.75,0.2,0.8"/>
        <Attribute name="LineColor" value="red"/>
        <Attribute name="MarkerColor" value="red"/>
        <Attribute name="TextColor" value="red"/>
        <Data x="10,20,30,40" y="60,50,70,10">
            <Attribute name="LineWidth" value="2.0"/>
            <Attribute name="DataType"
            value="DATA_TYPE_LINE|DATA_TYPE_MARKER"/>
            <Attribute name="MarkerType" value="MARKER_TYPE_HOLLOW_SQUARE"/>
            <Attribute name="Title" value="Red Axis"/>
```

```
            <ToolTip/>
        </Data>
    </AxisXY>
    <AxisXY>
        <Attribute name="Viewport" value="0.13,0.75,0.2,0.8"/>
        <Attribute name="LineColor" value="blue"/>
        <Attribute name="MarkerColor" value="blue"/>
        <Attribute name="TextColor" value="blue"/>
        <AxisX>
            <Attribute name="Type" value="AXIS_X_TOP"/>
        </AxisX>
        <AxisY>
            <Attribute name="Type" value="AXIS_Y_RIGHT"/>
        </AxisY>
        <Data y="1,8,3,7">
            <Attribute name="DataType"
            value="DATA_TYPE_LINE|DATA_TYPE_MARKER"/>
            <Attribute name="MarkerType" value="MARKER_TYPE_FILLED_CIRCLE"/>
            <Attribute name="Title" value="Blue Axis"/>
            <ToolTip/>
        </Data>
    </AxisXY>
</Chart>
```

Axis Offset

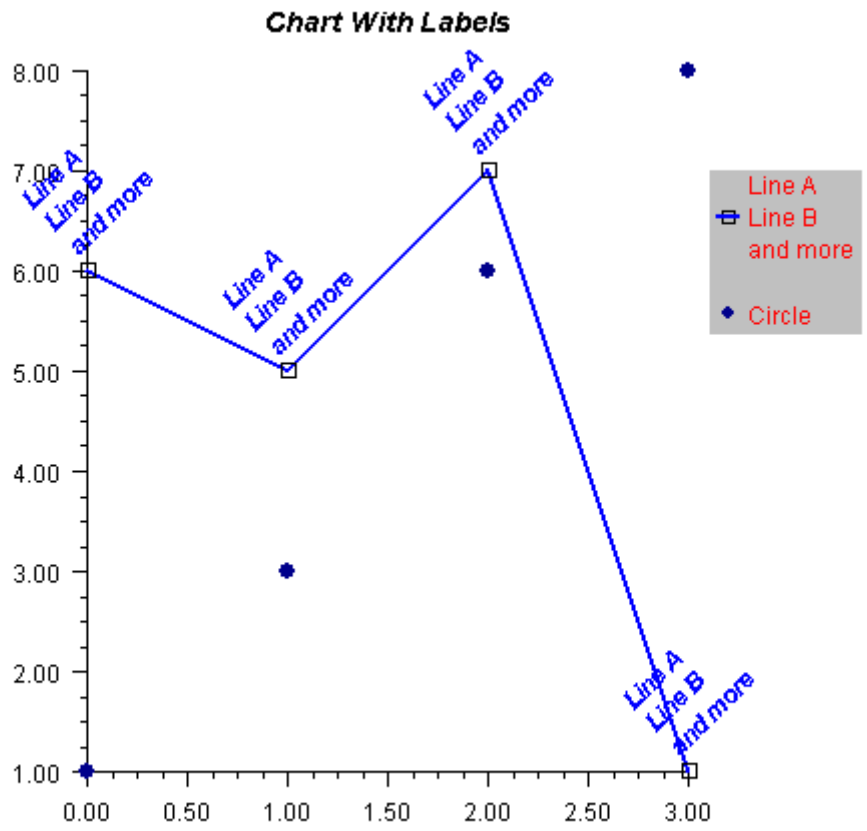


View code file

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE Chart PUBLIC "-//Rogue Wave Software//DTD JMSL Chart//EN"
"http://www.roguewave.com/products/jmsl/Chart.dtd">
<Chart>
    <AxisXY>
        <Attribute name="Viewport" value="0.3,0.9,0.1,0.7"/>
        <Attribute name="LineColor" value="gold"/>
        <Attribute name="MarkerColor" value="gold"/>
        <Attribute name="TextColor" value="gold"/>
        <AxisX>
            <AxisTitle>
                <Attribute name="Title" value="Gold X"/>
            </AxisTitle>
        </AxisX>
        <AxisY>
            <AxisTitle>
                <Attribute name="Title" value="Gold Y"/>
            </AxisTitle>
        </AxisY>
```

```
        <Data x="10,20,30,40" y="60,50,70,10">
            <Attribute name="LineWidth" value="2.0"/>
            <Attribute name="DataType"
            value="DATA_TYPE_LINE|DATA_TYPE_MARKER"/>
            <Attribute name="MarkerType" value="MARKER_TYPE_HOLLOW_SQUARE"/>
            <Attribute name="Title" value="Gold Axis"/>
        </Data>
    </AxisXY>
    <AxisXY>
        <Attribute name="Cross" value="-0.75,-0.2"/>
        <Attribute name="Viewport" value="0.3,0.9,0.1,0.7"/>
        <AxisX>
            <AxisTitle>
                <Attribute name="Title" value="Blue X"/>
            </AxisTitle>
        </AxisX>
        <AxisY>
            <AxisTitle>
                <Attribute name="Title" value="Blue Y"/>
            </AxisTitle>
        </AxisY>
        <Attribute name="LineColor" value="blue"/>
        <Attribute name="MarkerColor" value="blue"/>
        <Attribute name="TextColor" value="blue"/>
        <Data y="1,8,3,7">
            <Attribute name="DataType"
            value="DATA_TYPE_LINE|DATA_TYPE_MARKER"/>
            <Attribute name="MarkerType" value="MARKER_TYPE_FILLED_CIRCLE"/>
            <Attribute name="Title" value="Blue Axis"/>
        </Data>
    </AxisXY>
</Chart>
```

Rotated Labels

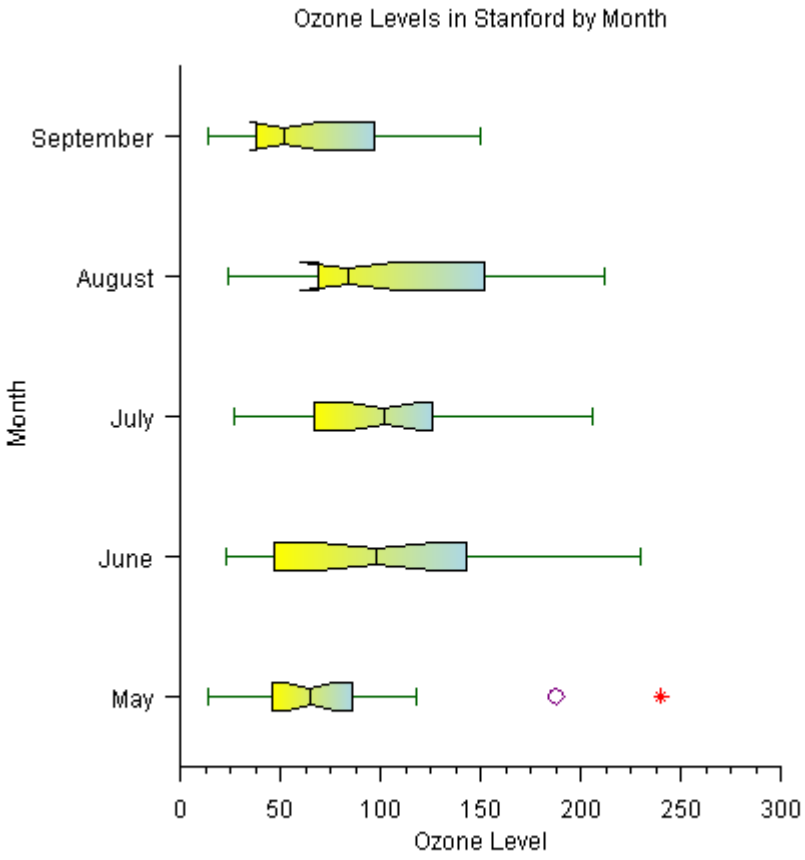


View code file

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE Chart PUBLIC "-//Rogue Wave Software//DTD JMSL Chart//EN"
"http://www.roguewave.com/products/jmsl/Chart.dtd">
<Chart>
    <ChartTitle>
        <Attribute name="Title">Chart With Labels</Attribute>
        <Attribute name="FontSize" value="14"/>
        <Attribute name="FontStyle" value="3"/>
    </ChartTitle>
    <Legend>
        <Attribute name="Paint" value="true"/>
        <Attribute name="TextColor" value="red"/>
        <Attribute name="FillType" value="FILL_TYPE_SOLID"/>
    </Legend>
    <AxisXY>
        <Attribute name="ClipData" value="false"/>
        <Data y="{6,5,7,1}">
            <Attribute name="LabelType" value="LABEL_TYPE_TITLE"/>
            <Attribute name="TextAngle" value="45"/>
```

```
            <Attribute name="LineColor" value="blue"/>
            <Attribute name="LineWidth" value="2.0"/>
            <Attribute name="DataType"
            value="DATA_TYPE_LINE|DATA_TYPE_MARKER"/>
            <Attribute name="MarkerType" value="MARKER_TYPE_HOLLOW_SQUARE"/>
            <Attribute name="TextColor" value="blue"/>
            <Attribute name="FontStyle" value="3"/>
            <Attribute name="Title">
<![CDATA[Line A
Line B
and more]]>
            </Attribute>
        </Data>
        <Data y="{1,3,6,8}">
            <Attribute name="MarkerColor" value="darkblue"/>
            <Attribute name="DataType" value="DATA_TYPE_MARKER"/>
            <Attribute name="MarkerType" value="MARKER_TYPE_FILLED_CIRCLE"/>
            <Attribute name="Title" value="Circle"/>
        </Data>
    </AxisXY>
</Chart>
```

Box Plot



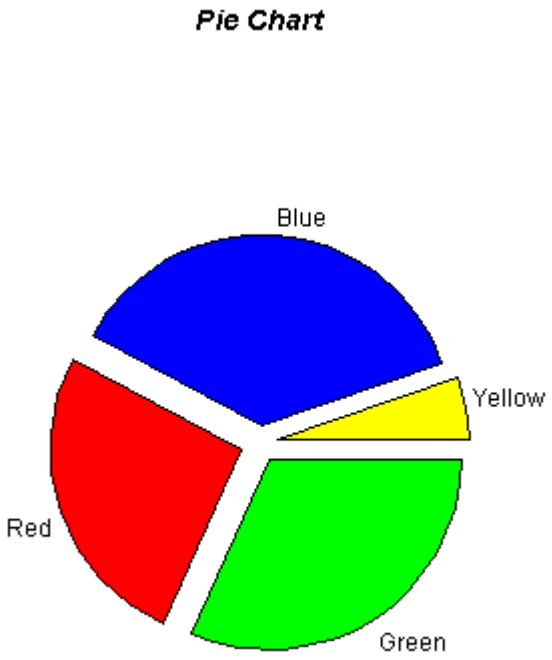
View code file

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE Chart PUBLIC "-//Rogue Wave Software//DTD JMSL Chart//EN"
"http://www.roguewave.com/products/jmsl/Chart.dtd">
<Chart>
    <Array id="ozone">
        <Array>
            <NumberList>
                    66.0, 52.0, 49.0, 64.0, 68.0, 26.0, 86.0, 52.0,
                    43.0, 75.0, 87.0, 188.0, 118.0, 103.0, 82.0,
                    71.0, 103.0, 240.0, 31.0, 40.0, 47.0, 51.0, 31.0,
                    47.0, 14.0, 71.0
            </NumberList>
        </Array>
        <Array>
            <NumberList>
                    61.0, 47.0, 196.0, 131.0, 173.0, 37.0, 47.0,
```

```
                215.0, 230.0, 69.0, 98.0, 125.0, 94.0, 72.0,
                72.0, 125.0, 143.0, 192.0, 122.0, 32.0, 114.0,
                32.0, 23.0, 71.0, 38.0, 136.0, 169.0
            </NumberList>
        </Array>
        <Array>
            <NumberList>
                152.0, 201.0, 134.0, 206.0, 92.0, 101.0, 119.0,
                124.0, 133.0, 83.0, 60.0, 124.0, 142.0, 124.0, 64.0,
                75.0, 103.0, 46.0, 68.0, 87.0, 27.0,
                73.0, 59.0, 119.0, 64.0, 111.0
            </NumberList>
        </Array>
        <Array>
            <NumberList>
                80.0, 68.0, 24.0, 24.0, 82.0, 100.0, 55.0, 91.0,
                87.0, 64.0, 170.0, 86.0, 202.0, 71.0, 85.0, 122.0,
                155.0, 80.0, 71.0, 28.0, 212.0, 80.0, 24.0,
                80.0, 169.0, 174.0, 141.0, 202.0
            </NumberList>
        </Array>
        <Array>
            <NumberList>
                113.0, 38.0, 38.0, 28.0, 52.0, 14.0, 38.0, 94.0,
                89.0, 99.0, 150.0, 146.0, 113.0, 38.0, 66.0, 38.0,
                80.0, 80.0, 99.0, 71.0, 42.0, 52.0, 33.0, 38.0,
                24.0, 61.0, 108.0, 38.0, 28.0
            </NumberList>
        </Array>
    </Array>
    <Array id="months">
        <String value="May"/>
        <String value="June"/>
        <String value="July"/>
        <String value="August"/>
        <String value="September"/>
    </Array>
    <Array id="grad">
        <Color value="yellow"/>
        <Color value="lightblue"/>
        <Color value="lightblue"/>
        <Color value="yellow"/>
    </Array>

    <ChartTitle>
        <Attribute name="Title" value="Ozone Levels in Stanford by Month"/>
    </ChartTitle>
    <AxisXY>
        <BoxPlot obs="#ozone">
            <Attribute name="BoxPlotType" value="BOXPLOT_TYPE_HORIZONTAL"/>
            <Attribute name="Labels" value="#months"/>
```

```
            <Attribute name="Notch" value="true"/>
            <Attribute name="ProportionalWidth" value="true"/>
            <Bodies>
                <Attribute name="FillType" value="FILL_TYPE_GRADIENT"/>
                <Attribute name="Gradient" value="#grad"/>
            </Bodies>
            <OutsideMarkers>
                <Attribute name="MarkerType"
                value="MARKER_TYPE_HOLLOW_CIRCLE"/>
                <Attribute name="MarkerColor" value="purple"/>
            </OutsideMarkers>
            <FarMarkers>
                <Attribute name="MarkerType" value="MARKER_TYPE_ASTERISK"/>
                <Attribute name="MarkerColor" value="red"/>
            </FarMarkers>
            <Whiskers>
                <Attribute name="MarkerColor" value="darkgreen"/>
            </Whiskers>
        </BoxPlot>
        <AxisY>
            <AxisTitle>
                <Attribute name="Title" value="Month"/>
            </AxisTitle>
        </AxisY>
        <AxisX>
            <AxisTitle>
                <Attribute name="Title" value="Ozone Level"/>
            </AxisTitle>
            <Attribute name="TextFormat" value="#"/>
        </AxisX>
    </AxisXY>
</Chart>
```

Pie Chart

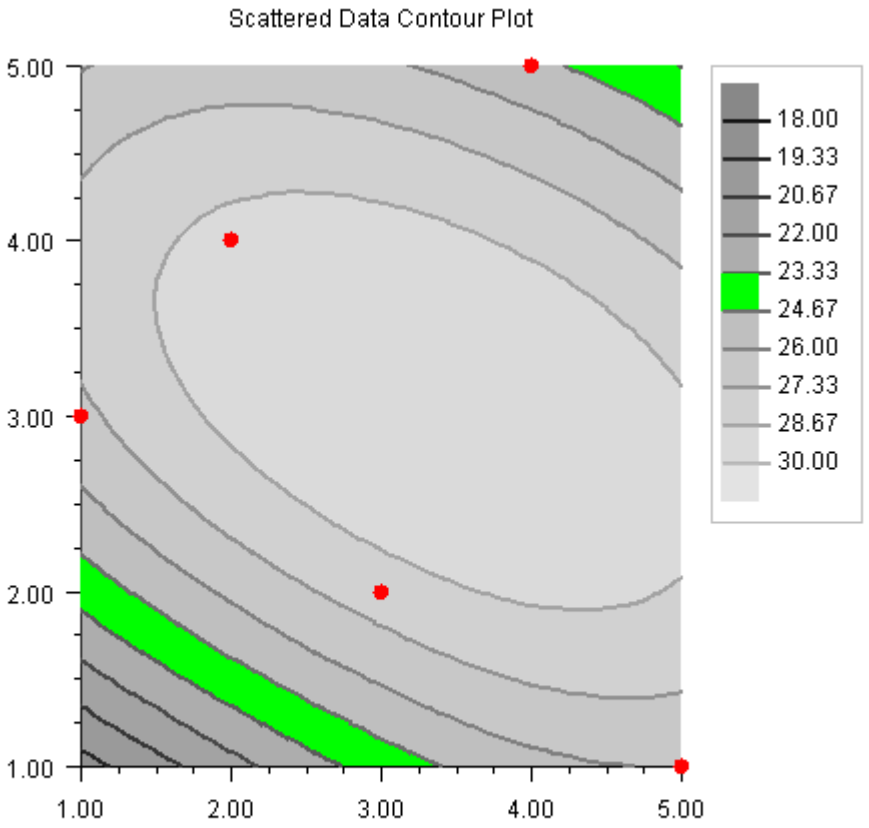


View code file

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE Chart PUBLIC "-//Rogue Wave Software//DTD JMSL Chart//EN"
"http://www.roguewave.com/products/jmsl/Chart.dtd">
<Chart>
    <ChartTitle>
        <Attribute name="Title">Pie Chart</Attribute>
        <Attribute name="FontSize" value="14"/>
        <Attribute name="FontStyle" value="3"/>
    </ChartTitle>
    <Pie y="{6,5,7,1}">
        <Attribute name="LabelType" value="LABEL_TYPE_TITLE"/>
        <Attribute name="Explode" value="0.1"/>
        <PieSlice index="0">
            <Attribute name="FillColor" value="lime"/>
            <Attribute name="Title"  value="Green"/>
            <ToolTip/>
        </PieSlice>
        <PieSlice index="1">
            <Attribute name="FillColor" value="red"/>
            <Attribute name="Title" value="Red"/>
            <ToolTip/>
        </PieSlice>
        <PieSlice index="2">
            <Attribute name="FillColor" value="blue"/>
            <Attribute name="Title" value="Blue"/>
```

```
            <ToolTip/>
        </PieSlice>
        <PieSlice index="3">
            <Attribute name="FillColor" value="yellow"/>
            <Attribute name="Title" value="Yellow"/>
            <ToolTip/>
        </PieSlice>
    </Pie>
</Chart>
```

Contour Chart

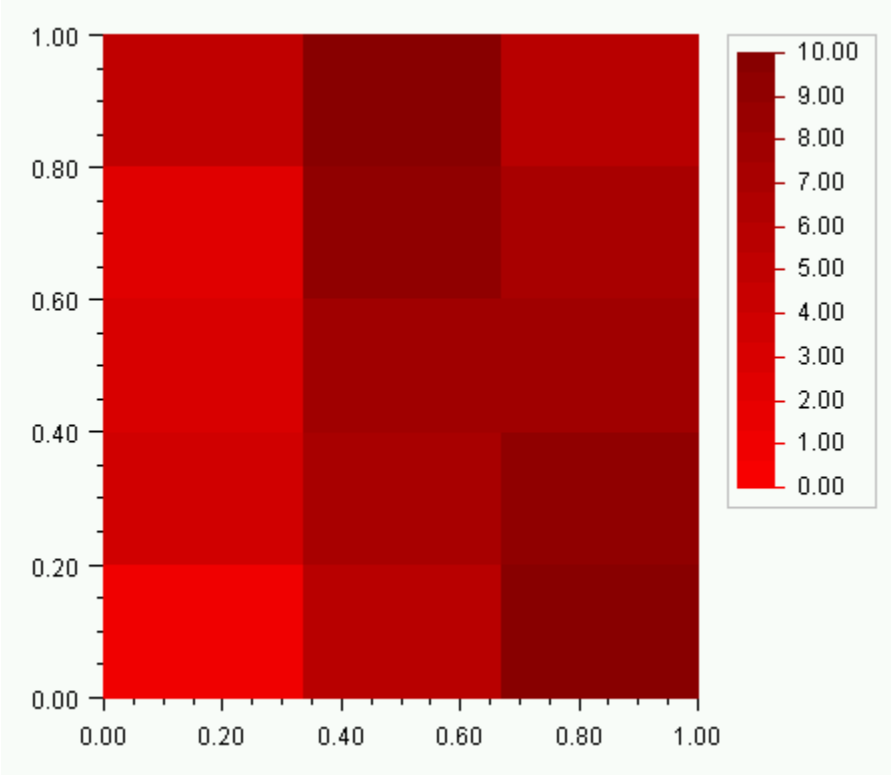


View code file

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE Chart PUBLIC "-//Rogue Wave Software//DTD JMSL Chart//EN"
"http://www.roguewave.com/products/jmsl/Chart.dtd">
<Chart>
    <Array id="x">
        <NumberList>5,3,1,2,4</NumberList>
```

```
    </Array>
    <Array id="y">
        <NumberList>1,2,3,4,5</NumberList>
    </Array>
    <Array id="z">
        <NumberList>26,28,27,29,25</NumberList>
    </Array>

    <ChartTitle>
        <Attribute name="Title" value="Scattered Data Contour Plot"/>
    </ChartTitle>
    <AxisXY>
        <Contour xGrid="#x"  yGrid="#y"  zData="#z">
            <ContourLegend>
                <Attribute name="Paint" value="true"/>
            </ContourLegend>
            <ContourLevel index="5">
                <Attribute name="FillColor" value="lime"/>
            </ContourLevel>
        </Contour>
        <Data x="#x" y="#y">
            <Attribute name="DataType" value="DATA_TYPE_MARKER"/>
            <Attribute name="MarkerType" value="MARKER_TYPE_FILLED_CIRCLE"/>
            <Attribute name="MarkerColor" value="red"/>
        </Data>
    </AxisXY>
</Chart>
```

Heatmap

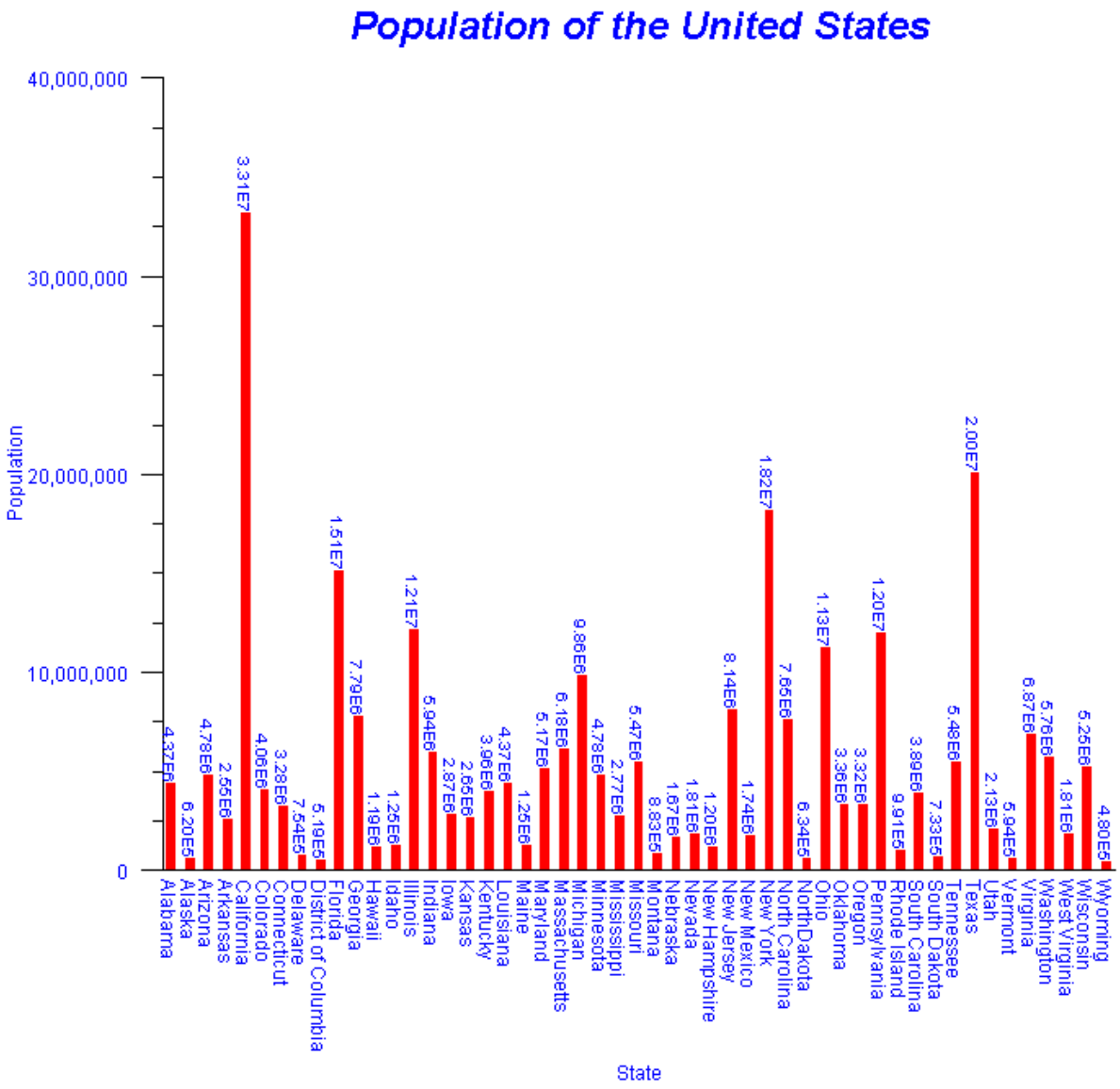


View code file

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE Chart PUBLIC "-//Rogue Wave Software//DTD JMSL Chart//EN"
"http://www.roguewave.com/products/jmsl/Chart.dtd">
<Chart>
    <Array id="data">
        <Array>
            <NumberList>23, 48, 16, 56</NumberList>
        </Array>
        <Array>
            <NumberList>89, 74, 54, 32</NumberList>
        </Array>
        <Array>
            <NumberList>12, 45, 18, 9</NumberList>
        </Array>
        <Array>
            <NumberList>72, 15, 42, 92</NumberList>
        </Array>
        <Array>
            <NumberList>63, 36, 78, 29</NumberList>
        </Array>
    </Array>

    <ChartTitle>
```

```
        <Attribute name="Title">Sample Heatmap</Attribute>
        <Attribute name="FontSize" value="14"/>
        <Attribute name="FontStyle" value="3"/>
    </ChartTitle>
    <AxisXY>
        <Heatmap xmin="0.0" xmax="5.0"
            ymin="0.0" ymax="4.0"
            zmin="0.0" zmax="100.0"
            data="#data" colormap="BLUE_RED">
            <HeatmapLegend>
                <Attribute name="Paint" value="true"/>
            </HeatmapLegend>
        </Heatmap>
    </AxisXY>
</Chart>
```

Bar Chart of States



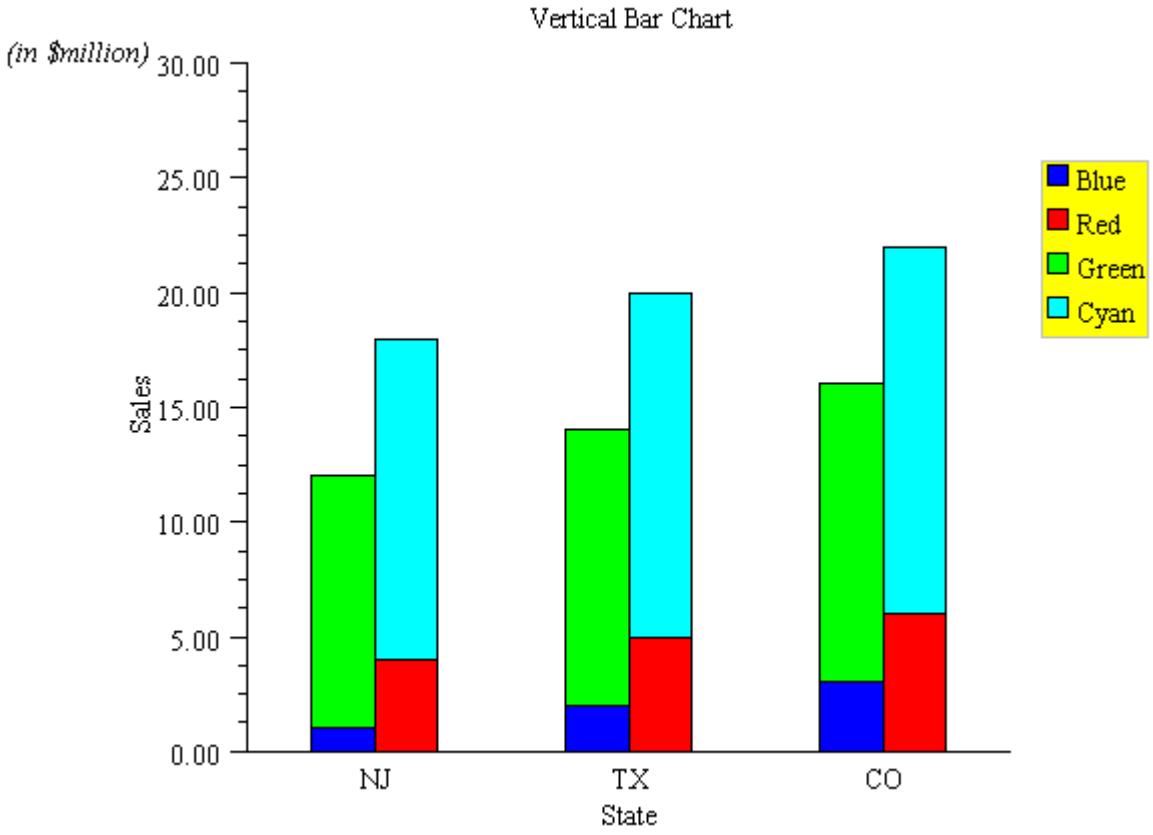
View code file

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE Chart PUBLIC "-//Rogue Wave Software//DTD JMSL Chart//EN"
"http://www.roguewave.com/products/jmsl/Chart.dtd">
<Chart>
<!--Data from http://www.census.gov/population/estimates/state/st-99-3.txt--
>
    <Array id="population">
        <NumberList>
            4369862,619500,4778332,2551373,33145121,4056133
            3282031,753538,519000,15111244,7788240,1185497,
            1251700,12128370,5942901,2869413,2654052,3960825,
            4372035,1253040,5171634,6175169,9863775,4775508,
```

```
            2768619,5468338,882779,1666028,1809253,1201134,
            8143412,1739844,18196601,7650789,633666,11256654,
            3358044,3316154,11994016,990819,3885736,733133,
            5483535,20044141,2129836,593740,6872912,5756361,
            1806928,5250446,479602
        </NumberList>
    </Array>
    <Array id="states">
        <String value="Alabama"/>
        <String value="Alaska"/>
        <String value="Arizona"/>
        <String value="Arkansas"/>
        <String value="California"/>
        <String value="Colorado"/>
        <String value="Connecticut"/>
        <String value="Delaware"/>
        <String value="District of Columbia"/>
        <String value="Florida"/>
        <String value="Georgia"/>
        <String value="Hawaii"/>
        <String value="Idaho"/>
        <String value="Illinois"/>
        <String value="Indiana"/>
        <String value="Iowa"/>
        <String value="Kansas"/>
        <String value="Kentucky"/>
        <String value="Louisiana"/>
        <String value="Maine"/>
        <String value="Maryland"/>
        <String value="Massachusetts"/>
        <String value="Michigan"/>
        <String value="Minnesota"/>
        <String value="Mississippi"/>
        <String value="Missouri"/>
        <String value="Montana"/>
        <String value="Nebraska"/>
        <String value="Nevada"/>
        <String value="New Hampshire"/>
        <String value="New Jersey"/>
        <String value="New Mexico"/>
        <String value="New York"/>
        <String value="North Carolina"/>
        <String value="NorthDakota"/>
        <String value="Ohio"/>
        <String value="Oklahoma"/>
        <String value="Oregon"/>
        <String value="Pennsylvania"/>
        <String value="Rhode Island"/>
        <String value="South Carolina"/>
        <String value="South Dakota"/>
        <String value="Tennessee"/>
```

```
        <String value="Texas"/>
        <String value="Utah"/>
        <String value="Vermont"/>
        <String value="Virginia"/>
        <String value="Washington"/>
        <String value="West Virginia"/>
        <String value="Wisconsin"/>
        <String value="Wyoming"/>
    </Array>

    <Attribute name="TextColor" value="blue"/>
    <Background>
        <Attribute name="FillColor" value="white"/>
        <Attribute name="FontSize" value="24"/>
    </Background>
    <ChartTitle>
        <Attribute name="Title">Population of the United States</Attribute>
        <Attribute name="FontSize" value="24"/>
        <Attribute name="FontStyle" value="3"/>
    </ChartTitle>
    <AxisXY>
        <AxisY>
            <AxisTitle>
                <Attribute name="Title" value="Population"/>
            </AxisTitle>
            <Attribute name="TextFormat">###,###</Attribute>
        </AxisY>
        <AxisX>
            <AxisTitle>
                <Attribute name="Title" value="State"/>
            </AxisTitle>
            <AxisLabel>
                <Attribute name="TextAngle" value="270"/>
            </AxisLabel>
        </AxisX>
        <Bar y="#population">
            <Attribute name="BarType" value="BAR_TYPE_VERTICAL"/>
            <Attribute name="Labels" value="#states"/>
            <Attribute name="FillColor" value="red"/>
            <Attribute name="FillOutlineType" value="FILL_TYPE_NONE"/>
            <Attribute name="LabelType"  value="LABEL_TYPE_Y"/>
            <Attribute name="TextAngle" value="270"/>
            <Attribute name="TextFormat" value="0.00E0"/>
            <Attribute name="FontSize" value="10"/>
        </Bar>
    </AxisXY>
</Chart>
```

Bar Chart



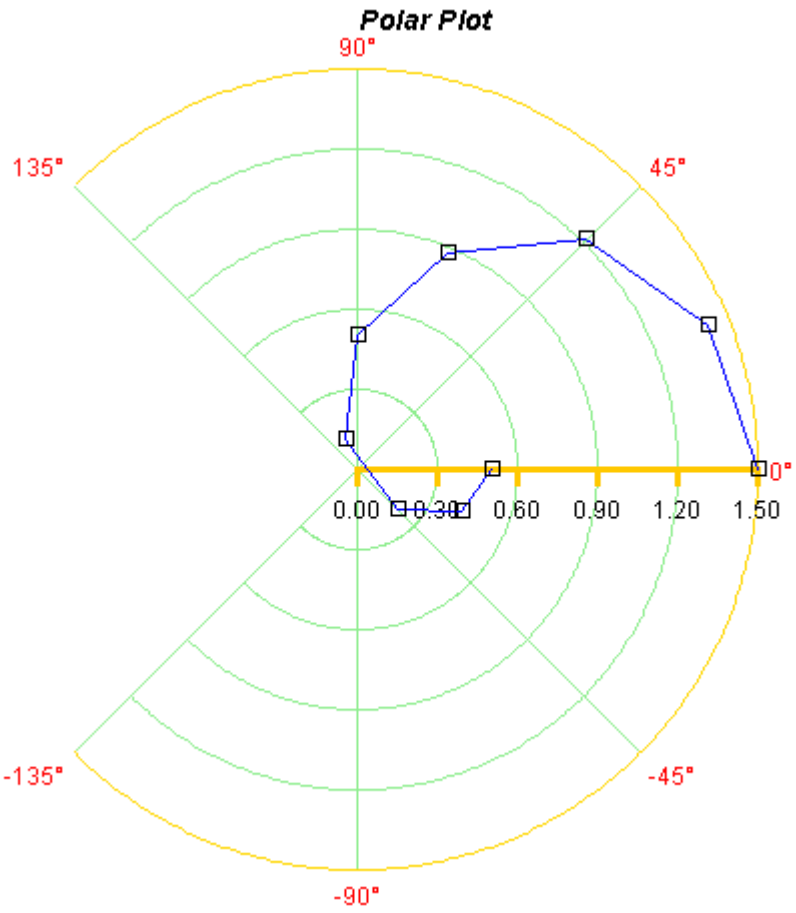
View code file

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE Chart PUBLIC "-//Rogue Wave Software//DTD JMSL Chart//EN"
"http://www.roguewave.com/products/jmsl/Chart.dtd">
<Chart>
    <Array id="foo">
        <Array>
            <Array>
                <NumberList value="1,2,3"/>
            </Array>
            <Array>
                <NumberList value="4,5,6"/>
            </Array>
        </Array>
        <Array>
            <Array>
                <NumberList value="11,12,13"/>
            </Array>
            <Array>
                <NumberList value="14,15,16"/>
            </Array>
```

```
    </Array>
</Array>
<Array id="states">
    <String value="NJ"/>
    <String value="TX"/>
    <String value="CO"/>
</Array>
<Array id="bg">
    <Color value="lightblue"/>
    <Color value="lightblue"/>
    <Color value="yellow"/>
    <Color value="yellow"/>
</Array>

<Attribute name="FontName" value="Serif"/>
<Attribute name="FontSize" value="14"/>
<ChartTitle>
    <Attribute name="Title" value="Vertical Bar Chart"/>
</ChartTitle>
<Background>
    <Attribute name="Gradient" value="#bg"/>
</Background>
<Legend>
    <Attribute name="Paint" value="true"/>
    <Attribute name="FillColor" value="yellow"/>
    <Attribute name="FillType" value="FILL_TYPE_SOLID"/>
</Legend>
<AxisXY>
    <AxisY>
        <AxisTitle>
            <Attribute name="Title" value="Sales"/>
        </AxisTitle>
        <AxisUnit>
            <Attribute name="Title" value="(in $million)"/>
            <Attribute name="FontStyle" value="ITALIC"/>
        </AxisUnit>
    </AxisY>
    <AxisX>
        <AxisTitle>
            <Attribute name="Title" value="State"/>
        </AxisTitle>
    </AxisX>
    <Bar x="{0,1,2}"  y="#foo">
        <Attribute name="BarType" value="BAR_TYPE_VERTICAL"/>
        <Attribute name="Labels" value="#states"/>
        <BarSet index="{0,0}">
            <Attribute name="FillColor" value="blue"/>
            <Attribute name="Title" value="Blue"/>
        </BarSet>
        <BarSet index="{0,1}">
            <Attribute name="FillColor" value="red"/>
```

```
            <Attribute name="Title" value="Red"/>
        </BarSet>
        <BarSet index="{1,0}">
            <Attribute name="FillColor" value="lime"/>
            <Attribute name="Title" value="Green"/>
        </BarSet>
        <BarSet index="{1,1}">
            <Attribute name="FillColor" value="cyan"/>
            <Attribute name="Title" value="Cyan"/>
        </BarSet>
    </Bar>
  </AxisXY>
</Chart>
```

Polar Plot



View code file

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE Chart PUBLIC "-//Rogue Wave Software//DTD JMSL Chart//EN"
"http://www.roguewave.com/products/jmsl/Chart.dtd">
```

```
<Chart>
    <ChartTitle>
        <Attribute name="Title">Polar Plot</Attribute>
        <Attribute name="FontSize" value="14"/>
        <Attribute name="FontStyle" value="3"/>
    </ChartTitle>
    <Polar>
        <Attribute name="LineColor" value="gold"/>
        <AxisR>
            <Attribute name="LineColor" value="orange"/>
            <AxisRLine>
                <Attribute name="LineWidth" value="3"/>
            </AxisRLine>
            <AxisRMajorTick>
                <Attribute name="LineWidth" value="3"/>
            </AxisRMajorTick>
        </AxisR>
        <AxisTheta>
            <Attribute name="Window"
            value="{-2.35619449019,2.35619449019}"/>
            <Attribute name="Number" value="7"/>
            <Attribute name="TextColor" value="red"/>
        </AxisTheta>
        <GridPolar>
            <Attribute name="LineColor" value="lightgreen"/>
        </GridPolar>
        <Data  x="{1.50,1.42,1.21,0.88,0.50,0.12,-0.21,-0.42,-0.50}"
               y="{0.00,0.39,0.79,1.18,1.57,1.96,2.36,2.75,3.14}">
            <Attribute name="LineColor" value="blue"/>
            <Attribute name="DataType"
            value="DATA_TYPE_LINE|DATA_TYPE_MARKER"/>
            <Attribute name="MarkerType" value="MARKER_TYPE_HOLLOW_SQUARE"/>
            <Attribute name="Title" value="Blue Line"/>
            <ToolTip/>
        </Data>
    </Polar>
</Chart>
```

Chapter 6 Actions

JMSL Actions

JMSL supports the following actions:

Picking

The pick mechanism allows ***MouseEvent***s to be associated with chart nodes. The pick mechanism follows the Java listener pattern.

The class that is to handle the pick events implements the ***PickListener*** interface. This requires the implementation of the method `pickPerformed(PickEvent)`.

The `PickListener` is then added to a ***Chart*** node, using the node's `addPickListener` method.

A pick is fired by calling the method `pick(MouseEvent)` in the top-level `Chart` node. Normally this is done in response to a mouse click. Mouse clicks can be detected by implementing the ***MouseListener*** interface and adding it to the chart's container.

Example

In this example, when a bar is clicked it toggles its color between blue and red.

A `MouseListener` is implemented as an inner class. Its `mouseClicked` method calls `Chart.pick(MouseEvent)`.

The class `SamplePick` implements the `PickListener` interface. It is not an inner class only because of the size of the `pickPerformed` method. The pick listener is added to the chart node bar. This means that it is active for that node and its children but is not active for the axes and other nodes.

View code file

```
import com.imsl.chart.*;
import java.awt.Color;

public class SamplePick extends JFrameChart implements PickListener {
    static private final double x[] = {0, 1, 2, 3};
    static private final double y[] = {4, 1, 2, 5};

    public SamplePick() {
        // Create a bar chart
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        Bar bar = new Bar(axis, x, y);
        bar.setBarType(Bar.BAR_TYPE_VERTICAL);
        bar.setLabels(new String[]{"A","B","C"});
        bar.setFillColor(Color.blue);

        // Add the pick listener to the bar
        bar.addPickListener(this);

        // Add the mouse listener
        addMouseListener(new java.awt.event.MouseAdapter() {
```

```
            public void mouseClicked(java.awt.event.MouseEvent event) {
                getChart().pick(event);
            }
        });
    }

    public void pickPerformed(PickEvent event) {
        // Track the number of hits on this bar
        ChartNode node = event.getNode();
        int count = node.getIntegerAttribute("pick.count", 0);
        count++;
        node.setAttribute("pick.count", new Integer(count));

        // Change the bar's color depending on the number of times
        // it has been picked.
        Color color = ((count%2==0) ? Color.blue : Color.red);
        node.setFillColor(color);
        node.setLineColor(color);
        repaint();
    }

    public static void main(String argv[]) {
        new SamplePick().setVisible(true);
    }
}
```

Zoom

JMSL provides the functionality to implement a variety of zoom policies instead of implementing a single specific zoom policy. The following example shows how to build a zoom policy that draws a rubber-band box in a scatter plot and zooms both the *x*-axis and *y*-axis to correspond to the region selected. Zoom out is also implemented.

This code can be used as the basis of an implementation of other zoom policies. For example, with a function plot one might want to only change the x-axis in response to a zoom and leave the *y*-axis alone. In other situations it may be desirable to zoom in or out a fixed amount around the location of a mouse click.

Example

In the constructor:

- A scatter plot is created,
- Zoom items are added to the main menu,
- A ***MouseListener*** is added to the panel. A ***JFrameChart*** contains a number of components, including a ***JPanel*** in which the chart is drawn. It is important that the MouseListener be added to the component in which the chart is drawn, so that the ***MouseEvent*** coordinates are the same as the device coordinates of the chart.

A mouse press event starts the zoom. In response to the event a ***MouseMotionListener*** is added to the panel to listen for mouse drag events. The location of this initial event is also recorded.

A mouse drag event resizes the area of the zoom. In response, the old rubber-band box is erased and a new box is drawn. The new box has the initial mouse press location as one corner and this drag event location as the opposite corner. The location of this event is stored in (`xLast`,`yLast`).

A mouse release event completes the zoom. In response,

- The `MouseMotionListener` is removed from the panel,
- The existing `Window` attributes for the *x*-axis and *y*-axis are saved on a stack. They may be used later to zoom out.
- The new `Window` is computed. The `Window` is the range, in user coordinates, along an axis. The method `Axis.mapDeviceToUser` is used to map the device (pixel) coordinates to the user coordinate space. The new windows are constrained to be inside of the existing window.
- Autoscaling is turned off, so that the new `Window` values will be used without further modification.
- The chart is redrawn with the new windows.

The menu items, added by the constructor to the main menu, can be used to zoom out. The added menu items added `"this"` as an ***ActionListener***, so the `actionPerformed` method is called when these menu items are selected. The `actionPerformed` method implements `"Out"` by popping previous Window attribute values off a stack. It implements `"Restore"` by re-enabling autoscaling.

`View code file`

```
import com.imsl.chart.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Stack;
import javax.swing.JMenu;
import javax.swing.JMenuItem;

public class SampleZoom extends JFrameChart
        implements MouseListener, MouseMotionListener, ActionListener {

    private javax.swing.JPanel panel;
    private AxisXY axis;
    private int xFirst,  yFirst;
    private int xLast,  yLast;
    private Stack stack;

    public SampleZoom() {
        // create a scatter plot with some random points
        Chart chart = getChart();
        axis = new AxisXY(chart);
        int n = 1000;
        double x[] = new double[n];
        double y[] = new double[n];
        for (int k = 0; k < n; k++) {
            x[k] = 10. * Math.random();
            y[k] = 10. * Math.random();
        }
        Data data = new Data(axis, x, y);
        data.setDataType(Data.DATA_TYPE_MARKER);

        // add menu "Zoom" menu
        JMenu jMenuZoom = new JMenu("Zoom");
        jMenuZoom.setMnemonic('Z');
        addMenuItem(jMenuZoom, "Out");
        addMenuItem(jMenuZoom, "Restore");
        getJMenuBar().add(jMenuZoom);

        // save x-axis and y-axis Window attributes on this stack.
        // they are used to zoom out one level.
        stack = new Stack();

        // listen for mouse press events
        panel = getPanel();
        panel.addMouseListener(this);
    }

    /**
     *  Add a menu item
     */
    private void addMenuItem(JMenu jMenuZoom, String title) {
```

```
        JMenuItem jMenuItem = new JMenuItem(title);
        jMenuItem.setActionCommand(title);
        jMenuItem.setMnemonic(title.charAt(0));
        jMenuZoom.add(jMenuItem);
        jMenuItem.addActionListener(this);
    }

    /**
     *  A mouse press starts the zoom
     *  Record location of this point and listen for drag (motion) events.
     */
    public void mousePressed(MouseEvent event) {
        xFirst = xLast = event.getX();
        yFirst = yLast = event.getY();
        panel.addMouseMotionListener(this);
    }

    /**
     *  A mouse release ends the zoom.
     *  Stop listening for drag events and update the
     *  Window attribute in the x and y axes.
     */
    public void mouseReleased(MouseEvent event) {
        panel.removeMouseMotionListener(this);

        // ignore degenerate zooms
        if (xFirst == xLast || yFirst == yLast) {
            repaint();
            return;
        }

        // get window and convert to user coordinates
        double windowX[] = (double[]) axis.getAxisX().getWindow();
        double windowY[] = (double[]) axis.getAxisY().getWindow();

        // save the windows on the stack (for zoom out option)
        stack.add(windowX);
        stack.add(windowY);

        // get user coordinate of left-upper corner of the box
        // limit it to stay inside current window
        double boxLU[] = new double[2];
        int x = Math.min(xFirst, xLast);
        int y = Math.min(yFirst, yLast);
        axis.mapDeviceToUser(x, y, boxLU);

        // get user coordinate of right-lower corner of the box
        // limit it to stay inside current window
        double boxRL[] = new double[2];
        x = Math.max(xFirst, xLast);
        y = Math.max(yFirst, yLast);
```

```
        axis.mapDeviceToUser(x, y, boxRL);

        // set axis window to range of rubber-band box
        // and disable autoscale to force use of window settings
        axis.setAutoscaleInput(ChartNode.AUTOSCALE_OFF);
        double xa = Math.max(windowX[0], boxLU[0]);
        double xb = Math.min(windowX[1], boxRL[0]);
        double ya = Math.max(windowY[0], boxRL[1]);
        double yb = Math.min(windowY[1], boxLU[1]);
        axis.getAxisX().setWindow(xa, xb);
        axis.getAxisY().setWindow(ya, yb);

        // force redraw of the chart
        repaint();
    }

    /**
     *  A drag event continues the zoom.
     *  Erase the old rubber-band box and draw a new one.
     *  Also keep track of the location of this event.
     */
    public void mouseDragged(MouseEvent event) {
        // erase previous rectangle
        Graphics g = panel.getGraphics();
        g.setXORMode(Color.cyan);  // complement of drawing color red
        drawBox(g);

        // draw new rectangle
        xLast = event.getX();
        yLast = event.getY();
        g.setPaintMode();
        g.setColor(Color.red);
        drawBox(g);
    }

    public void mouseMoved(MouseEvent event) {
    }

    public void mouseEntered(MouseEvent event) {
    }

    public void mouseClicked(MouseEvent event) {
    }

    public void mouseExited(MouseEvent event) {
    }

    /**
     *  Draw a box with (xFirst,yFirst) and (xLast,yLast) as its corners.
     */
    private void drawBox(Graphics g) {
```

```
            int x = Math.min(xFirst, xLast);
            int y = Math.min(yFirst, yLast);
            int w = Math.abs(xLast - xFirst);
            int h = Math.abs(yLast - yFirst);
            g.drawRect(x, y, w, h);
        }

        /**
         *  Handle menu item events.
         *  Command "Out" zooms out one level.
         *  Command "Restore" undoes all zooms.
         */
        public void actionPerformed(ActionEvent actionEvent) {
            String command = actionEvent.getActionCommand();
            if (command.equals("Out")) {
                try {
                    // zoom out by restoring window settings from the stack
                    axis.getAxisY().setWindow((double[]) stack.pop());
                    axis.getAxisX().setWindow((double[]) stack.pop());
                } catch (java.util.EmptyStackException e) {
                    // no more levels to zoom out
                    // restore original setting by turning on autoscale
                    axis.setAutoscaleInput(ChartNode.AUTOSCALE_DATA);
                }
            } else if (command.equals("Restore")) {
                // restore original setting by turning on autoscale
                // an empty stack
                axis.setAutoscaleInput(ChartNode.AUTOSCALE_DATA);
                stack.empty();
            }
            // force redraw of the chart
            repaint();
        }

        public static void main(String argv[]) {
            new SampleZoom().setVisible(true);
        }
    }
```

Printing

Printing from JFrameChart

The ***JFrameChart*** class, used to build most of the examples in this manual, includes a print option under the file menu. This option prints the chart as large as possible, without distortion, and centered on the page.

Printable Interface

The Java ***Printable*** interface is used to print a single page from Java. It is implemented by the class ***Chart***. The following code fragment shows how to print a chart using its `Printable` interface and ***PrinterJob***.

```
public void print() {
    PrinterJob printJob = PrinterJob.getPrinterJob();
    printJob.setPrintable(chart);
    if (printJob.printDialog()) {
        try {
            printJob.print();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

The `PrinterJob.printDialog()` method displays a dialog box that may contain a "Property Sheet." This property sheet is supplied by the hardware printer driver, not by Java. User changes to property sheet values are not reflected back to Java. Specifically, while a property sheet may allow the user to select landscape or portrait mode, these settings are not reflected back to the `PrinterJob` object. The correct way to set the orientation is by using the `PrinterJob.pageDialog(PageFormat)` method.

Pageable

The Java ***Pageable*** interface is used to print a complex document from Java. The method `Chart.paintChart(Graphics)` can be used to implement a `Pageable` interface. The following method is the implementation of the `Printable` interface in `Chart`. It uses the method `Chart.paintChart(Graphics)`.

Note that the chart is drawn to fit the Component associated with the chart. The `getScreenSize` method returns the size of this component. The following method scales the drawing to compensate for the difference in size between the screen and the paper.

View code file

```
public int print(Graphics graphics, PageFormat pageFormat,
```

```
            int param)
            throws PrinterException {
        if (param >= 1) {
            return Printable.NO_SUCH_PAGE;
        }

        // translate, so we do not clip on the left
        // scale to fill the page

        graphics.translate((int) pageFormat.getImageableX(),
                (int) pageFormat.getImageableY());

        // scale to fill the page
        double dw = pageFormat.getImageableWidth();
        double dh = pageFormat.getImageableHeight();
        Dimension screenSize = getScreenSize();
        double xScale = dw / screenSize.width;
        double yScale = dh / screenSize.height;
        double scale = Math.min(xScale, yScale);

        // center the chart on the page
        double tx = 0.0;
        double ty = 0.0;
        if (xScale > scale) {
            tx = 0.5 * (xScale - scale) * screenSize.width;
        } else {
            ty = 0.5 * (yScale - scale) * screenSize.height;
        }
        ((Graphics2D) graphics).translate(tx, ty);
        ((Graphics2D) graphics).scale(scale, scale);

        Dimension bounds = new Dimension((int) dw, (int) dh);
        paint(new Draw(graphics, bounds));
        return Printable.PAGE_EXISTS;
    }
```

Serialization

The chart tree can be serialized into a stream of bytes and recreated from that stream.

Serialization is used for lightweight persistence and for communication via sockets or Remote Method Invocation (RMI). See (http://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html) for more details about serialization in general.

Serialization of JMSL classes may not be compatible with future JMSL releases. The current serialization support is appropriate for short term storage or RMI between applications running the same version of JMSL.

To serialize a chart tree, create an ***ObjectOutputStream*** and then use its `writeObject` method to write out the ***Chart*** object. The chart tree can be reconstructed from the stream by creating a ***ObjectInputStream*** and using its `readObject` method to read the `Chart` object.

The chart tree cannot be serialized if it contains any non-serializable objects, such as attribute values. All of the JMSL chart nodes are serializable. ***Image*** objects are not serializable, but ***ImageIcon*** objects are serializable. Standard Java ***Paint*** objects, such as ***TexturePaint*** objects, are not serializable. But the Paint objects generated by the JMSL class ***FillPaint*** are serializable.

Example

When the following code is executed without any arguments, it creates a chart and serializes it to the file `sample.dat`. When it is executed with the single argument `-read`, it reads the file `sample.dat` and recreates the tree.

View code file

```
import com.imsl.chart.*;
import java.io.*;
import java.awt.*;
import javax.swing.ImageIcon;

public class SampleSerialization extends JFrameChart {

    private void createChart() {
        Chart chart = getChart();
        AxisXY axis = new AxisXY(chart);
        chart.getBackground().setFillType(ChartNode.FILL_TYPE_PAINT);
        Paint paint =
                FillPaint.checkerboard(24, Color.yellow, Color.orange);
        chart.getBackground().setFillPaint(paint);
        double y[] = {4, 6, 2, 1, 8};
        Data data = new Data(axis, y);
        data.setDataType(Data.DATA_TYPE_LINE);
        data.setLineColor(java.awt.Color.blue);
    }
```

```
    static private java.awt.Image loadImage(String name) {
        return new ImageIcon(Data.class.getResource(name)).getImage();
    }

    private void writeChart() throws IOException {
        FileOutputStream fos = new FileOutputStream("sample.dat");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(getChart());
        oos.flush();
        fos.close();
    }

    private Chart readChart() throws IOException, ClassNotFoundException {
        FileInputStream fis = new FileInputStream("sample.dat");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Chart chart = (Chart) ois.readObject();
        fis.close();
        return chart;
    }

    public static void main(String argv[]) throws Exception {
        if (argv.length == 0) {
            SampleSerialization sample = new SampleSerialization();
            sample.createChart();
            sample.writeChart();
            System.exit(0);
        } else if (argv[0].equals("-read")) {
            SampleSerialization sample = new SampleSerialization();
            Chart chart = sample.readChart();
            sample.setChart(chart);
            sample.setVisible(true);
        } else {
            System.out.println("usage: java SampleSerialization [-read]");
        }
    }
}
```

PART II

3D Charting

Chapter 7 Introduction to Chart 3D

Java 3D

The JMSL 3D chart package is built on top of Java 3D, an open source Java package which renders accelerated 3D graphics using OpenGL or Direct3D.

Java 3D Parent project is at `https://java3d.java.net/`.

Knowledge of Java3D is not required to use the 3D chart package, but Java 3D must be installed to run a chart 3D application.

Overview

The JMSL 3D chart package can be used to create customizable 3D charts using Java and Java3D. Knowledge of Java3D is not required.

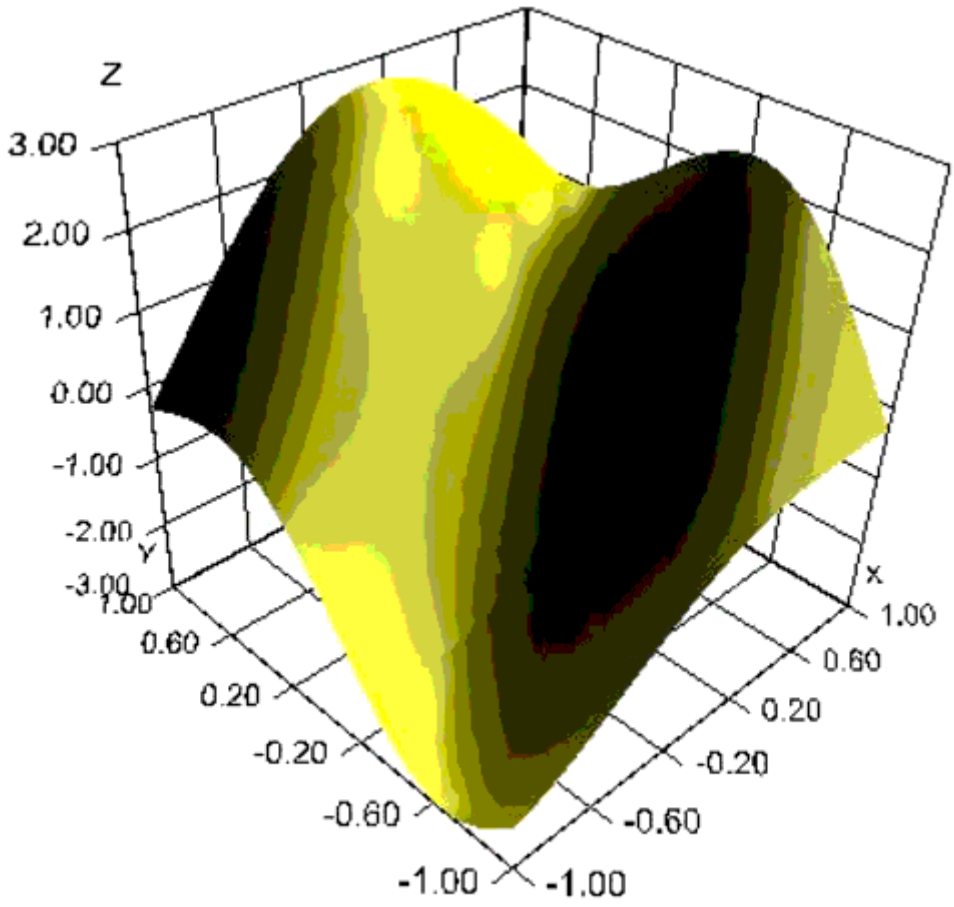
The JMSL 3D chart package is similar to the JMSL 2D chart package. In both, a chart is created by assembling chart nodes into a tree. This chart tree is then rendered to the screen.

The following class is a simple example of the use of the 3D chart package. It plots the function

$$z = 2sin(x + y) - cos(2x - 3y)$$

over the square [-1,1] by [-1,1]. The chart is displayed in a Swing frame.

The class for this example extends the ***JFrameChart3d*** class which creates the root ***Chart3D*** node. A tree of chart nodes is then created starting from the `Chart3D` root node, ***ChartNode3D***. After the tree is created, the render method is used to generate a Java 3D scene graph from the chart node tree. The scene graph is used to generate the image on the screen.

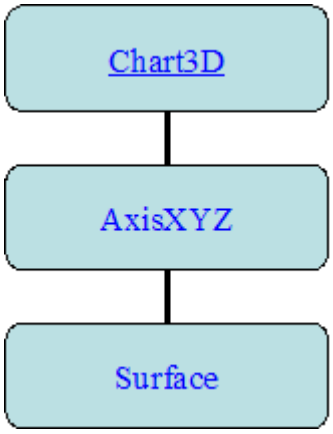


View code file

```
import com.imsl.chart3d.*;

public class SimpleSurface extends JFrameChart3D
        implements Surface.ZFunction {

    public SimpleSurface() {
        Chart3D chart = getChart3D();
        AxisXYZ axis = new AxisXYZ(chart);
        Surface surface =
            new Surface(axis, this, -1.0, 1.0, -1.0, 1.0);
        surface.setFillColor(java.awt.Color.yellow);
        surface.setSurfaceType(Surface.SURFACE_TYPE_NICEST);
        render();
    }

    public double f(double x, double y) {
        return 2*Math.sin(x+y) - Math.cos(2*x-3*y);
    }

    public static void main(String args[]) {
```

```
            new SimpleSurface().setVisible(true);
        }
}
```

The above example created and assembled three nodes into the following tree:



The general pattern of the ***AbstractChartNode*** class and derived classes have constructors whose first argument is the parent `AbstractChartNode` of the `AbstractChartNode` being created. The root node of the tree is always a ***Chart3D*** object.

3D `Chart` nodes can contain attributes, such as `FillColor` and `LineWidth`. Attributes are inherited via the chart tree. For example, when a ***Surface*** node is being painted, its `FillColor` attribute determines the color of the surface. If this attribute is set in the node being drawn, that is the value used. If it is not set, then its parent node (an `AxisXYZ` node in the above example) is queried. If the attribute is not set in the parent node, then its parent is queried. This continues until the root node is reached, after which a default value is used. Note that this inheritance is not the same as Java class inheritance.

Attributes are set using setter methods, such as `setFillColor(Color)` and retrieved using getter methods, such as `getFillColor()`.

Chart Movements

The mouse can be used to rotate, translate, or zoom a 3D chart.

Rotate

The chart is rotated when the mouse is moved with the main mouse button pressed. The rotation is in the direction of the mouse movement, with a rotation of 0.01 radians for each pixel of mouse movement.

Translate

The chart is translated when the mouse is moved with the right mouse button pressed (Shift-main mouse button on systems without a right mouse button). The translation is in the direction of the mouse movement, with a translation of 1% for each pixel of mouse movement.

The chart can also be translated using arrows keys (up, down, left, right). If the control key is also pressed, the motions are accelerated.

Zoom In or Zoom Out

The chart is zoomed when the mouse is moved with the middle mouse button pressed (or Alt-main mouse button on systems without a middle mouse button). The chart is zoomed by 1% for each pixel of mouse movement. Moving the mouse up moves the chart closer, moving the mouse down moves the chart further away.

The chart can also be zoomed using the plus ('+') or minus ('-') keys. The plus key zooms in and the minus key zooms out. If the control key is also pressed, the motions are accelerated.

Reset

Typing the Home key or the Equals ('=') key undoes any chart movements.

Chapter 8 Charting 3D Types

JMSL provides these 3D charting types:

Scatter Plot

This section describes the construction of scatter charts. The markers can be formatted using the Marker Attributes.

It is also possible to mix lines, tubes, markers and surfaces on a single chart (see Marker Attributes).

Fisher Iris Data

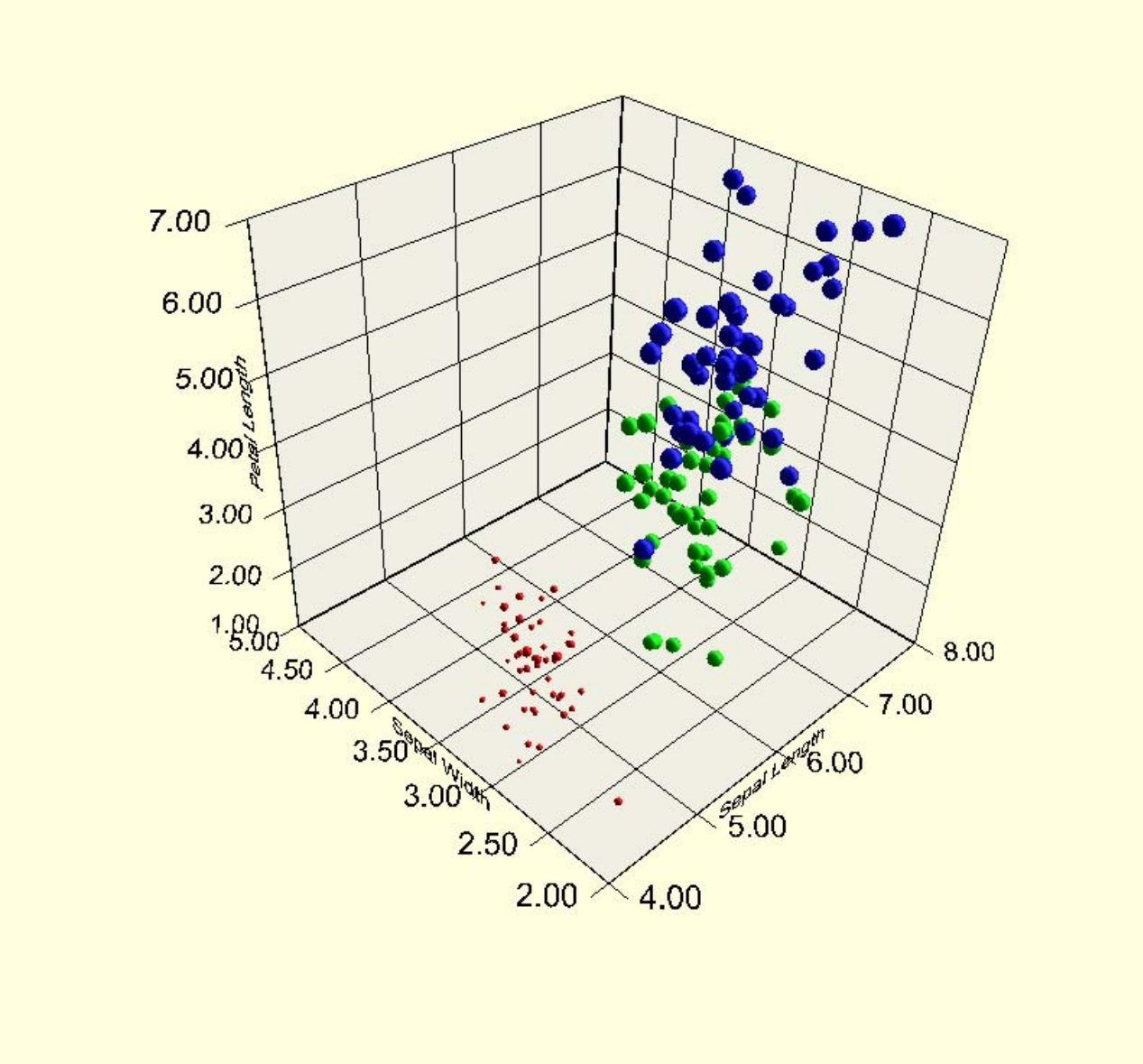
Fisher's Iris data set is read from a flat file and is charted. Each observation in this data set has five values: species, sepal length, sepal width, petal length, and petal width. The sepal length is mapped into the *x*-coordinate, the sepal width is mapped to the *y*-coordinate, and the petal length is mapped into the *z*-coordinate. The petal width is mapped into the marker size. The species is mapped into the marker color.

View code file

```
import com.imsl.chart3d.*;
import com.imsl.io.FlatFile;
import java.awt.Color;
import java.io.*;
import java.sql.SQLException;
```

```
import java.util.StringTokenizer;

public class SampleFisherIris extends JFrameChart3D {
    private int species[];
    private double sepalLength[];
    private double sepalWidth[];
    private double petalLength[];
    private double petalWidth[];

    public SampleFisherIris() throws IOException, SQLException {
        read();

        Chart3D chart = getChart3D();
        chart.getBackground().setFillColor("lightyellow");
        AxisXYZ axis = new AxisXYZ(chart);
        axis.setAxisTitlePosition(axis.AXIS_TITLE_PARALLEL);

        axis.getAxisX().getAxisTitle().setTitle("Sepal Length");
        axis.getAxisY().getAxisTitle().setTitle("Sepal Width");
        axis.getAxisZ().getAxisTitle().setTitle("Petal Length");

        axis.setDataType(Data.DATA_TYPE_MARKER);
        axis.setMarkerType(Data.MARKER_TYPE_SPHERE);
        Color color[] = {Color.red, Color.green, Color.blue};

        for (int k = 0;  k < species.length;  k++) {
            // marker type = Species
            // x = Sepal Length
            // y = Sepal Width
            // z = Petal Length
            // marker size = Petal Width
            double xp[] = {sepalLength[k]};
            double yp[] = {sepalWidth[k]};
            double zp[] = {petalLength[k]};
            Data data = new Data(axis, xp, yp, zp);
            data.setMarkerSize(Math.sqrt(petalWidth[k]));
            data.setMarkerColor(color[species[k]-1]);
        }
        setSize(800, 800);
        render();
    }

    void read() throws IOException, SQLException {
        InputStream is =
            getClass().getResourceAsStream("FisherIris.csv");
        FisherIrisReader fisherIrisReader =
            new FisherIrisReader(is);
        int nObs = 150;
        species = new int[nObs];
        sepalLength = new double[nObs];
        sepalWidth = new double[nObs];
```

```
            petalLength = new double[nObs];
            petalWidth = new double[nObs];
            for (int k = 0;  fisherIrisReader.next();  k++) {
                species[k] = fisherIrisReader.getInt("Species");
                sepalLength[k] =
                    fisherIrisReader.getDouble("Sepal Length");
                sepalWidth[k] =
                    fisherIrisReader.getDouble("Sepal Width");
                petalLength[k] =
                    fisherIrisReader.getDouble("Petal Length");
                petalWidth[k] =
                    fisherIrisReader.getDouble("Petal Width");
            }
        }

        static private class FisherIrisReader extends FlatFile {
            public FisherIrisReader(InputStream is)
                throws IOException {
                super(new BufferedReader(new InputStreamReader(is)));
                String line = readLine();
                StringTokenizer st = new StringTokenizer(line, ",");
                for (int j = 0;  st.hasMoreTokens();  j++) {
                    setColumnName(j+1, st.nextToken().trim());
                    setColumnClass(j, Double.class);
                }
            }
        }

        public static void main(String args[])
                throws IOException, SQLException {
            new SampleFisherIris().setVisible(true);
        }
    }
```


7.00
6.00
5.00
4.00
3.00
2.00
1.00
Petal Length
5.00
4.50
4.00
3.50
3.00
2.50
2.00
Sepal Width
4.00
5.00
6.00
7.00
8.00
Sepal Length

Tube Plot

Lines are drawn as flat, non 3D elements. Tubes are drawn as 3D versions of lines.

Spiral Example

A spiral is defined by the equations parameterized by:

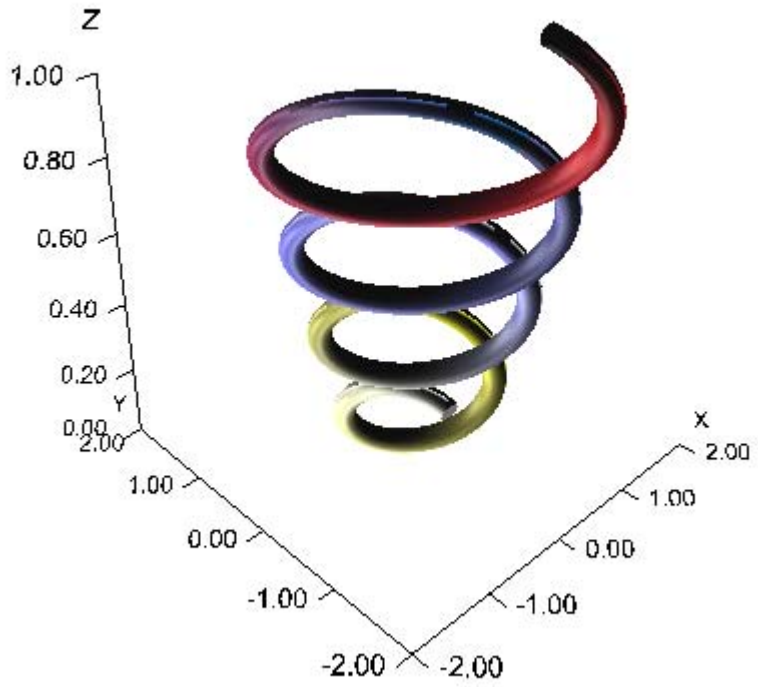
$$t, \quad 0 \le t \le 1$$
$$x(t) = (t + 0.6)\cos(8\pi t)$$
$$y(t) = (t + 0.6)\sin(8\pi t)$$
$$z(t) = t$$

The tube is shaded using the spectral `Colormap` from the 2D charting package.

View code file

```
import com.imsl.chart3d.*;
import java.awt.Color;

public class SampleSpiral extends JFrameChart3D implements ColorFunction {

    public SampleSpiral() {
        Chart3D chart = getChart3D();
        AxisXYZ axis = new AxisXYZ(chart);

        axis.getAxisBox().setPaint(false);

        int nSpiral = 400;
        double xSpiral[] = new double[nSpiral];
        double ySpiral[] = new double[nSpiral];
        double zSpiral[] = new double[nSpiral];
        for (int i = 0;  i < nSpiral;  i++) {
            double t = 8.0 * Math.PI * i / (double)(nSpiral-1);
            double r = 0.6 + (double)i / (double)(nSpiral-1);
            xSpiral[i] = r * Math.cos(t);
            ySpiral[i] = r * Math.sin(t);
            zSpiral[i] = (double)i / (double)(nSpiral-1);
        }
        Data spiral = new Data(axis, xSpiral, ySpiral, zSpiral);
        spiral.setDataType(spiral.DATA_TYPE_TUBE);
        spiral.setLineWidth(2);
        spiral.setColorFunction(this);
        render();
    }

    public Color color(double x, double y, double z) {
        return com.imsl.chart.Colormap.SPECTRAL.color(z);
    }
```

```
    public static void main(String args[]) throws Exception {
        new SampleSpiral().setVisible(true);
    }
}
```



If the function is defined by a set of points, the points in the *xy*-plane are triangulated. The resulting 3D triangles are plotted.

Surface Plot

The ***Surface*** class is used to draw a surface. Surfaces can be specified either by an interface defining a function over a rectangle or by a set of points in three dimensions.

If a function defined over a rectangle is used, the function is evaluated on a rectangular grid of points. The resulting quadrilaterals are plotted.

Shaded Surface

This example shades a surface with a user-specified function, instead of using shading. The same function is used as in the previous example: $z = 2sin(x + y) - cos(2x - 3y)$ over the square [-1,1] by [-1,1].

The shade `Color` function uses a ***Colormap*** object from the `com.imsl.chart` package. This maps a parameter to a ***Color***. In this example the surface is colored using the function $t = x^2 + y^2 + z^2$.

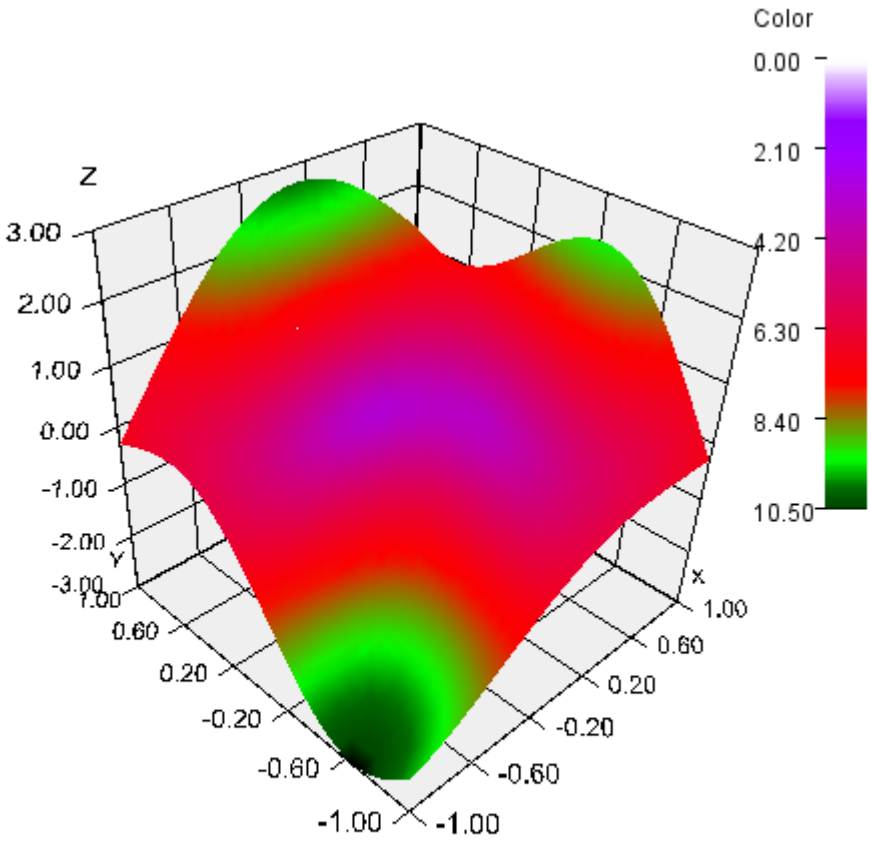
The function value is scaled to a `Colormap` parameter using $s = t^{1/4}/1.8$.

The chart also includes a ***ColormapLegend***, which is drawn on the background. It is positioned 5 pixels from the right edge and 5 pixels from the top edge of the canvas (see ***Canvas3DChart***).

View code file

```
import com.imsl.chart3d.*;
import com.imsl.chart.Colormap;

public class SampleShadedSurface extends JFrameChart3D
        implements Surface.ZFunction, ColorFunction {

    private Colormap colormap = Colormap.GREEN_RED_BLUE_WHITE;

    public SampleShadedSurface() {
        Chart3D chart = getChart3D();
        AxisXYZ axis = new AxisXYZ(chart);
        Surface surface =
            new Surface(axis, this, -1.0, 1.0, -1.0, 1.0);
        surface.setColorFunction(this);
        surface.setSurfaceType(Surface.SURFACE_TYPE_NICEST);

        double maxColor = Math.pow(1.8,4);
        ColormapLegend colormapLegend =
                new ColormapLegend(chart, colormap, 0., maxColor);
        colormapLegend.setTitle("Color");
        colormapLegend.setPosition(-5, 5);
        render();
    }

    public double f(double x, double y) {
```

```
        return 2*Math.sin(x+y) - Math.cos(2*x-3*y);
    }

    public java.awt.Color color(double x, double y, double z) {
        double t = x*x + y*y + z*z ;
        double s = Math.pow(t, 0.25) / 1.8;
        return colormap.color(s);
    }

    public static void main(String args[]) {
        new SampleShadedSurface().setVisible(true);
    }
}
```



Chapter 9 3D Drawing Elements

JMSL provides these Chart3D drawing elements:

Line Attributes

All lines, except for those used to draw markers or outline filled regions are affected by the attributes described in this section. These include axis lines and lines drawn when a ***Data*** node is rendered with its `DataType` attribute having its `DATA_TYPE_LINE` bit set.

Attribute LineColor

`LineColor` is a ***Color***-valued attribute that determines the color of the line. Its default value is `Color.black`.

Attribute ColorFunction

ColorFunction is a ***Color***-valued method which maps spatial points to colors. If `ColorFunction` is not defined then the line is drawn with a solid color set by attribute `LineColor`.

Attribute LineWidth

`LineWidth` is a double-valued attribute that determines the thickness of the lines. Its default value is 1.0.

Tube Attributes

Tubes are three dimensional shaded lines. A ***Data*** node is rendered as a tube if its `DataType` has its `DATA_TYPE_TUBE` bit set.

Attribute LineColor

`LineColor` is a ***Color***-valued attribute that determines the color of the tube. Its default value is `Color.black`.

Attribute ColorFunction

ColorFunction is a ***Color***-valued method which maps spatial points to colors. If `ColorFunction` is not defined then the tube is drawn with a solid color set by attribute `LineColor`.

Marker Attributes

Markers are drawn when a ***Data*** node is rendered with its `DataType` attribute having its `DATA_TYPE_MARKER` bit set. Drawing of markers is affected by the attributes described in this section. Note that when markers are drawn using lines, the line attributes do not apply to markers.

An alternative to markers are images, which can be used to draw arbitrary symbols instead of markers.

MarkerType

`MarkerType` is an integer-valued attribute that determines which marker will be drawn. There are constants defined in ***ChartNode3d*** for the marker types. The default value is `MARKER_TYPE_CUBE`. The following table defines marker types.

Marker Type	Description
`MARKER_TYPE_CUBE`	Cube with its edges outlined in a contrasting color.
`MARKER_TYPE_SIMPLE_CUBE`	Solid colored cube without edge outlining.
`MARKER_TYPE_SPHERE`	Shaded sphere
`MARKER_TYPE_TETRAHEDRON`	Tetrahedron with its edges outlined in a contrasting color.
`MARKER_TYPE_SIMPLE_TETRAHEDRON`	Solid colored tetrahedron without edge outlining.
`MARKER_TYPE_PLUS`	3D plus sign with its edges outlined in a contrasting color.
`MARKER_TYPE_SIMPLE_PLUS`	Solid 3D plus sign without edge outlining
`MARKER_TYPE_CUSTOM`	Marker defined by the user set using the attribute CustomMarkerFactory.

Attribute MarkerColor

`MarkerColor` is a ***Color***-valued attribute that determines the color of the marker. Its default value is `Color.black`.

Attribute MarkerSize

`MarkerSize` is a double-valued attribute that determines the size of the marker. The default value is 1.0.

Attribute MarkerPulsingCycle

Cycle time, in seconds, for pulsing the marker. If this time is less than or equal to zero the marker is not pulsed. The default value is zero.

Attribute MarkerPulsingCycleOffset

Offset time, in seconds, from the time rendering begins to when the marker begins pulsing. This attribute allows different markers to pulse with different phases.

Attribute MarkerPulsingMinimumScale

The minimum marker size during a pulse cycle is the value of the `MarkerPulsingMinimumScale` attribute times the value of the `MarkerSize` attribute.

Attribute MarkerPulsingMaximumScale

The maximum marker size during a pulse cycle is the value of the `MarkerPulsingMaximumScale` attribute times the value of the `MarkerSize` attribute.

Attribute MarkerRotatingCycle

Cycle time, in seconds, for rotating the marker. If this time is less than or equal to zero the marker is not rotated. The default value is zero.

Attribute MarkerRotatingCycleOffset

Offset time, in seconds, from the time rendering begins to when the marker begins rotating. This attribute allows different markers to rotate with different phases.

Attribute MarkerRotatingAxis

A double[3] array containing the axis of rotation. The default is (0,0,1), the z-axis.

Fill Attributes

Attribute FillColor

`FillColor` is a ***Color***-valued attribute that determines the color of a surface. The default value is `Color.black`.

Attribute ColorFunction

Colorfunction is a ***Color***-valued method which maps spatial points to colors. The default is `null`. If both `FillColor` and `ColorFunction` are set, `ColorFunction` is used to color the surface.

Attribute Material

Material defines the appearance of an object under illumination. Its value is a ***Material***. Its default value is the Material object created with default parameters.

Text Attributes

Attribute FontName

`FontName` is a string-valued attribute that specifies the logical font name or a font face name. The default value is "SansSerif". Java always defines the font names “Dialog”, “DialogInput”, “Monospaced”, “Serif”, “SansSerif”, and “Symbol”. Depending on the system, additional font names may be defined.

Attribute FontSize

`FontSize` is an integer-valued attribute that specifies the point size of the font. The default value is 12.

Attribute FontStyle

`FontStyle` is an integer-valued attribute that specifies the font style. This can be a bitwise combination of `Font.PLAIN`, `Font.BOLD` and/or `Font.ITALIC`. The default value is `Font.PLAIN`.

Attribute TextColor

`TextColor` is a ***Color***-valued attribute that specifies the color in which the text is drawn. The default value is `Color.black`.

Attribute TextFormat

`TextFormat` is a ***Format***-valued or a ***String***-valued attribute that specifies how to format string objects.

`TextFormat` can also be set using a `String`, which is used to generate a `Format` object when the attribute is accessed. The `Format` object is created using the value of locale in the chart node.

Some strings have special meanings (case is ignored in these strings):

- `"Date(SHORT)"` means use:
  `DateFormat.getDateInstance`(`SHORT`, ***Locale***)
- `"Date(MEDIUM)"` means use
  `DateFormat.getDateInstance`(`MEDIUM`, ***Locale***)
- `"Date(LONG)"` means use:
  `DateFormat.getDateInstance`(`LONG`, ***Locale***)
- `"Currency"` means use
  `NumberFormat.getCurrencyInstance`(***Locale***)
- `"Percent"` means use:
  `NumberFormat.getPercentInstance`(***Locale***)

If `TextFormat` has a string value that is not one of the above, then

`new DecimalFormat(value, new DecimalFormatSymbols(Locale))`

is used. For example, if its value is a `String` "0.00", then numbers will be formatted with exactly two digits after the decimal place. See ***DecimalFormat*** for a detailed description of these format patterns.

The default value of `TextFormat` is the value returned by the factory method `NumberFormat.getInstance(Locale)`.

Labels

Labels annotate data points. As a degenerate case they can be used to place text on a chart without drawing a point. Labels are controlled by the value of the attribute `LabelType` in a ***Data*** node.

Multiline labels are allowed.

Attribute LabelType

The attribute `LabelType` takes on one of the following values:

- `LABEL_TYPE_NONE` for no label. This is the default.
- `LABEL_TYPE_X` label with the value of the ***x***-coordinate.
- `LABEL_TYPE_Y` label with the value of the y-coordinate.
- `LABEL_TYPE_Z` label with the value of the z-coordinate.
- `LABEL_TYPE_TITLE` label with the value of the Title attribute.

AxisXYZ

The `AxisXYZ` node is the basis of both the scatter chart types and surface chart. Its parent node must be the root ***Chart3D*** node.

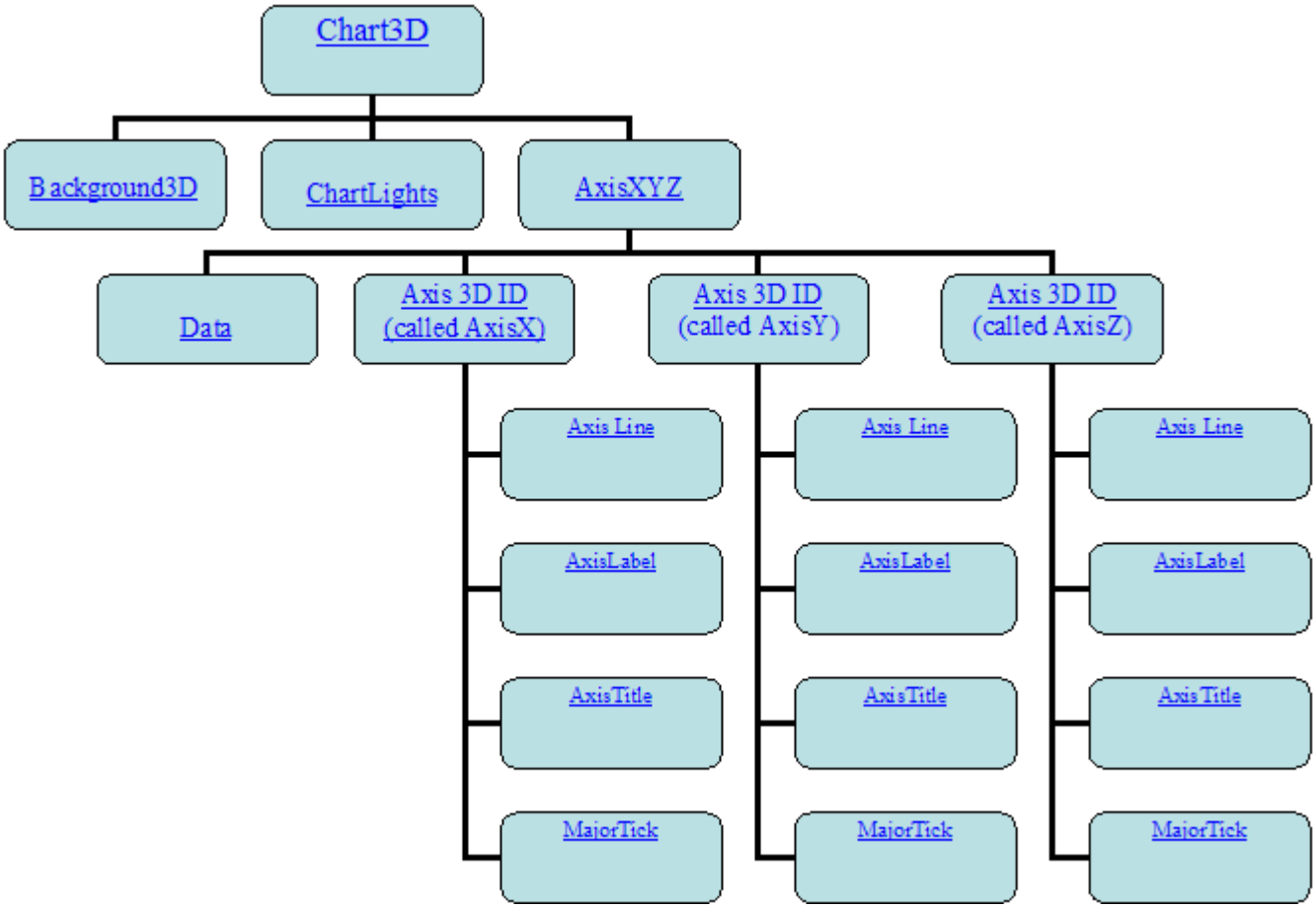
When an `AxisXYZ` node is created, it creates three ***Axis3D*** nodes. They can be obtained by using the methods `AxisXYZ.getAxisX()` and `AxisXYZ.getAxisY()`and `AxisXYZ.getAxisZ()`. Each of the Axis3D nodes in turn creates additional child nodes, as seen in the diagram below.

Accessor methods can be chained together, so the x-axis line can be retrieved using

```
axis.getAxisX().getAxisLine()
```

The code to set the ***x***-axis line to blue is

```
axis.getAxisX().getAxisLine().setLineColor(Color.blue)
```



Axis Layout

The layout of an `AxisXYZ` chart is controlled by the attributes `Window` and `Number`.

Attribute Window

`Window` is a double array-valued attribute that contains the axis limits. Its value is (*min, max*).

Attribute Number

`Number` is an integer-valued attribute that contains the number of major tick marks along an axis.

Transform

The *x*-, *y*- and *z*-axes may be linear or logarithmic, as specified by the `Transform` attribute. This attribute can have the values:

- `TRANSFORM_LINEAR` indicating a linear axis. This is the default.
- `TRANSFORM_LOG` indicating a logarithmic axis.

Autoscale

Autoscaling is used to automatically determine the attribute `Window` (the range of numbers along an axis) and the attribute `Number` (the number of major tick marks). The goal is to adjust the attributes so that the data fits on the axes and the axes have "nice" numbers as labels.

Autoscaling is done in two phases. In the first ("input") phase the actual range is determined. In the second ("output") phase chart attributes are updated.

Attribute AutoscaleInput

The action of the input phase is controlled by the value of attribute `AutoscaleInput` in the axis node. It can have one of three values.

- `AUTOSCALE_OFF` turns off autoscaling.
- `AUTOSCALE_DATA` scans the values in the ***Data*** nodes that are attached to the axis to determine the data range. This is the default value.
- `AUTOSCALE_WINDOW` uses the value of the `Window` attribute to determine the data range.

Attribute AutoscaleOutput

The value of the `AutoscaleOutput` attribute can be the bitwise combination of the following values.

- `AUTOSCALE_OFF` no attributes updated.
- `AUTOSCALE_NUMBER` updates the value of the `Number` attribute. This is the number of major tick marks along the axis.
- `AUTOSCALE_WINDOW` updates the value of the `Window` attribute. This is the range of numbers displayed along the axis.

The default is `AUTOSCALE_NUMBER | AUTOSCALE_WINDOW`; both the attributes `Number` and `Window` are adjusted.

Axis Label

The ***AxisLabel*** node controls the labeling of an axis. The drawing of this node is controlled by the text attributes..

Scientific Notation

If the values along an axis are large, scientific notation is more readable than a decimal format with many zeros. See ***DecimalFormat*** for details on number formatting patterns.

Date Labels

If the `TextFormat` attribute for an axis is an instance of ***DateFormat***, then the axis is scaled and labeled as a date/time axis, instead of as a real axis.

Date information passed to the `Date` constructor must be a `double` number representing the number of milli-seconds since the standard base time known as "the epoch", namely January 1, 1970, 00:00:00 GMT. This is used by the constructor for ***Date***.

String Labels

Any array of strings can be used to label the tick marks using the `setLabels(String[])` method. The `setLabels(String[])` method sets the `Number` attribute to the number of strings.

Axis Title

The ***AxisTitle*** node controls the titling of an axis. The drawing of this node is controlled by the text attributes.

Axes are titled with the value of the `Title` attribute in the `AxisTitle` node. By default, the titles are "X", "Y" or "Z".

Attribute AxisTitlePosition

This `AxisTitlePosition` controls the location of the axis title. Its value is one of the following:

- `AXIS_TITLE_AT_END` positions the axis title at the end of the axis. When the chart is rotated the axis title text always faces the viewer.
- `AXIS_TITLE_PARALLEL` positions the axis title parallel to the axis. When the chart is rotated the axis title stays parallel to the axis.

Major Tick Marks

The node ***MajorTick*** controls the drawing of tick marks on an ***AxisXYZ***.

The location of the major tick marks can be set explicitly, using the `Axis3D.setTicks(double[])` method. However, it is usually easier to allow autoscaling to automatically set the tick locations (see Autoscale).

Attribute TickLength

Tick marks can be made relatively longer or shorter by setting the attribute `TickLength`. Its default value is 1.0. If its value is negative, the tick marks are drawn in the opposite direction.

Attribute Number

`Number` is the number of major tick marks along an axis.

Attribute Ticks

The `Ticks` attribute, in an `Axis3D` node, contains the position of the major tick marks. If this attribute is not explicitly set, its value is computed from the attributes `Number`, `Window` and `Transform`.

Background

Background controls the drawing of the chart's background. It is created by ***Chart3D*** as its child. It can be retrieved from a Chart3D object using the `Chart3D.getBackground()` method.

Lights

`Lights` are used in conjunction with the surface fill color to shade surfaces where no color is defined.

The ***ChartLights*** node is automatically created by the ***Chart3D*** node. It provides a default set of lights. As with any node, it can be disabled by setting its `Paint` attribute to `false`.

More control over lighting is possible by using lighting chart nodes. There are three types of light nodes:

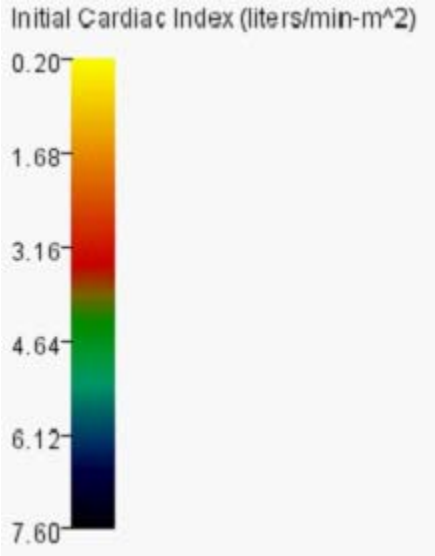
- ***AmbientLight*** is a light that seems to come from all directions.
- ***DirectionalLight*** is an oriented light with an origin at infinity. The `Direction` attribute defines the light's direction.
- ***PointLight*** is a light at a fixed point in space that radiates light equally in all directions away from the light source. The `Position` attribute defines the fixed point. The light's position is in a coordinate system in which the default viewport is the cube [-1,1] by [-1,1] by [-1,1].

Each of these nodes has a `Color` attribute to define the color of the light.

Color Map Legend

The ***ColorMapLegend*** is a color map gradient and scale for drawing on the background of the canvas.

The color map is an object that implements the ***Colormap*** interface in the JMSL Chart 2D package. The interface includes a number of predefined color maps. To see the Colormap in use, see ***example*** in the chart3d.Data.class.



PARTIII

Appendices

Appendix A: Web Server

Application

Using a Servlet

A JMSL chart can be returned as an image from a Web server using a servlet. The mechanism depends on the version of the Java runtime library being used.

Images are rendered using the standard Java class ***javax.imageio.ImageIO***. This class can be used on a headless server—that is, a server running without graphical interfaces. While most desktop Unix systems run the X Windows System, Unix servers are often run without it. Java runs in a headless mode if the system property `java.awt.headless` is set to `true`.

Java Server Pages (JSP) and `servlets` are the standard mechanism for adding Java functionality to Web servers.

Adding charts to a Web page is complicated by the fact that in response to a request a Web server can return *either* a text HTML page *or* a bitmap image. The response to the initial request is an HTML page that contains an image tag. When the browser parses the HTML page and finds the image tag, it makes a second request to the Web server. The Web server then returns the bitmap image in response to this second request.

The JMSL classes ***JspBean*** and ***ChartServlet*** are used to coordinate this process. During the first request, one or more JMSL chart trees are constructed and stored in the ***HttpSession*** object, obtained from the ***HttpServletRequest*** object, using the method `JspBean.registerChart`. The method `JspBean.getImageTag` is used to construct an HTML image tag with an identifier for the chart in the session object. This image tag refers to the servlet `ChartServlet`. The second browser request is therefore directed to the `ChartServlet`, which uses the identifier to retrieve the chart from the session object.

The use of the session object to hold the chart requires that the user's browser allow cookies, used to track sessions across requests.

Generating a Chart with a JSP

This example consists of a JSP working with a bean (`pieChart`) that contains a ***JspBean***. The HTML image tag is the value of the `imageTag` property from `pieChart`. The following is the JSP.

View code file

```
<%@page contentType="text/html"%>
<jsp:useBean id="pieChart" scope="page"
class="com.imsl.demo.jsp.SampleChartBean"/>
<html>
    <head><title>Pie Chart</title></head>
    <body>
        <h1>Pie Chart</h1>
        <jsp:setProperty name="pieChart"  property="*"/>
        <% pieChart.createChart(request); %>
        <jsp:getProperty name="pieChart"  property="imageTag"/>
    </body>
</html>
```

In the above JSP, line 1 flags that an HTML page is being generated. Line 2 creates the object `pieChart`, which is an instance of `SampleChartBean`. This object has page scope, so it is discarded after the page has been generated. Line 8 sets attributes in `pieChart` using parameters in the request. Attributes are set in `pieChart` using its setter methods, following the usual pattern for Java beans. Line 9 calls the `createChart` method to create the chart tree and store it in the session. This is Java code enclosed in the special delimiters. Line 10 inserts the HTML image tag obtained by calling `pieChart.getImageTag()`.

The class `SampleChartBean` uses a ***JspBean***. It creates a pie chart and registers it with the session obtained from the request. The `getImageTag()` method returns an HTML tag which refers to the ***ChartServlet*** servlet that returns the chart image to the browser.

View code file

```
/*
 * To change this license header, choose License Headers in Project
Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package com.imsl.demo.jsp;

import com.imsl.chart.*;
import java.awt.Color;

public class SampleChartBean {

    private JspBean bean;
```

```
    public SampleChartBean() {
        bean = new JspBean();
        bean.setSize(300, 300);
    }

    public void createChart(javax.servlet.http.HttpServletRequest request) {
        Chart chart = new Chart();
        bean.registerChart(chart, request);

        double y[] = {35., 20., 30., 40.};
        Pie pie = new Pie(chart, y);
        pie.setLabelType(Pie.LABEL_TYPE_TITLE);
        pie.setFillOutlineColor(Color.blue);

        PieSlice[] slice = pie.getPieSlice();

        slice[0].setFillColor(Color.red);
        slice[0].setTitle("Red");

        slice[1].setFillColor(Color.blue);
        slice[1].setTitle("Blue");
        slice[1].setFillOutlineColor(Color.yellow);

        slice[2].setFillColor(Color.black);
        slice[2].setTitle("Black");
        slice[2].setExplode(0.3);

        slice[3].setFillColor(Color.yellow);
        slice[3].setTitle("Yellow");
    }

    public String getImageTag() {
        return bean.getImageTag();
    }
}
```

Generating Charts with Client-side Imagemaps

Client-side image maps allow regions of an image to be linked to a URL. Clicking on such a region behaves the same as clicking on a hypertext link. This can be used to create a degree of interactivity. Target regions are created for chart nodes that have the HREF attribute defined.

In this example the sample pie chart as above is generated, but now when the user clicks on a slice the selected slice is exploded out. Now `pieChart` is an instance of `SampleImageBean`. The JSP gets the `pieChart`'s `imageMap` property to create a map tag in the HTML code.

View code file

```
<%@page contentType="text/html"%>
<jsp:useBean id="pieChart" scope="page"
class="com.imsl.demo.jsp.SampleImagemapBean" />
<html>
    <head><title>Pie Chart Imagemap</title></head>
    <body>
        <h1>Pie Chart Imagemap</h1>
        <jsp:setProperty name="pieChart"  property="*"/>
        <% pieChart.createChart(request); %>
        <jsp:getProperty name="pieChart"  property="imageMap"/>
        <jsp:getProperty name="pieChart"  property="imageTag"/>
    </body>
</html>
```

The code for `SampleImagemapBean` is similar to the code for the previous example `SampleChartBean`, but with the `HREF` attribute defined in each slice. The reference is back to `SampleImagemapJSP`, but with the explode parameter defined. The second change is the addition of a setter method for explode. The line

```
   <jsp:setProperty name="pieChart" property="*"/>
```

in the JSP file sets properties in the `pieChart` from parameters in the request. If explode is defined in the HTTP request, then the JSP calls this setter method.

View code file

```
/*
 * To change this license header, choose License Headers in Project
Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package com.imsl.demo.jsp;

import com.imsl.chart.*;
```

```
import java.awt.Color;

public class SampleImagemapBean extends JspBean {
    private JspBean bean;
    private int     explode;

    public SampleImagemapBean() {
        bean = new JspBean();
        bean.setSize(300,300);
        bean.setCreateImageMap(true);
    }

    public void createChart(javax.servlet.http.HttpServletRequest request) {
        Chart chart = new Chart();

        double y[] = {35., 20., 30., 40.};
        Pie pie = new Pie(chart, y);
        pie.setLabelType(pie.LABEL_TYPE_TITLE);
        pie.setFillOutlineColor(Color.blue);

        PieSlice slice[] = pie.getPieSlice();

        slice[0].setFillColor(Color.red);
        slice[0].setTitle("Red");

        slice[1].setFillColor(Color.blue);
        slice[1].setTitle("Blue");
        slice[1].setFillOutlineColor(Color.yellow);

        slice[2].setFillColor(Color.black);
        slice[2].setTitle("Black");

        slice[3].setFillColor(Color.yellow);
        slice[3].setTitle("Yellow");

        for (int k = 0;  k < slice.length;  k++) {
            slice[k].setHREF("SampleImagemapJSP.jsp?explode="+k);
        }

        try {
            slice[explode].setExplode(0.3);
        } catch (Exception e) {
            // ignore out of range values for explode
        }

        bean.registerChart(chart, request);
    }

    public void setExplode(int explode) {
        this.explode = explode;
    }
```

```
    public String getImageTag() {
        return bean.getImageTag();
    }

    public String getImageMap() {
        return bean.getImageMap();
    }
}
```

Servlet Deployment

A Web server that supports Java servlets will generally have a directory of Web applications, one per directory. In each Web application subdirectory there can be a WEB-INF directory containing the Java classes. This directory contains the subdirectory classes, containing individual class files, and the subdirectory lib, containing JAR files. For the above example, the file structure would be as follows:

```
webapps/
  ROOT/
    RogueWaveApps/
      SampleChartJSP.jsp
      SampleImagemapJSP.jsp
    WEB-INF/
      web.xml
      classes/
        com/
          imsl/
            demo/
              jsp/
                SampleChartBean.class
                SampleImagemapBean.class
      lib/
        jmsl.jar
```

The `web.xml` file may be different for different application servers. Its purpose is to configure the servlet class and its mapping. For Tomcat 8.0 and the examples on this page, `web.xml` would contain the following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--
 Licensed to the Apache Software Foundation (ASF) under one or more
  contributor license agreements.  See the NOTICE file distributed with
  this work for additional information regarding copyright ownership.
  The ASF licenses this file to You under the Apache License, Version 2.0
  (the "License"); you may not use this file except in compliance with
  the License.  You may obtain a copy of the License at

      http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
-->
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                      http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
  version="3.1"
```

```
  metadata-complete="true">

   <display-name>JMSL Examples</display-name>
   <description>
      Welcome to the JMSL Tomcat Examples
   </description>

   <servlet>
      <servlet-name>ChartServlet
      </servlet-name>
      <servlet-class>com.imsl.chart.ChartServlet
      </servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>ChartServlet</servlet-name>
      <url-pattern>/servlet/*</url-pattern>
   </servlet-mapping>

</web-app>
```

Appendix B: Writing a Chart as a Bitmap Image File

Using the ImageIO class

A JMSL chart can be saved as an image file using the Java ***ImageIO*** class.

The chart tree is constructed in the usual manner. Here a method called `createChart` is used to create a simple chart, with the null-argument `Chart` constructor.

The bitmap is generated by creating a ***BufferedImage*** object and painting the chart into the buffered image using the graphics object from the buffered image. The buffered image is written to a file using the `ImageIO.write` method.

This method of creating bitmaps does not require that a windowing system be running. The argument `-Djava.awt.headlesss=true` can be used with the java command to run Java in a "headless" mode.

View code file

```
import com.imsl.chart.*;
import java.awt.image.BufferedImage;
import java.io.File;

public class SampleImageIO {
    public static void main(String argv[]) throws java.io.IOException {
        Chart chart = createChart();
        chart.setScreenSize(new java.awt.Dimension(500,500));
```

```
        BufferedImage bi = new BufferedImage(500, 500,
            BufferedImage.TYPE_4BYTE_ABGR_PRE);
        chart.paintChart(bi.createGraphics());

        File file = new File("SampleImageIO.png");
        javax.imageio.ImageIO.write(bi, "PNG", file);
    }


    static Chart createChart() {
        Chart chart = new Chart();
        AxisXY axis = new AxisXY(chart);

        int  npoints = 20;
        double dx = .5 * Math.PI/(npoints-1);
        double x[] = new double[npoints];
        double y[] = new double[npoints];

        //  Generate some data
        for (int i = 0; i < npoints; i++){
            x[i] = i * dx;
            y[i] = Math.sin(x[i]);
        }
        new Data(axis, x, y);
        return chart;
    }
}
```

Using the Scalable Vector Graphics (SVG) API

Scalable Vector Graphics (SVG) is an XML vocabulary for vector graphics.

JMSL's SVG support is based on the Apache Batik toolkit. ***Batik is not included with JMSL Numerical Library***. It can be downloaded, for free, from the `Apache Batik` site. The `batik.jar` file needs to be added to the CLASSPATH.

Once `batik.jar` is installed, a JMSL chart can be saved as an SVG file by using the `Chart.writeSVG` method.

SVG can also be created from a ***JFrameChart*** window. Its File | SaveAs menu allows the chart to be saved as either a PNG image or as an SVG file. The SVG option is active only if `batik.jar` is in the CLASSPATH.

The following example generates SVG output. The output encoding format is set to UTF-8. This is done because SVG renderers do not support all encodings. The UTF-8 is the most general encoding.

`View code file`

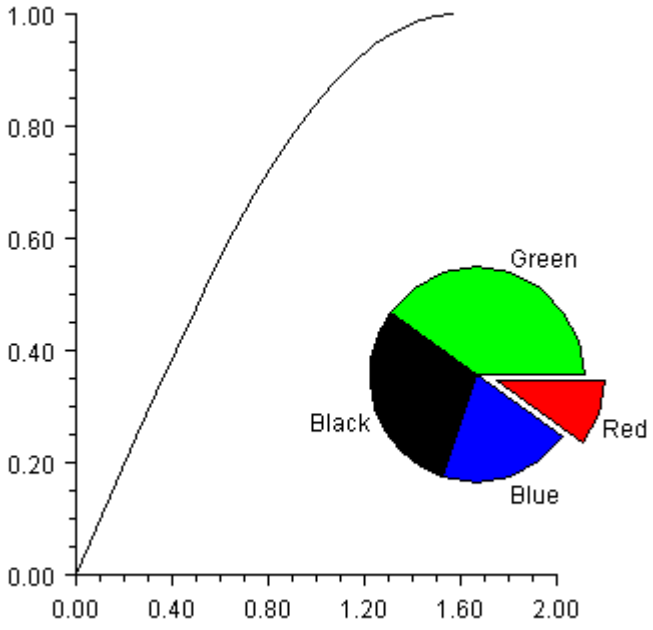
```
import com.imsl.chart.*;
import java.io.*;
import java.awt.*;

public class SampleSVG {
    public static void main(String argv[]) throws java.io.IOException {
        Frame frame = new Frame();
        frame.show();
        frame.setSize(500, 500);
        frame.setVisible(false);
        Chart chart = createChart(frame);

        OutputStream os = new FileOutputStream("SampleSVG.svg");
        Writer writer = new OutputStreamWriter(os, "UTF-8");
        chart.writeSVG(writer, true);

        System.exit(0);
    }


    static Chart createChart(Component component) {
        Chart chart = new Chart(component);
        AxisXY axis = new AxisXY(chart);

        int  npoints = 20;
        double dx = .5 * Math.PI/(npoints-1);
        double x[] = new double[npoints];
        double y[] = new double[npoints];
```

```
        //  Generate some data
        for (int i = 0; i < npoints; i++){
            x[i] = i * dx;
            y[i] = Math.sin(x[i]);
        }
        new Data(axis, x, y);
        return chart;
    }
}
```

Appendix C: Picture-in-Picture

Implementing Picture-in-Picture

The picture-in-picture effect can be obtained by using the `Viewport` attribute. This sets the fraction of the screen into which the chart is to be drawn. The `Viewport` attribute's value is a `double[4]` containing {*xmin, xmax, ymin, ymax*}, on a [0,1] by [0,1] grid with (0,0) at the top-left.



View code file

```
import com.imsl.chart.*;
import java.awt.Color;
```

```
public class SamplePnP extends JFrameChart {

    public SamplePnP() {
        Chart chart = getChart();
        createLineChart(chart);
        createPieChart(chart);
    }

    private void createLineChart(Chart chart) {
        AxisXY axis = new AxisXY(chart);

        int    npoints = 20;
        double dx = 0.5 * Math.PI/(npoints-1);
        double x[] = new double[npoints];
        double y[] = new double[npoints];
        for (int i = 0;  i < npoints;  i++){
            x[i] = i * dx;
            y[i] = Math.sin(x[i]);
        }
        new Data(axis, x, y);
    }

    private void createPieChart(Chart chart) {
        double y[] = {10., 20., 30., 40.};
        Pie pie = new Pie(chart, y);
        pie.setLabelType(pie.LABEL_TYPE_TITLE);
        pie.setViewport(0.5, 0.9, 0.3, 0.8);

        PieSlice[] slice = pie.getPieSlice();

        slice[0].setTitle("Red");
        slice[0].setFillColor(Color.red);
        slice[0].setExplode(.2);

        slice[1].setTitle("Blue");
        slice[1].setFillColor(Color.blue);

        slice[2].setTitle("Black");
        slice[2].setFillColor(Color.black);

        slice[3].setTitle("Green");
        slice[3].setFillColor(Color.green);
    }

    public static void main(String argv[]) {
        new SamplePnP().setVisible(true);
    }
}
```

Appendix D: Drawing to a Canvas

Drawing to a Canvas

It is possible to draw to the canvas using Java 2D methods using a class which implements `Canvas3DChart.paint`. This interface defines a single method `public void paint(java.awt.Graphics g)`.

The object passed to the paint method is actually a ***Graphics2D*** object. `Graphics` is used for consistency with standard Java usage. This method is called each time the 3D chart is redrawn.

An object which implements ***Canvas3DChart.Paint*** can be registered with `Canvas3DChart` using either `addPreRenderPaint` or `addPostRenderPaint`. The former method causes the 2D drawing to appear beneath the 3D drawing, the latter method causes the 2D drawing to appear on top of the 3D drawing.

Since the 3D chart is redrawn many times a minute, redrawing the 2D chart each time can add quite a bit of overhead. If the 2D drawing is not dynamic, the class ***BufferedPaint*** can be used to capture the 2D drawing into an image that can be quickly redrawn for each 3D redrawing. For example, the ***ColormapLegend*** class is implemented using ***BufferedPaint***.

Appendix E: Java 3D

Chart 3D

The JMSL Chart 3D package is built on top of Java 3D. For most purposes, using the Chart 3D package does not require knowledge of Java 3D.

A JMSL chart can be saved as an image file using the Java ***ImageIO*** class.

Using the Chart 3D with Swing

Swing components are lightweight, but the canvas used to render Java 3D is a heavyweight component. Generally, lightweight components cannot be drawn on top of heavyweight components

To allow Swing menu items to appear on top of the canvas, use

```
JPopupMenu.setDefaultLightWeightPopupEnabled(false);
```

To allow Swing Tooltips, use

```
JPopupMenu.setDefaultLightWeightPopupEnabled(false);
ToolTipManager.sharedInstance().setLightWeightPopupEnabled(false);
```

Video Card Drivers

Java 3D is implemented on top of OpenGL. In Windows, there is also an option to use Direct3D. Both of these APIs use the video card hardware to accelerate performance. The interface between OpenGL/Direct3D and the video card hardware is the video card driver.

Problems with the video card drivers can cause problems with Java 3D and therefore with the Chart 3D package. If there is a problem, check with the hardware manufacturer for a driver update. It may also be possible to work around problems by setting certain system properties.

System Properties

Java system properties can be set on the command line using the syntax

```
-Dname=value
```

They can also be set using methods in the `System` class.

There are many system properties which can be set to control Java 3D. The following table lists some of the most important settings:

Property	Values	Definition
j3d.rend	ogl | d3d	Windows-only. Specifies which underlying rendering API should be used thus allowing both Direct3D and OpenGL native DLLs to be installed on a singe machine. Default: ogl
j3d.deviceSampleTime	An integer	The sample time in milliseconds for non-blocking input devices. Default: 5
j3d.threadLimit	An integer	Controls how many threads may run in parallel regardless of how many cpu's the system has. Setting it to "1" will make the system act like a traditional OpenGL render loop. Default: number of CPUs plus one.
j3d.disableXinerama	true | false	Solaris only. Allows major performance boost when using dual screen environments with the X11 Xinerama extension enabled. Detailed information in the release notes. Default: false.
j3d.displaylist	true | false	OpenGL only. Enable use of display lists, an OpenGL performance enhancing feature. False to disable for debugging. Default: true
j3d.g2ddrawpixel	true | false	If true, use glDrawPixel to flush the graphics2D to the screen. If false, use texture mapping to flush the graphics2D to the screen. glDrawPixel is not accelerated on some older Windows video cards. Default: true
j3d.sharedctx	true | false	Shared contexts are used in OpenGL for DisplayLists and Texture Objects to improve performance. However some drivers have bugs causing weird rendering artifacts. This can be used to disable their use to see if this is the problem. Default: true for Solaris and false for Windows
j3d.debug	true | false	Prints out startup and running information. Useful for finding out information about the underlying hardware setup. Default: false
j3d.vertexbuffer	true | false	Use of vertex buffers, a D3D performance enhancing feature equivalent to OpenGL display lists. Some drivers have implementation problems so it might be worth turning this off if there are crashes. Default: true
sun.java2d.d3d	true | false	Windows only. Disable use of Direct3D by Java. Default: true

sun.java2d.ddoffscreen	true | false	Windows only. Disable use of DirectDraw and Direct3D by Java for off screen images, such as the Swing back buffer. Default: true
sun.java2d.noddraw	true | false	Windows only. Completely disable use of DirectDraw and Direct3D by Java. This avoids any problems associated with use of these APIs and their respective drivers. Default: false

For more details, see https://java.net/projects/java3d/pages/Java3DApplicationDev#Java_3D_System_Properties.

INDEX