VISUAL ENGINEERING

KavaChart Developer Documentation

KavaChart User Guide

VERSION 5.0

KavaChart User Guide

164 Main Street • Second Floor • Los Altos, CA 94022
Phone 650.949.5410 • Fax 650.949.5578

Table of Contents

Chapter

1

KavaChart Introduction

This chapter provides a broad overview of some of the ways you might use KavaChart to put charts into your application.

What is KavaChart?

KavaChart is a collection of tools for turning numbers into charts. Given one or more series of numbers, KavaChart can create a variety of common charts to help you absorb and interpret the information. KavaChart tools provide robust, well tested components that let software or web site developers translate numbers to graphics with minimal effort.

KavaChart is implemented in pure Java so that it can be used on virtually any computer operating system, ranging from mainframes to PDAs. The charting tools can be used from within HTML pages, Java applications, and various server technologies.

Java programming expertise is not required to use KavaChart, but we assume that our users will have some familiarity with one or more relevant technologies, such as HTML, applets, server scripting, servlets, database access, or object oriented programming. KavaChart is a complement to any of these skills.

Ways to use KavaChart

KavaChart can be used in a variety of ways. Many HTML pages include KavaChart applets that adjust automatically with dynamic data. Other web sites prefer using KavaChart to generate chart images on the server. KavaChart can be used to add charts to Java application programs. KavaChart can also be embedded within other tools, such as EJBs (Enterprise Java Beans) to add charting functions.

KavaChart on the Client: Applets

The easiest way to turn data into a chart is to use one of the preconfigured KavaChart applets. These applets use applet parameters ("params") to define the chart's overall appearance, as well as to define chart data. Unlike some

other charting products, KavaChart applets contain one chart type per applet. This means that each applet will download fast and start quick. In some cases, loading a KavaChart applet is faster than downloading an image.

Applets can also contain hyperlinks that will permit you to drill down to more detailed data or to open a reference page. You can have as many unique hyperlinks as you have data points.

KavaChart also implements tooltip labels within the applets, so that pointing to a specific bar (or pie slice, or line vertex, or marker, or...) will display the underlying data values and labels.

A chart applet that always shows the same values isn't very interesting, so most KavaChart applet definitions are created with some kind of server scripting language. Since KavaChart's applet data can be defined with simple textual param values, it's a simple matter to create a script that generates these values. When the values change, the chart will change, so building a dynamic chart is as easy as writing a script that generates dynamic page content. And since the values are all simple text, you can use any scripting language; perl, php, servlets, ASP, JSP, your choice.

In addition to params, KavaChart applets can open a URL to retrieve data. They can also poll a URL periodically to see if the data has changed. If the data has changed, the chart will automatically redraw to display the new data.

Of course, you can also use KavaChart's libraries to build your own custom applets.

KavaChart on the Server

KavaChart can run on an application or web server to generate image output. KavaChart includes a collection of chart "beans", encapsulated objects that will translate a set of property values into an image file. These beans are similar to applets, using a set of string values to describe the overall chart appearance and data. Unlike applets, these beans include imaging code and server cache management logic to generate almost any image format.

Within a JSP or ASP, KavaChart's simple bean definition runs on the server, and is replaced by an image tag. Logic inside the bean determines whether the chart already exists in the server's cache or needs to be rendered.

You can also use KavaChart server beans to generate a stream of image data based on your numbers.

These server beans can use an external property sheet to separate your overall page design (colors, fonts, etc.) from your application logic. This helps to make your server scripts simple and portable, making site maintenance easier.

These beans will also generate imagemaps that can be embedded into an HTML page for hyperlinks and tooltips, and they include specific methods to add watermarks, overlay images, and copyright notices.

The KavaChart custom Tag library is probably the best way to use these server beans in a JSP environment. These preconfigured Tag classes let you separate your data acquisition from your page presentation with simple, intuitive XML-like expressions. The tags also provide a way to easily apply ResourceBundles for localization, and to implement persistent data sources for applications that share data.

Beyond KavaChart's preconfigured server tools, the developer's API gives you all the tools you need to build your own server charting tools.

KavaChart in Software Applications

Many times, charting is a small but important part of a much larger application. For example, let's say you're building an enterprise inventory management system. While charts aren't your first concern, they're more than just a nice touch, they're intrinsic to communicating your application's value to the end user.

KavaChart has an extensive and intuitive API that lets you add charts to your application with a minimum of effort. KavaChart's "chart" package includes a range of preconfigured charts and charting components that will create almost any kind of chart. This package also includes utility classes that will let you zoom into your data, scroll your data, transform data values into pixel equivalents, transform pixel data into axis data, and identify internal chart objects based on pixel locations.

KavaChart's design also makes it easy to generate output for a variety of Graphics devices and protocols. This means your application can use the charting data on either the client or the server. It's also easy to integrate the charts into your printing contexts, and to export chart data to external formats, like GIF or PostScript.

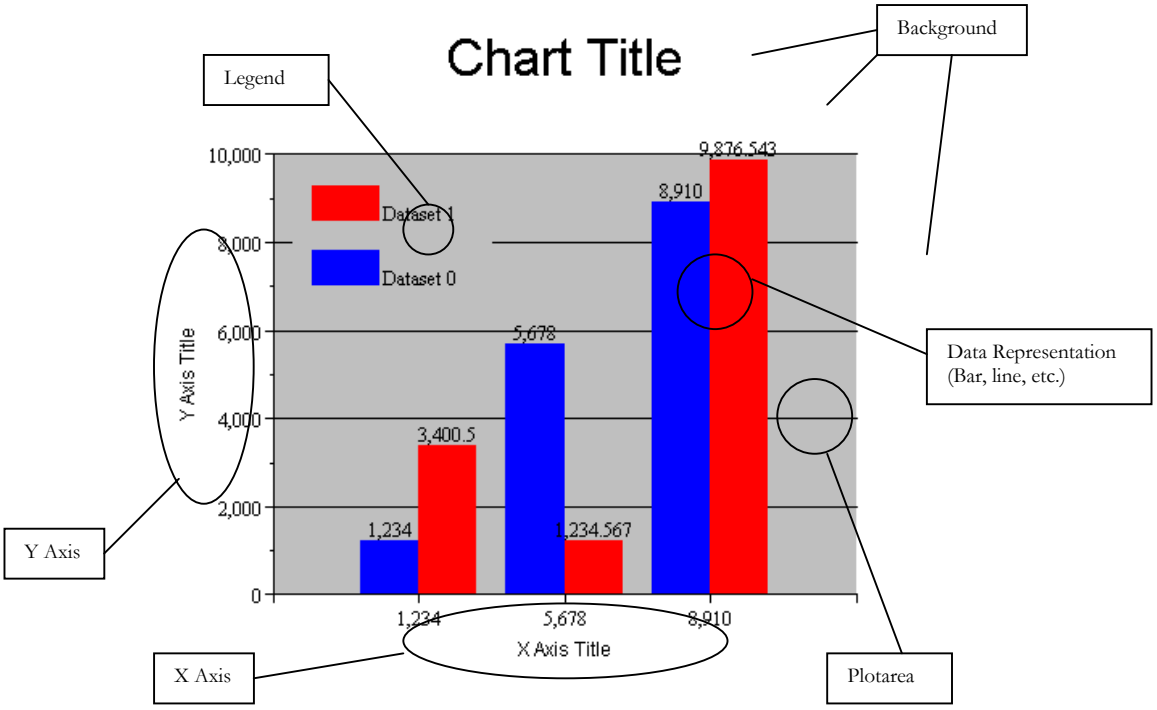
You can also use KavaChart with the latest J2EE application, or you can scale it to a smaller J2ME environment. It's also set up to automatically switch from Java2D to JDK1.1 if you need to support older virtual machines.

Chapter

2

Architecture Overview

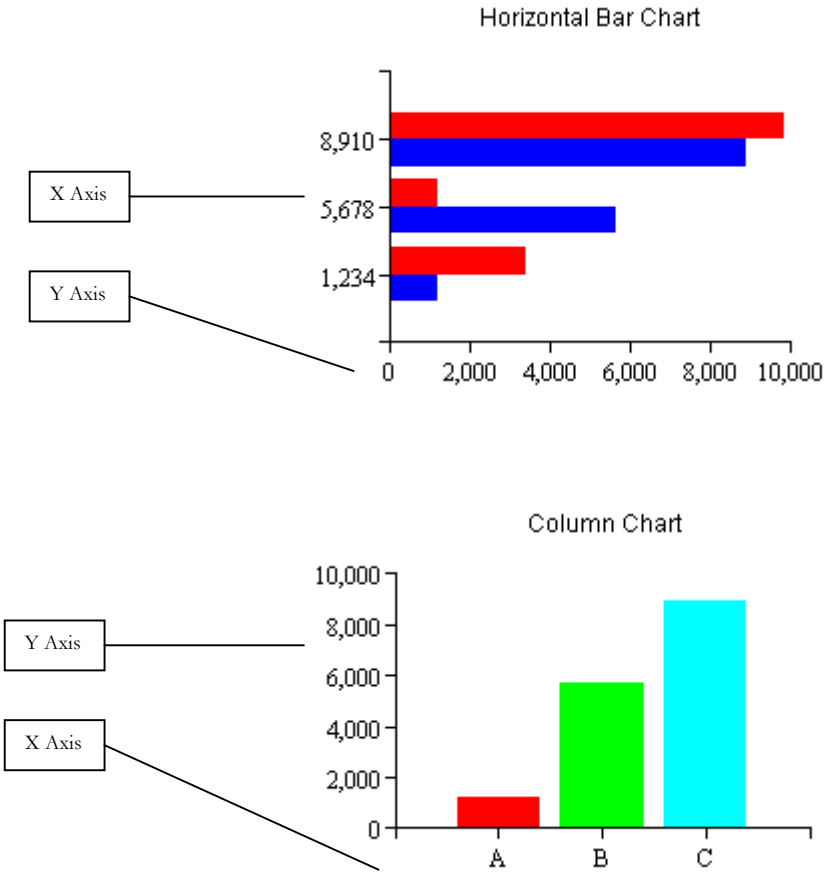
It's helpful to understand KavaChart's terminology. Here's a visual description of some of the most basic terms:



KavaChart charts use a standard set of graphical and non-graphical components to do the work of representing your data. To get the most out of your charts, it's helpful to understand how KavaChart refers to these components and how they fit together.

Chart Parts

X Axis and Y Axis



Axes can occur on the left, right, top or bottom of a Plotarea. A Y axis scales for Dataset Y values. Normally, these are represented vertically, and the Y axis is vertical. Horizontal Bar charts, Speedo charts, and Pie charts are exceptions.

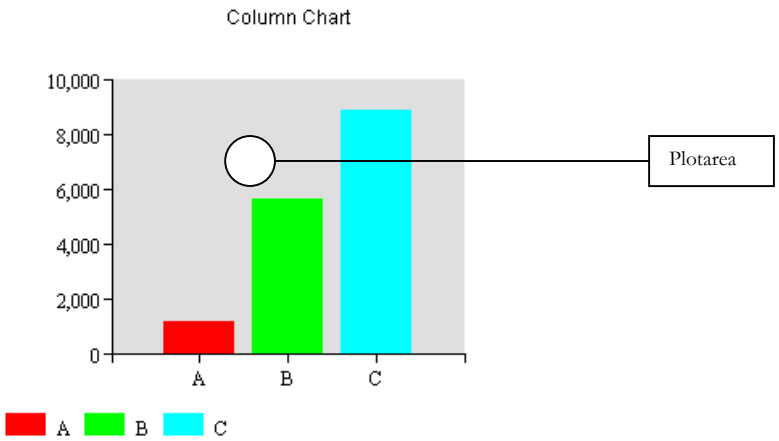
X axes scale for Dataset X values. For some charts, such as a Column chart or a Stacked Column chart, the X axis distributes the data evenly along the X axis, regardless of the Dataset's X values.

Several different types of Axes exist in KavaChart. A basic Axis automatically creates an aesthetically pleasing scale, arranged in even increments. An Axis can also scale logarithmically, which is appropriate for data with extremely wide variation. Some specialized axes, such as the DateAxis, are designed to handle specialized data. DateAxis arranges increments in months, weeks, or some other appropriate time value. A LabelAxis, such as those used for Column charts, will use user-defined labels. If no user-defined labels are present, the axis will try to determine appropriate labels.

Axes contain a number of elements that can be visible or not visible. These include the axis line, tick marks, minor tick marks, an axis title, labels, and grid lines. You can define the color of these elements, and in the case of labels and titles, the font. Labels can also use a number or date format of your choosing. By default, time and numeric labels are automatically localized for various locales.

Axes can be automatically scaled, semi-automatically scaled (you set the start and end, and let the axis determine labelling and increments), or manually scaled. A non auto-scaled axis requires you to set tick, grid, label, and minor tick counts as well as the axis start and end values.

Plotarea

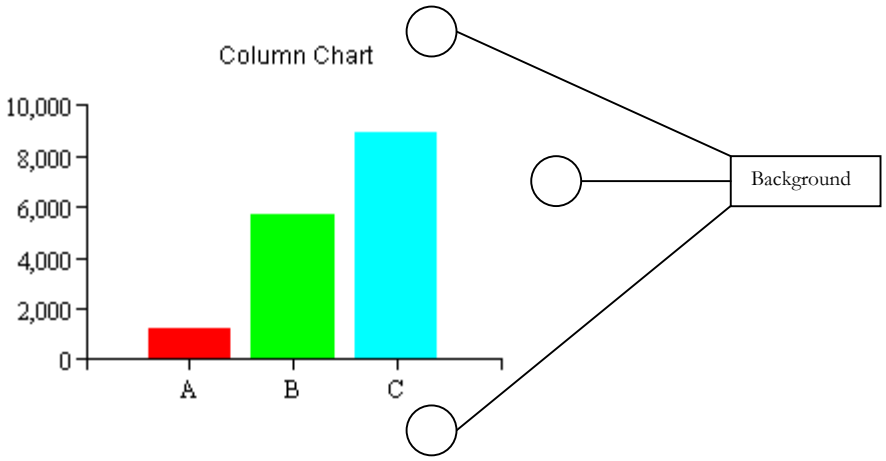


A Plotarea is the region bounded by an X and a Y Axis, which contains a DataRepresentation (such as a Line, Bar, Area, etc.). A Plotarea has a size and location determined by the upper right and lower left corners. The values that define the Plotarea size and location are percentages, relative to the overall chart. For example, an upper right corner value of (0.75, 0.75) means that the top of the Plotarea will be at 75% of the height, and the right side of the Plotarea will be at 75% of the width.

A Plotarea also has a user defined color and outline color.

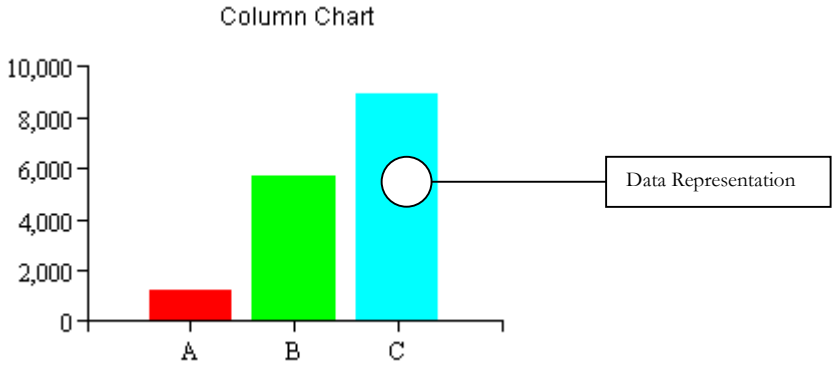
By changing the size and location of your Plotarea you implicitly change the size of your chart's margins. All Axis and DataRepresentation geometries will automatically adjust to accomodate your Plotarea definition.

Background



The rectangle underlying the entire chart is called a Background. The background also contains a title and sub-title. You can set the color of the background or use an image for the background if you prefer. You can also set the color and font of each of the title strings.

DataRepresentation

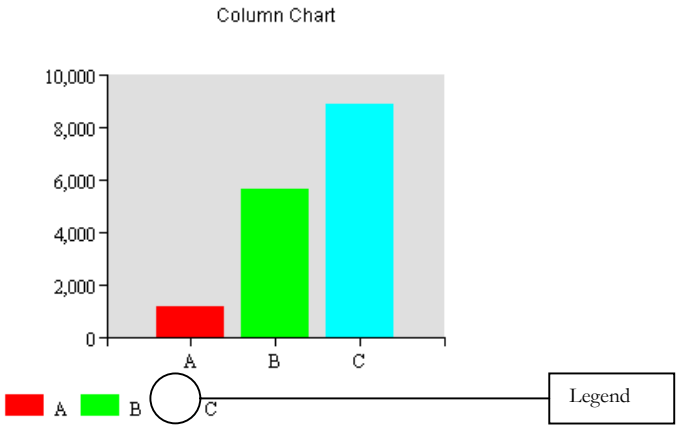


A DataRepresentation is the name KavaChart uses for a variety of objects. These include Line, Area, Bar, and Pie, as well as other more specialized DataRepresentations. These items visually describe a group of Datasets. For example, bar DataRepresentations exist that draw multiple series horizontally or vertically, and side by side or stacked. Bars also exist to represent high and low values, and to draw hi-lo-close, candlestick, histogram and other industry-specific visuals.

DataRepresentations obtain graphical information like colors and label fonts from the Datasets represented. Additionally, the X, Y (and other) magnitudes, as well as the bar/pie/etc. labels are derived from information in the Datasets.

Because DataRepresentations provide specific visual representations, they often have specialized properties. For example, bars can have variable cluster widths (the width of one group of bars), pies can vary the starting angle and toggle visibility on percent labels, speedos can have various types of needles, and so on.

Legend



A Legend contains a description of the Datasets in a particular chart. The icons and label text comes from the chart's Datasets. The X and Y values of a legend's lower left corner describe the legend's location. These values are in percentages of the overall chart. For example, a location of (0.5, 0.5) would place the lower left hand corner of the legend exactly at the center of your chart (50%, 50%).

Legends can have a background color, label font and font color. They can be arranged horizontally or vertically. You can also adjust the size of the legend's icons and the gap between the icon and the legend text (again, in percentages of the overall chart). Legends that are too large for the space you have defined will attempt to create a table of entries (rows and columns).

Various types of legends exist. These include standard Legends, which describe each dataset with a Dataset name and a rectangular icon, Pie legends, which describe each element in the first Dataset with an icon and the element name, and LineLegends, which use a line and optional marker for each Dataset.

Programmer Overview

KavaChart charts use a standard set of graphical and non-graphical components to do the work of representing your data. To get the most out of your charts, it's helpful to understand how KavaChart refers to these components and how they fit together.

Data

KavaChart arranges data into Datum and Dataset classes. A Datum class stores information for a particular observation, including values, labels, and graphical information. In a pie chart, each slice represents one Datum class. These classes store the slice values, colors, and labels. Datum classes are also represented by bars in a bar chart, points on a line or vertices of an area chart.

Dataset classes are used to organize Datum classes into a series or group. Datasets also contain a series name and common graphical attributes, like a line color or fill color. A bar chart with 2 groups of bars will contain 2 Datasets, each with Datum classes that contain the actual numbers that define the size of the

bars. The color of each series will be defined by its Dataset. The name that describes each icon in the chart's Legend is the Dataset name.

> **Tip:**
>
> Many applications need to manipulate data; either to update data from a realtime feed, to reduce the amount of data in the current view, or to let users modify the chart's internal data. Since a Dataset is simply a container for a java.util.Vector of Datum classes, it's easy to do most of these tasks by just working with this data Vector.

There are many ways to define Datum and Dataset classes, including JSP properties, applet params, Chart utility methods and bean datafeed classes. Ultimately, however, the data will end up in Datum classes, organized into Datasets.

Plotting

Data is plotted into bars, lines, areas, etc., by creating a set of axis scales and transforming the data geometrically into a specified region of your chart. This region is called a Plotarea in KavaChart. Usually it has an X axis on one side and a Y axis on the other. Some kind of DataRepresentation class, like com.ve.kavachart.chart.Bar or com.ve.kavachart.chart.Line determines what the actual output will look like.

Several objects work cooperatively to build the output for each of these DataRepresentations. First, the Y Axis looks at the chart's Dataset classes to determine how the Y axis should be scaled. It finds the minimum and maximum values in the Datasets, and creates a scale that will look nice for the given data range. Next, the X axis does the same for Dataset X values. Then the DataRepresentation class uses the values calculated by the axes and combines them with location and size information from the Plotarea to create appropriate drawing geometries. Finally, the DataRepresentation uses numbers, colors, and labels from its Datasets to draw a graphic within the Plotarea.

A simple chart uses a single DataRepresentation, a single Plotarea, and a pair of axes. A more complex chart can have several DataRepresentations, several Plotareas, and many axes. The basic structure is always the same, though: every DataRepresentation uses a Plotarea and Dataset and Axis classes to create its output.

How does an Axis determine what the scale should be? By examining its Datasets, and finding the extrema of X or Y values. Where does it draw? At the top, bottom left or right edge of its Plotarea, usually. Where does a DataRepresentation draw? Within the bounds of its Plotarea. How does it determine locations, colors, and so on? By examining its Datasets.

To change the location of your axes, then, just change the bounds of your Plotarea. Everything (bars, lines, etc.) follows along automatically. Also, if you're an advanced KavaChart user, you can build all sorts of custom charts by creating multiple groups of Axes, Plotareas, DataRepresentations and Dataset arrays. You can also move axes around as you draw to create layers of axes, each with its corresponding DataRepresentation.

Graphical Attributes

Almost all KavaChart graphical attributes are stored in Gc classes. These classes contain line colors and styles, fill colors and styles, images, and as appropriate (via helper classes) gradients and textures. Each charting component (Background, Axis, Dataset, etc.) uses one or more Gc classes to store this information.

For example, a Dataset contains a Gc that stores the overall colors for a series; a Line DataRepresentation would use the Dataset Gc's LineColor property to determine what color a particular line should be. The Legend would use the same Gc to draw the Dataset's icon within the Legend. A Pie would use the individual Datum Gc's FillColor property to draw each slice of the pie. A PieLegend uses these Gcs for its icons.

An Axis uses Gc classes to store color and line attributes for grid lines, tick marks, and the axis line itself.

A Background or Plotarea stores fill color and outline color information in their respective Gc classes.

Chapter

3

Programming with KavaChart Applets

This chapter is intended for users with at least some Java programming experience and a suitable Java programming environment. For basic information about KavaChart applets, see the KavaChart AlaCarte User's Guide.

While KavaChart applets are suitable for most applications without modification, they're also designed to be extended, so you can add all sorts of sophisticated functionality. You can even build your own custom charts, and use the KavaChart applet framework to deploy your charts in a web environment.

Overall Design

All KavaChart applets are subclasses of com.ve.kavachart.applet.ChartAppShell. Your Developers Kit includes the source code to this class.

ChartAppShell does the work of managing a single chart inside an applet. It handles thread management, URL and image loading, graphics resource management, and printing. It also takes care of dwell labels and launches hyperlinks.

ChartAppShell contains a variable named "chart" of type com.ve.kavachart.chart.ChartInterface. All basic KavaChart graphs extend com.ve.kavachart.chart.Chart, which implements this interface. The "chart" variable has protected scope, so it's available to all subclasses.

ChartAppShell doesn't parse applet parameters directly. This is done with a helper class, com.ve.kavachart.utility.ParameterParser, which is also shared by the com.ve.kavachart.servlet package. All general parameters and dataset oriented parameters are handled by ParameterParser. Parameters that are specific to a particular chart type, like "barClusterWidth", or "pie startAngle", are parsed by the individual applet subclass, as we'll see below.

A ChartAppShell subclass need only implement the "init" method, like this:

```
public class MyPieChart extends
com.ve.kavachart.applet.ChartAppShell {
      public void init(){
            initLocale();
            chart = new com.ve.kavachart.chart.PieChart();
                  getOptions();
      }
}
```

This chart applet will automatically pick up dwell labels, painting, resizing, and other behavior that other chart applets implement. It also supports all the parameters of any other KavaChart applet, such as titleString, backgroundImage, and so on.

When the applet calls "initLocale" a default font is selected and locale settings are created. The chart is then instantiated, and "getOptions" looks through the applet's parameters for KavaChart parameters.

If you want to make other initializations, you're free to do so. KavaChart applets also include a variety of useful method stubs to make easy work of most common modifications.

Applets and Server Beans

An important design feature of KavaChart applets lets you translate your applet to a server bean with virtually no coding. KavaChart's server beans share the overall applet design, but substitute a top level class, which manages things like image encoding and byte stream handling instead of applet threads and dwell labels.

This shared architecture means that you can probably translate your custom applet into a server-side object by simply extending com.ve.kavachart.servlet.Bean instead of com.ve.kavachart.applet.ChartAppShell. In addition to making your web architectures flexible enough to accommodate both client and server charting, this can be a powerful testing mechanism during the development phases of your project.

See the section at the end of this chapter for more information about how to do this.

Data Handling

All applets include a couple of data handling methods to let you override the process of adding data to your custom chart applet. These methods let you integrate your own JDBC connections, URL handling, computational data generation, or other techniques that may not be appropriate for the bulk of chart applet users.

DataProvider

A DataProvider implements a simple interface to supply Dataset classes to an applet. The DataProvider overrides params like "dataset0yValues", but applies params like "dataset0Color" to the incoming data. Here's a sample:

THE JAVA CODE:

```
public void CustomColumnApp extends columnApp {
     public void init(){
           setDataProvider(new RandomDataProvider());
           super.init();
     }

}

class RandomDataProvider implements DataProvider {
     public Enumeration getDatasets(){
           Dataset d = new Dataset();
           d.setName("random numbers");
           for(int i=0;i<20;i++){
                 d.addPoint((double)i, Math.random(), null);
           }
           Vector v = new Vector();
           v.addElement(d);
           return v.elements();
     }
     public String getUniqueIdentifier(){
           return "not unique";
     }
}
```

THE HTML:

```
<applet code=CustomColumnApp width=500 height=300>
<param name=dataset0Color value="blue">
<param name=dataset0Name value="interesting name">
</applet>
```

The applet will call the "getDatasets()" method in your custom class to obtain its data. The method "getUniqueIdentifier()" is ignored, since this is used for server-side caching logic.

getMyDatasets

Next, let's examine the "getMyDatasets" method. This method generally uses the value of the param "customDatasetHandler" to read its data. Here's a simple example.

THE JAVA CODE:

```
public void CustomColumnApp extends
                        com.ve.kavachart.applet.columnApp {
      public void getMyDatasets(String parameters){
            double[] yValues = {123,432,345};
            chart.addDataset("my data", yValues);
      }
}
```

THE HTML:

```
<applet code=CustomColumnApp width=300 height=200>
<param name=customDatasetHandler value="nothing">
</applet>
```

In the Java code, we're using the protected field "chart" from columnApp's parent, ChartAppShell. This is a "com.ve.kavachart.chart.ChartInterface" type that contains an instance of "com.ve.kavachart.chart.ColumnChart". We're just using that chart's addDataset method to populate the chart data. We can also use other com.ve.kavachart.chart.Dataset methods here to manipulate out data directly.

In the HTML, we're defining the param "customDatasetHandler". This param's value is passed to the method "getMy Datasets", but it's primary use is as a signal to the applet that we want to use the method "getMyDatasets".

Although this example is trivial, you can do anything you want in getMyDatasets, including JDBC connections, data calculations, or other Java code that makes sense for your application.

You can also call the applet's parameter parser to process dataset related params, like dataset0Color, dataset0FillStyle, and so on. To do this, send your data to the parser's "getDatasetParameters" method instead of adding the dataset directly. Here's our java code from above, modified to call getDatasetParameters:

```
public void CustomColumnApp extends
com.ve.kavachart.applet.columnApp {
   public void getMyDatasets(String parameters){
         double[] yValues = {123,432,345};
         parser.getDatasetParameters(0, null, yValues, null, null);
   }
}
```

If you examine the source code for KavaChart's time oriented charts (e.g. dateLineApp, dateAreaApp, etc.) you'll see that we implement this method with a custom version of ParameterParser, called DateStreamReader. DateStreamReader contains date-aware features for handling data as well as parsing axis features specific to time-oriented axes.

Generally, you'll find the next method, "getDataset" better for your own applet subclass. Why? The "getDataset" method makes for cleaner code implementations, and can be used for odd data types, like candlestick chart data.

getDataset

The "getDataset" method is called repeatedly until it returns a null. This method is called when the parameter "internalData" is set to "true". It will automatically process any dataset oriented params (dataset0Color, dataset0Name, etc.) to override values you set in your method. Here's a sample:

THE JAVA CODE:

```
public void CustomColumnApp extends columnApp {
     public Dataset getDataset(ChartInterface chart, int whichSet) {
           if(whichSet>0)
                 return null;
           double[] yValues = {123,432,345};
           return new Dataset("hello, world", yValues, whichSet, null);
     }
}
```

THE HTML:

```
<applet code=CustomColumnApp width=500 height=300>
<param name=internalData value=true>
</applet>
```

Of course, your applet subclass will do something much more interesting, but the basic concept remains the same.

If you're using a chart that uses discontinuities, such a disLineApp or disDateLineApp, you can set discontinuities in your data by using the com.ve.kavachart.chart.Datum.DISCONTINUITY flag. Here's a code example, using "getDataset":

THE JAVA:

```
public void CustomDiscontinuousLineApp extends disLineApp {
     public Dataset getDataset(ChartInterface chart, int whichSet) {
           if(whichSet>0)
                 return null;
           com.ve.kavachart.chart.Dataset d = new Dataset();
           for(int i=0;i<50;i++){
                 Datum dat = new Datum((double)i, Math.random(), null);
                 if(i%10==0)              dat.setLabel(Datum.DISCONTINUITY);
                 d.addDatum(dat);
           }
           return d;
     }
}
```

This example will create a line break at every tenth point.

Getting Custom Options

Your chart may require some custom parameters. KavaChart applets include a method stub for handling these parameters, after obtaining generalized options. Although you could include these options after the rest of your applet initialization, it's a good idea to use "getMyOptions" to make your applet code easy to translate to a KavaChart servlet bean. Here's a simple example:

```
public void getMyOptions() {
   LineChart l = (LineChart) chart;
   String str = getParameter("BlackLine");
   if(str!=null){
         if(str.equalsIgnoreCase("true")){
```

```
                l.getDatasets()[0].getGc().setLineColor(Color.black);
            }
        }
    }
```

Because the data acquisition methods have already been called, we can be sure we have data for dataset 0, and we just set the line color to black if the parameter instructs us to.

Custom Drawing

If you want to draw something on top of your chart, like a watermark, a copyright notice, or an axis threshold line, use the method stub "drawMyStuff". This method uses the same java.awt.Graphics class the applet uses to paint the image buffer, so anything you paint to this class will appear on your applet. In the example below, we'll just draw some text:

```
public void drawMyStuff(Graphics g) {
    g.setColor(Color.black);
    g.drawString("hey you", 0, getSize().height – 30);
}
```

Remember that the "chart" variable is also available to you at this point, so you can use classes in the com.ve.kavachart.chart package. In particular, you might want to use com.ve.kavachart.chart.DataTransform to draw threshold lines over your data, or com.ve.kavachart.chart.Highlighter to highlight portions of your chart.

Selected ParameterParser Methods

Every KavaChart applet (and server bean) uses an instance of ParameterParser. Since this class has some very useful utility methods that you can use from both applets and server beans, we'll quickly review some of them here. To obtain an applet's ParameterParser, just use the protected variable "parser", instantiated when the applet calls "getOptions" in its init method.

URL handling

ParameterParser has methods for opening and closing URLs, as well as for retrieving individual lines from a URL in a thread-safe manner. See your API documentation for more information.

Axis Option Handling

Custom applets frequently have more than one axis. Use ParameterParser's "parseAxOptions" method to get all the general options for any AxisInterface. This method parses and applies options like auxAxisTitleString, auxAxisColor, auxAxisStart, etc. In this example, we called our axis "auxAxis", but you can refer to it any way you'd like.

Color parsing

Because charts frequently need color definitions, ParameterParser includes a utility method for translating strings into colors. Use the "getColor" method to do this in a consistent way in your applet.

Font parsing

Similarly, charts need frequent font definitions from parameter strings. The utility method "getFont" translates a string into a java.awt.Font class, in a way consistent with other KavaChart font definitions.

Label arrays

To translate an list of items delimited by commas (or your specified delimiter) into an array of strings, use the "getLabels" method.

Parameter Handling

Although all applets support a getParameter method, you can make this portable to KavaChart's server beans by using "parser.getParameter(String)", which looks for applet parameters, servlet parameters or server bean properties, as appropriate.

Numeric arrays

Charting applets frequently need to translate parameter lists of number strings into arrays of numeric data. Use ParameterParser's "getVals" method to get an array of doubles from a delimited list of number strings. Strings that can't be translated become com.ve.kavachart.chart.Datum.DEFAULT. Translation is done according to localization defaults.

Image Loading

Use ParameterParser's "makeURLImage" method to load an image file in a context neutral way. In an applet, this is basically the same as calling "getImage", but uses a MediaTracker to ensure that the image is available before the chart is painted. In a servlet, the image can also be loaded from the local filesystem.

Styles and Colors

Almost all KavaChart style information is stored in com.ve.kavachart.chart.Gc classes. These classes exist in most KavaChart elements, such as Backgrounds, Plotareas, Datasets, and so on. ParameterParser has a utility method for setting a Gc's properties through parameters. Use "parser.populateGc" with a string that describes your Gc and the Gc, like this:

```
…
Gc gc = chart.getBackground().getGc();
parser.populateGc("secondBackground", gc);
```

Most of the graphics contexts for charts are already parsed, but you might find it convenient to use a Gc class for your own drawing, done with "drawMyStuff".

Locale Strings

Use ParameterParser's "getLocale" method to translate almost any locale string into a Java locale. Although KavaChart applets automatically localize to the client running the applet, you may want to obtain a locale string for data handling, labeling, or some other purpose.

Creating Custom Archives

Jar archives have many benefits and virtually no drawbacks. They're faster to start up, have a smaller footprint, and contain all your applet's resources in a single file. So how do you go about creating a jar file for a KavaChart based applet?

If your applet is based on an existing KavaChart applet, start by extracting the contents of your applet's superclass, using the jar command. For example, if your applet is based on com.ve.kavachart.applet.lineApp:

```
jar xvf lineApp.jar
```

This creates a hierarchy of class files that includes everything you need except the files you've added. Place your files into the directory hierarchy, remembering that if you created packages or are using KavaChart's packages, you'll need to place your class files in a directory that has the same name as your package.

Then create a new jar file based on these files:

```
jar cvf NewLineApp.jar .
```

Now you can use NewLineApp.jar in your applet definition.

Tip:

If you're not sure which class files you require, load a page that contains your applet definition, and then examine your server log to see which classes the ClassLoader requested.

Some integrated development environments (IDEs) will also export the minimum class files your applet requires, creating a jar file with the appropriate classes and resources automatically. If you're using an IDE like this, remember to include any classes you're loading by name, as these probably won't appear in the compiler's calculations.

The places this might be a problem with KavaChart include:

com.ve.kavachart.chart.SimpleDateAxis – a class that does simplified date arithmetic to work around problems in Internet Explorer version 4.

com.ve.kavachart.chart.GcHelper – a class that gets loaded automatically for Java2D functionality if those facilities exist in the runtime environment.

Changing an Applet into a Server Object

Applets that are constructed using KavaChart's applet framework can be quickly translated into an object that generates chart images on a server. KavaChart applets extend the abstract com.ve.kavachart.applet.ChartAppShell, and KavaChart server objects extend the abstract com.ve.kavachart.servlet.Bean. Explicit subclasses of either abstract class do the following:

Create an instance of a com.ve.kavachart.chart.ChartInterface (like "BarChart") and assign it to the protected variable "chart" in the "init()" method. Init() then calls the method "getOptions()" to proceed with initialization.

GetOptions() installs a "com.ve.kavachart.utility.ParameterParser", which does the work of translating parameters (or properties, in the case of server objects) into chart attributes.

Both applets and server objects use the method "getMyOptions()" to obtain parameters (or properties) related to a specific chart type.

Both applets and server objects use the same utility methods for obtaining data (getDataset, getMyDatasets) and drawing overlays (drawMyStuff).

Generally, you can translate a KavaChart applet into the equivalent server chart object by changing one line in the class definition. Simply change the class to extend com.ve.kavachart.servlet.Bean (or one of its subclasses) instead of extending com.ve.kavachart.applet.ChartAppShell, and you have a working server object, with all the facilities of any other server chart object, including image management, antialiasing, hyperlink generation, cache management, and so on.

Change this:

```
public class MyApplet extend com.ve.kavachart.applet.lineApp{ …
```

To this:

```
public class MyServerChart extends
      com.ve.kavachart.servlet.lineApp{ …
```

See the chapters on KavaChart's server objects for more information about how to use this object.

Chapter

4

Programming with KavaChart Server Objects

This chapter discusses how to create subclasses of KavaChart server objects that obtain data directly, draw overlay graphics, or install other behaviors that can't be controlled with properties. We'll also discuss the overall architecture, and describe how to use the KavaChart server object framework to build customized charts. See the ProServe User's Guide for foundation information on KavaChart server objects.

Overall Design

Off-the-shelf KavaChart server objects are subclasses of com.ve.kavachart.servlet.Bean.

"Bean" does the work of generating a single chart image and applying properties to an internal chart with a "ParameterParser" class. It uses a "CacheManager" object to manage image generation and cache management. It also takes care of generating link maps for tooltips or hyperlinks.

"Bean" contains a variable named "chart" of type com.ve.kavachart.chart.ChartInterface. All core KavaChart graphs extend com.ve.kavachart.chart.Chart, which implements this interface. The "chart" variable has protected scope, so it's available to all subclasses.

"Bean" doesn't parse properties directly. This is done with a helper class, com.ve.kavachart.utility.ParameterParser, which is also shared by the com.ve.kavachart.applet package. All general parameters and dataset-oriented parameters are handled by ParameterParser. Parameters that are specific to a particular chart type, like "barClusterWidth", or pie's "startDegrees", are parsed by the individual bean subclass, as we'll see below.

A Bean subclass need only implement the "init" method, like this:

```
public class MyPieChart extends
com.ve.kavachart.servlet.Bean {
      public void init(){
            initLocale();
            chart = new com.ve.kavachart.chart.PieChart();
                  getOptions();
      }
}
```

The "init()" method is called at the start of the chart image generation process.

This chart object will automatically inherit hyperlinking, image encoding, cache management, and other behavior that other chart types implement. It also supports all the properties of any other KavaChart server object or applet, such as titleString, backgroundImage, and so on.

When the server object calls "initLocale" a default font is selected and locale settings are created. The chart is then instantiated, and "getOptions" looks through the chart's property set for KavaChart properties.

If you want to make other initializations, you're free to do so. KavaChart server objects also include a variety of useful method stubs to make easy work of most common modifications.

Applets and Server Beans

An important design feature of KavaChart server beans lets you translate your server object to an applet with virtually no coding. KavaChart's applets share the overall server object design, but the top-level class is an "Applet" subclass, which manages threads and dwell labels.

This shared architecture means that you can probably start with a custom applet, for more interactive testing, and then translate your applet into a server-side object by simply extending com.ve.kavachart.servlet.Bean instead of com.ve.kavachart.applet.ChartAppShell. In addition to making your web architectures flexible enough to accommodate both client and server charting, this can be a powerful testing mechanism during the development phases of your project.

See the section at the end of this chapter for more information about how to do this.

Data Management Methods

Property strings are the easiest way to add data to a server chart object. However, if your chart has a lot of data, the overhead of translating information from its native format to a String, and then into a KavaChart Dataset can be inefficient. It's also a lot more efficient in a JSP to use a ChartTag with a DataProvider to simplify the page creation and maintenance process.

Like KavaChart applets, the server objects provide three method stubs to let you add data to a chart without using property strings.

DataProvider

A DataProvider implements a simple interface to supply Dataset classes to a server object or ChartTag. The DataProvider overrides properties like "dataset0yValues", but applies properties like "dataset0Color" to the incoming data. Here's a sample:

```
//a sample DataProvider implementation
class RandomDataProvider implements DataProvider {
     public Enumeration getDatasets(){
           Dataset d = new Dataset();
           d.setName("random numbers");
           for(int i=0;i<20;i++){
                 d.addPoint((double)i, Math.random(), null);
           }
           Vector v = new Vector();
           v.addElement(d);
           return v.elements();
     }
     public String getUniqueIdentifier(){
           return "random number:" + Math.random();
     }
}

com.ve.kavachart.servlet.Bean chart = new
com.ve.kavachart.servlet.columnApp();
chart.setDataProvider(new RandomDataProvider());
```

The server bean will call the "getDatasets()" method in your custom class to obtain its data. The method "getUniqueIdentifier()" is used to calculate cached chart image names. This is important to avoid confusion between chart images that have identical styles but divergent data.

getDataset

The "getDataset" method is called when you set the property "internalData" to "true". It will be called repeatedly until it returns a null. It will automatically process any dataset oriented params (dataset0Color, dataset0Name, etc.) to override values you set in your method. Here's a sample:

```
import com.ve.kavachart.chart.*;
public void CustomColumnApp extends
            com.ve.kavachart.servlet.columnApp {
  public Dataset getDataset(ChartInterface chart,
                                    int whichSet) {
      if(whichSet>0)
            return null;
      double[] yValues = {123,432,345};
      return new Dataset("hello, world",
                        yValues, whichSet, null);
  }
}
```

This chart will behave exactly like "columnApp" unless you set the property "internalData" to "true". In that case, it will use exactly 1 dataset, with the values "123,432,345".

You will likely want to override "getDataset" to use data from your JDBC connection, calculated values, or something much more interesting than our simple example, but the concepts are the same.

This situation frequently lends itself to the use of Java's anonymous class feature. The subclass in the example above could also be expressed like this:

```
MyChart = new columnApp{
  public Dataset getDataset(ChartInterface chart,
                                    int whichSet) {
      if(whichSet>0)
            return null;
      double[] yValues = {123,432,345};
      return new Dataset("hello, world",
                        yValues, whichSet, null);
  }
};
```

This syntax has the significant advantage of avoiding another Java source code file.

If the chart you've chosen uses discontinuities, such as disLineApp or disDateLineApp, you can set discontinuities in your data by using the com.ve.kavachart.chart.Datum.DISCONTINIUTY flag. Here's a code example, using "getDataset":

```
import com.ve.kavachart.chart.*;
public void CustomDiscontinuousLineApp
   extends com.ve.kavachart.servlet.disLineApp {
      public Dataset getDataset(ChartInterface chart,
                                       int whichSet) {
            if(whichSet>0)
                  return null;
            Dataset d = new Dataset();
            for(int i=0;i<50;i++){
                  Datum dat = new Datum((double)i,
                                    Math.random(), null);
                  if(i%10==0)
                        dat.setLabel(Datum.DISCONTINUITY);
                  d.addDatum(dat);
            }
            return d;
      }
}
```

This example will create a line break at every tenth point.

Note:

The Dataset class has many constructors. See the API documentation for more information on how to create Dataset classes.

getMyDatasets

Now let's examine the "getMyDatasets" method. This method is called when the property "customDatasetHandler" is defined. The value of that property is passed to the method "getMyDatasets. Here's a simple example.

```
public void CustomColumnApp extends
                    com.ve.kavachart.servlet.columnApp {
      public void getMyDatasets(String parameters){
            double[] yValues = {123,432,345};
            chart.addDataset("my data", yValues);
      }
}
```

Here we use the protected field "chart" from columnApp's parent, "Bean". "Chart" is a "com.ve.kavachart.chart.ChartInterface" type that contains an instance of "com.ve.kavachart.chart.ColumnChart". We're just using that Interface's addDataset method to populate the chart data. We can also use other com.ve.kavachart.chart.Dataset methods here to manipulate out data directly.

In the properties file, we're defining the property "customDatasetHandler". This property's value is passed to "getMyDatasets", and it signals the chart that we want to use the method "getMyDatasets" for our data.

Although this example is trivial, you can do anything you want in getMyDatasets, including JDBC connections, data calculations, or other Java code that makes sense for your application.

You can also call the chart object's parameter parser to process dataset related params, like dataset0Color, dataset0FillStyle, and so on. To do this, send your data to the parser's "getDatasetParameters" method instead of adding the dataset directly. Here's our java code from above, modified to call getDatasetParameters:

```
public void CustomColumnApp extends com.ve.kavachart.servlet.columnApp
{
      public void getMyDatasets(String parameters){
            double[] yValues = {123,432,345};
            parser.getDatasetParameters(0, null, yValues, null, null);
      }
}
```

If you examine the source code for KavaChart's time oriented charts (e.g. dateLineApp, dateAreaApp, etc.) you'll see that we implement this method with a custom version of ParameterParser, called DateStreamReader. DateStreamReader contains date-aware features for handling data. This class also parses properties related to uniqe time-oriented axis features

Generally, you'll find the simpler method, "getDataset", better for your own chart object subclass.

Getting Custom Options

Your chart may require some custom properties. KavaChart server beans include a method stub for handling these properties, after obtaining generalized options. Although you could include these options after the rest of your server chart bean initialization, it's a good idea to use "getMyOptions" to make your bean code easy to translate to a KavaChart applet. Here's a simple example:

```
public void getMyOptions() {
      LineChart l = (LineChart) chart;
      String str = getParameter("BlackLine");
      if(str!=null){
      if(str.equalsIgnoreCase("true")){

      l.getDatasets()[0].getGc().setLineColor(Color.black);
            }
       }
}
```

Because the data acquisition methods have already been called, we can be sure we have data for dataset 0, and we just set the line color to black if the property instructs us to.

Custom Drawing

If you want to draw something on top of your chart, like a watermark, a copyright notice, or an axis threshold line, use the method stub "drawMyStuff". This method uses the same java.awt.Graphics class the server object uses to paint the image buffer, so anything you paint to this class will appear on your final image. In the example below, we'll just draw some text:

```
public void drawMyStuff(Graphics g) {
      g.setColor(Color.black);
      g.drawString("hey you", 0, chart.getGloabls().getMaxX() - 30);
}
```

Remember that since the "chart" variable is also available to you at this point, you can use classes in the com.ve.kavachart.chart package. In particular, you might want to use com.ve.kavachart.chart.DataTransform to draw threshold lines over your data, or com.ve.kavachart.chart.Highlighter to highlight portions of your chart. We're using it here to get the width of our image.

Selected ParameterParser Methods

Every KavaChart server object (and applet) uses an instance of ParameterParser. Since this class has some very useful utility methods that you can use from both applets and server beans, we'll quickly review some of them here. To obtain an chart's ParameterParser, just use the protected variable "parser", instantiated automatically the "getOptions" method of a server chart.

URL handling

ParameterParser has methods for opening and closing URLs, as well as for retrieving individual lines from a URL. You can also open a file on your server using these methods. See your API documentation for more information.

Axis Option Handling

Custom charts frequently have more than one axis. Use ParameterParser's "parseAxOptions" method to get all the general options for any AxisInterface. This method parses and applies options like auxAxisTitleString, auxAxisColor, auxAxisStart, etc. In this example, we called our axis "auxAxis", but you can refer to it any way you'd like.

Color parsing

Because charts frequently need color definitions, ParameterParser includes a utility method for translating strings into colors. Use the "getColor" method to do this in a consistent way in your custom chart.

Font parsing

Similarly, charts need frequent font definitions from parameter strings. The utility method "getFont" translates a string into a java.awt.Font class, in a way consistent with other KavaChart font definitions.

Label arrays

To translate an list of items delimited by commas (or your specified delimiter) into an array of strings, use the "getLabels" method.

Property Handling

By using "parser.getParameter(String)" you can get property values in a platform neutral way. This method works much the same as "getParameter(String)" in any applet, but it works for both server object properties and applet parameters.

Numeric arrays

Charting objects frequently need to translate property lists of number strings into arrays of numeric data. Use ParameterParser's "getVals" method to get an array of doubles from a delimited list of number strings. Strings that can't be translated become com.ve.kavachart.chart.Datum.DEFAULT. Translation is done according to localization defaults.

Image Loading

Use ParameterParser's "makeURLImage" method to load an image file in a context neutral way. In an applet, this is basically the same as calling "getImage", but uses a MediaTracker to ensure that the image is available before the chart is painted. In a server object, the image can also be loaded from the local filesystem.

Styles and Colors

Almost all KavaChart style information is stored in com.ve.kavachart.chart.Gc classes. These classes exist in most KavaChart elements, such as Backgrounds, Plotareas, Datasets, and so on. ParameterParser has a utility method for setting

a Gc's properties through parameters. Use "parser.populateGc" with a string that describes your Gc and the Gc, like this:

```
…
Gc gc = chart.getBackground().getGc();
parser.populateGc("secondBackground", gc);
```

Most of the graphics contexts for charts are already parsed, but you might find it convenient to use a Gc class for your own drawing, done with "drawMyStuff".

Locale Strings

Use ParameterParser's "getLocale" method to translate almost any locale string into a Java locale.

Overriding Default Behaviors

You may wish to change some default behavior in a chart without creating a complete custom chart. For example, you might want to create a chart that always has ***exactly*** 4 Y axis labels, but calculates the minimum and maximum values automatically.

To do this, override the "getOptions()" method to modify the chart. This makes modifications to the internal chart class after it has been instantiated, but before the chart properties are processed to modify defaults. Here is an example:

```
MyChart = new com.ve.kavachart.servlet.lineApp(){
      public void getOptions(){
            super.getOptions();
            Axis a = new Axis(){
                  public void scale(){
                        super.scale();
                        numLabels = 4;
                  }
            };
            chart.setYAxis(a);
      }
};
```

In this example, we changed the internal chart's default Y axis into a new anonymous class that overrides the scaling behavior to always draw exactly 4 labels. The syntax might look a little bit obscure, but by using anonymous classes, we have changed the axis scaling behavior of our internal chart without creating any new Java source files.

Note:

If you're planning to use a customized server object in a ChartTag, remember to use the fully qualified class name (e.g. custom.CoolChart) with the ChartTag's chartType attribute.

Changing a Server Object into an Applet

Server objects that are constructed using KavaChart's framework can be quickly translated into applets, as long as they can run within the constraints of a browser's security limitations. KavaChart applets extend the abstract com.ve.kavachart.applet.ChartAppShell, and KavaChart server objects extend the abstract com.ve.kavachart.servlet.Bean. Explicit subclasses of either abstract class do the following:

Create an instance of a com.ve.kavachart.chart.ChartInterface (like "BarChart") and assign it to the protected variable "chart" in the "init()" method. Init() then calls the method "getOptions()" to proceed with initialization.

GetOptions() installs a "com.ve.kavachart.utility.ParameterParser", which does the work of translating parameters (or properties, in the case of server objects) into chart attributes.

Both applets and server objects use the method "getMyOptions()" to obtain parameters (or properties) related to a specific chart type.

Both applets and server objects use the same utility methods for obtaining data (getDataset, getMyDatasets) and drawing overlays (drawMyStuff).

Generally, you can translate a KavaChart server chart into the equivalent applet by changing one line in the class definition. Simply change the class to extend com.ve.kavachart.applet.ChartAppShell (or one of its subclasses) instead of extending com.ve.kavachart.servlet.Bean, and you have a working applet, with dwell labels, hyperlinks, printing facilities, and so on.

Change this:

```
public class MyServerChart extends
      com.ve.kavachart.servlet.lineApp{ …
```

To this:

```
public class MyApplet extends com.ve.kavachart.applet.lineApp{
      …
```

See the chapters on KavaChart's applets for more information about how to use client-side features.

Installing an Alternate Image Encoder

KavaChart server objects have built-in support for most popular image formats. However, you may want to install another image encoder that meets a different set of requirements.

To do this, first create a class that extends com.ve.kavachart.servlet.UserImagingCodec, and override one of these methods:

```
public byte[] drawChartToOutputStream(ChartInterface)

public byte[] encodeImageBytes(Image)
```

Use the first method if you want to preserve chart geometries in a scalable way, such as with PostScript output or SVG (Scalable Vector Graphics) output.

Use the second method if you want to translate image rasters into a data stream. You might want to do this to add watermarking, animation or transparency to GIF output, for example.

Finally, install the encoder like this:

```
MyChart.setUserImagingCodec(myEncoder);
```

If such an encoder is installed in the chart object, it will be used instead of the built-in imaging tools, regardless of how the property "imageType" is set.

The encoded output will be treated like any other KavaChart server object output. It can be cached in the cache directory with automatic naming conventions, written to a file with a specified name, or returned as a byte stream.

The Developers Kit includes three samples you can experiment with. First, "com.ve.kavachart.contrib.PostScriptEncoder" paints the server chart's internal ChartInterface to a PostScript stream, using a public domain PostScript generator. This output is suitable for inclusion in larger PostScript documents or PDF files.

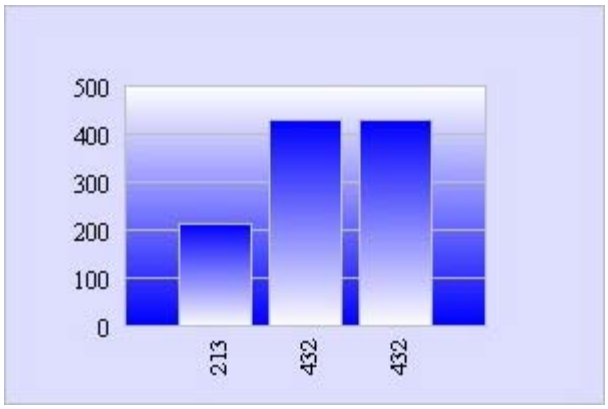
The second sample, "com.ve.kavachart.contrib.GifEncoderForKava" is based on the free Acme GIFEncoder class. This sample uses that encoder with an image filter to generate output with transparent backgrounds. When included on a document page, the background image of that document will show through the usually opaque chart background. Other chart elements, such as bars, pie slices, etc. will be opaque.

Finally, "com.ve.kavachart.contrib.JPEGEncoder" uses Sun's "com.sun.image.codecs.jpeg" classes to generate JPEG output. This encoder gives you the ability to adjust the quality of the JPEG output to achieve the desired balance of image quality versus image file size.

This simple standalone program uses JPEGEncoder to generate tunable output quality:

```
import java.io.IOException;
import com.ve.kavachart.chart.*;

import com.ve.kavachart.contrib.JPEGEncoder;

public class JPEGEncoderMagic {


      protected void doIt() {

         Bean bean = new columnApp();

         bean.setProperty("dataset0yValues", "213,432,432");
         bean.setProperty("xAxisLabelAngle", "90");
         bean.setProperty("dataset0Gradient", "2");
         bean.setProperty("outlineColor", "lightGray");
         bean.setProperty("plotAreaGradient", "2");
         bean.setProperty("plotAreaSecondaryColor", "blue");
         bean.setProperty("yAxisOptions", "gridOn,tickOff");
         bean.setProperty("yAxisLineColor", "lightGray");
         bean.setProperty("xAxisOptions", "tickOff");
         bean.setProperty("yAxisGridColor", "lightGray");
         bean.setProperty("noRotations", "true");
         bean.setProperty("backgroundColor", "ddddff");

         bean.setProperty("writeDirectory", ".");

         bean.setProperty("width", "300");
         bean.setProperty("height", "200");
         bean.setProperty("fileName", "test.jpg");

        JPEGEncoder encoder = new JPEGEncoder();
        // high quality but lower compression
        encoder.setQuality(0.9);
        bean.setUserImagingCodec(encoder);

         //generate our output here...
         try{
         System.out.println("filename:" + bean.getFileName());
         }catch(Exception e){
               System.out.println("oops");
               e.printStackTrace();
         }
      }

      public static void main(String[] s){
         JPEGEncoderMagic attempt = new JPEGEncoderMagic();
         attempt.doIt();
         System.exit(0);
      }
}
```

Output from this code looks like this:


500
400
300
200
100
0
213
432
432

Chapter

5

KavaChart's Core API

This chapter discusses the internal KavaChart architecture, and how to use the com.ve.kavachart.chart package. If you want to make extensions to KavaChart classes, or build custom charting components, this chapter will provide you with overview information to start. Consult the API documentation for implementation specifics.

When to Use the Core Charting Classes

Most applications can use one of KavaChart's built-in collections (applets, server objects, beans) to get quick results. These collections combine chart classes from the "com.ve.kavachart.chart" library with some additional functionality to meet a general requirement.

For example, the applet collection uses chart objects from "com.ve.kavachart.chart" for painting chart images. The applet collection adds parameter parsing, thread management, printing intelligence, hyperlinking, etc., all in the context of a subclass of "java.applet.Applet". Similarly, the server object collection uses chart objects from "com.ve.kavachart.chart", but adds image encoding, property parsing, cache management, and hyperlinking to facilitate use on a typical server.

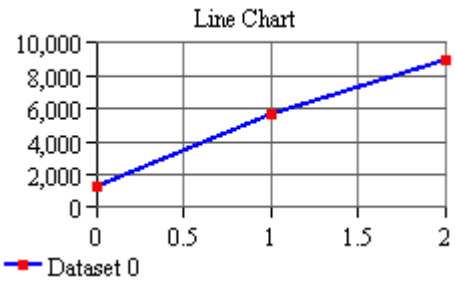
What's in "com.ve.kavachart .chart"?

Each of these collections consists of software that:

- Provides a convenient way to manipulate the internal classes and methods in "com.ve.kavachart.chart" with a minimum of Java code.
- Provides a graphics context to paint charts that are defined in "com.ve.kavachart.chart".
- Provides a context for using the output of "com.ve.kavachart.chart" chart objects.

Classes in "com.ve.kavachart.chart" do the actual work of creating a chart. These include Axis classes that calculate and draw various kinds of axis scales, DataRepresentation classes that provide specific visual descriptions of data, Dataset and Datum classes for storing and organizing data, and various helper classes that tie these objects together.

These classes are collected into specific kinds of charts. A chart class provides an organizational structure and implements a few convenience methods, but it is primarily a container for charting objects. For example, let's examine "com.ve.kavachart.chart.LineChart".



This chart object is a subclass of "com.ve.kavachart.chart.Chart". It implements the interface "com.ve.kavachart.chart.ChartInterface". It contains the following classes from "com.ve.kavachart.chart":

- "Background", a rectangle beneath all other objects, with a title and subtitle.

- "Line", a subclass of "DataRepresentation". This class is responsible for drawing plot lines and markers for an array of Dataset classes. It requires a "Plotarea" to define a drawing region, and two "AxisInterface" classes to obtain transform geometries (e.g. information about how numbers in the Datasets map into the region defined by the Plotarea).

- An array of "Dataset" classes, which are used to organize data. Each dataset class has a name, some color and gaphics information, and a Vector of Datum classes. The Datum objects contain the values we're plotting, along with some additional color and labeling information.

- A "Plotarea" class, which defines the region of the chart image to be used for plotting data. By default, this region is bounded by our chart's axes.

- Two "Axis" classes, one that calculates and draws a vertical axis (the Y axis in this chart), and another one that caculates and draws a horizontal axis (the X axis in this chart).

- A "Legend" class, which draws a description of each "Dataset" in this chart's Dataset array.

- The chart also contains a couple of invisible classes that provide important helper functions:

    - A "DisplayList" class stores drawing geometric information to identify other chart objects by location.

    - A "Globals" class is shared by all the objects in the chart to store shared information like default fonts, and locales.

    - A "RotateString" class provides text rotation facilities for Java versions that don't support Java2D.

Even though LineChart acts primarily as a container for these classes, it also inherits some useful methods from its parent, "com.ve.kavachart.chart.Chart":

- "addDataset" convenience methods. These methods translate arrays of doubles and Strings into Dataset and Datum classes, and install the Datasets into this chart.

- A set of "drawGraph" and "paint" methods. These methods are responsible for dispatching the "draw" methods of this chart's internal classes in the proper order.

- Access methods for the various internal chart components. For example, a "setDataRepresentation" or "setLine" method lets you replace the default "Line" class with some other dataset drawing object.

Note:

Since charts are primarily containers, KavaChart's "get" access methods are critical to using the core charting API. For example, if you want to change the color of the title, you'll first need to "get" the Background to access the "setTitleColor" method:
`chart.getBackground().setTitleColor(Color.green);`

LineChart is typical of other charts found in the "com.ve.kavachart.chart" package: it inherits from "com.ve.kavachart.chart.Chart". It uses other classes from "com.ve.kavachart.chart" to do drawing and data management. It acts primarily as a container and manager for these classes.

Build your own applet

Although KavaChart includes a collection of chart applets, you might need to add charts to an existing applet. To do this, you should probably use the core charting classes in "com.ve.kavachart.chart", or use a JavaBean from our "com.ve.kavachart.beans.chart" package. Our beans are described in later chapters.

A later section describes how to build a simple applet that uses KavaChart to paint the applet. You may also want to build your own JavaBean that encapsulates a KavaChart graphic in a Panel or JPanel subclass, and then add that to your applet. Later sections describe how to build a simple reusable chart panel suitable for use in an applet.

Build your own servlet

KavaChart's server objects meet the needs of the majority of server applications. However, you may want to build your own image generation servlet using KavaChart's core classes.

A later section describes in detail how to build a servlet using the core classes.

KavaChart in a standalone application

If you're building a Java application that doesn't run as an applet or a servlet, you will want to use KavaChart's core charting library or JavaBeans package. Using the core package directly has advantages if you want to create custom data representations, or somehow modify the way KavaChart works.

For example, if you want your application to label the Y axis using some kind of automatically generated labels like "big, bigger, biggest", then you could install this as an axis LabelFormat property.

Cook your own JavaBeans

KavaChart's core charting library is a great tool for building your own custom JavaBeans. In this case, you will probably follow the general model for a "reusable chart panel" described below, but will expose the chart and data features important to your application with "get" and "set" access methods.

You can also create data models, customizers, event handlers, and so on, similar to the pre-built KavaChart beans packages. A KavaChart based bean can be as simple or as complex as you want.

Extend or customize an existing chart

KavaChart's framework is designed to make it easy to build your own custom charts.

You can create a DataRepresentation that displays your data in with a different graphic than our defaults, and still take advantage of axes, legends, and so on.

You can build a custom axis that draws labels in reverse order, along a curve, or with some alternate scaling algorithm.

Perhaps you need to create an extended Datum class that contains more than just the usual numeric and string observations.

These sorts of extensions and customizations will require you to gain some familiarity with KavaChart's core charting framework.

Core Components

KavaChart's "com.ve.kavachart.chart" package comprises a set of graphical and non-graphical objects. These classes are combined into chart containers, which are generally subclasses of "com.ve.kavachart.chart.Chart", and implement "com.ve.kavachart.chart.ChartInterface". This overall organization is described above.

The following sections describe these components in more detail. We'll look at how KavaChart's internal structures are organized and examine some of the more common class hierarchies. For complete programming examples using the "com.ve.kavachart.chart" API, see the next chapter.

> **Note:**
>
> For class API details, method names and method signatures, consult the API javadoc.

Globals

The Globals class stores information that all other classes in a chart will require. For example, Globals is used to store a chart display list, and information about the overall height and width of a chart, in pixels. You shouldn't ever need to create a new instance of Globals unless you're creating a custom chart. This class is created automatically when charts are initialized.

Gc

All graphical classes (classes that appear visually) use at least one Gc class to store a graphical context. The Gc class determines fill and line colors for an object, gradient and texture styles, marker styles, line styles and so on.

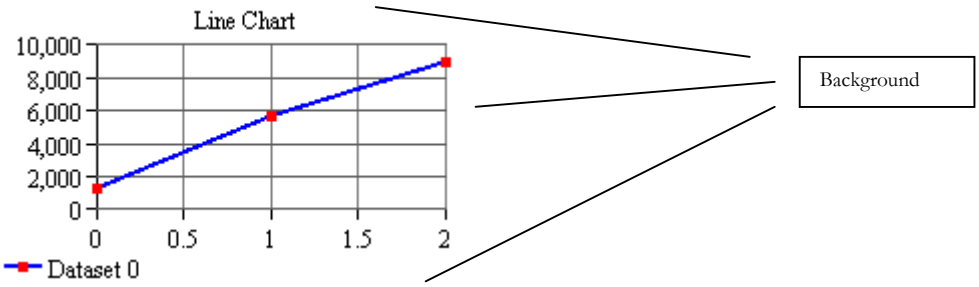
If a Gc is used to paint a filled object, that object will be outlined with the Gc's line attributes if the Gc's outlineFills property is set to "true".

Objects with a Gc colors set to "Gc.TRANSPARENT" are not drawn.

This class also implements various drawing functions used throughout KavaChart. Most drawing is actually done by an object's Gc class. If your application uses Java2D, Gc loads a helper class to implement advanced drawing functions.

Because all visual KavaChart classes use Gc for drawing functions, your applications can "get" and "set" graphical attributes in a uniform way.

Background



This class draws a rectangle beneath the overall chart. It also draws a title and subtitle. By default, the title and subtitle are centered at the top of the chart. Set the "titleX", "titleY", "subTitleX", or "subTitleY" properties to place the text at another location.

User defined title positions are expressed in percentages of the overall chart area. For example, a title drawn at 0.5, 0.5 will be drawn at the center of the entire chart image.

Background contains attribute properties for the title and subtitle font and color. The color, fill style, and outline color of a Background are determined by the Background's Gc class. To change a chart's Background to light gray:

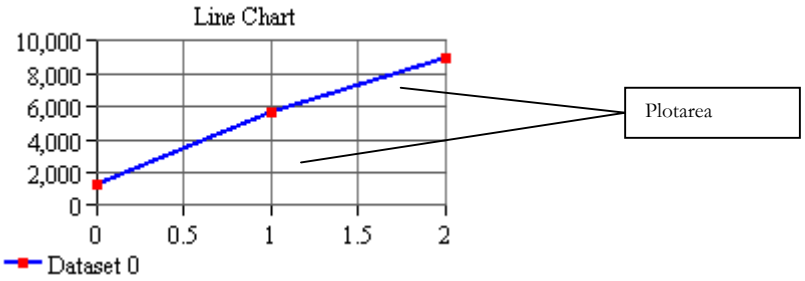
```
chart.getBackground().getGc().setFillColor(Color.lightGray);
```

To change a chart's title:

```
chart.getBackground().setTitleString("my new title");
```

You can create Background subclasses that add logos, watermarks, additional title lines, etc. Background provides a convenient place to insert graphics drawn before any other chart component.

Plotarea



A Plotarea draws a rectangle over the Background and beneath axes and DataRepresentations (such as Bar, Line, etc.). Plotarea has a Gc that determines colors and outlining, as well as a Globals that determines whether the Plotarea will be drawn with a 3D effect.

Plotarea has properties that determine the placement of the lower left corner (default 0.2, 0.2) and the upper right corner (default 0.8, 0.8). Since axes are typically attached to the edges of the Plotarea, these settings determine how much of the chart image the plotting region will occupy. A large plotarea will leave small margins (used for axis labels, legends, titles, etc.) and a small plotarea will leave large margins.

Plotarea coordinates are expressed as a percentage of the overall chart image size. For example, if your chart image is 500 pixels wide, the default Plotarea right value (0.8) would be at 400 pixels. The default Plotarea left value (0.2) would be at pixel 100.

DataRepresentation

KavaChart uses an abstract superclass called "DataRepresentation" as a starting point for all visual representations of data. These include various kinds of bars, lines, polygon shapes (Bubble, Area), and combinations of lines and polygons (BoxWhisker, Candlestick). DataRepresentations create output that represents an array of Dataset classes.

A DataRepresentation typically interacts with two Axis classes and a Plotarea. The Axis classes determine starting and ending values. This determines how the output should be scaled. The Plotarea determines where to place the scaled output. The process typically looks like this:

- The chart object draws the X and Y axes. This causes the axes to calculate starting and ending value properties.
- The DataRepresentation creates a "java.chart.Transform" based on the starting and ending values of the axis scales and the Plotarea location and size.

- The DataRepresentation goes through each Dataset in its array, using the Transform to translate Datum values into pixel values. These pixel values are used to create graphics based on the rules of the specific DataRepresentation. For example, bars will draw rectangles, lines will draw connected lines, and so on.

- Drawing commands are directed to an appropriate "com.ve.kavachart.chart.Gc" class. Some DataRepresentations use the Dataset Gc. For example, a Line typically uses a single color and marker to represent an entire set of data. A Pie uses the Datum Gc, since each slice needs to be a different color.

Generally, DataRepresentations can display Datum labels or data values along with the graphical representation (e.g. bar labels, line labels, etc.). When a numeric label is displayed, the DataRepresentation's LabelFormat property determines the appearance and formatting of the label. A LabelPrecision property lets you change the number of decimal places dislayed by numeric labels. DataRepresentation also stores label font and color information.

You can set general DataRepresentation properties by obtaining a chart's DataRepresentation object:

```
MyChart.getDataRepresentation().setLabelFont(myFont);
```

Properties of DataRepresentation subclasses can be set by casting the DataRepresentation into the appropriate object:

```
((Bar)MyChart.getDataRepresentation()).setClusterWidth(0.65);
```

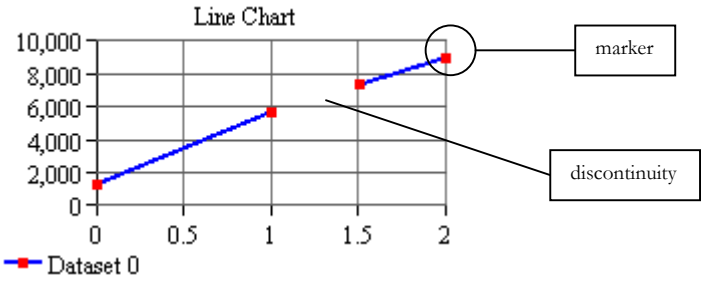
Most charts also include DataRepresentation accessors with a mnemonic name. This returns the internal DataRepresentation as a specific subclass. The example above is synonymous with this:

```
MyBarChart.getBar().setClusterWidth(0.65);
```

Line Classes: Line, DiscontinuousLine, Regress

Line classes are probably the simplest DataRepresentation. These classes are used to create various line charts and scatter plots.

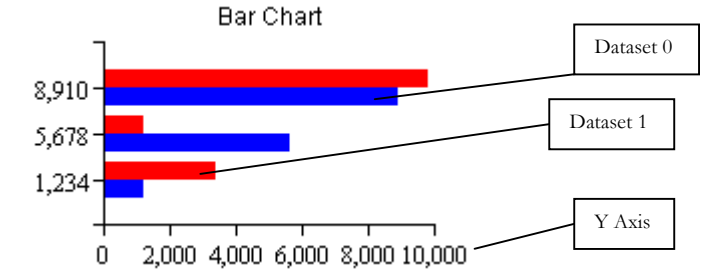
Line draws a line segment between each point in a Dataset, in the order of the Dataset's data Vector. DiscontinuousLine creates a break in the line at any point with a Datum label of "Datum.DISCONTINUITY". Regress calculates an ordinary least squares regression line, which is drawn to the edges of the Plotarea.

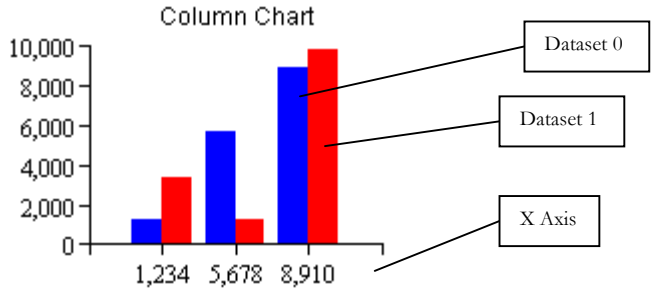


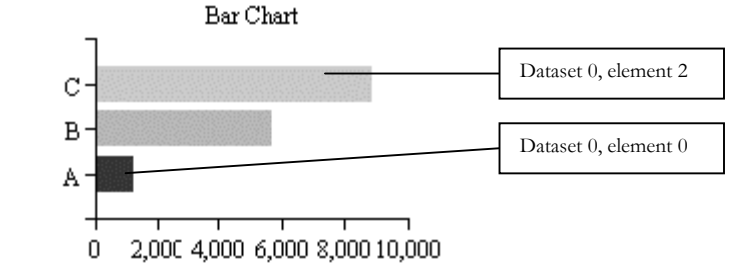
A "ScatterPlot" property turns the line off to create a scatter plot. Markers are drawn using the Dataset Gc's MarkerStyle property, unless the Line's IndividualMarkers property is set to "true". In this case, markers styles are taken from the individual Datum Gc classes. Datasets or Datum classes that have an Image in their Gc will use the Image as a marker.

Bar: Column, Bar, StackBar, StackColumn, Candlestick, Hi-Lo Bars, HLOC

Bar charts display discrete observations with some kind of rectangular graphic. KavaChart's Bar DataRepresentations can stack or cluster bars. Bars may be horizontal or vertical. Some bar types can also draw a hi and low (or start and end) value. Other specialized types draw extended Datum information, such as Candlestick and HLOC (4 numeric values).





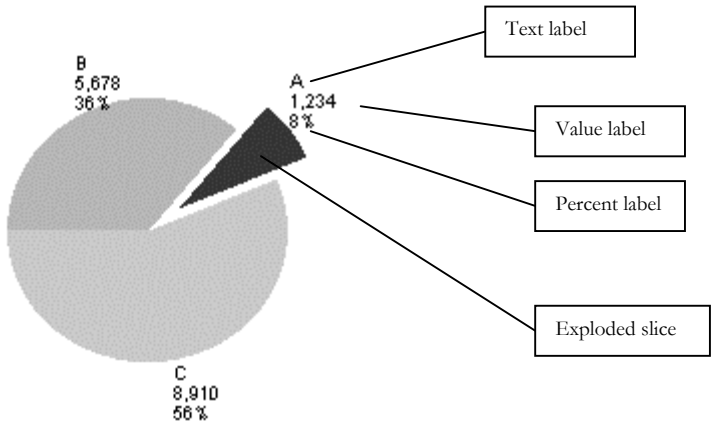


All Bars share properties and drawing methods common to the Bar superclass. Bars generally have 2-D or 3-D effects, and may or may not have visible labels. Other significant properties of Bars include:

- baseline sets the value from which bars will ascend or descend.
- labelsVisible determines whether bar labels will be displayed.
- labelPrecision determines how many decimal places will display for the Bar's automatic numeric labels.
- labelFormat determines how a Bar's numeric labels will appear. By default, they will be formatted to the conventions of the JVM's locale settings. You can override the format to any subclass of java.text.Format.
- labelAngle sets the angle at which bar labels draw.
- clusterWidth sets the width of a cluster of bars, as a percentage of the available space (1.0 means that clusters will abut, 0.5 means that clusters are as wide as the space separating clusters).
- xAxis, yAxis - the axes used to scale this bar.
- useValueLabels – determines whether bar labels will derive from the Datum Y value or the Datum label string.
- doClip – determines whether the bar will employ a Graphics clip rectangle to ensure that bars will be clipped at the edge of the Plotarea.
- unitScaling – Normally, bars are used to describe categories, such as "servers, routers, workstations". In this case, Datum X values are ignored, and the bars are distributed evenly along the X axis. The implicit X locations are "0, 1, 2, …". KavaChart refers to this as "unitScaling". In some cases, however, bars need to appear at an actual X value. For example, Candlesticks typically appear along a time scale, which uses millisecond values to calculate scaling heuristics. If the Candlestick bars are to appear at the proper location, they must transform the Datum's X value based on a time value rather than a category. KavaChart behaves this way when unitScaling is set to false.

Pie

Pie charts use a "com.ve.kavachart.chart.Pie" DataRepresentation. Pies draw a single Dataset, the first in the DataRepresentation's Dataset array. Slice color and fill styles are determined by each Datum's Gc class. The size of each slice is determined by each Datum's Y value.



Pies can be drawn with 2 or 3 dimensional effects. Each slice can be exploded away from the center of the pie by an individually variable amount. Labels can be placed at the edge of each slice, inside the slice, or at the end of a pointer. Labelling information can include the actual Y value, a calculated percentage, and each Datum's label value. Numeric labels are formatted according to the rules described by "LabelFormat" and "PercentFormat" properties. Defaults are determined by the JVM's Locale settings.

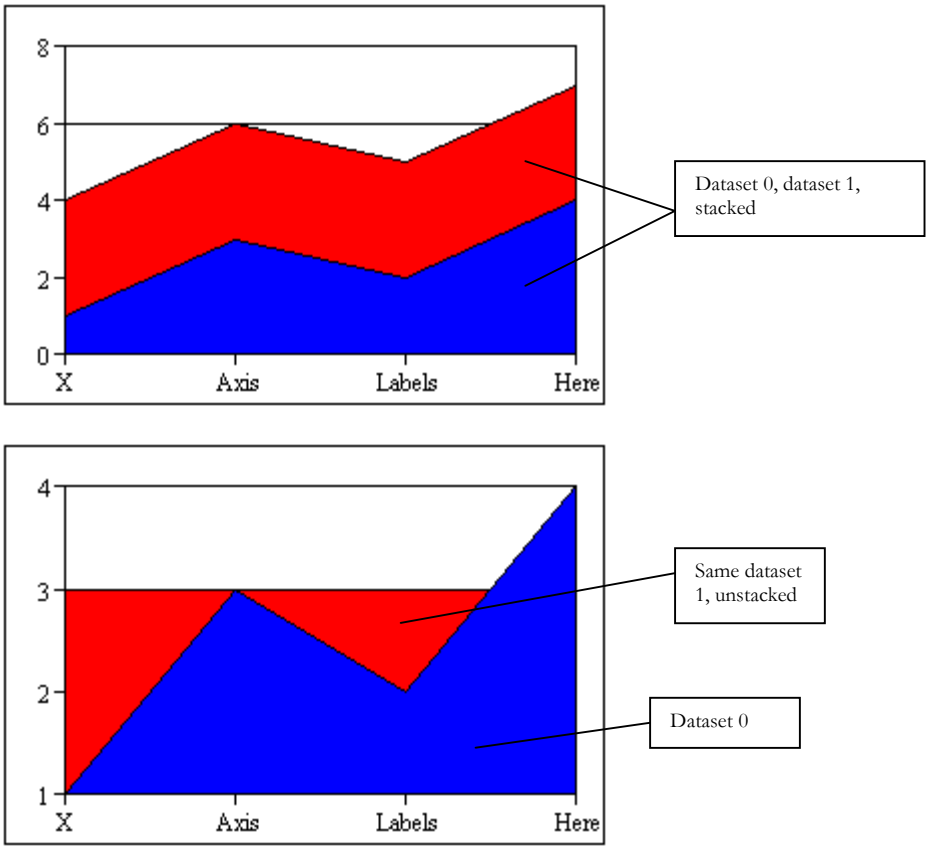
Pie chart legends display an icon for each slice, using the Datum label (if available) for a description.

Pies can be sized horizontally and vertically. Three dimensional pies generally look better if the width is greater than the height. The depth of a 3 dimensional pie is determined by the chart's Globals class.

- valueLabelsOn determines whether a numeric value should display for each pie slice.
- textLabelsOn determines whether a Datum label should display for each pie slice.
- percentLabelsOn determines whether a label showing each slice's percentage should display.
- startAngle determines the location of the first pie slice.
- pieSize determines the radius of this pie, which can be either an oval or a circle.
- pieLocation determines the location of the center of this pie.
- labelPrecision determines how many decimal places will display for value labels.
- percentFormat determines how percentages should be displayed.

Area, DiscontinuousArea

Area charts, including time oriented charts, use a "com.ve.kavachart.chart.Area" DataRepresentation. Areas display a solid color for each series. These regions are stacked one upon another by default.



If your Datasets have different X values, you should set the "stackValues" property to "false" to avoid confusion.

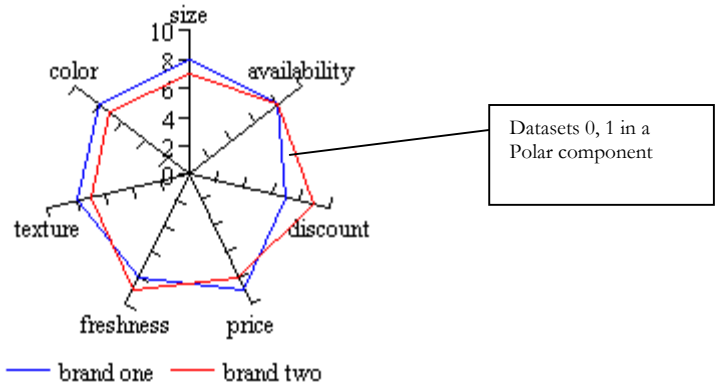
DiscontinuousAreas display a break wherever a Datum has a label value of "Datum.DISCONTINUITY". If your Datasets have discontinuities in different places, you should set the property "stackValues" to "false" to avoid confusion.

Areas may be displayed with two or three dimensional visual effects. Areas do not display labels. The color and fill style of each region is based on that region's Dataset Gc values.

Areas include a "doClip" property to ensure that polygons are clipped at the edges of the Area's Plotarea when the Area exceeds the bounds of the X or Y axis.

Kiviat: Polar

KavaChart's polar charts are also called "Kiviat" charts. The "com.ve.kavachart.chart.Polar" component draws a roughly circular line around a spoked axis. A polar component may draw markers in addition to, or instead of lines.

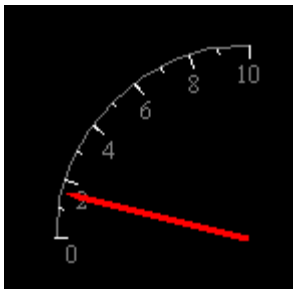


The Dataset Gc determines the line weight, style and color. It also determines the type of marker.

Polar requires a "com.ve.kavachart.chart.PolarAxis" to work properly.

Speedo, HistorySpeedo

A "com.ve.kavachart.chart.Speedo" is another special purpose DataRepresentation. This class is used to draw various kinds of gauges, in conjunction with "com.ve.kavachart.chart.SpeedoAxis". A Speedo draws the first value of the each dataset. Other values are unused, except for calculating a SpeedoAxis' extents.



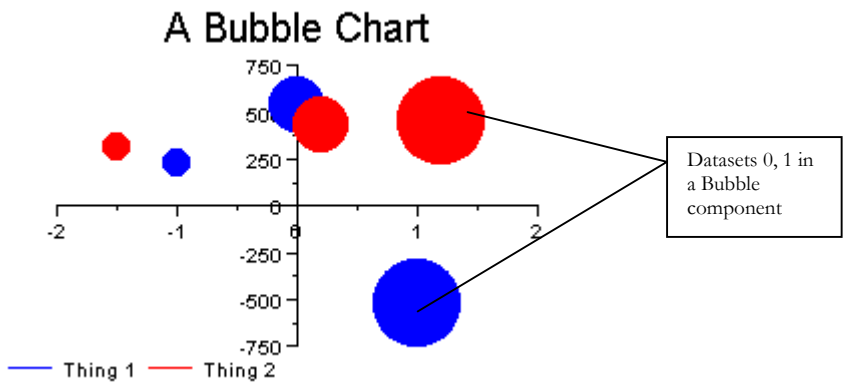
HistorySpeedo draws a sweep region in a separate color to record the historical activity of a speedo.

The shape of a speedo is dependent on its Axis. Speedo properties include:

- needleStyle - This is one of: 0 - a simple line with a small arrowhead, 1 - single line needle, 2 - polygonal "fat" needle, 3 - sweep arc.

Bubble

Bubble charts use a "com.ve.kavachart.parts.Bubble" together with conventional axes to construct a bubble chart. Bubbles may be filled or hollow. The X and Y values of each Datum in each Dataset determine the location of each bubble. The Y2 value of each Datum determines the radius of each bubble.



The radius of each bubble is scaled to some percentage of the Y axis. By default, Bubble automatically determines some appropriate value for this scalar. To override this, set the "zAutoScale" property to false, and set the scale value with "setZScale". Since most charts attempt to display relative Z values rather than absolute Z values, Z autoscaling is generally appropriate.

Other DataRepresentations : BoxWhisker, BoxJenkins, DailyPolar, StemLeaf, etc.

KavaChart's "com.ve.kavachart.parts" package includes a number of other interesting DataRepresentations. Generally these are part of a chart with a similar name (e.g. BoxJenkins is used in "com.ve.kavachart.contrib.BoxJenkinsChart"). Many of these DataRepresentations do mathematical calculations to generate metadata used in the final visual representation. See the API documentation for more information about available properties for individual components.

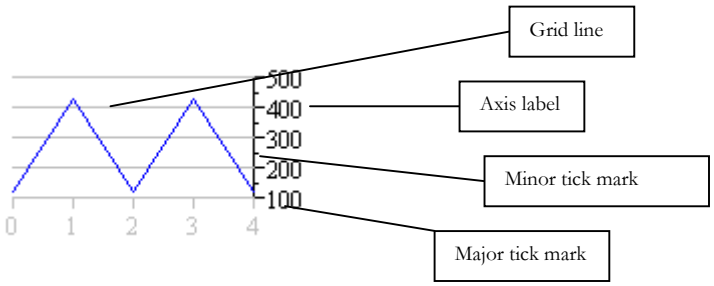
Components in the "parts" package are supplied in source code form to help you understand how to build your own custom components. Also, you may wish to change the metadata calculations to suit your own purposes.

Axis and its Subclasses

The "com.ve.kavachart.chart" package contains a number of classes to calculate and draw chart axes. These classes implement "com.ve.kavachart.chart.AxisInterface", and generally extend "com.ve.kavachart.chart.Axis". An Axis class draws a single axis.

Axis contains an array of Dataset classes. This is usually the same array used by the chart's DataRepresentation. Every AxisInterface implements a "scale" method, which examines the Datasets and sets internal scaling properties, including an axis start and end value. These start and end values are used by DataRepresentation classes to construct a "com.ve.kavachart.chart.Transform", which creates geometric transformations from data values to graphic values.

Axes draw scales that may include labels, major and minor tick marks, labels, and grid lines.



Axes include a Plotarea property. By default, an axis is drawn at an edge of the rectangular Plotarea region, defined by the axis constants TOP, BOTTOM, LEFT, and RIGHT.

An axis scaling calculation examines its Dataset classes to construct a scale. The simplest axes find the minimum and maximum values, and use these values to set the axis minimum and maximum. If the Axis' "isXAxis" property is set to "true", data X values are considered. Otherwise, Y values are used for this calculation.

The base class for most axes is "com.ve.kavachart.chart.Axis". This class uses a heuristic algorithm that attempts to create intuitive axis intervals. This algorithm may extend the start or end of the axis to accommodate this interval. For example, if your data ranges from 0.1 to 9.5, the axis scaling algorithm will create a scale that goes from 0 to 10.

Axis subclasses will use Dataset values in different ways. For example, the DateAxis subclass assumes that data values are based on Java's primitive millisecond time calculations, and creates time based intervals based on data values. StackAxis calculates the maximum sum of datapoints at specific locations. HiLoAxis considers both Y and Y2 values to create a scale that will accommodate both. And so on.

In general, axes operate as follows:

- Before a chart's DataRepresentation is drawn, the x and y axes are either drawn or scaled.
- When an axis is drawn, the "scale" method is called when the "autoScale" property is set to "true" (the default). Scale:
    - Examines the axis' Dataset classes to find minimums and maximums. If the "axisStart" or "axisEnd" properties are set, these are used instead of Dataset calculations.
    - Finds an appropriate interval, based on the axis type and the data range.
    - Calculates and sets the number of grids, ticks, and labels. (note: if you have set "autoScale" to false, these properties must be set manually for each axis).

- If labels are visible, the axis draws increment labels, based on the rules of this axis:
    - The method "whereOnAxis" is called iteratively for the number of labels set in "scale" to determine the location of the axis labels.
    - The method "getLabel" is called for each label to return a label formatted according to the rules of this axis.
- If grids, major ticks, or minor ticks are visible, these are drawn according to the rules of this axis:
    - The method "whereOnAxis" is called iteratively for the number of grids, etc. This method returns a pixel location for the item being drawn.
    - Colors and linestyles are set according to the properties of that item's Gc class (gridGc, for example).
- The DataRepresentation is subsequently drawn, and uses the remaining "axisStart" and "axisEnd" properties to set up a drawing transform.

Methods to Override

A number of axis methods have protected scope to permit extensive modification of default axis behaviors. In particular:

- "scale" – you can insert your own scaling algorithm here to examine datasets and set the necessary properties: axisStart, axisEnd, numGrids, numTicks, and numLabels.
- "whereOnAxis" – lets you modify the way items are layed out along the scale. For example, even though the standard axis distributes labels evenly along the axis, the DateAxis subclass distributes labels on time and date boundaries, which may be unevenly distributed.
- "getLabel" – even though you can install a Format class to format numeric labels, on the axis, you may want to do something more interesting. "getLabel" provides you with an item count and a numeric value, and lets you return any string you want.
- "draw" – you can override the method that draws the axis to great effect. For example, in a chart that has multiple Y axes, you might want to offset a Y axis to the left or right like this:

```
public void draw(Graphics g){
      Graphics shiftG = g.create();
      shiftG.translate(-20, 0);
      super.draw(shiftG);
      shiftG.dispose();
```

```
}
```

Since KavaChart's components (DataRepresentation, etc.) use AxisInterface instead of Axis, you can also build your own axis scaling logic to be combined with existing charts.

Axis Properties

The following properties are common to all Axis subclasses. Most are also relevant to implementations of AxisInterface:

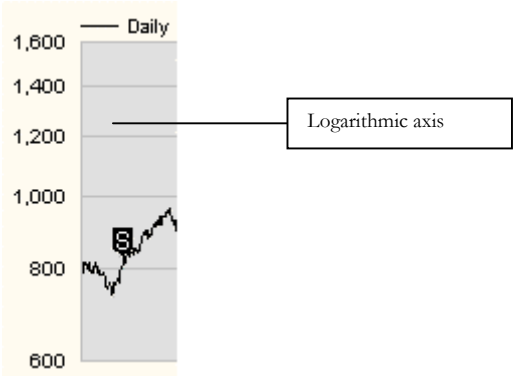
- The Plotarea bounded by and scaled by this Axis
- Gc (Colors, etc.)
    - Axis Line
    - Major Tick Mark
    - Minor Tick Mark
    - Grid Lines
- Length
    - Major Tick Mark
    - Minor Tick Mark
- Visibility
    - Axis Line
    - Major Tick Mark
    - Minor Tick Mark
    - Grid Lines
    - Labels
- labelColor
- labelFont
- labelAngle
- labelPrecision (how many decimal places to use)
- titleColor
- titleFont

- titleString
- titleRotated – true or false
- autoScale (either on or off, determines whether you or KavaChart will scale this axis)
- Number of Grids, Ticks, Labels
- startValue
- endValue
- xAxis (determines whether an Axis will be used for X or Y data)
- barScaling (if true, Y axes default to scale from 0)
- logScaling (not valid for all Axis subclasses)

com.ve.kavachart.chart.Axis

The base Axis class is used to create a numeric scale. By default this axis automatically chooses an appropriate scale based on the extrema of its Datasets. This class also contains most of the drawing and scaling heuristics used by its subclasses.

Axis creates linear or logarithmic increments. Log scaling automatically triggers logarithmic geometry transformation in the DataRepresentations using this Axis.



Logarithmic scales increment using common logarithms. These scales may not cross zero, although they are useful for dealing with extremely small numbers.

"com.ve.kavachart.chart.HiLoAxis"

A HiLoAxis works like a standard Axis, but it also evaluates the Y2 values in each Dataset to ensure that the axis extent is large enough for both Y and Y2 values. This type of Axis is appropriate for various hi-lo DataRepresentations, such as HLOC, HiLoBar, Candlestick, and so on.

"com.ve.kavachart.chart.StackAxis"

A StackAxis behaves like a standard Axis, but it calculates the extent by getting the largest stack of values at any given point. This type of axis is used with Area and StackBar DataRepresentations.

"com.ve.kavachart.chart.LabelAxis"

A LabelAxis draws textual labels instead of numeric labels. LabelAxis first attempts to use the label strings defined with "setAxisLabels". If no label string is defined for a particular item, the axis attempts to use the label string from dataset 0 at that increment. If no data label string is defined, the axis uses the value of dataset 0 at that increment.

This type of axis is typically used for bar charts, area charts, and any other chart that uses text labels instead of numbers. Its scaling behavior is also different than it's superclass, Axis. LabelAxis autoscales to create as many labels as there are observations (Datum classes) in Dataset 0.

This works well for charts without dense data. If your chart has many observations, a LabelAxis can become overcrowded. A subclass of LabelAxis, "com.ve.kavachart.parts.CullingLabelAxis" will automatically prune labels to display only as many labels as can be viewed at the chart size you defined.

The underlying numeric scale used by LabelAxis ranges from 0.0 to the number of observations in the first Dataset. This means that X values are implicitly ignored. This is appropriate for most Bar oriented charts. KavaChart also includes Line and Area charts that use LabelAxis (LabelLineChart, AreaChart). These charts may look odd if explicit X values are used (implicit X values are monotonically increasing integers).

You may want to use LabelAxis as your Y axis to display customized labels, but this is generally inadvisable. Users typically use a Y axis to understand the values presented by the chart's DataRepresentation. If you use a LabelAxis, you will end up setting start and end values and label increments that may not be meaningful to future code maintainers.

A better approach is to use Java's ChoiceFormat along with a conventional Axis. You can still control the axis extents and intervals if necessary, but your labels will have an explicit meaning, conveyed through the ChoiceFormat source code. For example, your ChoiceFormat might use labels of "bad", "adequate", "good", and "best", which would have explicit threshold values.

"com.ve.kavachart.chart.DateAxis"

KavaChart's DateAxis is especially useful for displaying time oriented data. This axis interprets Dataset X (or Y) values as times, expressed in "milliseconds since epoch, UTC". This value is available from all of Java's Date classes using the "getTime" method.

DateAxis then caculates a time span for the data, and selects a scaling model. Scaling models range from "second" oriented scaling through "year" oriented scaling.

The scaling model determines where axis increments should occur: on some multiple of hours, for instance, or on some multiple of days. An internal heuristic then chooses some "natural" multiple, such as 2 months, or 6 hours, to create an axis with approximately 4 to 7 labels. Finally, the beginning and end of the axis are extended to accommodate the scaling model together with the multiple used for increments.

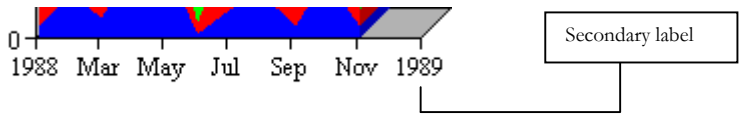
For example, if your data starts at 12:30 AM and ends at 5:17 AM, the axis might choose an hourly scaling model. Your data spans approximately 5 hours, so one label per hour might be appropriate. In order to place an axis label at the beginning of each hour, the Axis would likely extend the beginning of the axis to 12:00 AM, and extend the end of the axis to 6:00 AM, yielding a 6 hour axis with labels at the beginning of each hour.

Note:

DateAxis adds space to the beginning and ending of the axis to create natural labeling increments. This means your chart may have some empty space before or after your data.

Default DateAxis labels are based on the scaling model and the actual increment. For example, an hourly scaling model will use labels that display the hour and minute, but not the second nor the date. A yearly scaling model will display the year, but not the month.

In addition to the format class common to all Axis subclasses, DateAxis also uses a "secondaryLabelFormat". This format is used when your data crosses a special boundary. For example, if the axis scaling model is monthly scaling, the secondary label format will be used for the month of January to add information about the year.



By default, all automatic label formats use the default locale for the Java VM. This provides a sort of automatic localization for applications that use the default formats.

Note:

DateAxis also uses a localized Calendar class for calculating axis increments. It's important to keep this in mind when you supply data to your chart. The same millisecond value might be January 1 to a viewer in London, but December 31 to a viewer in San Francisco. You can avoid this by making sure the Date classes used to create your data use the timezone you intend viewers to see.

Special DateAxis properties:

- scalingType – by choosing one of the predefined scalingTypes you can override default scaling behaviors, even when the axis autoScale property is set to "true". For example, set the scalingType to HOUR_SCALING if you want labels to be some multiple of hours, even though the default might be labels every day.

- secondaryDateFormat – as discussed above, this property provides special formatting information for labels that occur on significant boundaries, such as a new year or a new day.

- calendar – you can override the default calendar behavior by installing your own calendar subclass.

- timezone – allows you to override the default timezone used for calendar calculations.

Although DateAxis has a very powerful scaling heuristic that generally creates a reasonable scale for most data ranges, you might want to set "autoScale" to "false" to obtain more control over the axis. This is generally inadvisable, even though DateAxis will attempt to accommodate your settings.

Note:

DateAxis attempts to accommodate manual scale properties, such as numLabels and numTicks, but it's usually better to let autoScale determine these properties. Use a LabelAxis if you need absolute control over labeling.

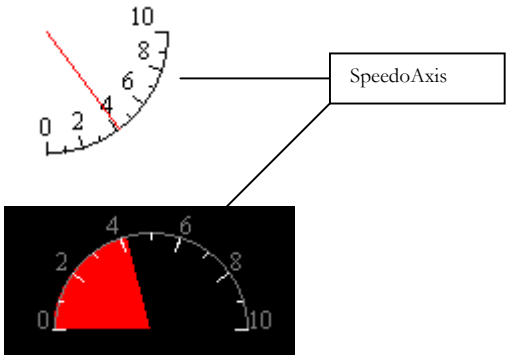
Time periods are irregular (months, days, and years have irregular sizes), and time labels are generally imprecise, so a manually scaled axis can lead to misleading labels. For example, if you use a label format that shows only the day of the week, a particular label of "Thursday" might occur at 3:01 AM or at 11:53 PM. This sort of behavior is difficult for users to understand.

If you need this sort of control over labeling, you should consider using a chart that contains a LabelAxis. This axis lets you set the number of labels and the actual labels that will occur. Data is handled differently for these charts as well, which might be easier for some applications.

You might also want to use the Axis subclasses "com.ve.kavachart.parts.InsetDateAxis" or "com.ve.kavachart.parts.FinanceAxis", discussed below. While these axes don't give you more control over scaling behaviors, the default behavior is more appropriate for some tasks.

com.ve.kavachart.chart.SpeedoAxis

A SpeedoAxis is used to draw a numeric scale for a "Speedo" DataRepresentation. This scale is a "Y" axis, since it examines dataset Y values for scaling and labeling. Unlike other Axis subclasses, this scale is not drawn at the edge of a Plotarea. Instead, a SpeedoAxis is a curve bounded by the Plotarea.



A SpeedoAxis examines all Dataset Y values to obtain axis extents, though the Speedo DataRepresentation uses only the first value of the first Dataset. SpeedoAxis labels may be inside or outside the axis. SpeedoAxis does not use logarithmic scaling.

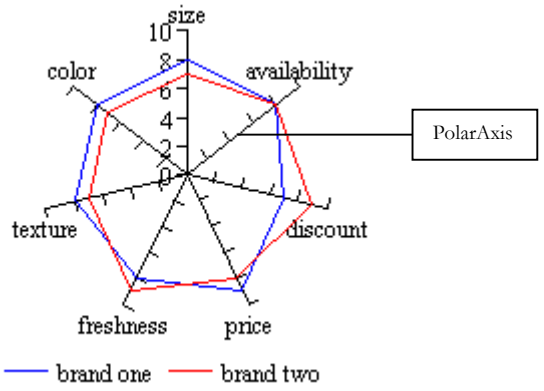
The shape and position of a SpeedoAxis is determined by the "speedoPosition" property:

- 0 - a nearly full circle, centered in the Plotarea
- 1 - half circle centered at bottom of Plotarea
- 2 - half circle centered at top of Plotarea
- 3 - half circle centered at left of Plotarea
- 4 - half circle centered at right of Plotarea
- 5 - quarter circle centered at ll of Plotarea

- 6 - quarter circle centered at lr of Plotarea
- 7 - quarter circle centered at ur of Plotarea
- 8 - quarter circle centered at ul of Plotarea

"com.ve.kavachart.chart.PolarAxis"

A PolarAxis draws a spoked axis for displaying Datasets in Polar DataRepresentations. Like SpeedoAxis, PolarAxis is used only for "Y" data. Polar uses Y values to determine the shape of a dataset's line or polygon.



A Polar DataRepresentation has one vertex for each Y observation. A PolarAxis provides a radial scaling framework for this data. PolarAxis can draw any number of spokes, but the default is equal to the number of observations in the first Dataset. This creates a "Kiviat" chart, such as the one displayed above.

Unlike other Axis classes, PolarAxis does not draw at the edges of its Plotarea. Instead, it draws at the center of a region bounded by its Plotarea.

PolarAxis properties include:

- manualSpoking – by default, this axis draws one spoke for each observation (or Datum) in the first Dataset. Set "manualSpoking" to false if you have a large Dataset to create a conventional Polar coordinate chart.
- numSpokes – when "manualSpoking" is set to "false" set "numSpokes" to determine how many auxiliary axis scales should be displayed.

Unlike other Axis classes, PolarAxis has multiple titles, one for each axis spoke. Define these titles with the "addLabels" and "replaceLabels" methods inherited from Axis. PolarAxis ignores title strings defined with the "setTitleString" method.

Other Interesting Axis Subclasses

There are several Axis subclasses that are not used in standard KavaChart charts, but that are very useful for minor customizations. These classes were

developed by Visual Engineering, but appeal to a subset of our users. The Developers Kit provides these in source code form in the "com.ve.kavachart.parts" package. Since the source code is available, they're a good source of information for building your own Axis subclasses. They're also easy to modify directly for specific requirements.

"com.ve.kavachart.parts.CullingLabelAxis"

Like LabelAxis, CullingLabelAxis draws an axis with user defined labels. LabelAxis defaults to one label per observation. With a large Dataset, this can lead to a very crowded Axis, with overlapping and unreadable labels.

CullingLabelAxis solves this problem by measuring label lengths and output size. Overlapping labels are eliminated.

Install a CullingLabelAxis like this:

```
Axis Ax = new com.ve.kavachart.parts.CullingLabelAxis();
Ax.setBarScaling(true);
Ax.setSide(Axis.BOTTOM);
Chart.setXAxis(Ax);
```

com.ve.kavachart.parts.SymbolAxis

Users with very large numbers in their data sometimes need special number formatting for legibility. SymbolAxis demonstrates one way to accomplish this, by overriding the "getLabel" method. Axis labels in the billions look like this: 3B, 4B, etc. Labels in the millions look like this: 1M, 2M, etc.

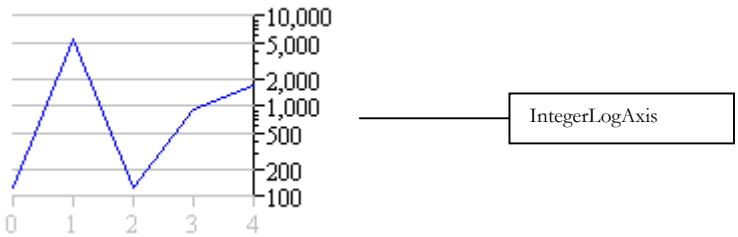
SymbolFormat provides another way to accomplish the same task. This class extends java.text.Format to do the same format transformations.

Install a SymbolFormat like this:

```
Chart.getYAxis().setFormat(new SymbolFormat());
```

"com.ve.kavachart.parts.IntegerLogAxis"

This Axis subclass uses the default logarithmic scaling in KavaChart version 3.2 and earlier. This scaling generally creates axes that start and end at integer powers of. For example, numbers that range from 12 to 750 would range from 10 to 1,000, with an intuitive labeling increment between. Since log scaling is best used to visually reduce extremely large movements in data, this is generally appropriate.

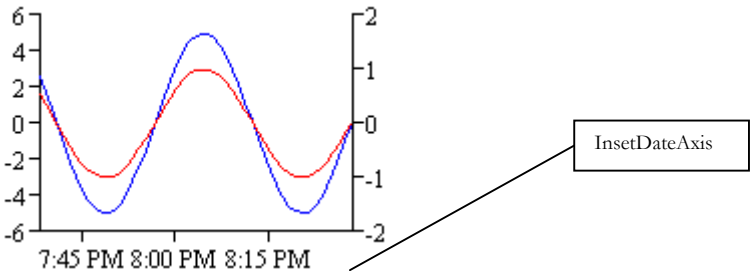


This type of scaling is appropriate for users that want to compare a trend when the data covers an extremely large range.

InsetDateAxis

KavaChart's DateAxis automatically creates breaks at natural time increments, such as multiples of minutes, month endings, etc. However, data doesn't usually begin and end at natural time increments. To create these increments, DateAxis adds additional space at the axis ends to reach a natural increment, and then distributes labels relatively evenly across the axis. Unfortunately, this may result in some blank space at the beginning and ending of the axis.

"com.ve.kavachart.parts.InsetDateAxis" extends DateAxis to remove the blank space. This axis may not have a label at the beginning and ending of the axis, and the remaining labels may be shifted to one side or the other, to label the data at natural increments.

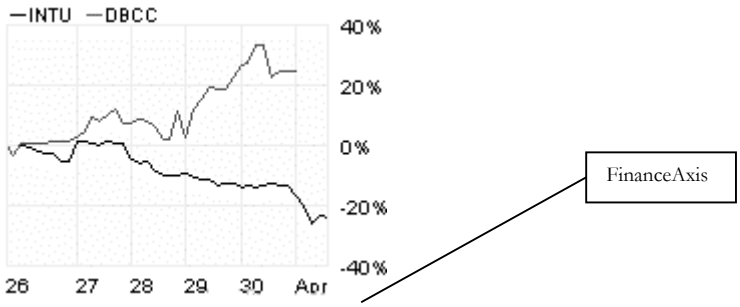


Because of the way InsetDateAxis works, it must be autoscaled. You can set the scaling model (MINUTE_SCALING, MONTH_SCALING, etc.), but other changes won't work correctly.

FinanceAxis

Finance applications present some interesting axis scaling challenges. Financial trading data is inherently discontinuous. Traders don't work 24 hour days, and exchanges are closed on weekends and holidays. However, users want to visualize data as if it were happening in one continuous stream, and users expect each bar (or line segment, candlestick, etc.) to occupy the same amount of space along the axis, even for partial trading periods. Furthermore, holidays and trading hours are different for different financial exchanges.

"com.ve.kavachart.parts.FinanceAxis" is designed to address these problems. This chart is designed to work with DataRepresentations that support "unitScaling". That means that the internal geometric scale for this axis ranges from 0 to the total number of observations. Candlesticks, Bars, HLOCs, and other DataRepresentations should use "unitScaling" when combined with a FinanceAxis. A special Line class, "com.ve.kavachart.parts.FinanceLine", is provided for this purpose.



Despite the internal scaling, FinanceAxis displays labels based on the X values in its Datasets. Like DateAxis, FinanceAxis assumes that these X values represent milliseconds since the Java Date epoch (Jan 1, 1970, UTC). Based on these values, FinanceAxis constructs a set of labels that describes the Dataset X values.

The actual labels depend on your JVM's locale settings, the size of the chart (in pixels), and the context of the data. For example, a very small chart may display labels every hour, while in a larger version of the same chart FinanceAxis may display labels every 10 minutes.

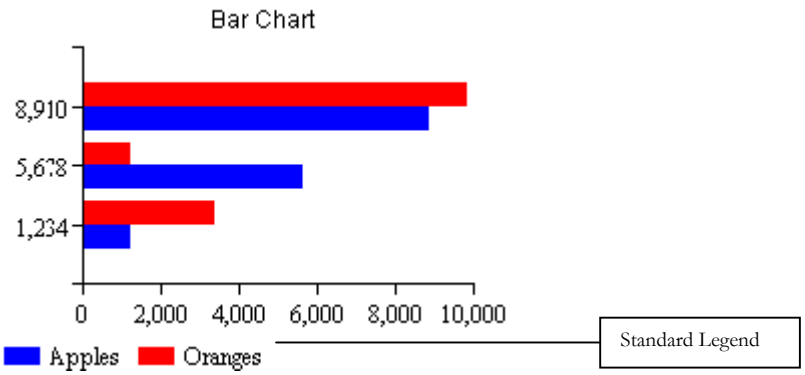
FinanceAxis also marks significant time transitions (such as a new week or a new year) with a secondary label format, similar to DateAxis. Axes that don't have any major time period transitions may display additional information on the initial label.

Because of the high level of automation in FinanceAxis, this axis can't be manually scaled. If you require that level of specification, use LabelAxis. Consult your API documentation for more information about the properties available to FinanceAxis.

Also, consult Appendix B for more information about pre-constructed chart objects that use FinanceAxis.

Legend and its Subclasses

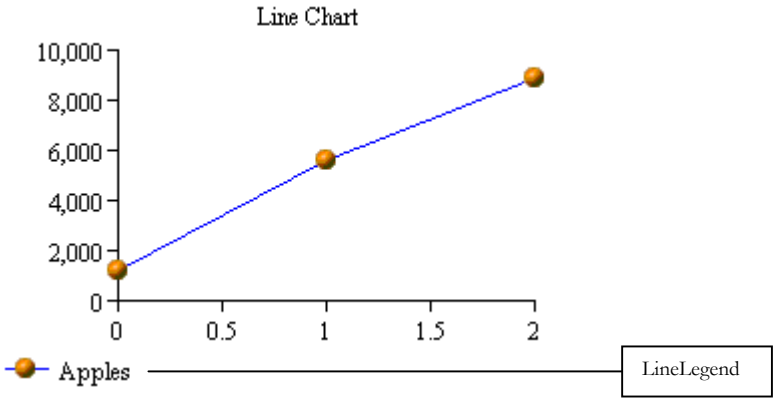
A Legend is a chart component that describes your chart's data. KavaChart's core components include 3 Legend classes, implementing "com.ve.kavachart.chart.LegendInterface". The base legend superclass, "com.ve.kavachart.chart.Legend" draws one rectangle plus a label for each dataset, using the Dataset name and Dataset Gc fillColor properties.



The basic legend is used for most types of bar charts, and area charts. It uses the dataset color and name to draw a rectangular icon and label for each dataset.

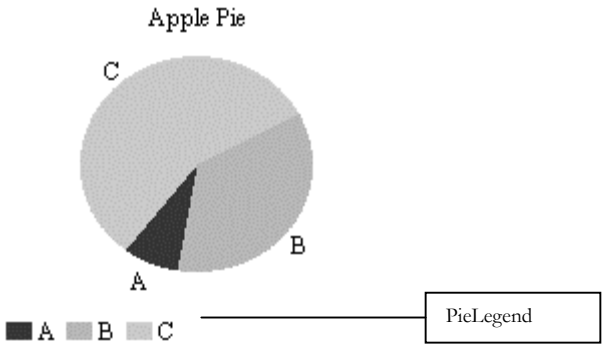
LineLegend

A LineLegend draws a line with (optionally) a marker or image for each dataset in the chart. The dataset color and marker are used for the line icon, and dataset names are used for the legend labels. This type of legend is used for all line chart types.



PieLegend

A PieLegend draws an icon and label for every data element in the first dataset. The individual Datum color and label are used to color the rectangular icon and obtain the legend labels. This legend type is also used for the bar and column charts when the "individualColors" property is set to "true".

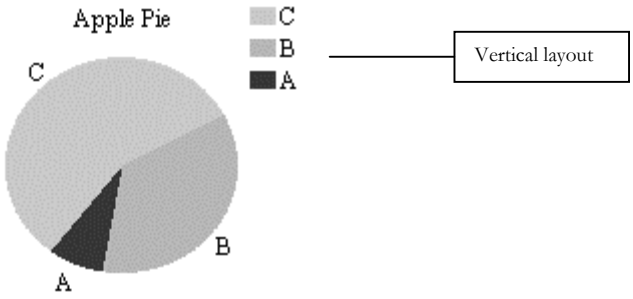


Legend Properties

A legend has a position, a background color, a layout, a label font, and a label color. You can also toggle the visibility of a legend's background. The horizontal and vertical size of the legend icon is adjustable, as is the gap between the icon and label text.

Legend Layout

Legend entries can be listed horizontally or vertically. A legend position is defined by the legend's lower left hand corner, in percentages of the overall chart (a position of .5, .5 would place the lower left hand corner of the legend at the center of the entire chart). If a legend won't fit horizontally or vertically in the prescribed location, it wraps to another column or line. For example, if a legend has a horizontal layout, but the number of entries would cause the legend to grow beyond the right hand side of the chart, the legend draws a second or third line of icons and labels



By adjusting the legend's label font, the icon size, and the icon gap, you can adjust the overall size of the legend.

Legend classes include a "recalculateSize()" method that calculates the upper right corner of the legend based on current legend property settings. To use this method to place your legend at the upper right hand corner of your chart:

```
Legend legend = (Legend)(chart.getLegend());
legend.setLlX(0.0); //put the legend's ll corner at the origin
legend.setLlY(0.0);
legend.recalculateSize(myGraphicsClass); //calculate legend size
```

```
legend.setLlX(1. - legend.getUrX()); //move it to the ur corner
legend.setLlY(1. - legend.getUrY());
```

DataTransform

This component translates data (axis coordinate) values into pixel values, and pixel values into axis coordinates. The internal transforms automatically update based on chart size and axis scaling.

For example, let's say you want to draw something on a specific date on a chart that uses DateAxis. Since DateAxis uses millisecond values for its internal geometries, the first step is to obtain that value, and create a "com.ve.kavachart.chart.Datum" object, like this:

```
Date specialDay = new Date(whenever);
Datum ageDat = new Datum((double)specialDay.getTime(), 0.0, null);
```

Now we create a DataTransform instance, and use it to obtain the "java.awt.Point" corresponding to that observation:

```
DataTransform dataTransform = new DataTransform(chart);
Point specialDayLocation = dataTransform.datumToPoint(ageDat);
```

Now we can use "specialDayLocation" to draw a marker, a note, or whatever else our application requires. If you use the yAxisStart instead of 0.0, this will be coincident with the X axis.

Chapter

6

Core API: A Programmer's Cookbook

This chapter describes how to use KavaChart's core "com.ve.kavachart.chart" API to build custom charting components. You may also want to become familiar with KavaChart Beans, which use a more self-contained architecture. If you're building a web-based application, you should consider extending one of KavaChart's applets or server objects.

This chapter assumes you have experience using the Java programming language. Use this chapter as a companion to the javadoc API documentation.

The Basics

KavaChart's core charting library consists of a collection of charting components, described in the previous chapter, and a set of charts. The charts extend the abstract class "com.ve.kavachart.chart.Chart", and implement the interface "com.ve.kavachart.chart.ChartInterface". Each chart acts as a container for other charting components (e.g. Axis, Plotarea, etc.) from the "com.ve.kavachart.chart" package.

Charts are organized into individual chart types, such as "BarChart", and "LineChart". Although "com.ve.kavachart.chart" does include some combination charts, there are some significant benefits to this "single chart" approach. It's easy to build custom charts by recombining KavaChart components in new ways. You can also extend KavaChart's core charting components and install new behaviors (e.g. customized axis labeling, new line drawing variants, etc.) quickly and easily. This approach also yields applet and application jar files with a minimum footprint size.

Unlike some other Java charting libraries, KavaChart's chart objects are not Panel or JPanel subclasses. Instead, they provide a framework that can draw to any "Graphics" class. This lets you use KavaChart to generate images, PostScript, SVG, or any other output your application requires. You can use KavaChart with a Java2d environment and Swing, or with JDK 1.1 and AWT.

This approach also means you need to build some sort of application context to do anything useful with KavaChart. Our first example, then, will show a Frame that draws a chart.

Example 1 - A Simple Chart Program

The first example extends java.awt.Frame to build a very simple, non-interactive chart.

1. Create a standalone program that draws a Frame. The following program draws an empty window (Frame) with a size of 400 pixels by 400 pixels:

```
import java.awt.*;

public class myChart extends Frame {
   public static void main(String[] args){
      myChart app = new myChart();
      app.setSize(400, 400);
      app.setVisible(true);
   }
}
```

2. Create a KavaChart class for use with this program, and set the size to be the same as the Frame. While we're creating a BarChart for this example, you could just as easily use any other subclasses of com.ve.kavachart.chart.Chart, such as LineChart or PieChart. This superclass contains most of the useful methods for building charts and for accessing the various constituent chart elements.

```
import com.ve.kavachart.chart.*;
BarChart chart = new BarChart("sample");
chart.resize(400, 400);
```

3. Override the Frame's paint method to do something (slightly) interesting. While our chart isn't complete yet, this is where we'll want to do the drawing. Frame (and most other graphics classes) use the paint method whenever they need to draw themselves.

```
public void paint(Graphics g){
   chart.paint(g);
}
```

4. Add some data. With this step, we have a fully functional program for drawing a BarChart. The following source code is the complete myChart.java file:

```
import java.awt.*;
import com.ve.kavachart.chart.*;

public class myChart extends Frame {

   BarChart chart = new BarChart("sample");

   public static void main(String[] args){
      double[] myData = {123, 432, 223, 345};
      myChart app = new myChart();
      app.setSize(400, 400);
      app.setVisible(true);
      app.chart.resize(400, 400);
      app.chart.addDataset("important numbers", myData);
   }

   public void paint(Graphics g){
      chart.paint(g);
   }
}
```

5. Compile myChart.java. If you're using the JDK from JavaSoft:

- a) make sure the directory containing "com" is part of your CLASSPATH environment (e.g. "set CLASSPATH=C:\KavaChart" for Windows)

- b) execute the command "javac myChart.java".

If you're using some other development environment, such as JBuilder, Eclipse, or NetBeans the steps are similar: make sure com.ve.kavachart.* is in your CLASSPATH and execute compilation. Alternatively, you may want to import the entire com.ve directory tree into your interactive development environment.

6. Run your program. If you're using JavaSoft's JDK, this would be done with the following command: "java myChart". Note that if your program is part of a Java package, such as com.ve.kavachart.standalone, you'll have to run "java com.ve.kavachart.standalone.myChart".

Example 2 - A Reusable Chart Panel

Our next example uses a very similar approach, but creates a reusable subclass of Panel that can be added to any other Java component, with only a few minor changes.

1. Create a standalone program that will:

   - create a java.awt.Panel subclass class

   - place that Panel into a Frame

```
import java.awt.*;
```

```
public class ChartPanel extends Panel {
        public static void main(String[] args){
                Frame myFrame = new Frame("Title Bar");
                ChartPanel myPanel = new ChartPanel();
                myFrame.add(myPanel);
                myFrame.setSize(400, 400);
                myFrame.setVisible(true);
        }
}
```

2. Add a method to initialize a simple chart. A few things to note:

    - We intentionally place the chart definition outside the intialization method to ensure the chart object is available elsewhere, such as from paint() for drawing, or from an external class, to change various chart properties.

    - we use the initialization method to change a couple of chart properties as well as to install some sample data. The initialization method sets a direct property, the chart's LegendVisibility, which is available for all Chart subclasses. We also set a title string for this chart by acquiring the chart's Background (chart.getBackground()) and then setting the Background's title string. This hierarchical approach to managing chart properties can seem confusing at first, but ultimately provides a well organized and modular mechanism for managing a lengthy list of properties.

```
public BarChart chart = new BarChart("sample");

protected void initChart(){
   double[] sampleData = {123, 432, 345, 432};
   chart.addDataset("Banana Sales", sampleData);
   chart.setLegendVisible(true);
   chart.getBackground().setTitleString("Sample Chart");
}
```

3. Override the Panel's paint method to draw a chart whenever this panel paints itself. We'll add a new twist this time, though. First, we'll override the update method, to eliminate update's default clearing and the resultant flicker. Second, we'll use a special KavaChart paint method. This method creates an off-screen image buffer (using Panel's createImage method) that will actually take the chart drawing. By drawing to the off-screen image buffer, and then drawing the buffer in a single operation, we'll have smooth, flicker free drawing.

```
   public void update(Graphics g){
      paint(g);
   }

   public void paint(Graphics g){
       chart.paint(this, g);
```

```
}
```

Note:

Use "Chart.paint(Component, Graphics)" to let KavaChart handle double buffered drawing and automatic resizing. This method also automatically installs the necessary code (RotateString, image buffer) to display rotated text labels.

Our total ChartPanel class can be used by any other graphics class with a few simple lines:

```
ChartPanel myChartPanel = new ChartPanel();
myChartPanel.initChart();
add(myChartPanel);
myChartPanel.chart.getBackground().setTitleString("new title");
```

The finished ChartPanel code looks like this:

```
import java.awt.*;
import com.ve.kavachart.chart.*;

public class ChartPanel extends Panel {
   public BarChart chart = new BarChart("sample");

   protected void initChart(){
      double[] sampleData = {123, 432, 345, 432};
      chart.addDataset("Banana Sales", sampleData);
      chart.setLegendVisible(true);
      chart.getBackground().setTitleString("Sample Chart");
   }

   public void update(Graphics g){
      paint(g);
   }

   public void paint(Graphics g){
      chart.paint(this, g);
   }

   public static void main(String[] args){
      Frame myFrame = new Frame("Title Bar");
      ChartPanel myPanel = new ChartPanel();
      myPanel.initChart();
      myFrame.add(myPanel);
      myFrame.setSize(400, 400);
      myFrame.setVisible(true);
   }
}
```

Example 3 – a Reusable Swing Panel

Our next example is nearly identical, except that it uses lightweight Swing components. KavaChart's flexible architecture can be easily used with either Swing or AWT environments.

1. Create a standalone program that will

   - create an extension to JPanel
   - place that JPanel into a Jframe

```
public class ChartPanel extends JPanel {
        public static void main(String[] args){
                JFrame myFrame = new JFrame("Title Bar");
                ChartPanel myPanel = new ChartPanel();
                myFrame.getContentPane().add(myPanel);
                myFrame.setSize(400, 400);
                myFrame.setVisible(true);
        }
}
```

2. Add a method to initialize a simple chart. This is identical to the previous AWT example.

```
public BarChart chart = new BarChart("sample");

   protected void initChart(){
      double[] sampleData = {123, 432, 345, 432};
      chart.addDataset("Banana Sales", sampleData);
      chart.setLegendVisible(true);
      chart.getBackground().setTitleString("Sample Chart");
   }
```

3. Override the JPanel's paint and update methods to draw a chart whenever this panel paints itself. Again, this is identical to the AWT modifications:

```
public void update(Graphics g){
   paint(g);
}

public void paint(Graphics g){
        chart.paint(this, g);
}
```

Like the AWT version, the lightweight ChartPanel class can be used by any other graphics class with a few simple lines:

```
ChartPanel myChartPanel = new ChartPanel();
myChartPanel.initChart();
getContentPane().add(myChartPanel);
myChartPanel.chart.getBackground().setTitleString("a title");
```

The finished ChartPanel looks like this:

```
import javax.swing.*;
import com.ve.kavachart.chart.*;

public class ChartPanel extends JPanel {
   public BarChart chart = new BarChart("sample");

   protected void initChart(){
      double[] sampleData = {123, 432, 345, 432};
      chart.addDataset("Banana Sales", sampleData);
      chart.setLegendVisible(true);
      chart.getBackground().setTitleString("Sample Chart");
   }

   public void update(Graphics g){
      paint(g);
   }

   public void paint(Graphics g){
      chart.paint(this, g);
   }

   public static void main(String[] args){
      JFrame myFrame = new JFrame("Title Bar");
      ChartPanel myPanel = new ChartPanel();
      myPanel.initChart();
      myFrame.getContentPane().add(myPanel);
      myFrame.setSize(400, 400);
      myFrame.setVisible(true);
   }
}
```

Charts As Containers

KavaChart Chart classes provide a container and organizational framework for classes from the "com.ve.kavachart.chart" and "com.ve.kavachart.parts" package. While charts do include some convenience methods, most of the interesting properties and functionality are contained in the internal classes.

Obtaining Internal Classes

Use KavaChart's accessor methods, like "getPlotarea" to obtain these internal classes. For example, to change the chart's title string:

```
Chart.getBackground().setTitleString("new Title");
```

To change an axis scale:

```
Chart.getYAxis().setAxisEnd(1000.);
```

Charts are also organized in a way that lets you quickly modify groups of attributes:

```
Dataset[] data = chart.getDatasets();
```

```
for(int i=0;i<data.length;i++){
      data[i].setName("Dataset " + i);
}
```

Or, to modify axis attributes:

```
SetupAxisAppearance(chart.getYAxis());
SetupAxisAppearance(chart.getXAxis());
```

Charts include accessors to obtain an interface or base class for objects that have an extensive hierarchy, such as Axis or DataRepresentation. For example, if you are using a BarChart, this will return a "Bar" class:

```
DataRepresentation bar = chart.getDataRepresentation();
```

This is adequate for general DataRepresentation properties, such as "labelsOn", or "labelFormat", etc. However, to set properties unique to the "Bar" subclass of DataRepresentation, we need a class of type "Bar":

```
Bar bar = (Bar)chart.getDataRepresentation();
bar.setClusterWidth(0.65);
```

Similarly, if we want to set properties for a specific type of Axis, we need to know what type of axis is in use, and cast the return value to that type of axis:

```
DateAxis ax = (DateAxis)chart.getXAxis();
ax.setScalingType(DateAxis.MINUTE_SCALING);
```

Modifying Chart Properties

Most chart properties are fairly easy to find. For example, if you want to set the label font for an axis, you'll find a "labelFont" property available to all Axis classes.

Classes that draw labels generally have font and label color properties. Classes that display numbers generally have a label format property, and have a label precision property to modify the default format. Classes that might rotate a label for better spacing generally have a label angle property. For example, we can add labels to the top of a Bar Chart's bars like this:

```
Bar bar = (Bar)chart.getDataRepresentation();
bar.setLabelsOn(true);
bar.setLabelColor(Color.red);
bar.setLabelAngle(90);
bar.setLabelFont(new Font("SanSerif", Font.BOLD, 20));
```

Note:

Because KavaChart classes always return objects by reference (instead of making copies of the objects), you don't need to use the "set" method to re-set the object into the chart. Changes to objects returned by "get" take effect immediately (upon repaint for visual properties).

Color properties are contained in "Gc" classes. Most charting components contain at least one "Gc" class. Some classes, such as Axis, have a Gc for each part of the Axis; gridGc, lineGc, tickGc, and so on.

This code uses "Gc" classes to modify our background color, plotarea color, and Y axis grid color:

```
Chart.getPlotarea().getGc().setFillColor(Color.lightGray);
Chart.getBackground().getGc().setFillColor(Color.black);
Chart.getYAxis().getGridGc().setLineColor(Color.white);
```

Tip:

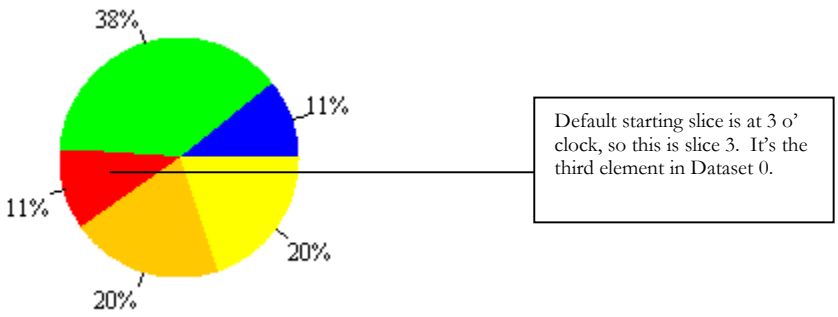
Whenever you need to change color, fill style, line style, or other graphical properties, look for the "getGc()" method in the API documentation.

Dataset Colors

If you're new to KavaChart, you might be surprised to find that DataRepresentation classes (such as Bar, Pie, etc.) do not have any Color properties. They don't even have "Gc" classes.

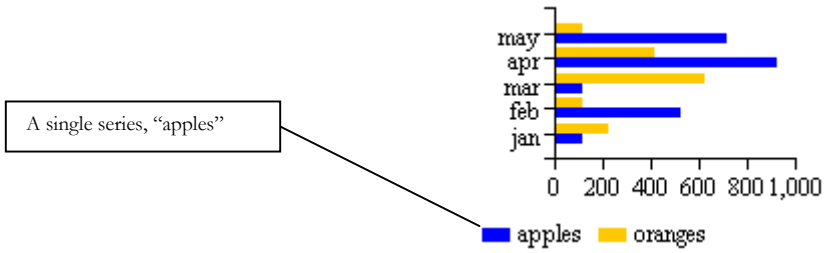
These colors are derived from the Dataset Gc or Datum Gc classes for the Data they represent. For example, to change the color of the third pie slice to red:

```
Datum slice = chart.getDatasets()[0].getDataElementAt(2);
slice.getGc().setFillColor(Color.red);
```



If a DataRepresentation displays a Dataset using a single color, such as most line and bar charts, the code looks like this:

```
Dataset series = chart.getDataset("apples");
series.getGc().setFillColor(Color.blue);
```



As you can see in the example above, the Dataset color is reflected in both the Legend and the Bar DataRepresentation. If we used the same Dataset classes with a Column, or a StackBar, or some other DataRepresentation, we would get the same colors, and the same Legend colors.

Data Management

KavaChart stores data in arrays of "com.ve.kavachart.chart.Dataset" classes. Each of these classes contains a Vector of "com.ve.kavachart.chart.Datum" classes, which contain the actual numeric observations and labels. A Dataset also contains a name and a Gc class to store colors, styles, and labeling information for the overall Dataset.

Most subclasses of "com.ve.kavachart.chart.Chart" use a single Dataset array. This array is shared by the chart's Axis classes, DataRepresentation class, and Legend.

More complicated Chart subclasses might use multiple Dataset arrays. For example, BarLineChart uses one Dataset array for the Bar DataRepresentation, and another Dataset array for the Line DataRepresentation. When the chart's "drawGraph" method is called, Dataset references are copied from the primary Dataset array into the Bar's array and the Line's array.

Note:

Chart components act on Dataset classes in their Dataset array. Changes to a Dataset, or to the Datum classes within the Dataset, will affect all chart components using that Dataset.

Dataset and Datum Classes

Dataset classes can be created in several ways. Charts include several "addDataset" methods that create and install Dataset objects based on arrays of numbers and text strings. For example:

```
double vals[] = {123.,345.,543.};
chart.addDataset("series one", vals);
```

This code is the equivalent of directly creating a Dataset class and installing it like this:

```
double vals[] = {123.,345.,543.};
Dataset dataset = new Dataset("series one", vals);
chart.addDataset(dataset);
```

The chart adds this Dataset class to its Dataset array, which is also used by the internal chart components. By obtaining this Dataset, you can change Dataset properties, such as colors, fill styles, and the series name. You can also change the Dataset's underlying Datum classes. To obtain the Dataset, request it by name:

```
Dataset thatSeries = chart.getDataset("series one");
```

Datasets are added sequentially to the Dataset array. If you know which series number you're working with, you can also obtain it like this:

```
Dataset thatSeries = chart.getDatasets()[0];
```

When we created our Dataset, a set of Datum classes was also created. Datum classes contain each observation's numeric values and label. These classes are added to a Dataset's "data" Vector. The Dataset constructor example above implicitly did this:

```
Datum d = new Datum(0., 123., null);
dataset. addDatum(d);
d = new Datum(1., 345., null);
dataset.addDatum(d);
d = new Datum(2., 543., null);
dataset.addDatum(d);
```

In this example, we used X values that weren't defined in our Dataset constructor or our Chart convenience method. Some DataRepresentations ignore X values entirely (Pie, unitScaled Bar and Column), but it's still a good idea to use a reasonable X value. This makes the Dataset portable to a new Chart container that may use X values, such as LineChart or LabelLineChart.

We can also add, change, or delete entries in an existing Dataset. For example, we can add a new observation to our exiting "series one" Dataset like this:

```
Dataset dataset = chart.getDataset("series one");
Datum d = new Datum(3., 654., chart.getGlobals());
dataset.getData().addElement(d);
```

Note that we added the chart's Globals class to the Datum constructor. A Globals class stores information about the chart's overall size, locale, default fonts, and so on. A valid Datum Globals isn't required by all charts, but some DataRepresentations need to know the chart's overall size at the Datum level, so it's good practice to do this. Calling "addDataset(Dataset)" will automatically set the Globals class for all Datasets and Datum classes to be the same as the chart. If we used the Dataset method "addDatum" instead of manipulating the data vector directly, the Dataset will automatically set the Globals class for us.

Example 4 - Using the Dataset's DataVector

From the discussion above, it should be clear that changes to items in a chart's Dataset array and individual Datum classes are reflected in the entire chart. What may not be as obvious, however, is that this provides a powerful paradigm for all sorts of chart manipulations. For example, we can create a chart that scrolls through a continuous data feed by adding Datum classes to the end of the data Vector and removing them from the beginning. Here's an extended version of our earlier ChartPanel example that does exactly that:

```
public class RunChart extends Panel implements Runnable {
   public BarChart chart = new BarChart("sample");
   static double[] newData = {123, 234, 345, 456};
   int i = 0;
   Thread myThread = null;

   public void start(){
               myThread = new Thread(this);
               myThread.start();
   }

   public void stop(){
               myThread = null;
   }
   public void run(){
               while(myThread!=null){
                   try{
                              Thread.sleep(500);
                         }catch(Exception ignored){
                         }
                         Dataset dataset = chart.getDatasets()[0];
                         Datum dat = new
             Datum(i+4,newData[i%newData.length],null);
                         dataset.getData().addElement(dat);
                         dataset.getData().removeElementAt(0);
                         repaint();
                         i++;
               }
```

```
            }
    protected void initChart(){
        chart.addDataset("Banana Sales", newData);
        chart.setLegendVisible(true);
        chart.getBackground().setTitleString("Sample Chart");
    }
    public void update(Graphics g){
        paint(g);
    }
    public void paint(Graphics g){
        chart.paint(this, g);
    }
    public static void main(String[] args){
        Frame myFrame = new Frame("Title Bar");
        RunChart myPanel = new RunChart();
        myPanel.initChart();
        myFrame.add(myPanel);
        myFrame.setSize(400, 400);
        myFrame.setVisible(true);
        myPanel.start();
    }
}
}
```

This code adds a Thread to refresh data once every 500 milliseconds. Our new data consists of a repetition of our original data, added one point at a time. The result is a chart that cycles through the data every half second.

Since this example doesn't use real data, we could accomplish the same thing by just manipulating the data Vector instead of creating new Datum classes, like this:

```
Vector data = chart.getDatasets()[0].getData();
Object datum = data.getElementAt(0);
data.removeElementAt(0);
data.addElement(datum);
```

The result is a continuous cycling of existing Datum objects. This works fine for our contrived example in a BarChart, but it would fail for a LineChart.

A simple Bar or Column chart ignores X values, and uses the position of Datum classes within the DataVector to arrange bars (this is called unitScaling). Line charts, on the other hand, use both X and Y values to draw lines.

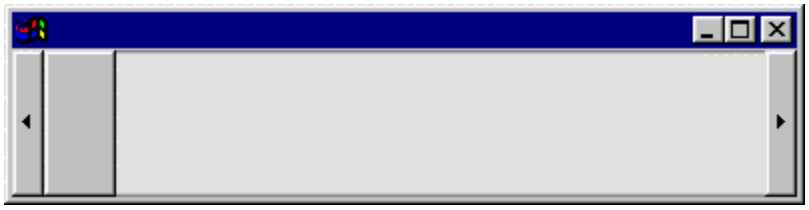
Example 5 – Managing Data with a Scrollbar

Our next example uses a scrollbar to manage our data vector. Scrollbars are usually integer-based, with a maximum, minimum, value, and visible amount. We'll create a chart that displays only the Datum classes indicated by the Scrollbar's position and handle size.

The first step is to create a Scrollbar subclass tied to a java.util.Vector. Here's our code:

```
import java.awt.*;
import java.awt.event.*;
import com.ve.kavachart.chart.*;
import java.util.*;

class DataScroller extends Scrollbar implements
      AdjustmentListener {
      Vector data;

public DataScroller() {
      super(Scrollbar.HORIZONTAL);
      addAdjustmentListener(this);
}
public void adjustmentValueChanged(AdjustmentEvent event){
      int start = getValue();
      int end = start + getVisibleAmount();
      System.out.println("start:" + start);
      System.out.println("end:" + end);
      System.out.println("start value:" +
      data.elementAt(start));
      System.out.println("end value:" + data.elementAt(end));
}
public static void main(String args[]) {
      Frame f = new Frame();
      f.setSize(400, 100);
      DataScroller scrollbar = new DataScroller();
      Vector v = new Vector();
      for(int i=0;i<100;i++){
            v.addElement(new Integer(i));
      }
      scrollbar.setData(v);

      f.add(scrollbar);
      f.setVisible(true);
}
public void setData(Vector vector) {
      data = vector;
      setMinimum(0);
      setMaximum(data.size()-1);
}
}
```

This code creates a Frame filled with a Scrollbar, like this:



As you move the scroll tab or press the buttons on the ends, the Java console displays values from the scrollbar's "data" vector, along with the scrollbar position. In this case, they're identical:

```
start: 10
end: 20
startValue: 10
endValue:20
start: 11
end: 21
startValue: 11
endValue: 21
```

The "main" method of this scrollbar loads up the data vector with Integer values. The "adjustmentValueChanged" method just displays those values. To make this scrollbar useful for a chart, we need to make a couple of modifications:

1. "setData" needs to make a copy of the entire Dataset data Vector.

2. "adjustmentValueChanged" needs to make sure the Dataset data Vector contains only the Datum classes indicated by the scrollbars position and visible amount. This is accomplished by moving data from the original copy into the chart's actual Dataset data Vector.

3. "adjustmentValueChanged" needs to repaint the chart.

Here is the modified source code:

```
import java.awt.*;
import java.awt.event.*;
import com.ve.kavachart.chart.*;
import java.util.*;

class DataScroller extends Scrollbar implements AdjustmentListener {
            Vector totalData;
Vector visibleData;
      Panel paintable;

public DataScroller() {
            super(Scrollbar.HORIZONTAL);
            addAdjustmentListener(this);
}
public void adjustmentValueChanged(AdjustmentEvent event){
            int start = getValue();
            int end = start + getVisibleAmount();
            visibleData.removeAllElements();
            for(int i=start;i<end;i++){
                  visibleData.addElement(totalData.elementAt(i));
            }
            if(paintable!=null)
                  paintable.repaint();
}
```

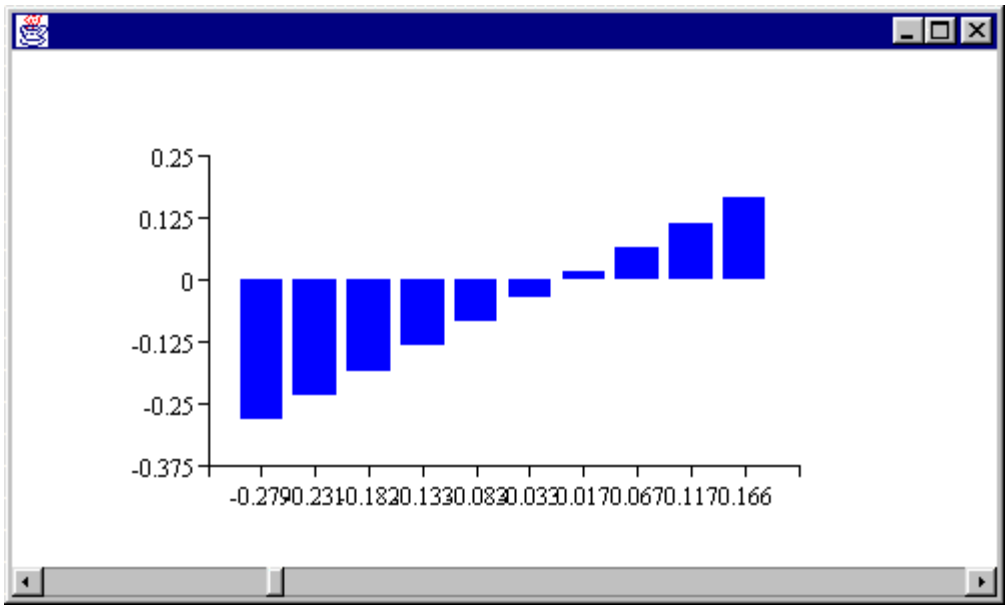
```
public void setData(Vector vector) {
            totalData = (Vector)vector.clone();
            visibleData = vector;

            setMinimum(0);
            setMaximum(totalData.size()-1);

            //set the current data window
            adjustmentValueChanged(null);
}
public void setPaintable(Panel p) {
            paintable = p;
}
}
```

Our modified code adds 2 new variables: "visibleData", and "paintable". "visibleData" is our chart's data Vector (remember that Java objects are passed by reference, so this is the same Vector used by the internal chart classes). "paintable" is a Panel class, which can be refreshed by calls to "repaint()". DataScroller is now ready to be connected to our earlier "ChartPanel" example.

Chart Panel will also need some very minor modifications to make sure we have enough data to scroll. We'll also modify the "main" method to connect ChartPanel to our DataScroller. Here's the new ChartPanel:

```
import java.awt.*;
import com.ve.kavachart.chart.*;
import java.util.*;

class ChartPanel extends Panel {
      Chart chart;
public ChartPanel() {
      chart = new BarChart();
      double[] data = new double[500];
      for(int i=0;i<data.length;i++){
            data[i] = Math.sin(i/20.);
      }
      chart.addDataset("sine wave", data);
}
public static void main(String args[]) {
      Frame f = new Frame();
      f.setSize(500, 300);
      ChartPanel chartPanel = new ChartPanel();

      DataScroller scrollbar = new DataScroller();
      scrollbar.setPaintable(chartPanel);

      Vector data =
chartPanel.chart.getDatasets()[0].getData();
      scrollbar.setData(data);

      f.setLayout(new BorderLayout());
      f.add(chartPanel, "Center");
      f.add(scrollbar, "South");
      f.setVisible(true);
}
public void paint(Graphics g) {
      chart.paint(this, g);
}
}
```

The new ChartPanel draws bars in a sine wave. The "main" method creates a Frame, sets a layout, and adds the ChartPanel and Scrollbar to it. The result is an animated chart that looks like this:



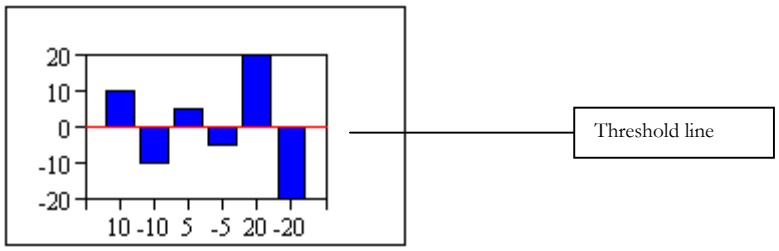
Movements along the scrollbar control what Datum classes are in the chart's data Vector. This controls what the chart's DataRepresentation (Bar) draws, what range the Y axis uses for scaling, and what labels are drawn along the X axis.

This version of DataScroller manages a single dataset, but you should see how to easily modify DataScroller to handle as many datasets as the chart contains. Also, the simplified version of DataScroller is designed to handle fixed data. You can extend it to handle live data.

Example 6 – Drawing with DataTransform

KavaChart is designed to make easy work of drawing additional graphics information on your charts, like logos or copyright notices. Applets and server beans provide a method that gives you a "java.awt.Graphics" class you can use for drawing in a conventional way.

Using KavaChart's DataTransform class, you can also draw information in axis coordinates. For example, suppose you want to draw a red horizontal line at exactly 0.0 on the Y axis, like this:



To do this, you need to find the location of zero on the Y axis. This can change depending on your data, since the Y axis doesn't always have the same scale. You'll also need to know other information that might change, like the left and right edges of the plot area.

DataTransform is key to doing this sort of activity, since it translates KavaChart Datum objects into pixel coordinates. Here's our applet subclass that adds a threshold line:

```
import java.awt.*;
import com.ve.kavachart.chart.*;
import com.ve.kavachart.applet.*;

public class ThresholdLineBarApplet extends columnApp {

  DataTransform dataTransform;
  Datum zeroDatum;

  public void drawMyStuff(Graphics g){
        //initialize if necessary
        if(dataTransform==null)
              dataTransform = new DataTransform((Chart)chart);
        if(zeroDatum==null)
              zeroDatum = new Datum(0.0, 0.0, null);

        //figure out where the left and right should be
        Dimension d = getSize();
        int plotLeft =
              (int)(d.width * chart.getPlotarea().getLlX());
        int plotRight =
              (int)(d.width * chart.getPlotarea().getUrX());
        Point p1 = dataTransform.datumToPoint(zeroDatum);

        g.setColor(Color.red);
        g.drawLine(plotLeft, p1.y, plotRight, p1.y);
  }
}
```

This code extends one of the standard KavaChart applets to implement the method "drawMyStuff". This method is available for both applets and server beans to add graphical overlays to your chart.

ThresholdLineBarApplet uses the protected "chart" variable from columnApp to initialize a DataTransform object. It also initializes a Datum object with both X and Y values set to zero. DataTransform uses that Datum class to construct a Point in pixel coordinates equivalent to the Datum.

DataTransform automatically adjusts as axis coordinates change, so we only need to construct a single class.

This applet retrieves the applet's Dimension and the internal chart's Plotarea to calculate the start and end X coordinates for this line. We could also use two Datum classes, and use the X axis start and end values to find starting and ending points for our line.

DataTransform does not work for Pie charts or Polar coordinate charts. For horizontal bar charts, DataTransform automatically swaps X and Y values, so that Datum X values are mapped into Point Y values.

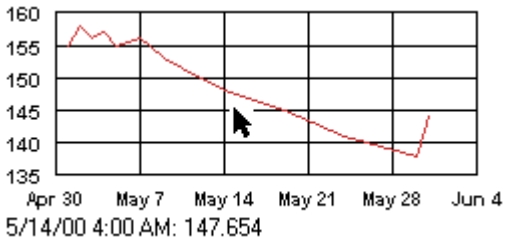
Note:

We can change this into the equivalent server imaging bean by simply changing "import com.ve.kavachart.applet.*" to "import com.ve.kavachart.servlet.*"

Example 7 – Using DataTransform for Mouse Interactions

You can also use DataTransform to translate pixel coordinates into KavaChart axis coordinates. The next example uses mouse events to obtain pixel coordinates. These points are then used to calculate what a data value would be at the mouse location, and display that information along with the chart.

The first step is to add mouse support to a charting class. We'll start with another applet, dateLineApp. The finished applet will look like this:



The display across the bottom updates as the cursor moves to new locations, displaying theoretical dates and values, based on the current axis system. Here's the code:

```
public class ReadoutDateLineApplet extends dateLineApp {
      DataTransform dataTransform;
      Font displayFont = new Font("Dialog", Font.PLAIN, 10);
      NumberFormat numberFormat = NumberFormat.getInstance();
      DateFormat dateFormat = DateFormat.getInstance();
/**
 * Method to display our current mouse location
 * @param p java.awt.Point
 */
public void displayMouseLocation(Point p) {
      if(dataTransform==null)
       dataTransform = new DataTransform((Chart)chart);
      Datum dat = dataTransform.pointToDatum(p);

      Graphics g = getGraphics();
      g.drawImage(chart.getImage(), 0, 0, this);

      g.setColor(Color.black);
      g.setFont(displayFont);
```

```
        Date now = new Date((long)dat.getX());
        String s1 = dateFormat.format(now) + ": ";
        String s2 = numberFormat.format(dat.getY());

        Dimension d = getSize();
        g.drawString(s1 + s2, 0, d.height - 5);
    }
    /**
     * overload the init method to add mouse support
     */
    public void init() {
        super.init();

        addMouseMotionListener(new MouseMotionAdapter()
         {
                public void mouseMoved(MouseEvent e){
                        displayMouseLocation(e.getPoint());
                }
         });
    }
    }
```

This class uses an anonymous inner class as a simple MouseMotionListener. A MouseEvent contains a Point representing the current location of the cursor. We send this point into the "displayMouseLocation" method, which does the real work of this class.

DataTransform translates the point into X and Y coordinates. Since this is a time oriented chart, we use the X value to create a "java.util.Date" class. NumberFormat and DateFormat classes create the display formats for this coordinate data.

Finally, we paint the chart's image buffer and then write our display string over the chart.

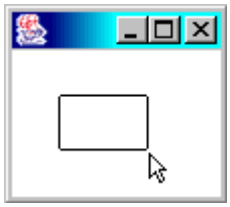
Example 8 – Using DataTransform and Mouse Interactions to Zoom a Chart

You can also use DataTransform to interactively zoom in to a subset of your chart's entire data. The first step will be to obtain a rectangular extent box around the data we want to see in more detail. Next, we'll set the chart's axis extents to match this data region and repaint.

This technique exploits a useful feature in KavaChart axes. Usually axes are either autoscaled or manually scaled. By default, autoscaled axes examine the Datasets to find the largest and smallest numbers. However, you can set an axis start or axis end value to override the Dataset values. By setting axis start and end values based on mouse input, the axes and DataRepresentation (bar, line, etc.) will behave as if only the mouse input values are important, creating a zoomed effect.

First, we'll create a "Zoomer" class that will draw a rubberband box based on mouse input. This class includes a "main" method for testing purposes.

Running this class results in a Frame that responds to mouse click-drag events like this:



Zoomer stores a Point when the mouse is pressed, and then draws lines in XOR mode when the mouse is dragged. When the mouse is released, the Panel is repainted. Here's the code:

```
import java.awt.*;
import java.awt.event.*;
import com.ve.kavachart.chart.*;

class Zoomer implements MouseListener, MouseMotionListener {
       Panel panel;
       ChartInterface chart;
       Point mousePressed;
       Point mouseReleased;
       int lastX, lastY;
       DataTransform dataTransform;

public Zoomer(Panel panel, ChartInterface chart) {
       this.panel = panel;
       this.chart = chart;

       panel.addMouseMotionListener(this);
       panel.addMouseListener(this);
}

/**
 * test method
 */
public static void main(String args[]) {
       Frame f = new Frame();
       f.setSize(100, 100);
       Panel p = new Panel();
       f.add(p);

       Zoomer z = new Zoomer(p, null);
       f.setVisible(true);
}
public void mouseClicked(MouseEvent e) {
}
public void mouseDragged(MouseEvent e) {
       Point p = e.getPoint();
       Graphics g = panel.getGraphics();
       g.setXORMode(Color.white);

       g.drawLine(mousePressed.x, mousePressed.y, mousePressed.x,
       lastY);
```

```
        g.drawLine(mousePressed.x, lastY, lastX, lastY);
        g.drawLine(lastX, lastY, lastX, mousePressed.y);
        g.drawLine(mousePressed.x, mousePressed.y, lastX,
        mousePressed.y);

        g.drawLine(mousePressed.x, mousePressed.y, mousePressed.x, p.y);
        g.drawLine(mousePressed.x, p.y, p.x, p.y);
        g.drawLine(p.x, p.y, p.x, mousePressed.y);
        g.drawLine(mousePressed.x, mousePressed.y, p.x, mousePressed.y);

        lastX = p.x;
        lastY = p.y;

}
public void mouseEntered(MouseEvent e) {
}
public void mouseExited(MouseEvent e) {
}
public void mouseMoved(MouseEvent e) {
}
public void mousePressed(MouseEvent e) {
        if(e.isMetaDown()){
              resetChart();
              return;
        }
        mousePressed = e.getPoint();
        lastX = mousePressed.x;
        lastY = mousePressed.y;
}
public void mouseReleased(MouseEvent e) {
        mouseReleased = e.getPoint();
        if(chart!=null)
              zoomChart();
        panel.repaint();
}

/*
 * resets the chart axes to autoscaling
 */
protected void resetChart() {
        if(chart==null)
              return;
        AxisInterface axis;
        axis = chart.getXAxis();
        axis.setAxisStart(Axis.AUTO_SCALE);
        axis.setAxisEnd(Axis.AUTO_SCALE);

        axis = chart.getYAxis();
        axis.setAxisStart(Axis.AUTO_SCALE);
        axis.setAxisEnd(Axis.AUTO_SCALE);

        //make sure mouseReleased doesn't re-zoom
        mousePressed = null;
}

/*
 * zooms the chart by setting the axes to the mouse locations
```

```
 */
protected void zoomChart() {
       if(mousePressed==null) //set by resetChart
             return;
       if(dataTransform==null)
             dataTransform = new DataTransform((Chart)chart);

       Datum mouseDownDatum = dataTransform.pointToDatum(mousePressed);
       Datum mouseUpDatum = dataTransform.pointToDatum(mouseReleased);

       double axisStart;
       double axisEnd;
       AxisInterface axis;

       axisStart = Math.min(mouseDownDatum.getX(),
       mouseUpDatum.getX());
       axisEnd = Math.max(mouseDownDatum.getX(), mouseUpDatum.getX());
       axis = chart.getXAxis();
       axis.setAxisStart(axisStart);
       axis.setAxisEnd(axisEnd);

       axisStart = Math.min(mouseDownDatum.getY(),
       mouseUpDatum.getY());
       axisEnd = Math.max(mouseDownDatum.getY(), mouseUpDatum.getY());
       axis = chart.getYAxis();
       axis.setAxisStart(axisStart);
       axis.setAxisEnd(axisEnd);
}
}
```

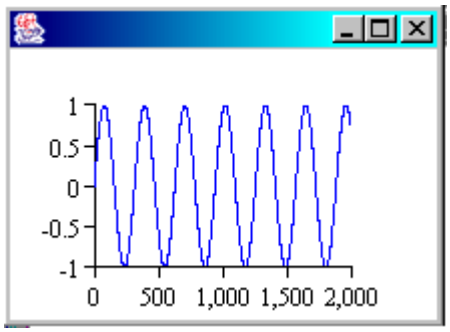
This code is designed to attach a listener to any Panel class. If we create a Panel subclass that also includes a chart, we can create a DataTransform, and obtain the axis coordinates from our zooming action. The "zoomChart" method gets these axis coordinates, and uses them to set the axis start and end values. The "resetChart" method, triggered when the user presses the right mouse button or meta-key, resets the axis scale values to "AUTOSCALE".

Since Zoomer is generalized to work with any Panel, it can be used with an Applet as well as within a larger application. Let's apply it to a version of our simple ChartPanel example above. Here's the code:

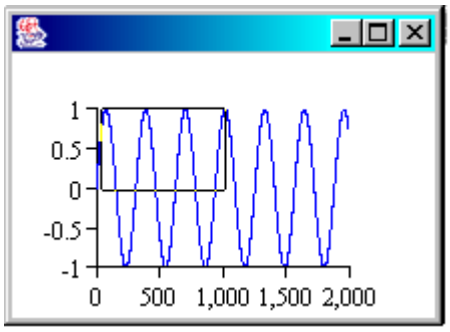
```
import com.ve.kavachart.chart.*;
import java.awt.*;

public class SimpleChart extends java.awt.Panel {
             public Chart                  chart;
             public SimpleChart(){
                   chart = new LineChart("Test Chart");
                   double values[] = new double[2000];
                   for(int i=0;i<values.length;i++){
                         values[i] = Math.sin(i/50.);
                   }
                   chart.addDataset("data", values);

                   //important for zooming:
                   ((LineChart)chart).getLine().setClip(true);
```

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

          SimpleChart p = new SimpleChart();

          //attach the Zoomer class
          Zoomer z = new Zoomer(p, p.chart);

          Frame f = new Frame();
          f.add(p);
          f.setSize(500, 300);
          f.setVisible(true);
    }
    public void paint(Graphics g){
          chart.paint(this, g);
    }
}
```

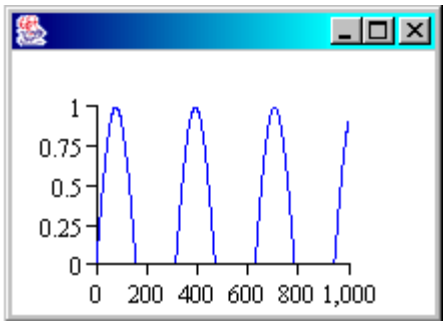
This simple class draws a sine wave with 2000 points. With a typical display resolution, you can't see all 2000 points, even if the chart is displayed over the entire screen. Here's the chart:



Zooming the chart like this:



Sets the Y axis to start at 0 and end at 1.0. The X axis will start at 0 and end at 1,000. Here's the result when the mouse button is released:



One important thing to notice is that the plot line stops at the axis extents. Plot line clipping is disabled by default. To turn it on, the "SimpleChart" constructor contains this line:

```
//important for zooming:
((LineChart)chart).getLine().setClip(true);
```

Most bar and line type DataRepresentations include a "Clip" property.

DisplayList Management

All KavaChart Chart subclasses include a DisplayList class for identifying chart objects based on pixel locations. A DisplayList contains a table of chart objects with their corresponding geometric information. Public methods let you all the chart objects at that location.

One Point may contain more than one chart component. For example, if you click on a Bar within a BarChart, the bar is the top item, but it is drawn above a Plotarea class, which is drawn above a Background class. DisplayList's "contains" method puts all these classes into your Vector, sorted by drawing order.

A single object may also be associated with more than one charting object. In our Bar example above, the Bar is actually associated with 3 items: the Bar DataRepresentation, the Dataset associated with that Bar series, and the Datum class used to draw that specific Bar. All these objects are added to the Vector filled by the "contains" method.

Example 9 – Identifying Objects from Mouse Events

In this example, we'll use the chart's DisplayList to get information about specific points on the chart. We'll start with a simple AWT Panel, although this same technique can be used with an Applet, a JPanel, or any other display component.

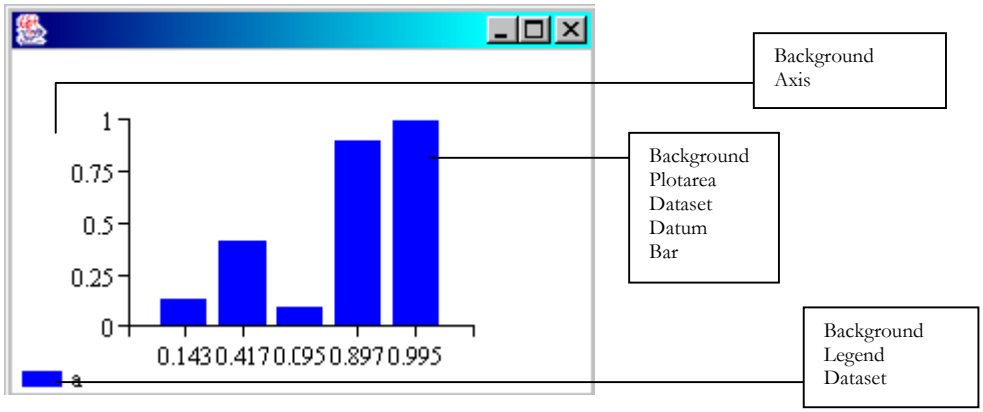
Our Panel subclass, called "Identifier", will draw a simple Bar Chart with a few random numbers. We turn on the legend to see what mouse clicks in that region will yield. We also enable the Chart's DisplayList with this line:

```
chart.setUseDisplayList(true);
```

"Identifier" also installs a MouseAdapter so we can capture mouse events. When the mouse button is pressed, the "tellMe" method is called, which queries the chart's DisplayList to find out what objects are beneath the cursor. These objects are printed in back-to-front order. Here's the code:

```
import com.ve.kavachart.chart.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;

public class Identifier extends Panel {
      Chart             chart;

      Identifier(){
            chart = new BarChart();
            double x[] = new double[5];
            double y[] = new double[5];

            for (int i = 0; i < 5; i++) {
              x[i] = (double) i;
              y[i] = Math.random();
            }
            chart.addDataset("a", x, y);
            chart.setLegendVisible(true);
            chart.setUseDisplayList(true);
            addMouseListener(new MouseAdapter(){
                  public void mousePressed(MouseEvent e){
                        tellMe(e.getPoint());
                        }
                  });
      }

      protected void tellMe(Point p){
            Vector list = new Vector();
            chart.getDisplayList().contains(p, list);

            for(int i=0;i<list.size();i++){
                  Object myObj = list.elementAt(i);
                  System.out.println(myObj);
            }
            System.out.println();
      }
      public static void main (String[] args) {
            Frame f = new Frame();
            f.setSize(300, 200);
            f.add(new Identifier());
            f.setVisible(true);
      }
      public void paint(Graphics g){
            chart.paint(this, g);
      }
}
```

Running this code displays a Frame containing a bar chart. Your "System.out" will display output in response to mouse clicks like this:



Notice that a mouse click on a bar or legend icon identifies both the "Bar" object and the "Dataset" object. "Bar" clicks also identify the "Datum" object associated with that particular bar.

DisplayList's access methods are public, so you can supplement the list of chart objects with your own objects. For example, you could add a logo to your charts, and then identify mouse clicks on the logo by identifying the logo object when processing DisplayList information.

Example 10 – Simple Tooltip Labels

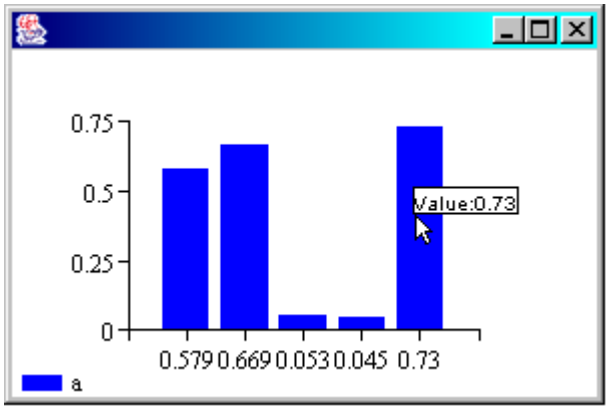
Our next example is very similar to Example 9, but we'll use the chart's DisplayList to create a simple tooltip label. This is similar to the "dwellLabel" facilities in KavaChart applets.

This class installs a MouseMotionAdapter to look for mouse motion events instead of mouse button presses. When the cursor passes over a "Datum" class, we will display the Y values from that class at the cursor location. Here's the code:

```
import com.ve.kavachart.chart.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.text.*;

public class ToolTipper extends Panel {
       Chart              chart;
       NumberFormat       formatter;
       Font               font = new Font("Dialog", Font.PLAIN, 10);

       ToolTipper(){
             chart = new BarChart();
             double x[] = new double[5];
             double y[] = new double[5];

             for (int i = 0; i < 5; i++) {
               x[i] = (double) i;
```

```
            y[i] = Math.random();
        }
        chart.addDataset("a", x, y);
        chart.setLegendVisible(true);
        chart.setUseDisplayList(true);
        addMouseMotionListener(new MouseMotionAdapter(){
            public void mouseMoved(MouseEvent e){
                doToolTip(e.getPoint());
                }
            });
        formatter = NumberFormat.getInstance();
    }

    protected void doToolTip(Point p){
        Vector list = new Vector();
        chart.getDisplayList().contains(p, list);

        Graphics g = getGraphics();
        for(int i=0;i<list.size();i++){
            Object myObj = list.elementAt(i);
            if(myObj instanceof Datum){
                g.drawImage(chart.getImage(), 0, 0, this);

                Datum dat = (Datum)myObj;
                String s = "Value:" +
                    formatter.format(dat.getY());

                g.setFont(font);
                FontMetrics fm = g.getFontMetrics();
                int width = fm.stringWidth(s);
                int height = fm.getMaxAscent() + 2;
                p.y = p.y - height; //move away from cursor
                g.setColor(Color.white);
                g.fillRect(p.x, p.y, width, height);
                g.setColor(Color.black);
                g.drawRect(p.x, p.y, width, height);
                g.drawString(s, p.x, p.y + height - 1);
                return;
            }

        }
        g.drawImage(chart.getImage(), 0, 0, this);
    }
    public static void main (String[] args) {
        Frame f = new Frame();
        f.setSize(300, 200);
        f.add(new ToolTipper());
        f.setVisible(true);
    }
    public void paint(Graphics g){
        chart.paint(this, g);
    }
}
```

The drawing logic in "doToolTip" uses a formatter and default font to measures the length of the string. The code then draws a rectangle beneath the label and outlines the rectangle. Finally, the string is drawn above the entire rectangle. The result is a chart that looks like this when the cursor passes over a bar:



For good display performance, this class paints the chart's image buffer instead of redrawing the entire chart when processing mouse events.

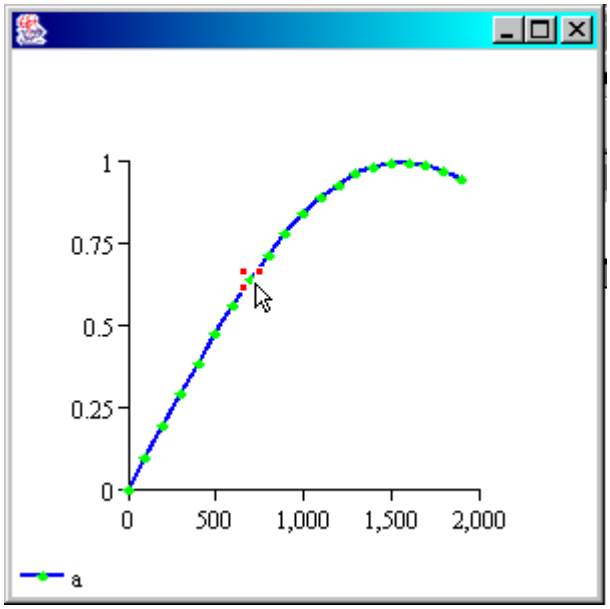
Note:

If you're creating your own chart classes that use DisplayList, it's very important to call the DisplayList "clear()" method before drawing the chart. If the DisplayList is not cleared, you will incur a substantial memory leak. Subclasses of "com.ve.kavachart.chart.Chart" can do this by calling "super.drawGraph()".

Example 11 – Highlighting Selected Objects

KavaChart's core charting library includes a class named "Highlighter" that can be used as a companion class to DisplayList. Highlighter retrieves a set of points outlining a particular chart object. When used in conjunction with DisplayList, Highlighter makes quick work of creating an interactive charting application.

In this example, we modify Example 10 to create a line chart with markers at each point. Instead of listening for mouse motion events we listen for mouse presses. When the user presses a mouse button, we send the mouse cursor location to the chart's DisplayList. If the point contains a Datum object, we send that object into a Highlighter class to obtain a set of boundary points for the object. Using those points, we draw red highlight boxes around the location, like this:



Mouse presses that don't contain any Datum objects are ignored. Here's the code for "HighlightObject":

```
import com.ve.kavachart.chart.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.text.*;

public class HighlightObject extends Panel {
        Chart              chart;
        Highlighter              highlighter;

        HighlightObject(){
              chart = new LineChart();
              double x[] = new double[20];
              double y[] = new double[20];

              for (int i = 0; i < x.length; i++) {
                x[i] = i * 100.;
                y[i] = Math.sin(i/10.);
              }
              chart.addDataset("a", x, y);

              //set style
              Gc seriesStyle = chart.getDataset("a").getGc();
              seriesStyle.setMarkerStyle(Gc.MK_DIAMOND);
              seriesStyle.setMarkerSize(7);
              seriesStyle.setLineWidth(2);
              seriesStyle.setFillColor(Color.green);

              chart.setLegendVisible(true);
              chart.setUseDisplayList(true);
              addMouseListener(new MouseAdapter(){
                    public void mousePressed(MouseEvent e){
```

```
                        doHighlight(e.getPoint());
                        }
                  });
            highlighter = new Highlighter(chart.getGlobals());
      }

      protected void doHighlight(Point p){
            Vector list = new Vector();
            chart.getDisplayList().contains(p, list);

            Graphics g = getGraphics();
            chart.paint(this,g);
            for(int i=0;i<list.size();i++){
                  Object myObj = list.elementAt(i);
                  if(myObj instanceof Datum){
                        g.drawImage(chart.getImage(), 0, 0,
this);
                        Point[] points =
highlighter.getHighlightPoints(myObj);
                        for(int j=0;j<points.length;j++){
                              drawPoint(g, points[j]);
                        }
                        return;
                  }

            }
      }

      private void drawPoint(Graphics g, Point p){
            Dimension d = getSize();
            p.y = d.height - p.y;
            g.setColor(Color.red);
            g.fillRect(p.x - 2, p.y - 2, 4, 4);
            g.setColor(Color.white);
            g.drawRect(p.x - 2, p.y - 2, 4, 4);
      }

      public static void main (String[] args) {
            Frame f = new Frame();
            f.setSize(300, 300);
            f.add(new HighlightObject());
            f.setVisible(true);
      }
      public void paint(Graphics g){
            chart.paint(this, g);
      }
      public void update(Graphics g){
            paint(g);
      }
}
```

Like the preceding examples, we created this sample by extending a "java.awt.Panel" class. The same technique works for Canvases, JPanels, applets, or other GUI components as well. You can also use this technique to

highlight objects other than "Datum" classes. For example, you might want to highlight an entire Dataset, or a Legend.

We could also extend this example to modify individual data items. For example, we might present a dialog box when a user double-clicks on a Datum object. The dialog box could permit the user to modify the Datum's x, y, or lable values. Repainting the chart would show the modified chart data.

Extending Chart Components

Most KavaChart core library components are designed for easy extension. This lets you customize the behavior of basic components like "Line" or "Bar" and combine these into your own "Chart" subclasses, creating complex new behaviors.

You will find pre-made examples of several extended chart components in the "com.ve.kavachart.parts" package. This section describes how to build your own extensions.

Before building extended chart components, it's important to understand what each part of a chart does, so you can determine the most appropriate extensions. For example, if you want to add something that affects the visual output of every series, you probably want to modify the DataRepresentation class. If you want to store some information that might vary for every observation, you probably want to extend the Datum class. If each series needs additional information, create an extended Dataset class. And so on.

Example 12 – An Extended Datum class

You may want to store information with your chart beyond the simple values, label, and style information included with KavaChart's "Datum" class. In this example, we're going to build a customized Datum class that will let us determine whether some extraordinary event has occurred. We'll use that information in the next few examples to create a chart with more information than a basic line chart.

In this example, we'll create a Datum class to identify an extraordinary event, like a network failure, or a stock split. In addition, we'll store a string that describes the extraordinary event. Our class looks like this:

```
import com.ve.kavachart.chart.*;

public class ExtraordinaryDatum extends Datum {
	String description = null;

	public ExtraordinaryDatum(double x, double y, Globals g){
		super(x, y, g);
	}

	public void setDescription(String s){
		description = s;
	}
```

```
        public String getDescription(){
              return description;
        }
 }
```

This Datum subclass will work like a conventional Datum for purposes of drawing lines, bars, etc. In the next example, we'll create a "Line" subclass that will use this extended Datum differently.

ExtraordinaryDatum isn't created automatically with the Chart and Dataset convenience methods, so we'll need to create it directly. Here's an example that turns the fifth data element into an ExtraordinaryDatum:

```
chart.addDataset("simple data", xArr, yArr);
Dataset d = chart.getDataset("simple data");
Datum oldDat = d.getDataElementAt(4);
ExtraordinaryDatum newDat = new ExtraordinaryDatum(
      oldDat.getX(), oldDat.getY(), oldDat.getGlobals());
d.getData().setElementAt(newDat, 4);
```

Alternatively, you might want to just create an ExtraordinaryDatum whenever you encounter an extraordinary event in your data stream, like this:

```
Dataset series = new Dataset();
for(int i=0;i<info.size();i++){
        if(info.element(i).isSpecial()){
              ExtraordinaryDatum d =
                    new ExtraordinaryDatum(info.x, info.y, null);
              series.getData().addElement(d);
        }
        else{
              Datum d = new Datum(info.x, info.y, null);
              series.getData().addElement(d);
        }
}
chart.addDataset(series);
```

Note that we could have used all ExtraordinaryDatum classes, and added a boolean class variable that tells us whether the Datum subclass is special. It's just as easy to only use the special Datum when we need it, however, and it creates more readable code. When we use this class, we can use logic like this:

```
Datum d = dataset.getDataElementAt(i);
if(d instanceof ExtraordinaryDatum){
      //do extraordinary stuff
}
else{
      //do regular stuff
}
```

Example 13 – Creating a Custom DataRepresentatio

Our next example creates a subclass of the basic "com.ve.kavachart.chart.Line" class. This example will draw a line that changes color when it reaches beyond a specified threshold. This gives a quick visual cue that something has exceeded a preset value.

If we have several datasets, we probably want to vary the over-threshold color on a dataset-by-dataset basis. One way to accomplish this is to create a custom Dataset subclass that stores an additional color. However, every Dataset stores both fill and line colors in its Gc class. If we're not planning to make threshold bars and threshold areas, we can just use the basic dataset Gc's line color for regular lines, and the fill color for the over-threshold lines.

The first step is to create a Line subclass that behaves the sames as "Line", but stores a threshold value. We create get and set accessor methods for thresholdValue. By setting thresholdValue to "Double.Nan", you can also clear the threshold value.

We also create our own "draw" method that overrides the "draw" method in the Line superclass. To start with, this class will simply draw the same thing as a Line. Here's the starting code:

```
import java.awt.*;
import com.ve.kavachart.chart.*;

public class ThresholdLine extends Line {

	Double thresholdValue = null;

	public void draw(Graphics g){
		super.draw(g);
	}

	public void setThreshold(double val){
		if(val == Double.NaN)
			thresholdValue = null;
		else
			thresholdValue = new Double(val);
	}
	public double getThreshold(){
		if(thresholdValue==null){
			return Double.NaN;
		}
		return thresholdValue.doubleValue();
	}
}
```

Since this is a subclass of DataRepresentation and Line, we can change "LineChart" to use this class like this:

```
Chart = new LineChart();
Chart.setDataRepresentation(new ThresholdLine());
```

This isn't very interesting, though, because it does the same thing as "Line". Let's add drawing logic to make this line draw in two colors. First we'll draw the entire line in the line color. Our draw method already does this with a "super.draw(g)".

Next, we need to redraw the line in the Dataset fill color. We also want to clip the line at the axis position of our thresholdValue, if it exists. This will prevent the threshold color from overwriting the existing normal color.

To do this we need to create a "com.ve.kavachart.chart.Transform" class. This class is used by just about every chart to translate data values (axis values, actually) into pixel values. This method builds a Transform class that translates the axis coordinates into the Plotarea space:

```
protected Transform buildTransform(){
      double xStart = xAxis.getAxisStart();
      double xEnd = xAxis.getAxisEnd();
      double yStart = yAxis.getAxisStart();
      double yEnd = yAxis.getAxisEnd();

      int plotRight = (int)(plotarea.getUrX() * globals.getMaxX());
      int plotLeft = (int)(plotarea.getLlX() * globals.getMaxX());
      int plotTop = (int)(plotarea.getUrY() * globals.getMaxY());
      int plotBase = (int)(plotarea.getLlY() * globals.getMaxY());

      return new Transform(xStart, yStart, xEnd, yEnd,
            new Point(plotLeft, plotBase),
            new Point(plotRight, plotTop));
}
```

This method is simpler than it appears at first. It finds the start and end of the axes, and then finds the pixel rectangle that these axes describe (the Plotarea). The method just creates a new Transform class based on this information, which will do the work of translating all our Datum values into pixel drawing space.

With a Transform in place, the work of drawing the data lines is very simple. Here's our method:

```
protected void doThresholdLine(Graphics g, int series){
      double[] xarr = datasets[series].getXValues();
      double[] yarr = datasets[series].getYValues();
      Point[] points = dataTransform.pointList(xarr,yarr);
      Gc gc = datasets[series].getGc();
      Color saveColor = gc.getLineColor();
      gc.setLineColor(gc.getFillColor());
      gc.drawPolyline(g, points);
      gc.setLineColor(saveColor);
}
```

We extract arrays of X and Y values from the Dataset class, and translate the values into Points, using our Transform class. We then switch the FillColor and

LineColor properties on our Dataset Gc class. Using this Gc class, draw a Polyline with the transformed points, and then switch the colors back.

Our final step creates a clipping rectangle to make sure the threshold line only draws in a region above the threshold value. This code is similar to the code that created our Transform:

```
if(thresholdValue==null)
          return; //nothing more to do
dataTransform = buildTransform();
g = g.create();
double startX = xAxis.getAxisStart();
double endX = xAxis.getAxisEnd();
Point rightPoint = dataTransform.point(startX,
          thresholdValue.doubleValue());
Point leftPoint = dataTransform.point(endX,
          thresholdValue.doubleValue());
g.clipRect(rightPoint.x, 0, leftPoint.x, globals.getMaxY() -
          leftPoint.y);
```

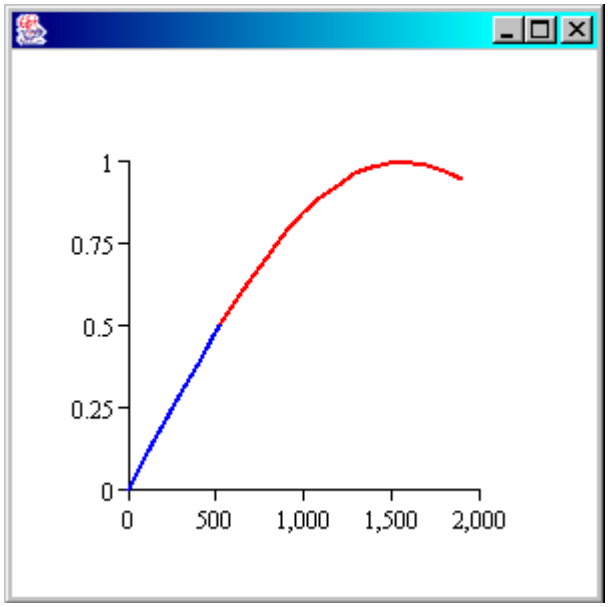
It's important to note that in creating our Graphics clip rectangle, we subtract the Y values from the size of our panel (gloabals.getMaxY()). This changes the coordinate system from KavaChart's default (0,0 at lower left) to AWT's default (0,0 at upper left).

We'll add a constructor with a threshold value to simplify things, and here's our completed class:

```
import java.awt.*;
import com.ve.kavachart.chart.*;

public class ThresholdLine extends Line {

          Double thresholdValue = null;
          Transform dataTransform;

          public ThresholdLine(){
            super();
          }
          public ThresholdLine(double value){
            super();
            thresholdValue = new Double(value);
          }

          public void draw(Graphics g){
            super.draw(g);
            if(thresholdValue==null)
                 return; //nothing more to do
            dataTransform = buildTransform();
            g = g.create();
            double startX = xAxis.getAxisStart();
            double endX = xAxis.getAxisEnd();
            Point rightPoint = dataTransform.point(startX,
```

```
        thresholdValue.doubleValue());
        Point leftPoint = dataTransform.point(endX,
        thresholdValue.doubleValue());
        g.clipRect(rightPoint.x, 0, leftPoint.x,
        globals.getMaxY() - leftPoint.y);

        int i=0;
        while(datasets[i]!=null){
            doThresholdLine(g, i);
            i++;
        }
    }

    public void setThreshold(double val){
        if(val == Double.NaN)
            thresholdValue = null;
        else
            thresholdValue = new Double(val);
    }
    public double getThreshold(){
        if(thresholdValue==null){
            return Double.NaN;
        }
        return thresholdValue.doubleValue();
    }

    protected Transform buildTransform(){
        double xStart = xAxis.getAxisStart();
        double xEnd = xAxis.getAxisEnd();
        double yStart = yAxis.getAxisStart();
        double yEnd = yAxis.getAxisEnd();

        int plotRight = (int)(plotarea.getUrX() *
            globals.getMaxX());
        int plotLeft = (int)(plotarea.getLlX() *
            globals.getMaxX());
        int plotTop = (int)(plotarea.getUrY() *
            globals.getMaxY());
        int plotBase = (int)(plotarea.getLlY() *
            globals.getMaxY());

        return new Transform(xStart, yStart, xEnd, yEnd,
            new Point(plotLeft, plotBase),
            new Point(plotRight, plotTop));
    }

    protected void doThresholdLine(Graphics g, int series){
        double[] xarr = datasets[series].getXValues();
        double[] yarr = datasets[series].getYValues();
        Point[] points = dataTransform.pointList(xarr,yarr);
        Gc gc = datasets[series].getGc();
        Color saveColor = gc.getLineColor();
        gc.setLineColor(gc.getFillColor());
        gc.drawPolyline(g, points);
        gc.setLineColor(saveColor);
    }
}
```

We'll add this new DataRepresentation to a simplified version of example 11, to create a chart that looks like this:



Here's the new chart panel Java code:

```
import com.ve.kavachart.chart.*;
import java.awt.*;

public class CustomLinePanel extends Panel {
           LineChart         chart;

           CustomLinePanel(){
                 chart = new LineChart();
                 double x[] = new double[20];
                 double y[] = new double[20];

                 for (int i = 0; i < x.length; i++) {
                   x[i] = i * 100.;
                   y[i] = Math.sin(i/10.);
                 }
                 chart.addDataset("a", x, y);

                 chart.setLine(new ThresholdLine(0.5));

                 //set style - lines are blue, threshold is red
                 Gc seriesStyle = chart.getDataset("a").getGc();
                 seriesStyle.setLineWidth(2);
                 seriesStyle.setFillColor(Color.red);

           }

           public static void main (String[] args) {
                 Frame f = new Frame();
                 f.setSize(300, 300);
                 f.add(new CustomLinePanel());
```

```
            f.setVisible(true);
        }
        public void paint(Graphics g){
            chart.paint(this, g);
        }
        public void update(Graphics g){
            paint(g);
        }
}
```

Our chart panel sets the threshold value to 0.5, and the Dataset's Gc fillColor to red, so all values above 0.5 are red.

Now let's modify ThresholdLine to add support for handling the "ExtraordinaryDatum" class we defined in Example 12. Let's say we want to circle data flagged as extraordinary, and add a label that describes why. We'll add a method to ThresholdLine to do this work:

```
protected void doExtraordinaryIndicators(Graphics g, int series){
        Point circleSize = new Point(20,20);
        Dataset d = datasets[series];
        for(int i=0;i<d.getData().size();i++){
           Datum dat = d.getDataElementAt(i);
           if(dat instanceof ExtraordinaryDatum){
                Point p = dataTransform.point(dat.getX(),
        dat.getY());
                d.getGc().drawArc(g, p, circleSize, 0, 360);
                g.setColor(d.getLabelColor());
                ExtraordinaryDatum ed = (ExtraordinaryDatum)dat;
                d.getGc().drawString(g, p.x, p.y,
        ed.getDescription());
           }
        }
}
```

This method is added to ThresholdLine's "draw", like this:

```
           i = 0;
           while(datasets[i]!=null){
                doExtraordinaryIndicators(g,i);
                i++;
           }
```

Note:

When possible, use KavaChart's Gc class to draw strings, lines, and filled objects. The Gc class automatically adjusts for KavaChart's lower-left origin, and takes advantage of features in Java2D (when available), such as antialiasing, line styles, etc. Also, Gc's drawing methods are generally more friendly to charting functions. In our ThresholdLine, for example, we use Gc's fillArc,

which is centered. Java's Graphics fillArc method aligns to the upper right corner.

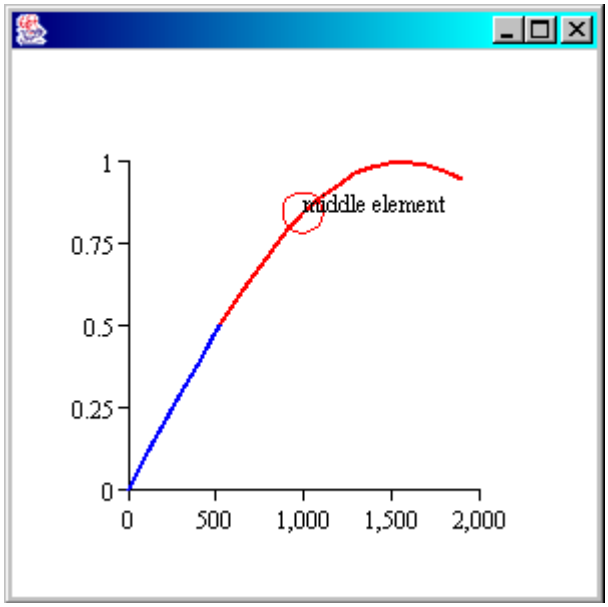
Finally, we'll modify our CustomLinePanel's constructor to create an ExtraordinaryDatum in the middle of our generated data, like this:

```
CustomLinePanel(){
            chart = new LineChart();
            Globals g = chart.getGlobals();

            Dataset series = new Dataset();
            for (int i = 0; i < 20; i++) {
              double x = i * 100.;
              double y = Math.sin(i/10.);
              if(i==10){
                    ExtraordinaryDatum dat = new ExtraordinaryDatum(x,
            y, g);
                    dat.setDescription("middle element");
                    series.getData().addElement(dat);
              }
              else{
                    Datum dat = new Datum(x, y, g);
                    series.getData().addElement(dat);
              }
            }
            chart.addDataset(series);

            chart.setLine(new ThresholdLine(0.5));

            //set style
            Gc seriesStyle = series.getGc();
            seriesStyle.setLineWidth(2);
            seriesStyle.setFillColor(Color.red);
            seriesStyle.setLineColor(Color.blue);

}
```

This chart generates the same data as the previous version, but it doesn't use the convenience methods in Chart to do this. Instead a Dataset class is created, and individual Datum classes are created for each observation. We flagged observation 10 as extraordinary, even though it resides in a Vector with other Datum classes.

ThresholdLine recognizes the ExtraordinaryDatum, and draws a circle along with the ExtraordinaryDatum's description text.

This code results in a chart that looks like this:


1
0.75
0.5
0.25
0
0
500
1,000
1,500
2,000
middle element

Creating Custom Chart Classes

KavaChart charts are containers that organize groups of cooperative components. As we've discussed in previous examples, you can change a wide range of chart attributes and behaviors by setting properties on internal chart components or by creating component subclasses.

KavaChart also makes it easy to create your own chart containers that recombine the charting components in new ways. For example, you could build a chart that contains multiple internal charts. You could build a chart with any mixture of DataRepresentation classes. You could build a chart with multiple axes, or plot areas, or a chart that shares some axes and distributes other axes among datasets.

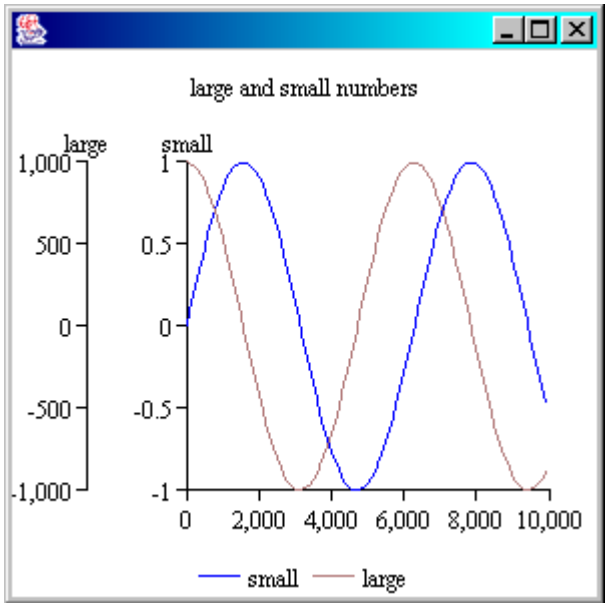
By using the "com.ve.kavachart.chart.Chart" superclass or the "com.ve.kavachart.chart.ChartInterface" classes to build custom charts, you can also use KavaChart's existing applet and servlet mechanisms to re-use existing tested code. For example, if you implement "ChartInterface", you can easily create a server bean that has built-in support for multiple image types, a server image cache, internal and external property management, and client-side tooltip labels.

This section discusses how to build a custom chart class, and then demonstrates how to put that chart into the existing applet and servlet framework.

Example 14 – A Custom Line Chart with Multiple Axes

With our next example, we'll create a chart that has one Y axis for each dataset, and a shared X axis. This type of chart is very useful for displaying related data that has disparate scaling. For example, suppose we want to chart megawatts of electricity, and Celsius outdoor temperature on the same chart. Unless each dataset uses a separate axis, we won't be able to see whether temperature affects electricity consumption or not.

Our finished chart will look similar to this:



In this sample, the "large" series ranges from –1,000 to +1,000. The "small" series ranges from –1 to 1. Both series are clearly sinusoidal waves. If both waves were on the same axis, the "small" series would appear to be a nearly straight line.

This chart's X axis shares both datasets. Each Y axis scales only a single dataset, however. All the plot lines appear on the same Plotarea. All datasets appear in the same legend.

Our first step is to create an organizational class to contain all the parts that will be separate for each dataset. This class will need to contain a Y axis, a Line class, and a Dataset.

Since Line and Axis require a Plotarea class, our helper class will need to know about the chart's Plotarea. Line also needs an X axis to obtain scaling information. Here's the code for our helper class, "AxisAndLineContainer":

```
import java.awt.*;
import com.ve.kavachart.chart.*;

public class AxisAndLineContainer {
        Axis yAxis;
        Line line;
        Dataset[] dataset = new Dataset[1];

        public AxisAndLineContainer(Dataset series,
              AxisInterface xAxis,
              Plotarea plotarea){
              dataset[0] = series;

              yAxis = new Axis(dataset, false, plotarea);
              line = new Line(dataset, xAxis, yAxis, plotarea);
        }
```

```
        public void draw(Graphics g, int offset){

              //label the axis with the dataset name
              yAxis.setTitleString(dataset[0].getName());
              g.translate(offset, 0);
              yAxis.draw(g);
              g.translate(-offset, 0);
              line.draw(g);
        }

}
```

Note that our class also includes a "draw" method. This method shifts the graphics by some offset, draws the Y axis, and then shifts back to draw the Line.

Note:

It's important to draw the axes before drawing the Line so the Line can obtain accurate scaling information.

Now we'll move on to creating a Chart subclass that uses our "AxisAndLineContainer". Chart subclasses typically override "initChart", "initAxes", and "drawGraph" methods.

If you need to use a locale other than the JVM's default, use the chart constructor that includes a Locale class. Setting locales after a chart has been initialized is problematic, since objects like axes, and DataRepresentation number formats have already been created with the default locale.

"InitChart" also installs a Background, a Plotarea, and calls "initDatasets". Datasets must be initialized before initializing dataset dependent classes, such as DataRepresentations, Legends, and Axes.

"InitAxes" creates axis instances for this chart subclass. In our case, we don't know in advance how many datasets the user will create, so we only initialize an X axis, using the protected variable "datasets", which was initialized by "initDatasets".

Most of the time you won't need to worry about these dependencies. You can simply extend a chart class that does the initialization for you, and then override only those items that need to be changed. For example, you can change the Y axis to a DateAxis like this:

```
public void initAxes(){
      super.initAxes();
      chart.setYAxis(new DateAxis();
}
```

Our multi-axis chart class Java code looks like this:

```
import java.awt.*;
import com.ve.kavachart.chart.*;

public class MultiAxisLineChart extends Chart {

	AxisAndLineContainer[] lines =
	  new AxisAndLineContainer[Chart.MAX_DATASETS];

	protected void initChart() {
	  initGlobals();
	  setPlotarea(new Plotarea());
	  setBackground(new Background());
	  initDatasets();
	  initAxes();
	  setLegend(new LineLegend());
	}

	protected void initAxes() {
	  setXAxis(new Axis());
	  xAxis.setSide(Axis.BOTTOM);
	}

	public void drawGraph(Graphics g){
	  super.drawGraph(); //clears DisplayList
	  background.draw(g);
	  plotarea.draw(g);
	  if(xAxisVisible)
		xAxis.draw(g);

	  int axisOffset = 0;
	  for(int i=0;i<datasets.length;i++){
		if(datasets[i]!=null){
		      if(lines[i]==null){
			    lines[i] = new
	AxisAndLineContainer(datasets[i], xAxis, plotarea);
		      }
		      lines[i].draw(g, axisOffset);
		      axisOffset -= 50; //shift the next axis over
		}
	  }
	  if(legendVisible)
		legend.draw(g);
	}
}
```

The "initChart" and "initAxes" methods are nearly identical to those in "LineChart", and we could have simply extended that chart here, overriding the "draw" method to add the new multiple axis functionality.

"DrawGraph" draws a Background, a Plotarea, and an X Axis. Then it loops through the protected "datasets" array, creating an "AxisAndLineContainer" for any dataset that doesn't yet have one.

Using the "draw" method found in "AxisAndLineContainer", the Y axis and Line is drawn for each Dataset in the datasets array. The chart increments an "offset" value for each line drawn, so that the Y axis is shifted progressively to the right.

With a small amount of additional coding, we could create multiple X axes, place Y axes on the left and right, create "stacks" of Y axes, and so on. The chart container works like any other KavaChart chart class. Here's the code for a Panel subclass that uses our chart class:

```
import com.ve.kavachart.chart.*;
import java.awt.*;

public class MultiAxisLinePanel extends Panel {
          MultiAxisLineChart            chart;

          MultiAxisLinePanel(){
                chart = new MultiAxisLineChart();
                Globals g = chart.getGlobals();

                Dataset series = new Dataset();
                series.setName("small");
                for (int i = 0; i < 100; i++) {
                  double x = i * 100.;
                  double y = Math.sin(i/10.);
                  Datum dat = new Datum(x, y, g);
                  series.getData().addElement(dat);
                }
                chart.addDataset(series);
                series.getGc().setLineColor(Color.blue);

                series = new Dataset();
                series.setName("large");
                for (int i = 0; i < 100; i++) {
                  double x = i * 100.;
                  double y = 1000. * Math.cos(i/10.);
                  Datum dat = new Datum(x, y, g);
                  series.getData().addElement(dat);
                }
                chart.addDataset(series);
                series.getGc().setLineColor(Color.yellow);

                //give enough room for multiple Y axes
                chart.getPlotarea().setLlX(0.30);
                chart.getPlotarea().setUrX(0.92);

                //turn a legend on
                chart.setLegendVisible(true);
                chart.getLegend().setLlX(0.30);

                //set an overall title
                Background b = chart.getBackground();
                b.setTitleString("large and small numbers");

          }
```

```
	public static void main (String[] args) {
		Frame f = new Frame();
		f.setSize(300, 300);
		f.add(new MultiAxisLinePanel());
		f.setVisible(true);
	}
	public void paint(Graphics g){
		chart.paint(this, g);
	}
	public void update(Graphics g){
		paint(g);
	}
}
```

This code is nearly the same as our previous Panel based examples, except that we used our new MultiAxisLineChart instead of BarChart or LineChart or some other basic charting class. We also added some code for a second Dataset with larger values than the first, to demonstrate how the chart handles disparate data.

Example 15 – Using a Custom Chart in an Applet

Now that we have a custom chart class, based on KavaChart's "chart" components, we might want to create an Applet that takes advantage of our chart. KavaChart's Applet framework makes this almost ridiculously simple. Here's the complete applet:

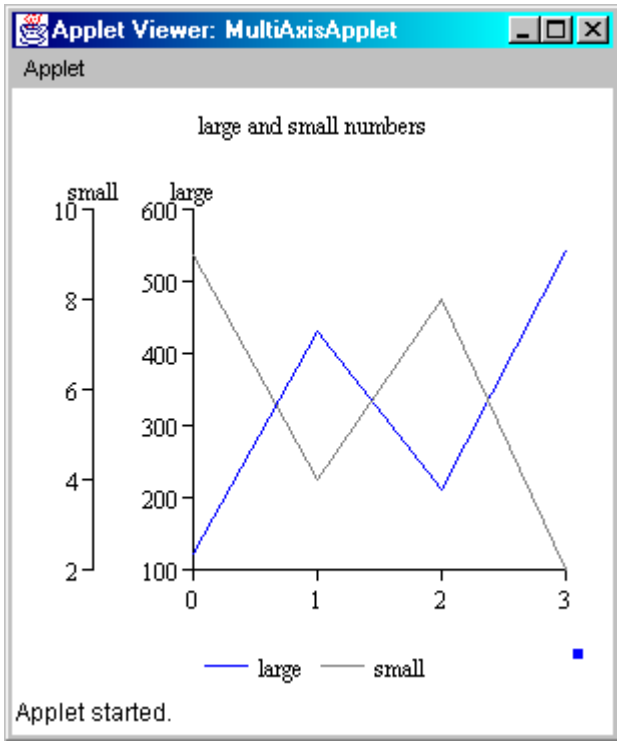
```
import java.applet.Applet;
import com.ve.kavachart.applet.*;

public class MultiAxisApplet extends ChartAppShell {
	public void init(){
		initLocale();
		chart = new MultiAxisLineChart();
		getOptions();
	}
}
```

This applet supports all the parameters of any other KavaChart applet. It also supports dwell labels, printing, live data updates, it works in Java 1.1 and Java 2 environments, and includes every other feature of any other KavaChart applet. To create a chart that mimics the example above (albeit with much less data) we can use this HTML applet definition:

```
<applet code=MultiAxisApplet width=300 height=300>

<param name=dataset0yValues value=123,432,213,543>
<param name=dataset0Color value="blue">
<param name=dataset0Name value="large">

<param name=dataset1yValues value="9,4,8,2">
<param name=dataset1Color value="yellow">
<param name=dataset1Name value="small">
```

```
<param name=legendOn value="true">
<param name=legendLlX value="0.30">
<param name=plotAreaLeft value="0.30">
<param name=plotAreaRight value="0.92">
<param name=titleString value="large and small numbers">
</applet>
```

The resulting chart looks like this:



Example 16 – Using a Custom Chart in a Server Imaging Bean

If you've reviewed the chapters on KavaChart server beans, you know that it's quite easy to convert an applet definition into a server bean definition. This is just as true for custom charts. The code for creating an imaging bean with our multiple axis chart is nearly trivial:

```
import com.ve.kavachart.servlet.*;

public class MultiAxisServerBean extends Bean {
      public void init(){
            initLocale();
            chart = new MultiAxisLineChart();
            getOptions();
      }
}
```

This code is the same as our applet example, except that we renamed the class and extended "Bean" from the "com.ve.kavachart.servlet" package instead of "ChartAppShell" from the "com.ve.kavachart.applet" package.

This object inherits the ability to process properties, create multiple image types, use a server image cache, create client-side link maps, and every other feature of other KavaChart server beans.

A JSP that uses this chart to create an image like the applet above is very simple, but it's actually more complicated that the bean class itself:

```
<!--
  A JSP with a Custom Chart class
-->

<jsp:useBean id="chart" class="MultiAxisServerBean"
        scope="page"/>

<html>
<head><title>Custom Chart Generation</title></head>
<body bgcolor="white">
<font size=4>

<%
    chart.setProperty("width", "300");
    chart.setProperty("height", "300");
    chart.setProperty("dataset0yValues, "123,432,213,543");
    chart.setProperty("dataset0Color, "blue");
    chart.setProperty("dataset0Name, "large");

    chart.setProperty("dataset1yValues, "9,4,8,2");
    chart.setProperty("dataset1Color, "yellow");
    chart.setProperty("dataset1Name, "small");

    chart.setProperty("legendOn, "true");
    chart.setProperty("legendLlX, "0.30">
    chart.setProperty("plotAreaLeft, "0.30");
    chart.setProperty("plotAreaRight, "0.92");
    chart.setProperty("titleString, "large and small numbers");

    chart.setProperty("imageType", "j_png");

%>
<p>
here's the chart:
<img src=/<%= chart.getFileName() %>>
</body>
</html>
```

Using Core Charts in a Servlet

KavaChart makes it easy to build servlets, JSPs, and other web applications using the framework and classes in the "com.ve.kavachart.servlet" package. However, you can also build your own servlet using the core charting classes.

Like all servlets, a chart servlet will extend the Servlet or HttpServlet classes in Sun's J2EE. These classes provide a request object and a response object that provide the tools needed to construct a chart and reply with an image.

Example 17 – A Simple Servlet

Our servlet will construct an image in memory, and then draw the chart into that image. The image will be encoded into some standard format. The servlet will set the content type to match that format, and will then write the image bytes to the servlet response's output stream.

Here's our code:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.awt.*;
import java.awt.image.*;
import java.io.*;

import com.ve.kavachart.chart.*;
import com.ve.kavachart.utility.GifMaker;

public class SimpleChartServlet extends HttpServlet {

        static final int WIDTH = 450;
        static final int HEIGHT = 300;

        public void doGet(HttpServletRequest request,
                          HttpServletResponse response) throws
                                              ServletException,
                                                   IOException {

              //create an image to draw to
              BufferedImage image =
                    new BufferedImage(WIDTH, HEIGHT,
                                BufferedImage.TYPE_INT_RGB);
              Graphics g = image.getGraphics();

              //create the chart
              BarChart chart = new BarChart("Hello World");
              double[] data = new double[5];
              for(int i=0;i<data.length;i++)
                    data[i] = Math.random() * 500.;
              chart.addDataset("Stuff", data);
              chart.resize(WIDTH, HEIGHT);
              chart.drawGraph(g);

              //encode the image and send it
              response.setContentType("image/gif");
```

```
        ServletOutputStream out =
                    response.getOutputStream();
        GifMaker gifMaker = new GifMaker(out, image);
        gifMaker.write();
    }
}
```

This example uses the sample GifMaker encoder available at Visual Engineering's web site. Other encoders are available at minimal or no cost, including Acme.com's "GifEncoder", Sun Microsystem's "jimi" package, and the encoders found in Java's "com.sun.image.codec" package.

Note:

Most GIF encoders use LZW compression, which is patented by Unisys Corporation. Before using a GIF encoder for more than experimental purposes, you must obtain a license from Unisys. Other image formats, such as PNG and JPEG have no such requirement

Chapter

7

Cooking With KavaChart Beans

The JavaBeans model provides a way to encapsulate classes into portable components that can be used ina standard way. This chapter discusses KavaChart's collection of JavaBeans, and provides programming examples to implement these components in your applications. This chapter is intended for users familiar with Java programming and the general JavaBeans component model.

Why Beans? An Overview of KavaChart Chart Beans

Much of Java's power as a programming language derives from its object-oriented design. Object-oriented languages make programmers more productive, and have the potential to make large-scale projects easier to complete and maintain. Object-orientation encourages code reuse, unit testing, and intelligible program code, but it doesn't happen automatically.

Like any programming language, Java can also be used to create a mess. A computer language cannot completely prevent programmers from creating bug-filled programs that are hard to extend and maintain.

Well written software generally uses collections of objects grouped into an intuitive organization of some kind. These objects might in turn consist of groups of other objects. This sort of grouping follows the way most complex human endeavors are organized.

For example, if you were to build a bridge, you wouldn't start with atoms, molecules, or even raw steel. Instead, you'd start with manufactured components like girders, bolts, rivets, and cables. Then you'd organize these objects into your bridge design to create an overall working bridge.

The JavaBeans component model organizes groups of programming code into a uniform set of classes with predefined behaviors. Fundamentally, this is similar to any other Java class, such as an Applet, with it's predefined "init", "start",

"stop", and "destroy" methods. The JavaBeans model adds some important features to facilitate the more generalized use of software components. This model also uses (and probably spurred the creation of) some important features of the Java programming language, such as introspection and serialization.

JavaBeans have not realized their initial promise, and have perhaps been supplanted in some areas by technologies like Enterprise JavaBeans, Java Server Pages, and so on. However, they still represent an important technology, especially for use in creating graphical user interfaces.

The KavaChart Developer's kit really includes 3 different collections of "beans": the applet collection (technically, applets are beans), the server bean collection, and the "com.ve.kavachart.beans" packages. The server bean collection was created to meet users' requirements for generating chart images in a web based application environment. The "com.ve.kavachart.beans" packages are designed for use in adding charts to custom applets and applications with graphical user interfaces.

Encapsulation

Perhaps more than anything else, the JavaBeans component architecture is designed to encapsulate objects. Encapsulation models are implicitly designed for a general level of object complexity. For example, it's possible to create JavaBeans that represent primitive objects, like "java.lang.Integer", but it's not particularly useful. Programs using these beans wouldn't gain any advantage over programs that just use "Integer" directly. Similarly, it's possible to create a JavaBean that represents an entire database application. However, the complexity in dealing with an actual significant database with multiple users, complicated queries, and real life performance requirements makes using the simple JavaBean model more bother than it's worth.

Most JavaBeans components are relatively simple objects with a small set of properties. Although the JavaBeans model has provision for editing internal objects within a component, in practice most components use simple properties, and provide access methods ("getters" and "setters") for these properties.

This works well for simple objects like Buttons, TextFields, or Scrollbars. It doesn't work as well for objects that have many internal components, or for objects that are, themselves, combinations of malleable components.

KavaChart's core charting librarary, "com.ve.kavachart.chart" is a collection of charting components, like axes, bars, and datasets. These components are designed to be re-used and extended in flexible ways. To create JavaBeans from these charting components, we used a collection of standard charts from the "com.ve.kavachart.chart" pacakage. These charts were each encapsulated into a GUI component that can be used like any other GUI JavaBean. These bean charts are less flexible than the core charting components, but simpler to use.

A GUI can be used to modify most standard chart properties. The beans respond to standard events for live data updates. Resizing, repainting, and double buffering behaviors are automatic. The KavaChart chart beans include just about all standard charts:

- com.ve.kavachart.beans.chart.AreaChart
- com.ve.kavachart.beans.chart.BarChart
- com.ve.kavachart.beans.chart.DateAreaChart
- com.ve.kavachart.beans.chart.DateLineChart
- com.ve.kavachart.beans.chart.HorizBarChart
- com.ve.kavachart.beans.chart.IndBarChart
- com.ve.kavachart.beans.chart.IndColumnChart
- com.ve.kavachart.beans.chart.LabelLineChart
- com.ve.kavachart.beans.chart.PieChart
- com.ve.kavachart.beans.chart.RegressChart
- com.ve.kavachart.beans.chart.SpeedoChart
- com.ve.kavachart.beans.chart.StackBarChart
- com.ve.kavachart.beans.chart.StackColumnChart
- com.ve.kavachart.beans.chart.StickChart

Each of these beans encapsulates the "com.ve.kavachart.chart.Chart" subclass of the same name. For example, "com.ve.kavachart.beans.chart.BarChart" encapsulates the "com.ve.kavachart.chart.BarChart" core charting class.

Each of these beans extends "com.ve.kavachart.beans.chart.ChartBean", which provides the event handling capabilities all KavaChart beans share.

Note:

By default KavaChart beans are extended "java.awt.Panel" classes, since this fits into most current applet environments. The basic developers kit includes the source code for "com.ve.kavachart.chart.ChartBean" so you can change all the bean classes to extend "javax.swing.JPanel" for lightweight component applications.

Introspection

Using Java's introspection capabilities, most IDEs are able to automatically create property sheets for simple JavaBean components. Introspection looks for matching "get/set" methods at the top level of an object, and creates a simple GUI editor to let users set these values without coding.

In a graphical layout environment that uses beans like this, the IDE generates code to match the property settings from the GUI editor. Code generated for a simple Button class might look like this:

```
Button b1 = new Button();
b1.setLabel("Press Me");
```

This approach presents some serious problems for more complex classes. The most important difficulty is "exposing" all the functionality of a complex object at the top level of a class. If a class has hundreds of potential properties to set, the source code can become unwieldy and difficult to maintain.

It's also a problem when the user is presented with an unordered property sheet that doesn't provide any visual organization to the many available properties. Searching through a long scrolling list to find a property by name can be difficult, but it's even worse if you're not intimately familiar with the available properties.

It's also a problem for non-standard properties. For example, KavaChart's time series axes let you set time and date format properties, and internal calendar classes for axis calculations. While these are standard Java classes, most IDEs don't have GUI mechanisms for editing these sorts of properties. While it's possible to create custom property editors for more complex properties like these, a length list of property editors is overwhelming, even when presented in a GUI.

KavaChart components use a different approach. Each KavaChart bean includes a GUI customizer that lets you interactively edit the chart's appearance with more intuitive graphical controls. See the section below about customizers and serialization for more information on this topic.

Because the chart beans use customizers instead of property lists, the property sheet for the chart bean will display only a name. All other properties are edited by using the chart's customizer, and by attaching data sources.

Serialization

While serialization isn't an intrinsic part of the JavaBeans specification, it was released at the same time to provide object persistence for complex components. KavaChart beans rely on serialization to save and restore chart properties edited with a chart customizer.

Unfortunately, many IDEs do not provide support for automatic serialization and deserialization. This means that you might need to create your own

serialized objects. See the section below on customizers and serialization for more information on this topic.

> **Note:**
>
> Serializing KavaChart chart and data beans may result in serialized data that is invalid for future KavaChart releases. In other words, if you choose to use serialized charts and data sources, you may need to re-serialize these objects in the future to take advantage of new KavaChart releases.

An Event Model

The JavaBeans specification includes an event model adequate for most GUI applications, including charting applications. Many beans use "propertyChange" events to communicate with each other. An object will fire a propertyChange event that is received by one or more listeners. Depending on the type of event, the "vetoable" status of the property on a particular listener, and so on, the events may register, and possibly even trigger another event fired to another listener.

KavaChart does not use propertyChange events to communicate with other objects. Charts are primarily concerned with plotting data rather than generating data, and so KavaChart beans are generally the recipients of events, not the initiator of new events.

KavaChart beans define a new "com.ve.kavachart.beans.data.DataEvent" that extends "java.util.EventObject" to provide new data to a chart. Any object interested in receiving these events must implement the interface "com.ve.kavachart.beans.data.DataFeedListener", as all KavaChart chart beans do.

Chart bean data is provided by data beans, defined in the "com.ve.kavachart.beans.data" package. These beans generate DataEvent events to classes that implement the DataFeedListener interface. The JavaBeans EventSetDescriptor in each data bean's BeanInfo file tells the IDE that KavaChart data beans are able to send a DataEvent to a chart bean or any other DataFeedListener. This generally means that you can use the graphical design tool within the IDE to make a connection from the data bean to the chart bean.

It also means you can connect the two with a simple bit of code, like this:

```
dataSource.addDataFeedListener(chart);
```

KavaChart data sources can send data to multiple charts at once. For example, if your data feed polls a URL for new data, each time the data changes, it is broadcast to every chart listening for new data. New data **replaces** existing data.

A KavaChart DataEvent includes modifiers to let listeners know what parts of the incoming data stream are relevant. For example, some charts are not interested in secondary Y (y2) data. Other data streams should not replace the dataset name, and so on.

See the code examples later in this chapter for more information on creating and using KavaChart data beans.

The Bottom Line

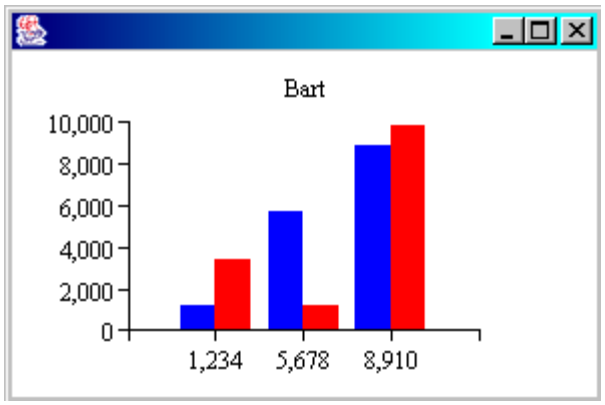
The JavaBeans component model provides an elegant model for component reuse in certain kinds of applications. This model isn't appropriate for every programming task, but it works well for creating interactive applications. Using KavaChart chart and data beans to build interactive charting applications can result in shorter development cycles, better code re-use, and simpler application architectures.

Using Chart Beans

The beauty of the JavaBeans model is that it's nearly trivial to add a component to a bean-friendly programming framework. While this is easy to accomplish with various graphical layout tools, it's just as easy to write the code directly. Here's a chart similar to the standalone "ChartPanel" examples in the "Core API" chapter:

```
import java.awt.*;
import com.ve.kavachart.beans.chart.*;

public class BeanFrame extends Frame {
      public BeanFrame(){
            add(new BarChart());
      }

      public static void main(String[] args){
            Frame f = new BeanFrame();
            f.setSize(300, 200);
            f.setVisible(true);
      }
}
```

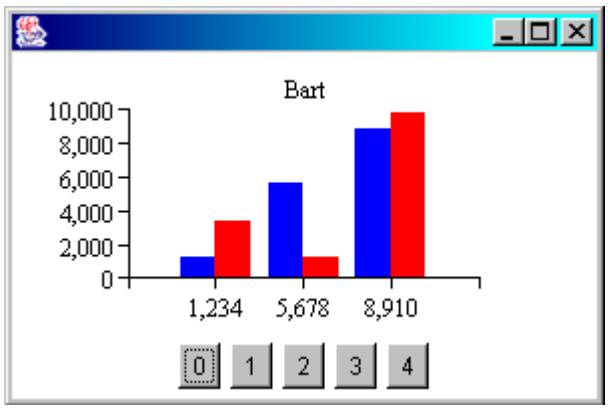
Since the "BarChart" bean extends "java.awt.Panel", we can add it to our Frame just like any other Panel. BarChart contains default data and settings, so we end up with a chart that looks like this:



A little bit of coding to add a (non working) GUI, results in code like this:

```
import java.awt.*;
import com.ve.kavachart.beans.chart.*;

public class BeanFrame extends Frame {
          public BeanFrame(){
                setLayout(new BorderLayout());
                add(new BarChart(), "Center");
                Panel p = new Panel();
                for(int i=0;i<5;i++){
                      Button b = new Button(Integer.toString(i));
                      p.add(b);
                }
                add(p, "South");
          }

          public static void main(String[] args){
                Frame f = new BeanFrame();
                f.setSize(300, 200);
                f.setVisible(true);
          }
}
```

This code isn't particularly useful, since it just adds non-working buttons, but it illustrates how the chart responds to the layout manager's instructions to accommodate the button container:



Since this chart is in a resizable Frame, we could also resize the frame, and the chart bean will resize appropriately.

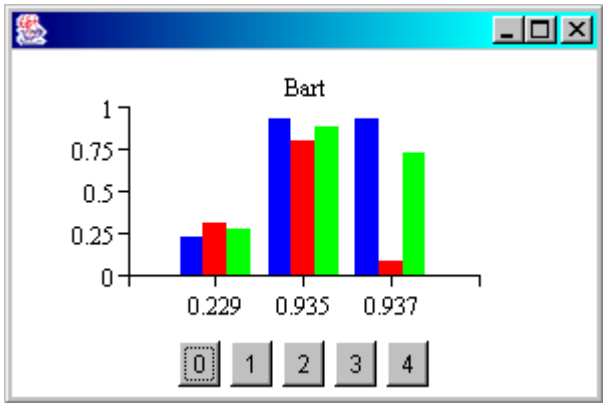
This chart still isn't particularly useful, since it uses default data. In a real application we would probably use an existing KavaChart data bean or create a data bean to feed data to the chart. The KavaChart data bean collection includes a useful "DataFeedSimulator" that generates random data to help us visualize the chart during development. Here's the code for a chart with a DataFeedSimulator:

```
import java.awt.*;
import com.ve.kavachart.beans.chart.*;
import com.ve.kavachart.beans.data.*;

public class BeanFrame extends Frame {
    ChartBean chart;

    public BeanFrame(){
         setLayout(new BorderLayout());
         chart = new BarChart();
         add(chart, "Center");
         Panel p = new Panel();
         for(int i=0;i<5;i++){
           Button b = new Button(Integer.toString(i));
           p.add(b);
         }
         add(p, "South");

         //add a data feed
         DataFeedSimulator dataSource =
                      new DataFeedSimulator();
         //new data every 2 seconds
         dataSource.setInterval(2);
         dataSource.setNumberOfSets(3);
         dataSource.setNumberOfPoints(3);
         dataSource.addDataFeedListener(chart);
         dataSource.update();
         dataSource.start();
    }

    public static void main(String[] args){
```

```
        Frame f = new BeanFrame();
        f.setSize(300, 200);
        f.setVisible(true);
    }
}
```

DataFeedSimulator lets us set the number of datasets and the number of points we want to simulate. We set both to 3. It also lets us set an update interval. The simulated data feed refreshes the data at every interval, updating all the chart listeners. We set this value to 2 to simulate an animated real time data feed. Next, we call "update()" before displaying the chart to make sure the default bean data is gone before the chart is displayed. Finally, we "start" the simulation.

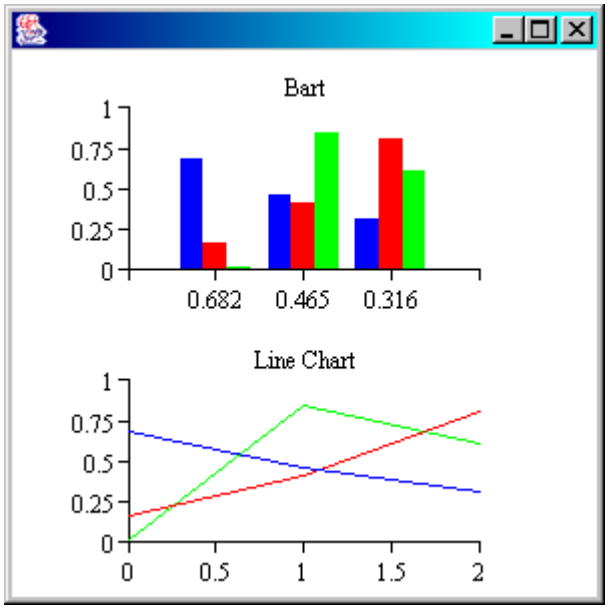
The result is a chart that updates every 2 seconds, and displays 3 series of 3 bars each:



While we're still dealing with fake data, this is much closer to an actual application; we have a visible chart that's actually reading data from a live data source. To change this to your own data source, it's a simple matter of replacing DataFeedSimulator with your actual data feed class.

Find out how to build your own data beans in the sections below.

This same data source can feed data to more than one chart simultaneously. We'll modify BeanFrame to create a two chart sample like this:



As the data source updates, both charts change simultaneously to describe data in different ways; line charts communicate trends, while bar charts communicate comparisons of individual categories.

Here's our code:

```
import java.awt.*;
import com.ve.kavachart.beans.chart.*;
import com.ve.kavachart.beans.data.*;

public class TwoBeanFrame extends Frame {
        ChartBean chart1, chart2;

        public TwoBeanFrame(){
            setLayout(new GridLayout(2, 1));
            chart1 = new BarChart();
            add(chart1);

            chart2 = new LineChart();
            add(chart2);

            //add a data feed
            DataFeedSimulator dataSource = new DataFeedSimulator();
            //new data every 2 seconds
            dataSource.setInterval(2);
            dataSource.setNumberOfSets(3);
            dataSource.setNumberOfPoints(3);

            //apply the data to both charts
            dataSource.addDataFeedListener(chart1);
            dataSource.addDataFeedListener(chart2);

            dataSource.update();
            dataSource.start();
        }
```

```
        public static void main(String[] args){
            Frame f = new TwoBeanFrame();
            f.setSize(300, 300);
            f.setVisible(true);
        }
    }
```

Bean Customizers and Serialization

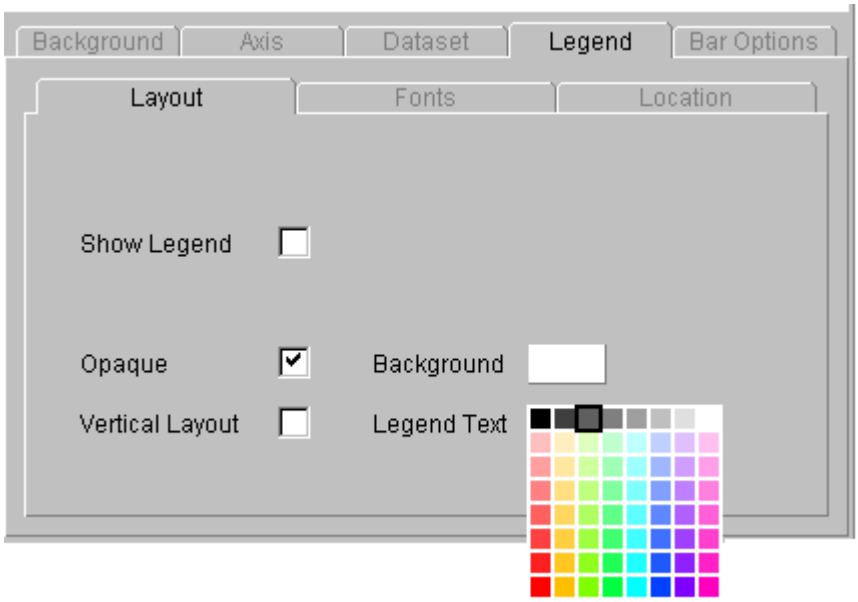
As we described in the sections above, KavaChart beans rely on object serialization to achieve a persistent state. This is different than exposing a long list of properties and using "getters" and "setters" to manage this information through a property sheet.

If your IDE supports automatic serialization of JavaBeans, it will create a resource that represents the bean in a particular state. KavaChart's beans are completely serializable, except for images you might assign to the charts (Image is not a serializable class).

Once you've obtained a serialized resource that represents your chart, you need only read that object using Java's "ObjectInputStream" to re-create the chart in the state it was in before serialization.

Using a Customizer

You will generally use a customizer to initialize a chart you intend to serialize. Each "Chart" class in "com.ve.kavachart.beans.chart" includes a customizer that looks something like this:



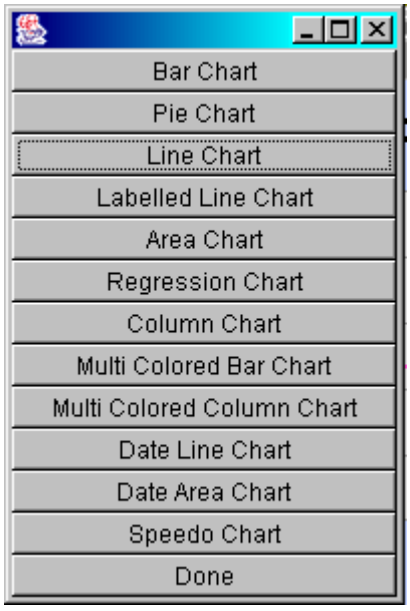
This customizer is included with "com.ve.kavachart.beans.chart.BarChart". The customizer provides a graphical user interface for interactively changing the

properties of a chart without using a property sheet. Since KavaChart uses a relatively fine-grained object orientation (each chart is a separate object, charts consist of sets of internal objects), this is much more effective and intuitive than using a property sheet. Changes to the customizer are immediately reflected in the chart managed by the customizer.

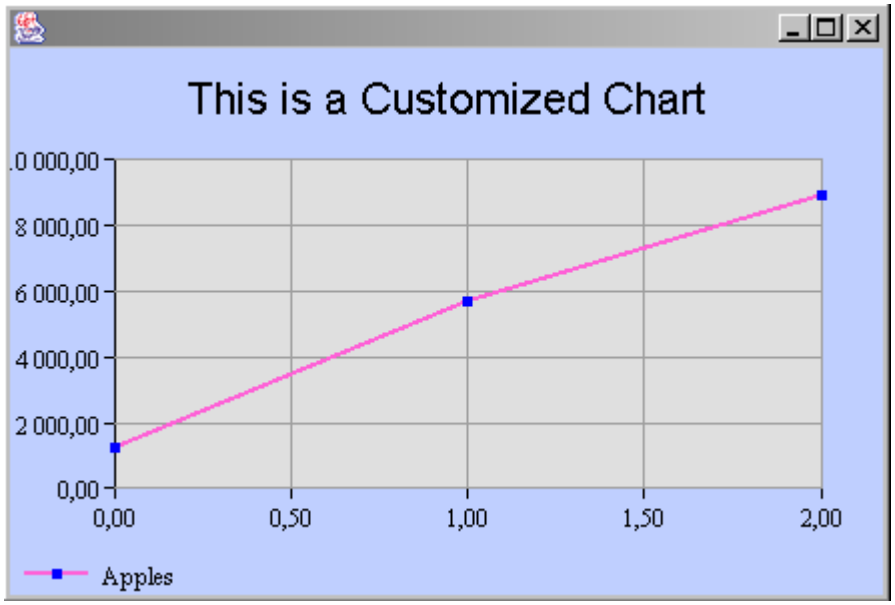
Each chart's customizer consists of groups of tabs that are common to all charts (background, legend information, etc.), and one or more tabs that are specific to the chart you're managing. For example, this customizer has a "Bar Options" tab that doesn't appear on line charts and pie charts. The pie chart customizer has a "Pie Options" tab, and no "Axis" tab.

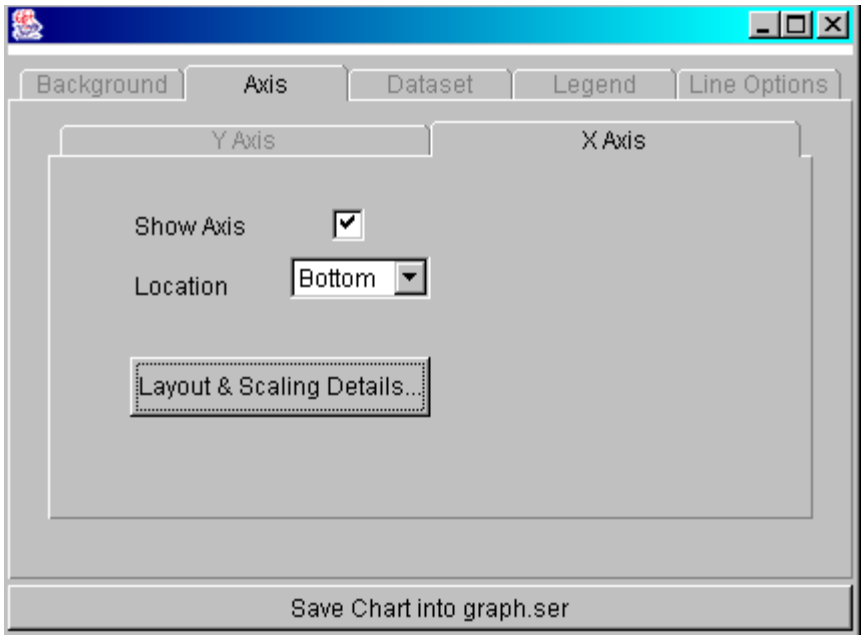
Figuring out how to launch the customizer may be a challenge in some IDEs. Many IDEs treat customizers as an oddity, and bias the user interface towards simple objects like "Button" or "Label".

If you can't find the customizer option in your IDE, use the program "com.ve.kavachart.beans.chart.ChartWriter" to create a serialized chart object. This chart launches a simple button box like this:



Select a button on this box to launch a default chart with a customizer. Use the customizer to make changes to the appearance of the chart. Here's a customizer and chart window created by "ChartWriter":





When you finish making changes to your chart, save it by pressing the "Save Chart" button. This creates a file named "graph.ser" containing the serialized chart.

Using Serialized Chart Data

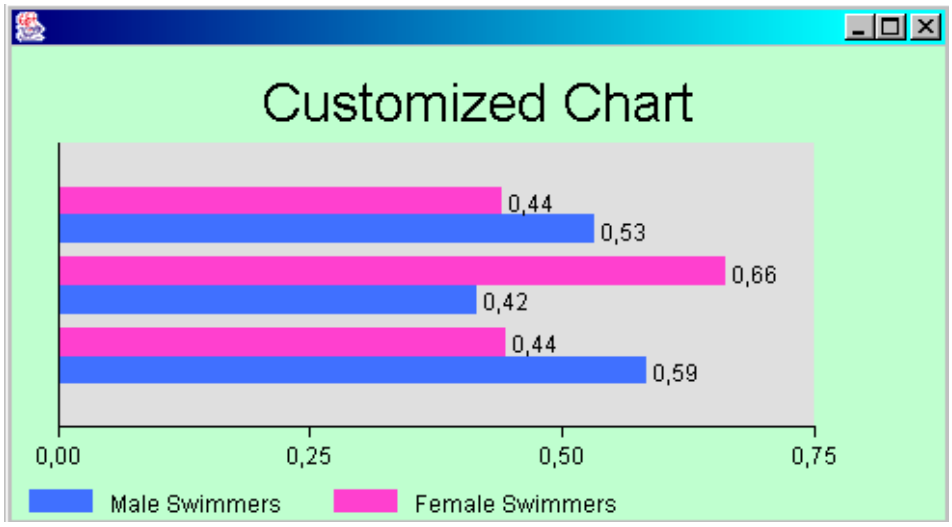
Your IDE may make provisions for including this file as a resource. If not, you may need to read the file from a jar file or from some other input stream. You can find an example of how to read these files in "com.ve.kavachart.beans.chart.ChartReader". We'll create a sample that follows our "BeanFrame" example above to read the serialized output from our customizer and create a chart object from it.

Here's the source code:

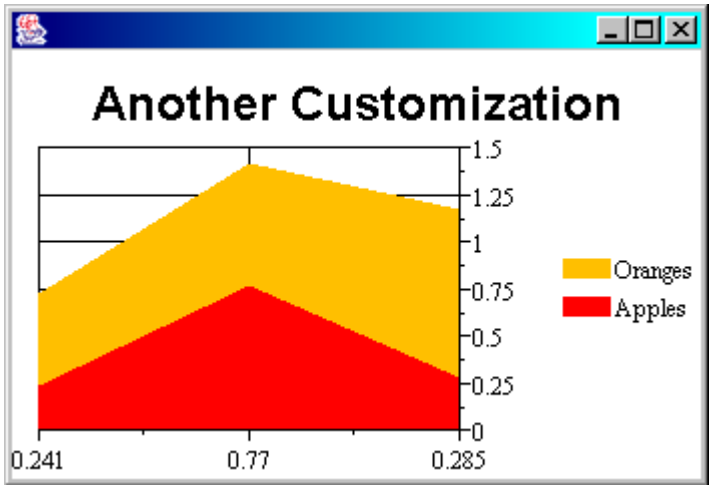
```
import java.io.*;
import java.awt.*;
import com.ve.kavachart.beans.chart.*;
import com.ve.kavachart.beans.data.*;

public class SerializedBeanFrame extends Frame {

        ChartBean chartBean;
        public SerializedBeanFrame(){

           chartBean = newChart("graph.ser");
           setSize(chartBean.getSize().width, chartBean.getSize().height);
           add(chartBean);

           //add a data feed like BeanFrame has
           DataFeedSimulator dataSource = new DataFeedSimulator();
           //new data every 2 seconds
           dataSource.setInterval(2);
           dataSource.setNumberOfSets(2);
           dataSource.setNumberOfPoints(3);
           dataSource.addDataFeedListener(chartBean);
           dataSource.update();
           dataSource.start();


        }
        public static void main (String[] args) {
           SerializedBeanFrame me = new SerializedBeanFrame();
           me.setVisible(true);
        }
        private ChartBean newChart(String fileName){

           ChartBean bean = null;

           InputStream is = this.getClass().getResourceAsStream(fileName);
           if (is == null) {
                 System.err.println("no chart data at " + fileName);
           }
           try {
                 ObjectInputStream ois = new ObjectInputStream(is);
                bean = (ChartBean) ois.readObject();
           } catch (Exception ex) {
                 System.err.println("trouble reading: "+ex);
           }
           return bean;
        }
}
```

This little bit of source code is nearly the same as "BeanFrame", except that instead of instantiating a chart with a "new" statement, we read it from serialized data with a "newChart" method. "NewChart" is copied directly from ChartReader.

In this case we declared our chart to be a "ChartBean", so we can read in any sort of serialized bean; a bar chart, line chart, pie chart, etc. The chart is created an initialized as if we had made all the method calls to customize colors, dataset names, chart titles, fonts, and so on. Applying this code to one serialized file yields a chart like this:



While applying the same code to another serialized resource file yields a chart like this:



As we've demonstrated in the previous sections, we could also create instances of multiple charts in the same layout, supply both charts with the same (or different) live data sources, and add other GUI components as well.

The KavaChart Bean Data Model

Part of the power of bean-based charting derives from separating the data model from the presentation. Data generators become portable; they can be used for many charts, and can even drive multiple charts concurrently.

In common object oriented parlance, this paradigm is called "model view controller". KavaChart data feeds connect the model (probably a database, live data feed, etc.) to a view (the chart or charts). The code that controls the process (usually some sort of user interface) is referred to as a controller. While it might take some initial work to create a data feed based on the KavaChart data model, it's worth the effort, since the same data feed can be reused for many kinds of charts and in many different situations.

KavaChart's bean packages include utility classes to help you build your own data feeds, but it's useful to understand how these work internally. You may find that it's easier and more efficient to build your own data interface "from scratch".

KavaChart's DataEvent

Central to the KavaChart's data model is "com.ve.kavachart.beans.data.DataEvent". This event class extends "java.util.EventObject" to add some very simple functionality. Essentially, this class contains an array of "com.ve.kavachart.chart.Dataset" classes and a "mask" used to describe what items a listening chart object may be interested in. This listener mask is defined like typical event masks, by "OR" ing different modifiers together:

The modifiers:

```
Y_DATA_MASK
X_DATA_MASK
Z_DATA_MASK
AUX_DATA_MASK
LABELS_MASK
SET_NAMES_MASK
```

Each modifier specifies whether the listener should use a particular item. For example, if we combine Y_DATA_MASK and LABELS_MASK, then the listeners will ignore incoming dataset names, Y2 and Y3 values, and X values in the incoming data:

```
int myDataMask = DataEvent.Y_DATA_MASK|DataEvent.LABELS_MASK;
```

Every bean data feed creates DataEvent classes, which are sent to DataFeedListeners. The listeners are usually either charts or data filters. A filter makes changes to the incoming data and forwards it to another listener, forming

a chain of data feeds and listeners. At the end of the data feed chain, a chart will receive the DataEvent and copy the incoming data into its internal datasets.

The DataFeedListener Interface

DataEvents are sent to classes that implement the DataFeedListener Interface. This interface consists of a single public method "newData" with a DataEvent as the argument. The "newData" method is called whenever incoming data changes. If a chart bean receives a DataEvent via a call to "newData", the chart updates immediately to display the new information.

> **Note:**
>
> Some DataEvent recipients are filters. A filter may or may not pass the new information to its listeners, depending on what it's designed to do. For example, you might use a filter to start and stop updates to your chart bean, even when the filter continues to collect the latest information from a data source.

Example – A Sine Wave Data Feed

Using DataEvent and DataFeedListener, we'll create a simple data feed that will generate sine waves. The class will contain a thread that sends new information periodically, sending information to any listeners with an "update" method like this:

```
private void update(DataEvent event){
       for(int i=0;i<listeners.size();i++){
           ((DataFeedListener)listeners.elementAt(i)).newData(event);
       }
}
```

The core of the data feed is a sine wave generator. This code creates "com.ve.kavachart.chart.Dataset" classes that contain Y data in a sine wave. The dataset classes are placed in a DataEvent, which is broadcast to all listeners in the "update" method.

Here's the sine wave generation code:

```
 void generateData(){
      //do this the first time through
      if(myEvent == null){
           //we want X and Y data, and dataset names
           myEvent = new DataEvent(this,
                 DataEvent.X_DATA_MASK|
                 DataEvent.Y_DATA_MASK|
                 DataEvent.SET_NAMES_MASK);
           //provide some names for the data:
           set1.setName("sine wave");
           set2.setName("cosine wave");
           myEvent.datasetArray[0] = set1;
           myEvent.datasetArray[1] = set2;
           for(double i=lastCount;i<lastCount+WINDOW_SIZE;i++){
```

```
                Datum d = new Datum((double)i,
                      Math.sin(i/WINDOW_SIZE), null);
                set1.getData().addElement(d);
                d = new Datum((double)i, Math.cos(i/WINDOW_SIZE),
                       null);
                set2.getData().addElement(d);
            }
            lastCount = lastCount + WINDOW_SIZE;
            update(myEvent);
            return;
        }
        //after that, just recycle data
        Datum d = set1.getDataElementAt(0);
        set1.getData().removeElementAt(0);
        d.setX(lastCount);
        d.setY(Math.sin(lastCount/WINDOW_SIZE));
        set1.getData().addElement(d);

        d = set2.getDataElementAt(0);
        set2.getData().removeElementAt(0);
        d.setX(lastCount);
        d.setY(Math.cos(lastCount/WINDOW_SIZE));
        set2.getData().addElement(d);
        lastCount+= 1.0;

        update(myEvent);
        return;
    }
```

The first time this data generator is called, it creates a DataEvent class that will be fired repeatedly to users. We could generate new DataEvents for each update, but in this case, that would only make the code more complicated.

The DataEvent starts with this modifier:

```
DataEvent.X_DATA_MASK|DataEvent.Y_DATA_MASK|DataEvent.SET_NAMES_MASK
```

This tells all the listeners that the data feed sets the correct information for X, Y and dataset name properties.

The data generator populates the DataEvent with 2 Dataset classes. These are used to store the original source data. We start with a sine wave for one series, and a cosine wave for the other.

Subsequent calls to the "generateData" method continue the sine and cosine waves. The first Datum class is removed from the Dataset's data vector, and the X and Y values are updated to place the point at the end of the series.

Each time we change the data, the "update" method is called to notify listeners that new data is available.

To complete the class, we'll add methods to add and remove listeners, and we'll add a Thread to generate new data every second. The completed class looks like this:

```
import com.ve.kavachart.beans.data.*;
import com.ve.kavachart.chart.*;
import java.util.*;

public class SineWaveGenerator implements Runnable {

      static int WINDOW_SIZE = 20;
      Vector listeners = new Vector();
      Dataset set1 = new Dataset();
      Dataset set2 = new Dataset();
      DataEvent myEvent = null;
      Thread myThread = null;

      double lastCount = 0.;

      public void addListener(DataFeedListener l){
           listeners.addElement(l);
      }

      public void removeListener(DataFeedListener l){
           listeners.removeElement(l);
      }

      private void update(DataEvent event){
           for(int i=0;i<listeners.size();i++){

           ((DataFeedListener)listeners.elementAt(i)).newData(event);
           }
      }

      public void run(){
           while(myThread!=null){
                 generateData();
                 try{
                       Thread.sleep(1000);
                 }catch(Exception ignored){
                 }
           }
      }

      public void start(){
           if(myThread==null){
                 myThread = new Thread(this);
                 myThread.start();
           }
      }

      public void stop(){
           if(myThread!=null){
                 myThread = null;
           }
      }
```

```
    void generateData(){
        //do this the first time through
        if(myEvent == null){
            //we want X and Y data, and dataset names
            myEvent = new DataEvent(this,
                DataEvent.X_DATA_MASK|
                DataEvent.Y_DATA_MASK|
                DataEvent.SET_NAMES_MASK);
            //provide some names for the data:
            set1.setName("sine wave");
            set2.setName("cosine wave");
            myEvent.datasetArray[0] = set1;
            myEvent.datasetArray[1] = set2;
            for(double i=lastCount;i<lastCount+WINDOW_SIZE;i++){
                Datum d = new Datum((double)i,
    Math.sin(i/WINDOW_SIZE), null);
                set1.getData().addElement(d);
                d = new Datum((double)i,
    Math.cos(i/WINDOW_SIZE), null);
                set2.getData().addElement(d);
            }
            lastCount = lastCount + WINDOW_SIZE;
            update(myEvent);
            return;
        }
        //after that, just recycle data
        Datum d = set1.getDataElementAt(0);
        set1.getData().removeElementAt(0);
        d.setX(lastCount);
        d.setY(Math.sin(lastCount/WINDOW_SIZE));
        set1.getData().addElement(d);

        d = set2.getDataElementAt(0);
        set2.getData().removeElementAt(0);
        d.setX(lastCount);
        d.setY(Math.cos(lastCount/WINDOW_SIZE));
        set2.getData().addElement(d);
        lastCount+= 1.0;

        update(myEvent);
        return;
    }
}
```

We now have a complete data generator that can be attached to any KavaChart chart bean. The code to connect the two is nearly trivial. This modification to a previous example does the job:

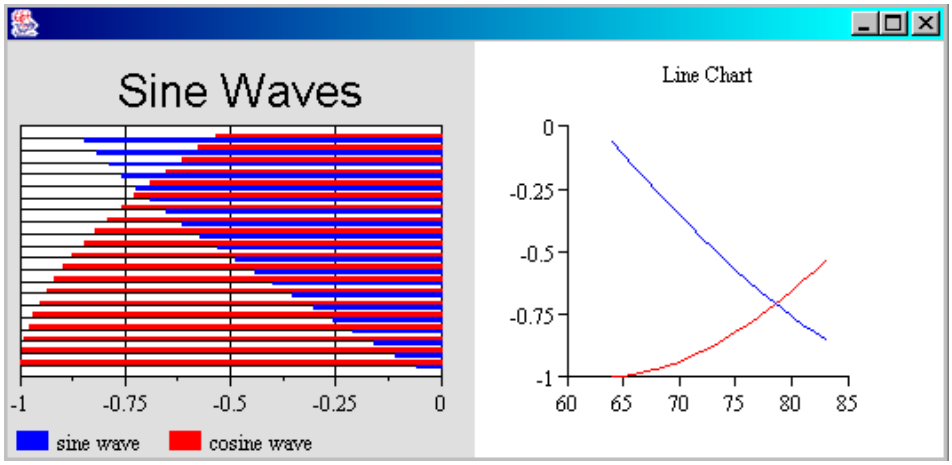
```
import java.awt.*;
import com.ve.kavachart.beans.chart.*;
import com.ve.kavachart.beans.data.*;

public class SineBeanFrame extends Frame {
    ChartBean chart;
```

```
        public SineBeanFrame(){
             chart = new LineChart();
             add(chart);

             //add our data feed
             SineWaveGenerator swg = new SineWaveGenerator();
             swg.addListener(chart);
             swg.start();
        }

        public static void main(String[] args){
             Frame f = new SineBeanFrame();
             f.setSize(300, 200);
             f.setVisible(true);
        }
}
```

This example will draw a chart within a Frame that is updated every second to display new data. If the chart has a legend, the dataset names within the legend are updated to reflect the names sent by the data feed.

This data feed is completely portable. You can use it with any chart bean, and even send data to multiple chart beans at once.

We'll use our previous example of reading serialized chart data, and add another chart to the same Frame. Connecting the sine wave data feed to both charts creates a multi-panel display like this:



The same data feed drives both charts, with updates every second. However, the first chart is based on serialized information created with a BarChart bean customizer. Like all horizontal bar charts, the Y axis is horizontal. The LineChart bean used for the second chart uses the same data, but displays it on a vertical Y axis and horizontal X axis.

The first chart includes a legend. Since we specified a SET_NAMES_MASK as part of our DataEvent modifier, the dataset names are updated to match the settings in our data feed.

Here's the code for this multiple chart panel:

```
import java.io.*;
import java.awt.*;
import com.ve.kavachart.beans.chart.*;
import com.ve.kavachart.beans.data.*;

public class SerializedSineFrame extends Frame {

      ChartBean chartBean1, chartBean2;
      public SerializedSineFrame(){

           setLayout(new GridLayout(1,2));

           //read the first chart from data
           chartBean1 = newChart("graph.ser");
           add(chartBean1);

           //create the second chart directly
           chartBean2 = new LineChart();
           add(chartBean2);

           //add our data feed
           SineWaveGenerator swg = new SineWaveGenerator();
           swg.addListener(chartBean1);
           swg.addListener(chartBean2);
           swg.start();

      }
      public static void main (String[] args) {
           SerializedSineFrame me = new SerializedSineFrame();
           me.setVisible(true);
      }

      //stuff to read serialized data
      private ChartBean newChart(String fileName){

           ChartBean bean = null;

           InputStream is =
      this.getClass().getResourceAsStream(fileName);
           if (is == null) {
                 System.err.println("Could not locate the chart data
      at " + fileName);
           }
           try {
                 ObjectInputStream ois = new ObjectInputStream(is);
              bean = (ChartBean) ois.readObject();
           } catch (Exception ex) {
                 System.err.println("trouble reading from serialized
      data: "+ex);
           }
```

```
            return bean;
        }
    }
```

The actual program logic for this class is very simple. The constructor creates a GridLayout to make room for both charts. The first chart is read from serialized data. The second chart is created directly with a constructor.

Our sine wave data feed is then created and both charts are added as listeners. The data feed is started, and the application is made visible. Since the data feed updates the sine wave every second, these charts continuously update.

Using SimpleDataFeed

Though it's relatively easy to build a data feed like the one above, KavaChart data beans include a helper class to make a full featured data feed even easier to create.

SimpleDataFeed includes pre-built methods for getting and setting DataEvent modifiers. It also includes predefined protected variables that you can use to create and modify dataset information:

```
double xArray[][] -  dataset x values
double yArray[][] -  dataset y values
double labelArray[][] -  dataset y values
Dataset myDatasetArray[] – the data repository in KavaChart form
```

To use SimpleDataFeed, dimension and complete the X and Y data arrays. Then call "updateDatasets()", and "update()". Data will be copied from the data arrays into the Dataset arrays and sent to each DataFeedListener attached to your data feed.

For a concrete example, we'll make a version of our SineWaveGenerator with SimpleDataFeed. The core of the code is similar to our previous example:

```
void generateData(){
      for(int i=0;i<xArray[0].length;i++){
        double val = (double)(i+lastCount);
        xArray[0][i] = val;
        xArray[1][i] = val;
        yArray[0][i] = Math.sin(val/WINDOW_SIZE);
        yArray[1][i] = Math.cos(val/WINDOW_SIZE);
      }
      lastCount = lastCount + WINDOW_SIZE;
      updateDatasets();
      update();
}
```

Unlike our previous example, however, our actual data generation method is extremely simple. We don't need to worry about creating datasets, initializing Datum classes, and so on.

We also inherit useful methods for getting and setting DataEvent modifiers. Using SimpleDataFeed involves a little bit of extra overhead because data has to be copied to and from the data arrays. In most applications, however, this overhead is negligible. The simplicity of resulting classes is probably reason enough for most applications to use SimpleDataFeed.

Here's the entire SimpleDataFeed subclass:

```
import com.ve.kavachart.beans.data.*;

public class SimpleSineWaveDataFeed
        extends SimpleDataFeed
              implements Runnable {


       int lastCount = 0;
       static int WINDOW_SIZE = 20;
       Thread myThread = null;

       public SimpleSineWaveDataFeed(){
         xArray = new double[2][];
         yArray = new double[2][];
         xArray[0] = new double[WINDOW_SIZE];
         xArray[1] = new double[WINDOW_SIZE];
         yArray[0] = new double[WINDOW_SIZE];
         yArray[1] = new double[WINDOW_SIZE];
       }

       public void run(){
         while(myThread!=null){
               generateData();
               try{
                     Thread.sleep(1000);
               }catch(Exception ignored){
               }
         }
       }

       public void start(){
         if(myThread==null){
               myThread = new Thread(this);
               myThread.start();
         }
       }

       public void stop(){
         if(myThread!=null){
               myThread = null;
         }
       }

       void generateData(){
         for(int i=0;i<xArray[0].length;i++){
               double val = (double)(i+lastCount);
               xArray[0][i] = val;
```

```
                xArray[1][i] = val;
                yArray[0][i] = Math.sin(val/WINDOW_SIZE);
                yArray[1][i] = Math.cos(val/WINDOW_SIZE);
            }
            lastCount = lastCount + WINDOW_SIZE;
            updateDatasets();
            update();
        }
}
```

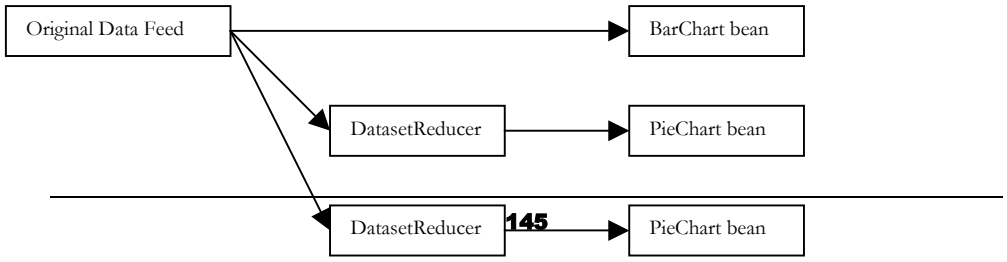
This data feed is connected to a chart bean like any other data feed:

```
import java.awt.*;
import com.ve.kavachart.beans.chart.*;
import com.ve.kavachart.beans.data.*;

public class SimpleSineBeanFrame extends Frame {
      ChartBean chart;

      public SimpleSineBeanFrame(){
        chart = new LineChart();
        add(chart);

        //add our data feed
        SimpleSineWaveDataFeed swg = new SimpleSineWaveDataFeed();
        swg.addDataFeedListener(chart);
        swg.start();
      }

      public static void main(String[] args){
        Frame f = new SimpleSineBeanFrame();
        f.setSize(300, 200);
        f.setVisible(true);
      }
}
```

And like other KavaChart data sources, this data feed can send DataEvents to more than one chart at a time, and can be attached to any kind of chart.

Data Beans As Filters

A data bean can be a DataFeedListener as well as a DataEvent sender. Some of KavaChart's built-in data beans work this way. A data bean like this is generally used to filter incoming data in some way. For example, KavaChart's "DatasetReducer" bean can be used to select one of several incoming datasets. In this way, you can use the same data feed to create, say, a bar chart with two series of bars, and two different pie charts.



While this might be difficult to create without the bean data model, it's really quite easy to do using a data bean as a filter.

The program code looks something like this:

```
//create the chart beans
BarChart barChart = new BarChart();
PieChart pie1 = new PieChart();
PieChart pie2 = new PieChart();

//create the data beans
DataFeed originalDataFeed = new DataFeed();
DatasetReducer reducer1 = new DatasetReducer();
DatasetReducer reducer2 = new DatasetReducer();

//connect everything together
originalDataFeed.addListener(barChart);
originalDataFeed.addListener(reducer1);
originalDataFeed.addListener(reducer2);

reducer1.setSelectedDataset(0);
reducer1.addDataFeedListener(pie1);

reducer2.setSelectedDataset(1);
reducer2.addDataFeedListener(pie2);
```

Whenever "originalDataFeed" fires a "newData" event, the bar chart will receive it. So will the filters "reducer1" and "reducer2". These filters will forward one series each to the pie charts "pie1" and "pie2". A change in data from "originalDataFeed" will automatically propagate to all three charts.

KavaChart's built-in data beans include both "DatasetReducer" and "DatapointReducer". As its name implies, "DatapointReducer" reduces an incoming data stream to a single observation. This is mostly useful for "Speedo" charts, which describe a single point. You can use these filters in combination to create multiple gauges for multiple series and points.

KavaChart Built-In Data Beans

KavaChart includes a number of other built-in data feed beans to help you provide data to your applications. In addition to DataFeedSimulator, DatasetReducer, DataPointReducer, and the abstract SimpleDataFeed, these include the following:

- URLDataFeed – this abstract class provides capabilities to read data from a URL. Concrete implementations of this class include:

- RowColumnDataFeed – reads data in rows or columns, which may or may not include data labels

    - DateDataFeed – like RowColumnDataFeed, but expects date string values instead of numeric X values.

- OnePointURLDataFeed – periodically reads a single number or pair, and accumulates that into a series.

- SocketListener – a datafeed that monitors a socket for incoming data strings. SocketListener accumulates observations until it reaches its defined capacity. At that point, data scrolls to display only the most recent observations.

- QuoteMonitorBridge – a sample datafeed that translates "propertyChange" events from the "QuoteMonitor" bean in Sun's BDK into KavaChart data.

- TimeStampAccumulator – a sample datafeed filter that associates time stamps with incoming data for time oriented charts (DateLineChart, DateAreaChart, etc.). Incoming X data is replaced with a time stamp created at the time of the data event.

Note:

KavaChart Data Beans are all provided in source code form so you can follow their form or make modifications to create your own data beans. You will likely need to create your own data sources to exploit KavaChart's model completely.

Server Push – a Complete Example

Standard KavaChart applets rely on "client-pull" to retrieve data over a network. That is, by using URL dataset definitions, the applet will contact your server via http periodically to update its information. While this is convenient and adequate for many applications, it suffers from 2 distinct drawbacks. First, the overhead of negotiating a new http connection can be significant. This is particularly significant if your data changes frequently. Creating a new http connection, say, every second to update your chart's data could become a major drain on server resources, particularly if many users are viewing the charts simultaneously.

Second, your applet is running asynchronously from data changes. If your applet is set to update every 30 seconds, it might well be 30 seconds late in displaying some important data change. For some kinds of information, this doesn't matter. Some information is extremely time critical, however.

For charts that deal with time sensitive data, a much better approach is to permit the server to trigger updates instead of waiting for the client to request updates. This approach is generally called "server-push". In this case, our client application opens a socket to the server, which remains open waiting for additional information to arrive.

One of the KavaChart datafeed beans is specifically designed to deal with this situation. The SocketListener bean opens a socket to a specified server, using a specified port, and listens for lines of data. Like all other datafeed beans, SocketListener then sends update notifications to any DataFeedListener that has registered with the datafeed bean. Here's what the body of our application looks like:

```
import java.awt.*;

import com.ve.kavachart.beans.chart.*;
import com.ve.kavachart.beans.data.*;

public class SocketChart extends Frame {

      public SocketChart(){
            ChartBean chart = new LineChart();
            SocketListener data = new SocketListener();
            data.addDataFeedListener(chart);
            add(chart);
            setSize(500, 300);
            setVisible(true);
      }
      public static void main(String[] args){
            new SocketChart();
      }
}
```

Pretty easy. We're just using SocketListener defaults (open a socket to localhost, and listen to port 830), so no additional setup is required. SocketListener (and Java) takes care of the details of Socket I/O and reading lines of incoming data.

The server side isn't quite as easy to deal with, since it involves creating a server application that listens for connections and sends updates from your original data source. Fortunately, Java provides some nice facilities here, so we furnished a simple server application that just happens to work with our standalone client app. In the "misc" directory, you'll find an application called Broadcaster that does the trick. Broadcaster sends out sine and cosine curves, so you can see that we're really getting some recognizable data at our client. Step one is to launch Broadcaster:

```
java Broadcaster
```

The next step is to launch our client, which can also be found in the misc directory:

```
java SocketChart
```

You will see some notification from Broadcaster that a connection was established, and SocketChart starts to plot sine and cosine waves that correspond to incoming data. Our SocketListener bean is set up to send a maximum of 50 observations, so it will start scrolling the data when it reaches 50 data points.

What happens now if we launch a second copy of SocketChart? You'll see that the new copy of SocketChart updates right along with the old copy. In fact, you can have as many copies of SocketChart running as you can stand to see. Each one will respond to updates coming from the single Broadcast application.

This is a very powerful paradigm for creating custom charts with real time data. With very little burden on your server, you can service a large number of clients, each one reflecting the latest changes in data as they arrive. Also, because of the multi-threaded nature of KavaChart's data model, your client application is completely "alive" for editing, resizing, highlighting, etc., while it awaits more incoming data.

We implemented this example with an application instead of an applet, but the approach is the same for an applet. Keep in mind, though, that Java's SecurityManager might prevent an applet from opening sockets to a machine other than the one serving KavaChart classes. You can overcome this limitation by creating a signed version of your applet.

KavaChart Beans And Swing

You have undoubtedly noticed that KavaChart's chart beans and examples are based on the "java.awt" package, rather than the newer Swing package. While most applet environments lack support today for the Swing GUI, most new client applications are being written to use a lightweight component architecture.

It's quite easy to implement KavaChart beans in either a Swing or AWT environment. In fact, KavaChart's internal core charting classes are set up to automatically detect the capabilities of your Java Runtime Environment (JRE) and optimize KavaChart's feature set to take best advantage of what's available. For example, if you use KavaChart graphs in an environment that has Java2d available, the charts can use gradient fills and user defined textures. The same chart can run in an AWT environment, but the gradients and textures will be replaced by solid color fills.

Unfortunately, we can't do this kind of auto-detection and conversion at the top level. KavaChart beans are either "Panel" subclasses or "JPanel" subclasses. To

convert the default beans to JPanel subclasses, modify "com.ve.kavachart.beans.chart.ChartBean" as follows. Change this line:

```
public abstract class ChartBean extends java.awt.Panel
       implements Serializable, DataFeedListener,
              PropertyChangeListener {
```

To this:

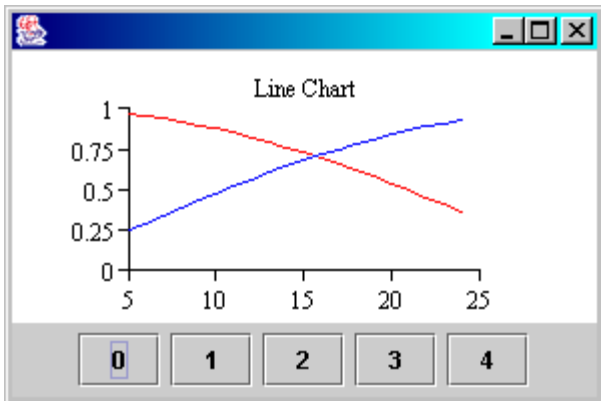
```
public abstract class ChartBean extends javax.swing.JPanel
       implements Serializable, DataFeedListener,
              PropertyChangeListener {
```

Then recompile "ChartBean.java". This will translate the entire collection of charting beans into lightweight components. Use these components the same way you would use any Swing component. Here's a sample JFrame that draws a simple chart bean:

```
import javax.swing.*;
import com.ve.kavachart.beans.chart.*;
import com.ve.kavachart.beans.data.*;

public class SwingSineBeanFrame extends JFrame {
       ChartBean chart;

       public SwingSineBeanFrame(){

              getContentPane().setLayout(new java.awt.BorderLayout());
              chart = new LineChart();
              getContentPane().add(chart, "Center");

              JPanel p = new JPanel();
              for(int i=0;i<5;i++){
                     JButton b = new JButton(Integer.toString(i));
                     p.add(b);
              }
              getContentPane().add(p, "South");
              //add our data feed
              SineWaveGenerator swg = new SineWaveGenerator();
              swg.addListener(chart);
              swg.start();
       }

       public static void main(String[] args){
              JFrame f = new SwingSineBeanFrame();
              f.setSize(300, 200);
              f.setVisible(true);
       }
}
```

This code is very similar to our previous AWT examples. In fact, all the basic concepts are identical. DataFeeds are agnostic about the GUI, and the overall

charting and drawing properties are completely independent of the GUI environment. Here's the chart created by our code:



Appendix

A

Headless Server Operation

The Java graphics environment uses native code that varies from one operating system to the next. Unix servers that lack an X Windows console and have a version of Java lower than 1.4 require special installation steps to generate images.

To generate an image, KavaChart employs various classes in Java's AWT package. Java's AWT package maps Java abstractions into specific operating system methods by using something called peer classes. These peer classes are usually created using native code (hence "native peer methods") for performance purposes. Windows implementations use code that calls the Windows GDI, Macintosh implementations the Mac Toolbox, and Unix implementations use Xlib

This generally requires actual display hardware of some sort on the server, even though nothing will actually appear on the screen. For Windows and NT servers, this isn't a problem. Unix servers must have access to an X-windows display. The display needn't be local (although performance may degrade if it's not), and it can even be a "virtual" display, such as the xvfb (X Windows Virtual Framebuffer), which can be freely downloaded as an RPM from most Linux sites.

Links to download various binary versions of xvfb can be found at the Visual Engineering web site at :

http://www.ve.com/kavachart/solutions/xwindows.html.

Once you have created an X windows environment, there are some additional issues you may need to address:

- When making a connection to an X server, one specifies a display number and a screen number, in addition to the hostname of the server. The default connection is usually "localhost:0.0", which means "the X server running on localhost, for display 0 and screen 0". This default is

usually changed for a particular shell by modifying the environment variable "DISPLAY". For example, "setenv DISPLAY goldfish:1.0" would change the default X server to the one running on the machine "goldfish" for display 1, screen 0. Most X applications (also called "client processes") also support a -display flag, which lets you override the DISPLAY environment directly.

- X Servers have a modicum of security built in, so that "foreign" clients can't connect to an X server unless permission is granted, using the xhost program. Consult your system's documentation for xhost specifics, but usually "xhost +" will instruct the default X server to permit connections from any other host or process.

- Usually, xvfb is started as the server for DISPLAY localhost:1.0.

- Some X servers default to a 256 color PseudoColor Visual model, which may yield poor color management when generating images.

- You can set up your X windows (and xvfb) configuration to be set at startup by modifying the appropriate files in /etc/rc*. Systems vary in startup configuration, so leave this task to your system administrator.

- If you're using your local display's X Server to make sure everything's working, and then log out, you'll be unable to connect to that server after logging out.

Another way to provide a graphics environment to your Unix server is to use PJAToolkit, from ETEKS.com. This freely available download provides a pure Java implementation of the graphics support classes normally provided by native peers. The ETEKS web site provides download and installation instructions in the French and English languages. Our users report that the performance of this tool is comparable to that of native classes in most environments.

JDK 1.4 implements a "Headless Support" option along with new GraphicsEnvironment methods, and can run without the X windows requirement. To use this option , the following property may be specified at the `java` command line:

```
-Djava.awt.headless=true
```

You can also specify this property by using environment variables or configuration files. Some application servers may also provide administration tools to set this property.

If the “java.awt.headless” property is not set to true, the server will throw a “HeadlessException”, new to JDK 1.4.

See Sun's online documentation for more information about headless server operation

Appendix

B

Financial Market Charting

Many, if not most, financial market applications require some level of customized charting. KavaChart includes a flexible financial combination chart that will meet most charting requirements. You will almost certainly need to add some Java code to tie your data sources into this chart. This appendix describes the properties and overall structure of this pre-built chart object.

People who deal with financial market data regularly are accustomed to using charts to absorb and interpret the large amount of data these markets generate. While the charts may seem pedestrian at first glance, in fact they are quite different from typical charts in many ways.

Financial market data is inherently discontinuous. Trading doesn't generally occur 24 hours a day, 7 days a week. Instead, markets have discrete trading hours, and each market has a unique set of holidays and other non-trading hours.

However, users want to visualize market data as if it were one continuous stream of observations, and they want the axes to describe the time periods as if time only exists for trading hours.

Trading data representations are also somewhat more complicated than for other applications. Users want to combine open, high, low, and close data within a single candlestick or OHLC. You may want to overlay special symbols that describe buy or sell indicators, statistical information, or technical studies. You may want to mark stock splits and reverse splits.

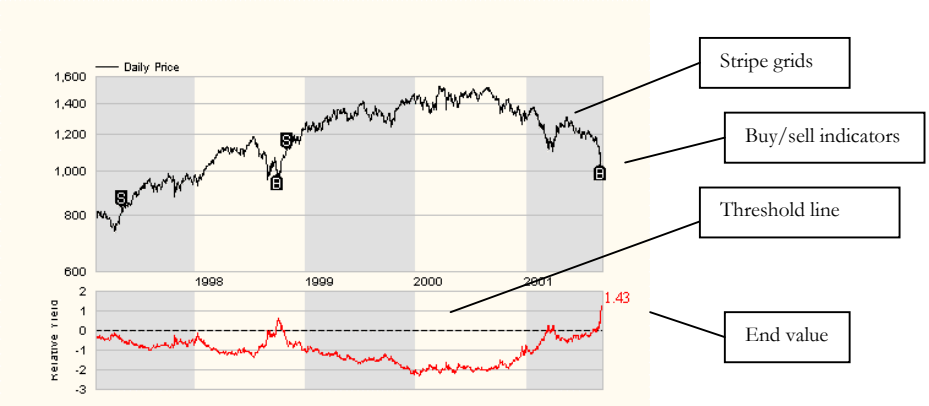
Finally, financial data is typically displayed in multiple Y Axis scales. One scale may hold pricing information. Another might contain volume information. A third scale could hold the range of numbers for a technical study, like a Bollinger Band, or a Moving Average caculation.

Fortunately, KavaChart's flexible architecture is able to accommodate these requirements and also let you add your own customizations.

KavaChart's "kcfinance" package contains a small set of classes that combines other KavaChart classes into a finance oriented combination chart called "CandlestickComboChart". This chart acts as a container for a set of "ChartWindow" objects, each of which contains a Y axis and one or more data representations (candlestick, line, stick, etc.). The overall container manages chart window positions, legends, and data distribution. For a better understanding of how KavaChart's core components are arranged, see the preceding chapters on "Core Components".

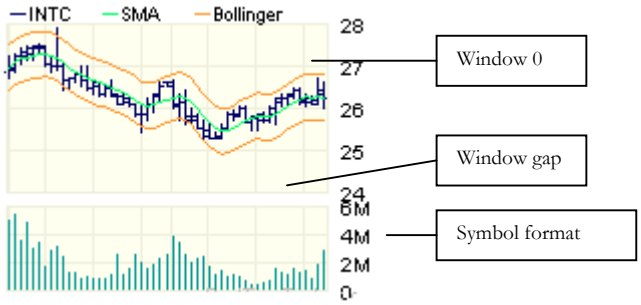
Chart Samples

These example charts were created using "kcfinance.CandlestickComboChart":

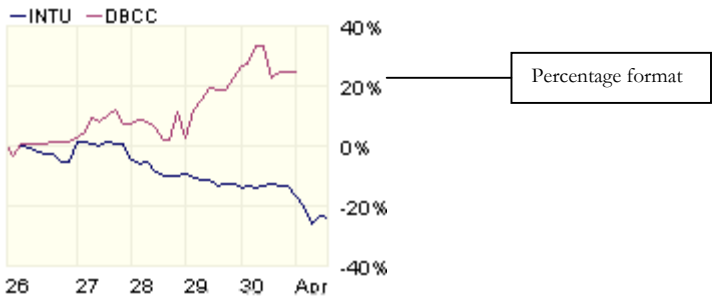


This chart uses buy and sell indicators, a threshold line, and a last-value label. The X axis is shared among two windows; one that describes pricing behavior, and another that shows a technical study.

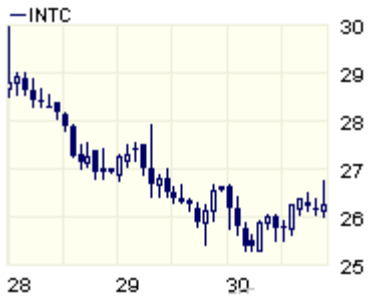
Like all charts created with CandlestickComboChart, the multiple windows in this chart share an X axis.



This chart also contains 2 windows. It has a shared legend, Y axes on the right hand side of the chart, and more conventional gridding. The X axis is not displayed. Pricing behavior is described with an OHLC class, which combines 4 observations into each unit (open, high, low, and close prices).



This chart uses a single window with 2 series. The data is described as percentages removed from a starting point. The X axis shows the behavior of "com.ve.kavachart.parts.FinanceAxis", which displays discontinuous market data as a continuous stream.



Another chart created with CandlestickComboChart, this graph shows open-high-low-close data as candlesticks.

These charts were created using "kcfinance.ChartImageBean", and "kcfinance.ChartApplet". These classes use KavaChart's applet and server imaging bean architecture to exploit the capabilities of CandlestickComboChart. As with other applets and server beans, the properties are shared, and the source code is nearly identical. In our experience, most users choose to use the server imaging bean because this applet is somewhat larger than most other KavaChart applets. However, for design and testing purposes, you may prefer to use the applet, since the applet parameters are identical to the server bean's properties. See the chapters on applet and server bean architecture for more information about how these objects are related.

Properties for Finance Charts

The following table describes properties for "kcfinance.ChartImageBean" and "kcfinance.ChartApplet":

Property	Value Type	Effect
dataset0Type	ohlc|line|stick| candle	Sets dataset 0 (0 through 39 available) to one of: open-high-low-close, line, stick, or candlestick data representations

dataset0Window	Integer	Assigns dataset 0 to a particular window. Windows are numbered sequentially starting with 0.
dataset0VisibleInLegend	true|false	Determines whether an icon for dataset 0 will be shown in the chart's legend.
window0HeightFactor	number	Value from 0.0 to 1.0 describes how much of the total charting area should be devoted to window 0.
window0Color	Color	The color for window 0's plotarea
window0AxisFormat	symbol|percentage	If set, this property changes the Y axis format for window 0 to be percentages, or a shorthand symbol (1M for 1 million, for example)
window0AxisOptions	List	Similar to yAxisOptions, etc. from the standard applets and imaging beans
window0AxisStart	number	Similar to yAxisStart from the standard applets and imaging beans
window0AxisEnd	Double	Similar to yAxisStart for standard applets and image beans.
window0AxisGridColor, tickColor, titleString, titleFont, etc.	Various	All "window0Axis" options apply to the Y axis for window 0 as the "yAxis" options apply to a conventional applet or image bean.
window0XLabelsVisible	true|false	Determines whether X axis labels should be displayed for window 0.
window0DisplayEndValue	true|false	Determines whether the final values of datasets in window 0 should be displayed at the right hand side of the chart. The first example above shows this property in use.
endValuePrecision	int	Integer precision for the end value
endValueUsesDollarSign	true|false	Determines whether the end value should be preceded by a dollar sign.
endValueUsesPercentageFormat	true|false	Determines whether the end value is displayed as a percentage.
window0ThresholdLineValues	List of numbers (e.g. "23.0,32,5")	Threshold lines will be displayed at these locations in window 0.

window0ThresholdLineColors	List of colors	Color assignments for the threshold lines described in the previous property
window0ThresholdLineStyles	List of SOLID|DASH, e.g "SOLID,SOLID"	Line style assignments for the threshold lines described above.
indicatorThreshold	Number	Values in dataset 1 that pass within this amount of threshold line 0 in window 1 will trigger buy or sell indicators along dataset 0. This behavior is shown in the first chart example above.
stripeGrids	true|false	If true, the X axis will use alternating colors, like the first chart example above. The default behavior is shown in the other chart examples.
gap	Number	This is a percentage value (e.g. 0.05) that describes how big the gap between chart windows should be.
dayFormat, monthFormat, yearFormat, hourFormat	Date format string	This format string describes how time stamps on the X axis should be formatted. See the applet and server bean sections above for more information about date format strings.

Financial Data

These charting objects are not designed to use conventional KavaChart data acquisition. Because the charts require non-conventional data (e.g. candlestick data), and are typically connected to a live data store, we recommend that you create your own implementation of the "getDataset" method to use ChartImageBean and ChartApplet.

Note:

CandlestickComboChart assumes that all datasets are "aligned". That is, every dataset has the same number of observations, and all X values are identical.

Datasets for these chart classes consists of one or more Y values for each observation, together with an X value describing the date. The date value is like other KavaChart time series charts, the number of milliseconds since Java's epoch. You can obtain this values from java.util.Date or java.sql.Date like this:

```
double dateValue = (double)(myDate.getTime();
```

You can add datasets containing conventional Datum classes or CandlestickDatum classes by implementing the "getDataset" method. You'll find more documentation about "getDataset" in the chapter on programming with server beans.

The default implementation in ChartImageBean and ChartApplet reads data from a text file that looks like this:

```
Daily Price and Relative Yield Data

DATE   PRICE RELYLD

02/04/1997  789.26      -0.25
02/05/1997  778.28      -0.20
02/06/1997  780.15      -0.22
02/07/1997  789.56      -0.26
...
```

Contact VE's consulting group (consulting@ve.com) for a quote on obtaining a custom data feed connection to your own data sources.

Unique Behaviors

Because these charts are somewhat different than KavaChart's other charts, they have some unique behaviors. These are outlined below:

X Axis behavior – The X axis is a "com.ve.kavachart.parts.FinanceAxis". This axis attempts to place labels at "intuitive" increments for both intraday and interday trading periods. Small charts will display fewer labels than large charts. This axis can also be set to use stripes of varying colors rather than grid lines.

Legend behavior – although this chart uses a standard KavaChart "LineLegend" class, the datasets that appear in the legend can be set by property. Also, the legend always appears at the top left corner of the chart. Spacing for the legend and chart windows is calculated automatically.

Plotarea behavior – the plotarea left and right property values are used to set the right and left margins of this chart, but the individual window plotarea values are determined automatically by the "windowHeightFactor" value.

Other Considerations

These charting components can be used to create a high performance financial chart server. However, like any high-demand server application, you will need to do extensive testing to find and eliminate performance bottlenecks.

Data Server

In our experience, financial information data servers are a frequent source of charting application performance bottlenecks. Many data servers are single

threaded internally, and some consolidate minute-by-minute bar data into larger periods for interday and intraday history requests.

By building an intermediate data cache for the most frequent queries, you can minimize the effects of a slow data server.

Our experience has also shown that local storage of pre-organized historical data usually provides the fastest response times.

Visual Engineering's consulting group can advise you on creating multi-tier data access environments for optimal charting performance.

Image Generation

Because Java makes use of native peer classes when generating images, some environments can benefit tremendously from hardware graphics accelerators. This is true even though nothing is visible on the server while images are being generated.

In a high traffic environment, you may also want to employ chart image caching to avoid regenerating frequently requested information.

As with any application, adequate memory and fast disk drives are important performance considerations. Since image generation is intrinsically memory intensive, this is particularly true for server-side image generation.

Index