VISUAL ENGINEERING

KavaChart Developer Documentation

KavaChart ProServe Users Guide

VERSION 5.0

KavaChart ProServe Users 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 Server

KavaChart ProServe is specifically targeted to programs that run on an application or web server to generate image output. This product includes tools for extending the "tags" you're using in a Java Server Page (JSP), as well as

a collection of flexible chart objects that will translate data into chart output for a variety of environments.

These chart "beans" are encapsulated objects that will translate a set of property values into an image file. They're the basis for KavaChart's server chart imaging operations, so it's useful to understand how they operate.

These beans use a set of string pairs to describe the overall chart appearance, defining things like title strings, colors, and legend definitions. These objects also include imaging code and server cache management logic to generate almost any image format. They can take data from a variety of sources, including definition strings compatible with KavaChart applets, or from classes that implement the simple KavaChart DataProvider interface.

These beans, or the KavaChart custom tags that use the 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.

Note:

Although this document refers to KavaChart ProServe output using the word "image", you can also produce other formats, such as Macromedia Flash output or Scalable Vector Graphics.

The KavaChart custom Tag library is probably the best way to use these server objects 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 Enterprise Edition API gives you all the tools you need to build your own server charting tools.

The KavaChart Wizard

KavaChart ProServe licensees can design charts by hand, using a text editor and documentation about KavaChart's server object properties. It's a lot easier, though, to design your charts visually, using the KavaChart Wizard. This on-line tool provides a graphical interface for designing chart appearance, and the ability to combine your local data sources with chart designs. The Wizard produces complete output templates for a variety of server technologies and data sources.

The KavaChart Wizard is on on-line tool available to all KavaChart users, with enhanced capabilities available to licensed users and maintenance subscribers.

Chapter

2

Quick Start Guide

If you're the kind of user that wants to see results ASAP, follow this quick start guide to get KavaChart busy producing images right away. You can then poke around with the examples and get an overall understanding of KavaChart ProServe, and then come back to this guide for a more detailed reference.

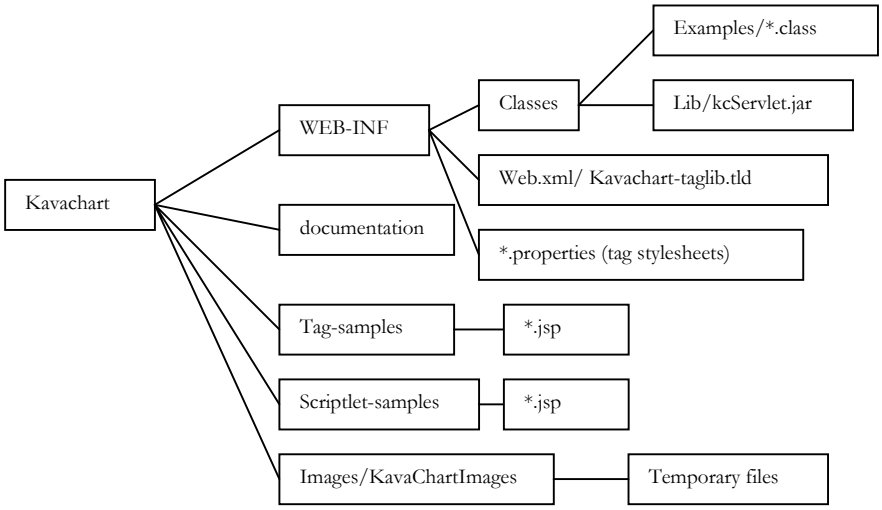
The first step in using KavaChart ProServe on your own server depends somewhat on your server. If you're using ColdFusion MX, scan through this chapter, and then check Appendix B for information about setting up your server's configuration files and using ColdFusion data sources.

Getting Started

If you're using a standard J2EE server that supports ".WAR" web archive files, the easiest way to see KavaChart ProServe in action is to let your server install "kavachart.war". Generally, you can simply place this file into your server's "webapps" directory, and then browse the "/kavachart" application.

This "application" is really just a collection of Java Server Pages that demonstrate a variety of ways to use KavaChart. Feel free to modify these JSP files to experiment with KavaChart and get a feel for how it might be used with your application.

The application hierarchy created by "kavachart.war" looks like this:



The top-level directory index contains links to various JSP samples, summary HTML documentation, and the on-line Chart Wizard.

If you're not using "kavachart.war" to create an initial installation of KavaChart, you'll need at least "kcServlet.jar", which contains the Java classes and other resources used in KavaChart ProServe.

If you also want to use KavaChart's tag library (recommended), you'll need to integrate "web.xml" with your own, and add "kavachart-taglib.tld" to your server's configuration information. You can find more detailed information about this in Chapter 8.

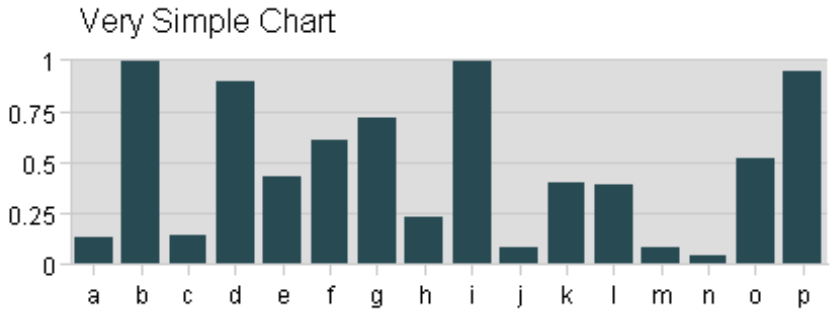
Sample Code

The demos are divided into two categories: JSP scriptlets and JSP tag examples. You can also use KavaChart ProServe to create your own image-generating servlets. In general, the tag library simplifies your page creation and maintenance tasks dramatically, but you're free to choose the technique that best suits you application.

Here's a minimalist example from the tag library examples:

```
<%@ taglib uri="/WEB-INF/jsp/kavachart-taglib.tld" prefix="chart" %>
<jsp:useBean id="foo" class="examples.RandomNumberDataProvider" />
<chart:streamed style="WEB-INF/paramchart.properties"
                chartType="columnApp"
                dataProviderID="foo" />
```

This code produces a chart that looks like this:



Styles for this chart are taken from the file "WEB-INF/paramchart.properties". Data is provided by an example DataProvider called "RandomNumberDataProvider, available in source code in the "WEB-INF/classes/examples" directory.

The generated chart includes tooltip labels that describe the charts data as you pass the cursor over each bar.

Note:

If your server is generating an error instead of a chart, and you're using a Unix server, you may need to set the System property "java.awt.headless" to "true". See Appendix A for more information about this important property.

Here's a simple JSP scriptlet that generates a similarly simple chart:

```
<%
 //add some random data here:
 for(int i=0;i<5;i++){
        double d = Math.random();
        chart.accumulateProperty("dataset0yValues", Double.toString(d));
 }

 //set a few other properties to make the chart look nice
 chart.setProperty("width", "400");
 chart.setProperty("height", "250");
 chart.setProperty("titleString", "hello, world");
 chart.setProperty("writeDirectory", application.getRealPath("/images"));
 chart.setProperty("yAxisOptions", "gridOn, minTickOn");
%>
<center>
<img src="/images/<%= chart.getFileName() %>">
</center>
```

Notice that KavaChart server beans operate by setting various string property pairs, described in detail elsewhere in this document. The same property pairs are used for chart tag "styles", which are really just Java properties files.

In the scriptlet example, one property in particular merits close examination: "writeDirectory" describes where this server object should write a chart image. In our example, we're using the servlet utility method "getApplicationRealPath" to locate the images directory in a portable way.

To produce the chart, the scriptlet calls the object method "getFileName()", and places that file name in an HTML <IMG> tag.

If you get an error message instead of a chart when you run this JSP, and you're running on a Unix-based server, you might need to set up your server's graphics environment. If you're running JDK 1.4 or newer, try adding this line to the top of your scriptlet, and then restart your server:

```
<% System.getProperties().setProperty("java.awt.headless", "true"); %>
```

If this workaround gets your charts running, add "-Djava.awt.headless=true" to your server's startup script.

If you're seeing charts, go ahead and experiment with the demo samples. You should find something similar to your own requirements, and be able to see fairly quickly how to integrate KavaChart ProServe objects into your own application. The Chart Wizard can be used to generate properties files or styles to replace the versions provided in the demos.

Use this manual and the KavaChart Wizard as a reference guide to refining your own application's charting capabilities.

Chapter

3

Terminology 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.

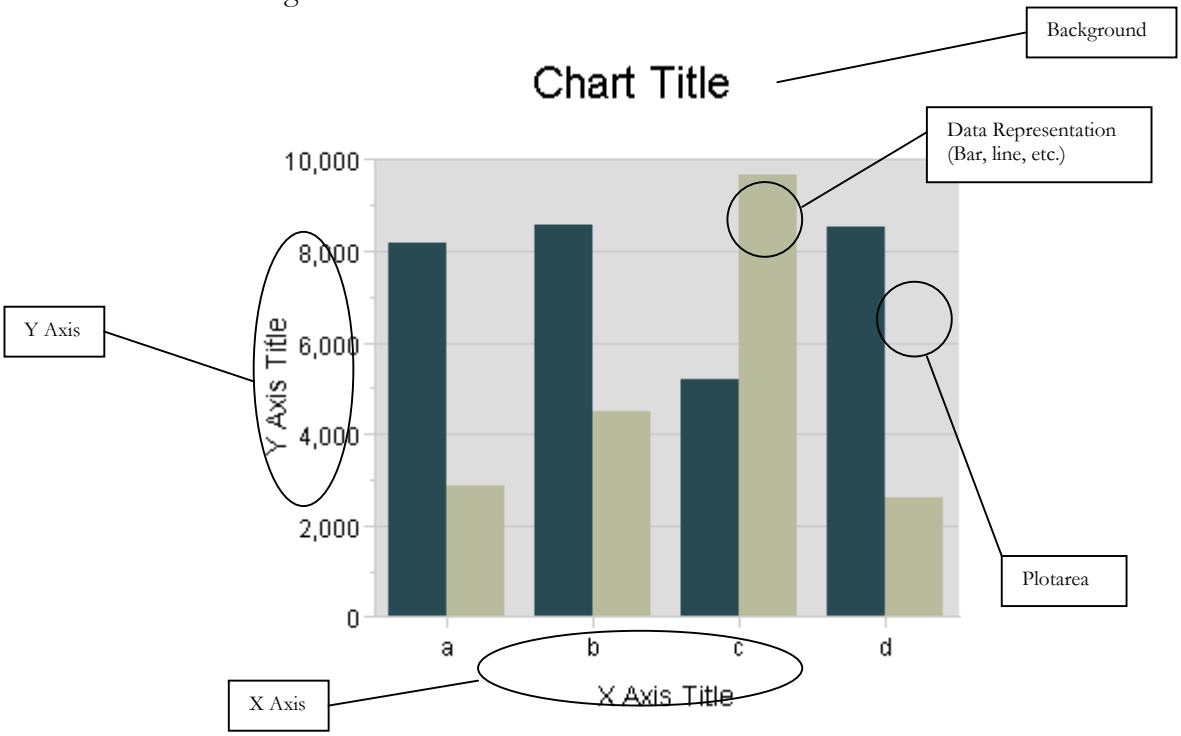
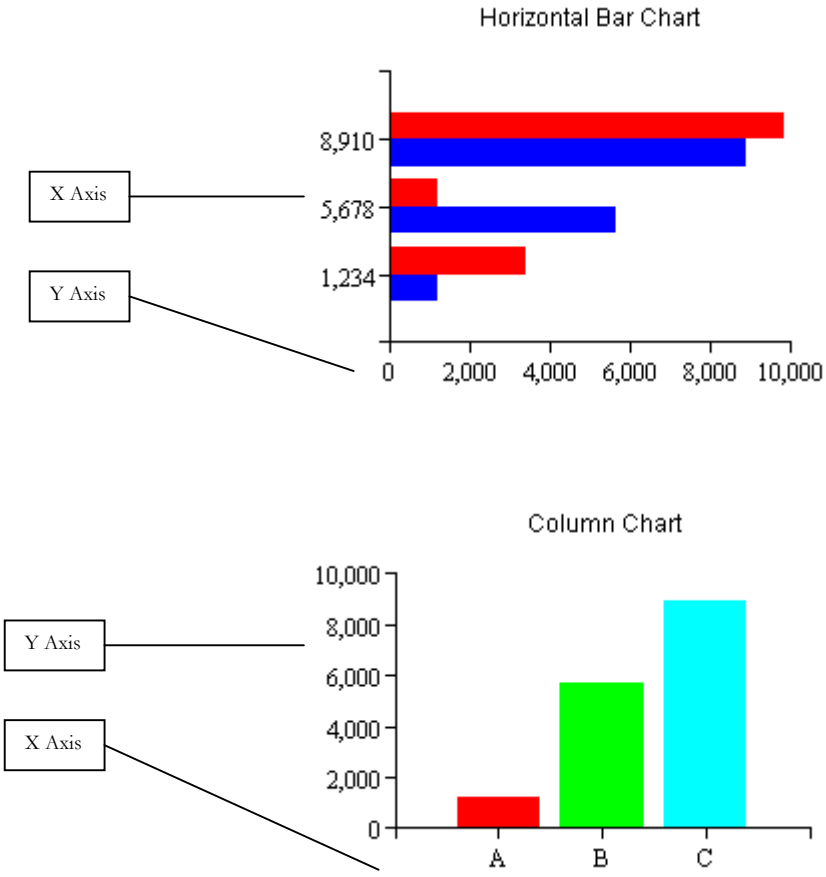


Chart Parts

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.

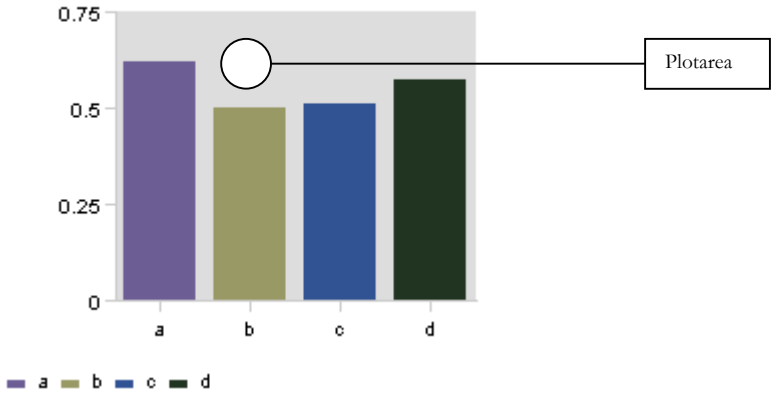
X axes scale for Dataset X values. For some charts, such as a Column chart or a Stacked Column chart, the X axis distributes the data evenly along the X axis, regardless of the Dataset's X values.

Several different types of Axes exist in KavaChart charts. 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

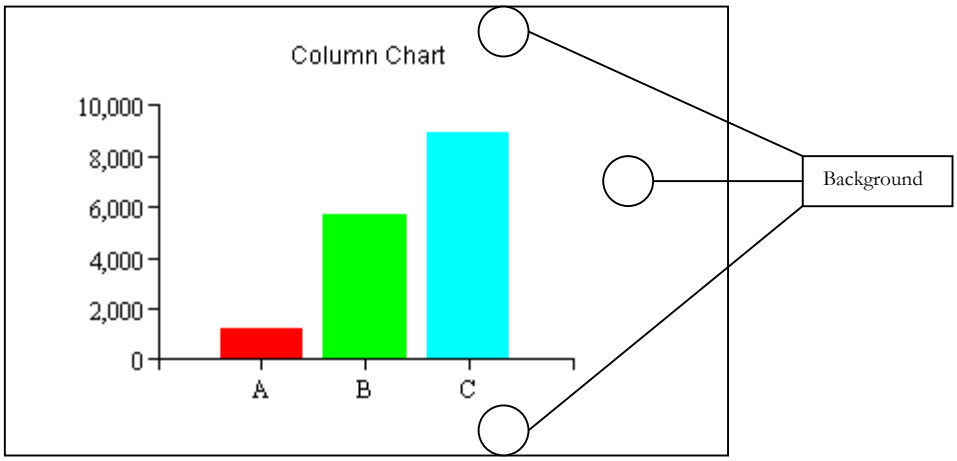


A Plotarea is the region bounded by an X and a Y Axis, which contains a DataRepresentation (such as a Line, Bar, Area, etc.). A Plotarea has a size and location determined by the upper right and lower left corners. The values that define the Plotarea size and location are percentages, relative to the 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.

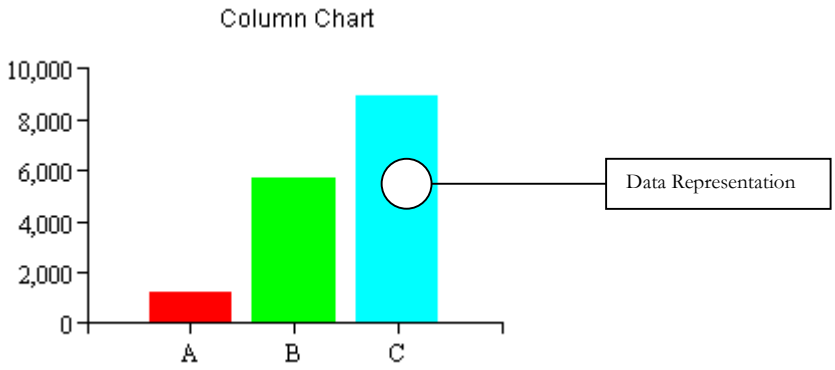
By changing the size and location of your Plotarea you implicitly change the size of your chart's margins. All Axis and DataRepresentation geometries will automatically adjust to accomodate your Plotarea definition.

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

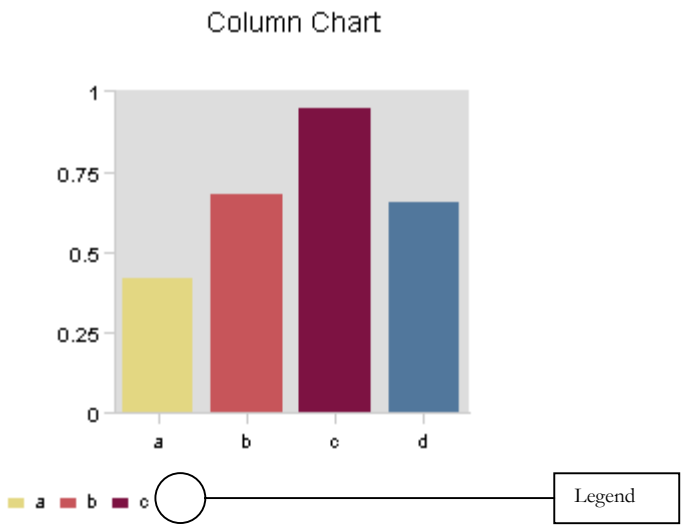


A DataRepresentation is the name KavaChart uses for a variety of objects. These include Line, Area, Bar, and Pie, as well as other more specialized DataRepresentations. These items visually describe a group of Datasets. For example, bar DataRepresentations exist that draw multiple series horizontally 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.

Most server applications should obtain data from a "DataProvider" class, which supplies Dataset classes to chart definitions. See the sample chart pages for examples of DataProvider, Dataset and Datum classes in action.

Chapter

4

KavaChart ProServe Imaging Operations

This chapter discusses server based charting in general, and gives a broad overview of KavaChart's factory server components. It also discusses some operational details, such as server image-cache management.

Benefits and Drawbacks to Using Server Objects

Since KavaChart lets you create your charts using applets on the client or using server objects, let's examine the benefits and drawbacks of using your server to create chart images.

Complete Control

Control is the overriding issue for most developers that want to create chart images at the server. These developers don't trust that every client machine and browser will be configured in a way that ensures that applets will run consistently. And if the applets don't run consistently, then the charts won't always look the same on every machine. You can be reasonably certain that an image generated at the server will look the same on every client. It doesn't matter whether they're using the latest browser revision on Linux, an old Macintosh at home, or even a wireless device with an experimental browser. Images will always look the same.

Images Can Be Saved

One distinct advantage of server-generated images is the ability to treat these images like any other image at the client. Users can copy images to the clipboard, save the image into a file, and so on. Since applets are live program code, they can't be copied and pasted like image data.

"Local" Data Stores

A perceived advantage of server objects lies in their nearness to enterprise data storage. Most application and web servers have the means to access databases in an efficient way, using pre-existing connection pools, database drivers, and other tools that are better suited for server use than applet use. Even though you can generate data to populate applet parameters, generating a chart image seems like a natural extension of other server activities, like building tabular displays.

External Requirements

Often, the developer doesn't get to choose where a chart is rendered. Someone else has already decided that a document must include dynamically generated chart images. Sometimes this is just the result of misinformation or bad experiences with poorly written applets.

Centralized Imaging Load

By generating your charts on the server instead of sending applet definitions to your clients, you're assigning the task of imaging every chart to your application or web server. It might not seem like a big deal, but the compute load can be significant for busy servers. Each image is first rendered in memory. The raw image memory is 24 x 500 x 300 bits, or nearly half a megabyte for a small chart image. Add to this whatever overhead is involved in managing fonts, colors, transparency, antialiasing, imaging encoders, etc., and you get some idea of what's going on behind the scenes.

Even though modern application servers and virtual machines are very good at load management and memory management, applets give you a way to distribute these tasks automatically.

Servers and Image Streams

Many applications require a mixture of text and graphics information to be sent from server to client. In a simple HTML page, one HTTP request retrieves the overall text for a page or document. The server identifies the contents of the request as HTML or text, and streams the information to the browser. This text includes markup tags like this:

```
<img src="logo.gif">
```

This tag causes the browser to make another HTTP request for the image file. In response, the server sends the contents of the image file, with a header that identifies the content-type as image information.

The browser decodes this byte stream with the appropriate image decoders, and displays the image according to the overall page layout, which was defined by the initial HTML text.

Most server transactions send a single content type

For most web developers, this process seems completely obvious. However it's important to keep it in mind when you're developing applications that generate dynamic graphics, because pages with dynamic content usually have dynamic text as well as graphics.

Servers can use KavaChart directly to stream image bytes to clients. For example, if your server supports Java servlets, you can use com.ve.kavachart.servlet.ChartServlet to create a stream of image bytes. This servlet sets the output's content-type to the appropriate image format, builds an in-memory version of the chart, encodes the data in some compression scheme (such as JPEG, PNG, etc.), and sends it directly to the browser in a single HTTP transaction.

However, the result is a chart image without any textual context. No page titles, no descriptive tables, no explanation, no formatting, just a bare chart image. This isn't particularly useful for most real world applications.

It's also possible to create separate server objects that generate the text content and the graphics content. The first component would generate output with a content-type of text, and the second would create output with image content. If you're retrieving information from a database, however, this means you'll have to do multiple database transactions to get information you could retrieve with a single transaction.

You could also explore creating a more exotic document that has multiple content-types, with content type delimiters and mixed multimedia headers. Of course, you'll also have to maintain a fairly complex code base in the future.

Finally, you could create a communication mechanism to let your text-generating object communicate with your chart-generating object. In a typical multi-threaded server environment, however, you'll have to create a significant amount of code to make sure each set of text-plus-image requests is processed independently.

Note: In a JSP, KavaChart's Tag library solves most imaging problems

If you're generating images for a Java Server Page, use KavaChart's custom ChartTags to automatically manage image generation and retrieval. See the chapter below for more information about ChartTags.

Server-side image cache

One solution to this problem is to create the text content and image content at the same time, but store the image content on the server to be retrieved in a separate transaction.

In this case, our server object might create text output like this:

```
<html>
<p>the number is 1.432</p>
<img src="image.png">
</html>
```

When our object creates the output, it also creates the image file "image.png". The client makes another request for the image file, since it's needed to complete the page.

One problem with this approach is that it leaves an image file on our server. Since we're reasonably certain that the file will be used soon, we can remove the file with another process, such as a Unix "cron" job, or a Windows scheduled

task. We could also use a thread on our server to remove this file at some time in the future.

Another potential problem lies in the contents of the image file. If we always name our output file "image.png", then we might have problems in a busy server. If user 1 and user 2 request the dynamic page at approximately the same time, it's possible that user 1 will get the "image.png" file created for user 2.

Image Cache: The KavaChart Approach

KavaChart's server charting objects give you a variety of ways to create image output, but the best mechanism for most applications uses something like the image cache approach described above.

KavaChart server objects create chart images based on a set of properties. Each property has a name and a value. For example, the property "titleString" might have a value of "hello, world". The property set, combined with charting data, defines the complete chart image. This image can be created in memory and retrieved as an array of bytes (encoded in some standard format, like JPEG or PNG), or it can be written to a server file.

By default, the name of the server file is based on an algorithm that considers all the chart properties to ensure that unique charts will have unique file names. You can override this naming convention, but you'll have to deal with the possible problem of users contending for the same filename, described above.

KavaChart also employs a special server object, CacheCleaner, that will periodically remove files that are older than some specified age.

If you're using the Tag library, images can be cached in memory (the *streamed* tag), and then automatically cleared from memory when the request has been fulfilled.

Since KavaChart's unique filenames are based on the chart's properties, identical charts will have identical names. By default, KavaChart calculates the filename before creating a chart file, and won't generate an image file if it already exists in a cache directory. This means you can use the KavaChart naming convention as an automatic image cache for frequently requested charts.

For example, if you have a set of company-wide reports that changes every day, these charts will be generated into server files when the first user requests them. For every other user, the KavaChart objects calculate the filenames, examine the cache, and use the pre-existing image files. You could run CacheCleaner automatically, or run it in a Thread that removes files older than 24 hours, and automatically keep only the image files needed for today's reports. This approach can dramatically boost the performance of an image server under the right circumstances.

Even if you're using KavaChart to retrieve image byte arrays, you can use this approach. KavaChart objects will write the image files upon the first request,. Afterwards, the objects will just read the data from the files to construct image

byte arrays, rather than doing the more compute-intensive imaging and encoding tasks involved in initial chart rendering.

Various Imaging Approaches Using KavaChart

You can create server images with KavaChart in a variety of ways. KavaChart ProServe includes a range of server objects that can be used directly, in a servlet or scriptlet, or you can use the KavaChart factory ChartTag classes to encapsulate the details of generating image data.

The KavaChart server bean approach, described briefly above, uses paired properties to define a chart. You can also extend these objects to add overlay drawing, data acquisition, and other common behaviors. These activities are described later in this chapter.

KavaChart ProServe objects offer many advantages over creating your own server chartig tools: They've been well tested, and are used in thousands of installations. They support virtually all the popular image encoders you might need. They include support for hyperlinking (client side imagemaps) and tooltip labeling for all popular browsers. They've also been around long enough to go through several performance enhancements and revisions. They even include method stubs to let you install your own image encoders or other output mechanisms.

Perhaps equally important is that KavaChart ProServe objects are 100% compatible with KavaChart applets. The same parameter pairs used by KavaChart applets can be redeployed as server object properties. This gives you a way to easily switch between server chart imaging and client chart imaging.

If you're using Java Server Pages, this compatibility is underscored with KavaChart ProServe's ChartTags. These tags let you switch from server-side imaging to client-side imaging (applets) with a single key word.

KavaChart Server Objects and JSP Scriptlets

Java Server Pages provide a convenient way to create dynamic content within an HTML context. These pages are a hybrid of HTML statements and Java program code. The first time a JSP is accessed, the server compiles it into a servlet. Subsequent accesses call the appropriate "service" methods of this servlet.

JSPs Emit Text, Not Graphics

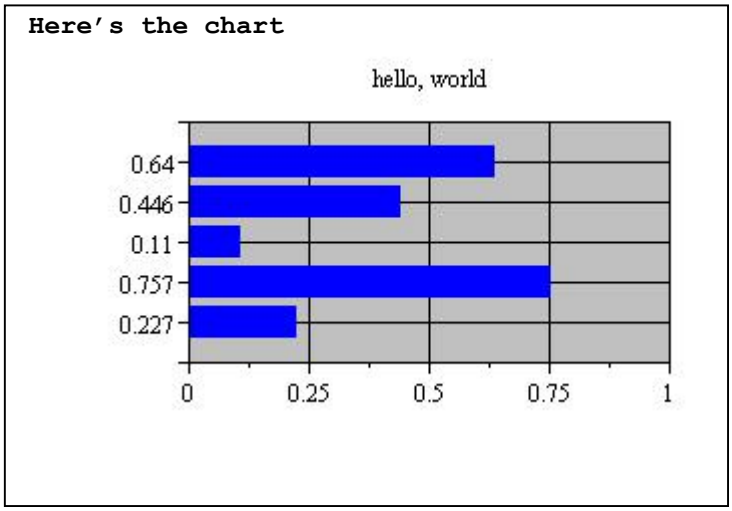
Because JSPs are designed to mix dynamic programmatic content into text and HTML content, they set the output's content-type to text/html. Any graphics you send through a JSP must also be text/html. To do this, your JSP will send an IMG tag as a part of its output. If you want this image to be a dynamically generated chart, the IMG tag must point to either an image file created by the JSP, or another URL that will create the image dynamically.

In the example below, our JSP creates a very simple page using a KavaChart server object to build a chart image.

```
<html>
<head><title>Simple Chart Generation</title></head>
<body bgcolor="white">
<font size=4>
<%
      com.ve.kavachart.servlet.Bean chart =
                  new com.ve.kavachart.servlet.barApp();
    //add some random data here:
    for(int i=0;i<5;i++){
          double d = Math.random();
          chart.accumulateProperty("dataset0yValues", Double.toString(d));
    }

    //set a few properties to make the chart look nice
    chart.setProperty("titleString", "hello, world");
    chart.setProperty("writeDirectory", "webpages");
    chart.setProperty("yAxisOptions", "gridOn, minTickOn");
    chart.setProperty("xAxisOptions", "gridOn");
    chart.setProperty("plotAreaColor", "lightGray");
    chart.setProperty("width", "400");  chart.setProperty("height",
"200");
%>
<p>
Here's the chart:
<br>
<img src=/<%= chart.getFileName() %>>
</font>
</body>
</html>
```

This JSP creates a page that looks like this:



The HTML portion of this page consists of some headers and a one-sentence paragraph. The graphics are generated automatically by a KavaChart server object. Because this chart displays a series of random numbers, a new image file will be generated each time the page is reloaded. Here's the HTML created by the JSP:

```
<html>
<head><title>Simple Chart Generation</title></head>
<body bgcolor="white">
<font size=4>
<p>Here's the chart:
<br>
<img src=03e2c9f00aff12534c4896a0309.jpg>
</font>
</body>
</html>
```

The only thing added to the JSP output by the KavaChart server object is the strange name of a JPEG file, "03e2c9f00aff12534c4896a0309.jpg". The client machine will request this file from the server in a separate HTTP transaction, which returns the image content.

Behind the scenes, here's what happened:

The JSP created an instance of KavaChart's "barApp" object.

The object's accumulateProperty method was called repeatedly to add data.

The object's setProperty method was called to set a title, size, plotarea color, and to turn on axis grids.

The chart's "getFilename" method was invoked, which triggered the actual chart computation:

The properties were examined and a filename was calculated that identified this unique chart.

The chart object checked the server's cache directory (set with the "writeDirectory" property) to see if the file had been previously created.

Since no previous image with this name existed, an internal chart was created. Note: if the file already existed, this would have been returned immediately to the JSP.

The properties were examined and the internal chart was modified accordingly: data was added, the title string set, plotarea color was changed, and so on.

An image was created in memory and the internal chart was drawn to the image.

Pixels were then extracted from this image, and the data was passed through a JPEG image encoder.

Finally, the image was written to the caculated filename, and the filename was returned to the JSP.

The filename was placed on the JSP's output stream, within an IMG tag, so the client can make another HTTP request for the image content.

Note that the JSP wrote a file on the server somewhere, and that the HTML points to an image somewhere on the server. "Somewhere" needs to be the same location. In our example, the JSP wrote the file in a directory named "webpages", and the HTML output assumed the file exists in the same document base as the HTML output.

Use "getRealPath" to find your images

Servlets have a convenience method that can help you write your images in the proper location. A ServletContext's "getRealPath" will return a fully qualified directory path based on a relative URL. For example, if you want to put your chart images into a directory that appears as **`http://yourserver/dynamic_charts/`** on the web, write your images into the actual directory returned by **`getServletContext().getRealPath("/dynamic_charts/")`**. Set the directory you're writing your charts into with the "writeDirectory" property.

Note:

JSP implementations use an instance variable "application" that will return a servlet context appropriate to a specific web application. If your web server supports this, you can write your images into a directory specified like this:

```
chart.setProperty("writeDirectory",application.getRealPath("char
t_cache"));
```

See the section below for more information about how KavaChart's server image cache operates.

KavaChart ChartTags and JSPs

Although the scriptlet described in the section above is fairly intuitive to a Java programmer, it's not necessarily intuitive to your page designers. The JSP will also become much more complicated when you start to add realistic data sources, such as those from database connection pools, resultset adapters, and so on.

ChartTags Hide Complexity

JSP includes a facility to define custom XML-like tags that perform Java processing. KavaChart ProServe includes a collection of Tag implementations that apply this concept to chart generation.

ChartTags use KavaChart server objects to create chart image streams, but encapsulate the processing into a simple tag. The tag does some compile-time checks to make sure the arguments are formed properly, and then expands into a part of the servlet created by the JSP.

Perhaps even more important than intuitive encapsulation, ChartTags give you a way to separate your chart data sources from your chart style to make your code more re-usable and testable. Similarly, ChartTags hide the details of chart localization and ResourceBundle application.

The example below uses a ChartTag to duplicate the functionality of our previous JSP scriptlet.

```
<html>
<head><title>Simple Chart Generation</title></head>
<body bgcolor="white">
<font size=4>
<%@ taglib uri="http://www.ve.com/kavachart-taglib" prefix="chart" %>
<p>
Here's the chart:
<br>
<chart:streamed
          style="WEB-INF/simplechart.properties"
          chartType="barApp"
          dataProviderID="foo" >
</chart:streamed>
</font>
</body>
</html>
```

Although the page looks like HTML/XML, the ChartTag becomes servlet code when it's compiled by the JSP compiler. This code creates a "barApp" server object, like the previous scriptlet. The style properties for this chart are contained in a properties file located in "WEB-INF/simplechart.properties". The data for this chart comes from a DataProvider class placed into the page context with an attribute ID of "foo".

One significant difference, compared with the scriptlet, is that the image is streamed back to the page without being written to the server's filesystem. How is this possible, since the page contains both text and graphics? ChartTags use a special servlet to store images to be streamed back when requested. The resulting HTML from this tag contains an IMG statement that looks like this:

```
<IMG
SRC="/servlet/com.ve.kavachart.servlet.ChartStream?sn=39cd
7cec8210c44f">
```

The servlet stores the image stream in memory temporarily, and deletes it after completing the request.

ChartTags are designed to be more efficient than most scriptlets. They implement optimizations to make sure re-used information persists in memory, while transient information is regenerated. They separate data from presentation in a way that makes it easier to test your data sources in a non-http context, even with chart designs, using the KavaChart Wizard. They make it harder for non-programmers to make mistakes in pages that contain charts. They also accommodate custom extensions you make to KavaChart's server objects.

If you're using JSP to generate your charts, ChartTags are probably the best way to implement your applications.

Using a Servlet for Image Content

Servlets provide more flexibility than JSPs in generating server output. In addition to text and HTML output, servlets can produce various types of media streams, including image streams. For example, it's perfectly acceptable to place and IMG tag in your HTML that looks like this:

```
<IMG SRC="/servlets/ChartGenerator">
```

This snippet assumes that "ChartGenerator" will produce a stream of bytes recognizable as image data by the browser. The image data must not include extraneous text, tables, etc., and the servlet must identify the output's content-type as some sort of image.

This approach does have some advantages. First, it doesn't require the server to generate an image file. The byte stream can be created in memory directly. It also provides some portability in situations where you might need to generate the same chart for several different pages.

Servlets Can Generate Image Data Directly

There are also some significant disadvantages to this approach. You can't take advantage of the performance boosts available from server image caches. You will also need to create separate server code for dynamic text output and dynamic image output. This might make database accesses inefficient or complicated.

The code below creates the same chart we created in our JSP example above, but sends the output as a stream of bytes.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class SimpleChartServlet extends HttpServlet {
     public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws
                  ServletException, IOException {
     com.ve.kavachart.servlet.Bean chart =
                  new com.ve.kavachart.servlet.barApp();
            //add some random data here:
            for(int i=0;i<5;i++){
               double d = Math.random();
               chart.accumulateProperty("dataset0yValues",
                        Double.toString(d));
            }
            //set a few properties
            chart.setProperty("titleString",
                              "hello, world");
            chart.setProperty("useCache", "false");
            chart.setProperty("byteStream", "true");
            chart.setProperty("yAxisOptions",
                            "gridOn, minTickOn");
            chart.setProperty("xAxisOptions", "gridOn");
            chart.setProperty("plotAreaColor",
                             "lightGray");
            chart.setProperty("width", "400");
            chart.setProperty("height", "200");
            ServletOutputStream out =
                             response.getOutputStream();
            response.setContentType("image/png");
            out.write(chart.getImageBytes());
     }
}
```

The core of this servlet is nearly identical to the JSP example above. The servlet creates a KavaChart server object (barApp) instance, and sets some properties. The drawing and imaging activities are triggered by a call to the chart's "getImageBytes" method.

The useCache Property

One significant difference between the properties in the JSP example and the properties in this example is the use of the "useCache" property. By setting this property to "false", we ensure that the server object generates a new image, regardless of whether an image exists in the cache or not.

The byteStream Property

Another difference is the use of the "byteStream" property. By setting this property to "true", we're telling the server object that we're only interested in byte streams, and that we don't ever want to use the cache.

KavaChart Server Objects and ASP

You may be surprised to find that KavaChart server objects can be used with non-Java languages in non-traditional Java server environments. The most prominent of these environments is, of course, Microsoft's Active Server Page environment, built into IIS servers.

By placing KavaChart's classes into a Windows 2000 Java CLASSPATH and loading the chart objects with ASP syntax, you can generate chart images like you can with any other Java enabled server.

The primary limitation, as of this writing, is that IIS does not have Java 2 support, so you won't be able to take advantage of some of KavaChart's advanced imaging features, like texture drawing and antialiasing. Microsoft has also dropped support for their Java Virtual Machine, so combining an application server like Apache Tomcat with .NET may be a better alternative.

Here's an example ASP that uses VBScript to build a chart:

```
<%@ LANGUAGE = VBScript %>
<%
'Create an instance of the chart object
Dim chart
Dim str
Dim chartData
Dim chartLabels
set chart =
getObject("java:com.ve.kavachart.servlet.barApp")
chart.setProperty "dataset0yValues",
         "123,453,345,654,345,654,567"
chart.setProperty "titleString",
          "Charting from ASP with Visual Basic"
chart.setProperty "writeDirectory","/Inetpub/wwwroot"
chart.setProperty "yAxisOptions",
                         "gridOn, minTickOn"
chart.setProperty "xAxisOptions", "gridOn"
chart.setProperty "plotAreaColor", "lightGray"
chart.setProperty "width", "400"
chart.setProperty "height", "200"
chart.setProperty "imageType", "j_png"
%>
<p>
Here's the chart:
```

```
<br>
<img src=/<%= chart.getFileName() %> >

</font>
</body>
</html>
```

Since you can access virtually all of KavaChart's server objects' functionality with property string pairs, or zero-argument methods, you can use these objects with any ASP scripting language, including VBScript and Jscript.

To make KavaChart accessible this way, unpack the server jar file, "kcServlet.jar" into the windows "trustlib" folder. By default, this is located in "C:\WINNT\java\trustlib". You can unpack this with the "jar" command from a Java Developers Kit like this:

```
jar xvf kcServlet.jar
```

This command will create a "javachart" directory that contains the necessary charting packages and classes. Make sure you don't change the location of class files within this hierarchy. Java maintains its namespace by organizing classes into parallel packages and directories (e.g. com.ve.kavachart.servlet.barApp must exist in com\ve\kavachart\servlet\barApp.class).

This command will also create a META-INF directory, which isn't necessary and can be removed.

You can also use a zip utility, like the popular WinZip product, to unpack the jar file. Jar files are compressed in zip format, with header information stored in the META-INF folder.

Note:

When using a zip utility to unpack a Java jar file, make sure you maintain the directory structure and file name capitalization. Changes to either of these will result in Java class loader problems.

The example code above uses the JSP command "getObject" to load the KavaChart server object. If you prefer to use the "CreateObject" method available from the ASP Server object, first add the KavaChart objectes you wish to use to the system's registry. Microsoft provides a small program, "javareg" for this purpose.

Consult Microsoft's server documentation for more information on using Java objects within IIS. As is generally the case, O'Reilly's technical books on ASP are also a rich source of information about using ASP.

KavaChart's Image Cache

To get the most out of KavaChart's server objects, you must understand how and why images are cached. As we discussed earlier in this chapter, application and web servers are generally designed to send one content type at a time.

Most pages with dynamic chart images also have dynamic text content, however. By default, KavaChart generates output into uniquely named image files so you can centralize your code that retrieves data and creates dynamic content. If your charts can be re-used, this file naming algorithm doubles as a performance boosting image cache.

Unique Names for Unique Charts

The key to this mechanism is an algorithm that can represent every visual difference in a chart within the filename. Every KavaChart output image starts with a generating class (such as com.ve.kavachart.servlet.pieApp), which has a set of default values. Any change from the defaults is performed by setting a property, using the "setProperty" method:

```
MyPie.setProperty("titleString", "Nice Pie Chart");
```

The combination of the class name, the property strings and the property value strings could give us a complete representation of the chart as one long string. Unfortunately, this might become a very long string. It would be particularly long for charts with a lot of data.

Fortunately, this is very similar to a problem that has already been addressed by security experts. Java's security classes give us some mechanisms to create a compressed table of these strings that's guaranteed to be unique. By creating a digest of this table, we get a string that is guaranteed to uniquely represent a particular chart.

Note:

KavaChart server objects don't reject chart definitions that have undefined properties. However, these properties are considered when creating filenames. This means you can create artificial differences between charts by adding properties of your own. For example, if you set the property "now" to the system's current time in milliseconds, every chart would have a unique filename. You may need to use an approach like this if you plan to create object subclasses that create chart data without properties.

Setting "writeDirectory"

One of the most critical properties is "writeDirectory", which instructs the server object to write its output into a particular directory. This is the directory your JSP or ASP will point to when it creates an IMG tag.

However, the server object has a different view of your server's filesystem than server clients do. Your server's view of the filesystem looks like it would to a local user. For example, your server's root directory might be located at "/usr/apache/wwwroot", or at "E:\apache\wwwroot". If you want to create an image named "chart.jpg" that clients can find at "http://yourserver/chart.jpg", you would create that file in "/usr/apache/webpages" or "E:\apache\wwwroot".

The writeDirectory property tells the KavaChart server object which directory you want to use for storing the output image. If we want to place our image in the server's root, and our server is configured like the example above, we would set "writeDirectory" to "/usr/apache/wwwroot", or "E:\\apache\\wwwroot".

Note:

Because Java uses a "\" character to "escape" special characters, you should use two slashes ("\\"), or a forward slash (e.g. "E:/apache/webpages") to specify your writeDirectory on Windows based platforms.

Instead of using the server's root directory, it's usually better practice to put your image files in a special directory for chart images. Not only does this prevent clutter in your server's root, but it also lets you use something like CacheManager (discussed below) to automatically clear out old chart images.

Using our example above, we would just set "writeDirectory" to "/usr/apache/webpages/chart_images" or "E:/apache/webpages/chart_images". We would also start with "/chart_images/" as the prefix for chart images, and call the server object's "getFilename()" method to add the rest of the complete URL.

Note:

Servlets have a convenience method that can help you write your images in the proper location. A ServletContext's "getRealPath" will return a fully qualified directory path based on a relative URL. For example, if you want to put your chart images into a directory that appears as **`http://yourserver/dynamic_charts/`** on the web, write your images into the actual directory returned by **`getServletContext.getRealPath("/dynamic_charts/")`**. Set the directory you're writing your charts into with the "writeDirectory" property.

Image Cache Related Properties

Property Name	Value String	Description
writeDirectory	String	KavaChart writes image files into this directory. By default this is set to "public_html/images". Since writeDirectory is "public_html/images" by default, the server will attempt to write images into $SERVER_ROOT/public_html/images/file_name. You can also specify an absolute path, such as /usr/lib/webserver/images for this directory. The write directory must be writable by the servlet engine, and readable by the web server. NOTE: this property is currently disabled in ChartServlet for security reasons.
useCache	boolean	If "true", the server objects will attempt to use an image in the cache directory matching this server object's parameters. If "false" the image will be generated each time. Note: if you are reading data by reference (e.g. dataset0yURL) and the data changes from time to time, you should either use this parameter to generate a fresh image, or clear the image cache whenever your data changes. Image caching is enabled by default. If useCache is false and byteStream is true, the bean won't write any output to the server's disk. NOTE: this property is currently disabled in ChartServlet for security reasons.
byteStream	boolean	By default, the server objects write an image to disk, and then create a file name to become part of an <IMG SRC=...> message to the response OutputStream. By combining useCache=false and byteStream=true you can avoid using the server's disk entirely. NOTE: this property is currently disabled in ChartServlet for security reasons.
fileName	String	An image file name for this servlet. By default, the server objects use a Secure Hash Algorithm (SHA) to create a digest of all the parameters in a given chart definition. This digest is used to create a filename that uniquely identifies a chart defined by a given set of parameters. The server object's CacheManager looks for the unique SHA filename in the image cache, and sends that image without regenerating it, if the image exists. Any change in the parameters, even the addition of an unused parameter, will create a new file name, and will cause the image to be regenerated. You can override the default image name with this property to force the image to be a specified name NOTE: this property is currently disabled in ChartServlet for security reasons.

Using CacheCleaner

Although there are many advantages to writing image files on your server's disk., there is one significant problem. Chart image files will accumulate on your server, creating a mass of small files. CacheCleaner was created to address this problem.

CacheCleaner is a utility class that deletes old chart images periodically. This class can run as a standalone application, or it can run within your server. If this class is run within the server, it uses a single lightweight thread. If it runs standalone, it will use an entire process.

This utility sleeps for a specified period of time (default 6 hours), and then removes all the files in a specified directory (this should be a chart image cache directory) older than a specified age (default 10 minutes).

CacheCleaner Method	Effect
public void setCacheDir(String s)	Sets the directory to be cleared. (default "E:\\apache\\htdocs\\images")
public String getCacheDir()	Retrieves the cache directory setting
public void setExpiredTimeInMinutes(int min)	Sets the minimum age of files to

	be deleted (default 10)
public int getExpiredTimeInMinutes()	Retrieves the minimum age of files to be deleted
public void setSleepIntervalInMinutes()	Sets the period of time CacheCleaner should sleep between removing files. (default 360)
public int getSleepIntervalInMinutes()	Retrieves the sleep time.
public void start()	Starts the cleaner thread.
public void stop()	Stops the cleaner thread.

KavaChart server objects have a common, static variable for your convenience in storing a running copy of CacheCleaner. Because this variable is static, and is the same for all instances of KavaChart server objects, you only need to instantiate a single CacheCleaner. Also, using this variable cleans only a single cacheDirectory, since new instances of CacheCleaner overwrite existing instances. You can automatically install a CacheCleaner by setting the appropriate group of properties. Here's an example:

```
<html>
<head><title>Simple Chart Generation</title></head>
<body bgcolor="white">
<font size=4>
<%
com.ve.kavachart.servlet.Bean chart =
                    new
com.ve.kavachart.servlet.areaApp();
//add some random data here:
for(int i=0;i<5;i++){
      double d = Math.random();
      chart.accumulateProperty("dataset0yValues",
           Double.toString(d));
}
String writeD =
          "/usr/ apache/webpages/chart_images";
//make sure we have a CacheCleaner
chart.setProperty("useCacheCleaner", "true");
chart.setProperty("cacheCleanerDirectory", writeD);
chart.setProperty("cacheCleanerInterval", "60");
chart.setProperty("cacheCleanerExpirationTime", "30");

//set a few properties to make the chart look nice
chart.setProperty("titleString", "hello, world");
chart.setProperty("writeDirectory", writeD");
chart.setProperty("yAxisOptions", "gridOn,
          minTickOn");
chart.setProperty("xAxisOptions", gridOn");
chart.setProperty("plotAreaColor", "lightGray");
chart.setProperty("width", "400");
chart.setProperty("height", "200");
%>
<p>
Here's the chart:
<br>
<img src=/chart_images/<%= chart.getFileName() %>>
```

```
</font>
</body>
</html>
```

In this example, we set the cacheDir property to the same value as our chart's writeDirectory property. We also set the sleep interval to 1 hour, and the expiration time to 30 minutes.

Every hour, our CacheCleaner will run, removing all files older than 30 minutes from our cache directory, "webpages/chart_images".

Notice that we also pre-pended "chart_images" to the filename to make sure the browser finds it in the correct cache directory.

Property Name	Value String	Description
useCacheCleaner	true/false	Determines whether an instance of CacheCleaner will be created to clean up the cache directory.
cacheCleanerDirectory	String	This sets the cache directory to be regulated by CacheCleaner. Note that CacheCleaner cannot tell the difference between chart images and other files so it will delete any files in this directory over the expiration age.
cacheCleanerInterval	integer	Determines the sleep interval in minutes for this instance of the CacheCleaner
cacheCleanerExpirationTime	integer	Any file in the cache directory over this age in minutes will be automatically deleted by CacheCleaner

If you want to run CacheCleaner in a separate process, you can set the properties via arguments:

Argument	Effect
-d*directory*	Sets the directory to be cleared. (default "E:\\apache\\htdocs\\images")
-x*minutes*	Sets the minimum age of files to be deleted (default 10 minutes)
-s*minutes*	Sets the period of time CacheCleaner should sleep between removing files. (default 360)
-v	Sets verbose mode
-r	Run CacheCleaner one time and exit

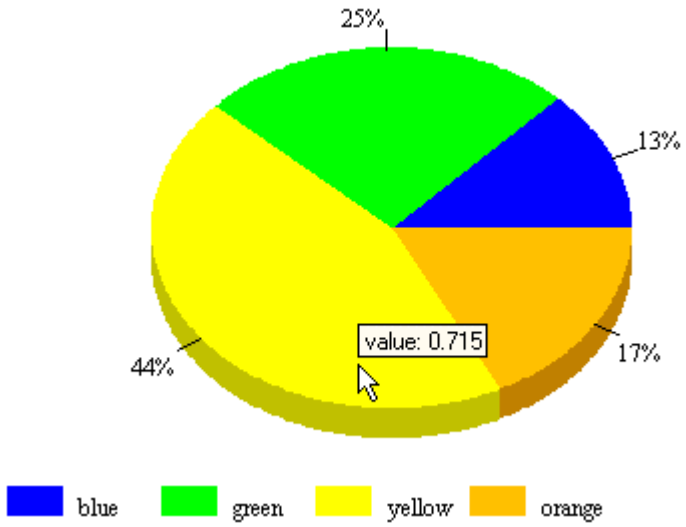
For example, the following command would clear images from "/usr/htdocs/image_cache" older than 10 minutes, and then exit:

```
java com.ve.kavachart.utility.CacheCleaner –d/usr/htdocs/image_cache –
          x10 –r
```

If you decide not to use CacheCleaner, every operating system has some facilities to accomplish the same task. For example, on a Unix server, you can create a cron job that runs a shell script to do the same task. An NT server might use a scheduled task or a service. If your application is such that you don't accumulate many image files, you could even remove the files manually from time to time.

Tooltips and Hyperlinks

One of the big advantages to using KavaChart's factory server tools is their built-in support for tooltip labels and hyperlinks. Tooltip labels are motion-sensitive labels that appear over a region of chart to provide additional information. This example shows actual data values as the mouse passes over individual pie slices:



These labels can show X values, Y values, data labels, or dataset labels. They can also be formatted with additional text, and show some combination of these labels.

The most basic mechanism for generating these labels creates a "client side imagemap", which describes a set of shapes for hyperlinks. These imagemaps are frequently used in web applications like maps or graphical menus. By generating chart images dynamically, KavaChart can also calculate the geometric values to create a map of shapes dynamically.

An ALT label for each region specifies the labeling for that region. Note: for browsers that don't support ALT tags in imagemaps, KavaChart also supports a translation facility to create JavaScript labels, which is described below.

Note:

If you want to generate tooltip output without writing an image into the server's image cache, use a ChartTag, which will generate the tooltip text while using a cooperative servlet to handle the image bytes.

Because each region is identified by way of a hyperlink imagemap, you can also place a hyperlink at each location, along with a "target" frame to specify where the hyperlink should open. The default link location is "_self", but you can also specify any other location with the "target" property.

Here is the HTML output for a chart with tooltip labels:

```
<MAP NAME=map0>
<AREA SHAPE=POLY COORDS="81, 148, 81, 132, 134, 132, 134,
148" ALT="value: 0.227">
<AREA SHAPE=POLY COORDS="81, 128, 81, 112, 261, 112, 261,
128" ALT="value: 0.757">
<AREA SHAPE=POLY COORDS="81, 108, 81, 92, 106, 92, 106,
108" ALT="value: 0.11">
<AREA SHAPE=POLY COORDS="81, 88, 81, 72, 186, 72, 186, 88"
ALT="value: 0.446">
<AREA SHAPE=POLY COORDS="81, 68, 81, 52, 233, 52, 233, 68"
ALT="value: 0.64">
</MAP>
<img src=03e2c9f00aff12534c4896a0309.jpg BORDER=0 ISMAP
USEMAP=#map0>
```

The first item is a "MAP" definition, named "map0". This is our imagemap, which has a list of shapes, defined in pixel coordinates. Each shape also contains an "ALT" tag that provides some text for the tooltip label.

The next item is our image, which has been defined with an "ISMAP" attribute. This identifies the image as having an associated imagemap. The "USEMAP" attribute identifies which map definition we want to associate with this image.

Note:

Macromedia Flash and Scalable Vector Graphics output includes tooltips and hyperlink logic within the output stream. No image map is required.

To build this output stream, KavaChart supplies the image file name and the map definition. You supply the rest of the HTML, including the IMG attributes that tell whether the chart is associated with an imagemap, and what that map's name is. This JSP scriptlet would create the stream above:

```
<%
Bean chart = new pieApp();
chart.setProperty("dataset0yValues",
                        ".227,.757,.11,.446,.64");
chart.setProperty("writeDirectory", "webpages");
chart.setProperty("toolTips", "true");
chart.setProperty("dwellUseXValue", "false");
chart.setProperty("dwellYString", "value: XX");
chart.setProperty("width", "400");
chart.setProperty("height", "200");
%>
<%= chart.getLinkMap() %>
<img src=/<%= chart.getFileName() %> BORDER=0 ISMAP
USEMAP=#map0>
```

The KavaChart server object's "getLinkMap()" method supplies a String that contains the entire imagemap. We used the default map name, so our IMG tag's USEMAP attribute specifies "#map0". Tooltips are enabled by setting the "toolTips" property to "true".

Two other properties manage the appearance of this chart's tooltips. First, we set "dwellUseXValue" to "false". X values don't have any meaning in a pie chart, so we disable them. Next, we defined "dwellYString" to be "value: XX". The string "XX" is replaced by each pie slice's Y value. The rest of the property value is used to describe this number.

This table describes tooltip related properties:

Property	Value	Description
toolTips	true/false	Tells the chart object whether to calculate a client-side MAP for tool-tip dwell labels. This map is returned by the "getLinkMap()" method.
dwellUseDatasetName	true/false	Tells the chart object whether to use the dataset name in the popup tooltip labels
dwellUseLabelString	true/false	Tells the chart object whether to use each data point's label as a part of the popup tooltip labels.
dwellUseXValue	true/false	Tells the chart object whether to use each data point's X value as a part of the popup tooltip labels.
dwellUseYValue	true/false	Tells the chart object whether to use each data point's Y value as a part of the popup tooltip labels.
dwellXString	String	A text string containing the characters "XX" to add descriptive text to the tooltip label X value. Example: "Category XX"
dwellYString	String	A text string containing the characters "XX" to add descriptive text to the tooltip label Y value. Example: "Unit Sales: $XX"
dwellLabelDateFormat	String	A text string providing formatting information for time oriented chart tooltip labels. For example, "MM/yyyy" would display month numbers, then year, like this: 02/2003
dwellXPercentFormat	true/false	Determines whether the X label will use a percent format
dwellYPercentFormat	true/false	Determines whether the Y label will use a percent format

dwellXCurrencyFormat	true/false	Determines whether the X label will use a localized currency format
dwellYCurrencyFormat	true/false	Determines whether the Y label will use a localized currency format
dwellXLabelPrecision	Integer	Number of digits of precision for dwell label values. For example, if precision is "2", labels will look like this: 123.45 or 123,45.
dwellYLabelPrecision	Integer	Number of digits of precision for dwell label values. For example, if precision is "2", labels will look like this: 123.45 or 123,45.

Tooltip Labels in JavaScript

If you need to generate tooltip labels for browsers that don't support use of the "ALT" tag in imagemaps, KavaChart provides a utility class that will translate the default imagemap into a browser-independent JavaScript.

For these situations, your image attributes and definition will remain the same. However, instead of using the chart object's "getLinkMap()" method directly, you would us it indirectly, like this:

```
com.ve.kavachart.utility.ToolTipMaker.translate(chart.getLinkMap());
```

These tooltips use document styles and layers to create context sensitive labels that track your mouse movements. If you are creating complex documents that already have multiple layers, with named spans, you may need to modify this JavaScript slightly. ToolTipMaker is provided in source code form so you can make these modifications if they're required.

Hyperlinks

KavaChart server objects can also create output streams that contain hyperlinks to other pages. This is useful if you want to create "drill-down" reports keyed to individual data items. For example, a pie chart might represent revenues in each district of a county. Clicking on a slice could take you to another chart that describes the revenue breakdown for that individual district.

Hyperlinks work the same as tooltips, but use some additional properties:

Property Name	Value	Description
mapName	String	ToolTips and hyperlinks use client side imagemaps to link strings and URLs to image geometries. These imagemaps must be named. The default name is "map0". This property lets you override the map name.
hasLinkMap	true/false	Tells the chart object whether to calculate a client-side MAP for hyperlinks (defined using DatasetNLinks). If this is set to false, tooltips without hyperlinks will be returned with getLinkMap().
datasetNLinks	List	A list of comma-separated URLs that will be used in a client-side imagemap as hyperlinks for each Datum (bar, pie slice, line marker, etc.) in Dataset N.

datasetNTargets	List	A list of comma-separated frame targets that will be used in conjunction with client-side imagemap hyperlinks for Dataset N. Note that you can also specify JavaScript objects here for additional user-defined client-side functionality.

The "mapName" property is shared with the tooltip imagemap property set, since the imagemap is the same. "hasLinkMap" determines whether the imagemap will contain hyperlink information. "toolTips" determines whether "ALT" tags will be added for tooltip labeling. If both properties are set to "true", then the imagemap will create both tooltips and hyperlinks.

Like tooltips, hyperlink imagemaps require you to add the imagemap to your output stream. Here's a sample, based on our previous example:

```
<%
Bean chart = new pieApp();
chart.setProperty("dataset0yValues",
                        ".227,.757,.11,.446,.64");
chart.setProperty("dataset0Links",
        "1.html,2.html,3.html,4.html,5.html");
chart.setProperty("writeDirectory", "webpages");
chart.setProperty("toolTips", "true");
chart.setProperty("useLinkMap", "true");
chart.setProperty("dwellUseXValue", "false");
chart.setProperty("dwellYString", "value: XX");
chart.setProperty("width", "400");
chart.setProperty("height", "200");
%>
<%= chart.getLinkMap() %>
<img src=/<%= chart.getFileName() %> BORDER=0 ISMAP
USEMAP=#map0>
```

Now our output imagemap will include links to the pages named in the property "dataset0Links". These links can include any URL, including URLs from another server.

Note:

Hyperlinks can also contain javascript references, which permit charts to manipulate information contained on a web page.

Using ChartServlet

KavaChart's server objects include a pre-built servlet that's handy for testing and setup, and perhaps even for production use. The class file is "com.ve.kavachart.servlet.ChartServlet", located in the "kcServlet.jar" file. This class implements both "doGet" and "doPost" methods, so you can call it in a variety of ways. Also, this servlet can return either text with hyperlinks and tooltip labels, or a stream of image bytes. (Note: because of security considerations, you'll have to modify ChartServlet.java to return anything but a memory-created byte stream).

After KavaChart's Web Application Archive (kavachart.war) hase been installed on your server you can access ChartServlet like this:

```
http://localhost:8080/kavachart/servlet/com.ve.kavachart.servlet.ChartServlet?chartType=barApp
  &titleString=boo
```

This URL assumes that your server is running on your local machine at port 8080, and that the servlet path is /servlet. We can set charting properties by sending them as parameters to ChartServlet. In this example, we're telling ChartServlet that we want to use the "barApp" object, and that we want the title to be "boo".

The "byteStream" property is set to "true" by default. The "useCache" property is set to "false" by default. This ensures that the servlet will generate an image in memory without writing it to a server cache. Since we're not defining the imageType, it will default to JPEG. We haven't defined any data or chartType, so the bean will use sample data in a default bar chart.

Note:

The "byteStream" and "useCache" properties cannot be changed for ChartServlet without recompiling the ChartServlet source code. Similarly, the "fileName" and "writeDirectory" properties cannot be altered without recompiling this file. You should carefully consider how these properties might be used on your server before making any changes to prevent malicious users from overwriting system files or filling your cache directories with unwanted image files.

A URL pointing to a ChartServlet in this manner will reach the "doGet" method to retrieve charting information. You can also use ChartServlet within an IMG tag like this:

```
<IMG
```

```
SRC="http://localhost:8080/kavachart/servlet/com.ve.kavachart.serv
let.ChartServlet?… >
```

Unfortunately, a realistic chart definition will quickly become so large as to make URLs like this unreasonable. You may find POST operations a better option with ChartServlet.

Some servers also implement "server-side-includes" that let you use an applet-like definition, similar to this:

```
<servlet code=com.ve.kavachart.servlet.ChartServlet>
<param name=chartType value=barApp>
<param name=width value=300>
<param name=height value=200>
<param name="dataset0yValues" value=321, 234, 234, 456>
<param name="xAxisLabels" value="apples, oranges, peaches, pears">
</servlet>
```

A server-side include will replace the ChartServlet definition above with a simple IMG tag:

```
<IMG SRC="03e2c9f00aff12534c4896a0309.jpg">
```

You should notice that the ChartServlet definition above is nearly identical to the parallel applet definition:

```
<applet code=com.ve.kavachart.applet.barApp width=300 height=200>
<param name="dataset0yValues" value="321, 234, 234, 456>
<param name="xAxisLabels" value="apples, oranges, peaches, pears">
</applet>
```

Since KavaChart server objects use the same parameter handling code as KavaChart applets, it's easy to switch between the two technologies.

To integrate ChartServlet into your own application, you must modify your "web.xml" specification to add the servlet into your application context.

ChartServlet Properties

ChartServlet has some special properties that aren't used by the server chart objects. Other properties are disabled for ChartServlet because of security issues. These are described in the table below:

Property	value type	Effect
chartType	String	the kind of chart you want to generate. These are all the "App" files in the servlet package, such as "barApp", "lineApp", and so on. The default chart type is a horizontal bar chart. If you want to reference a chart that's not in the com.ve.kavachart.servlet package, it must be a subclass of com.ve.kavachart.servlet.Bean, and you must provide the fully qualified name (e.g. "chartType=com.ve.kavachart.contrib.TwinAxisDateLineServlet").

writeDirectory	String	This property is disabled in ChartServlet for security reasons.
readDirectory	String	This property is disabled in ChartServlet for security reasons.
useCache	boolean	This property is disabled in ChartServlet for security reasons. ChartServlet always has a useCache value of “false”.
byteStream	boolean	This property is disabled in ChartServlet for security reasons. ChartServlet always uses a value of “true” for “byteStream”
fileName	String	This property is disabled in ChartServlet for security reasons.

Until you modify ChartServlet and recompile it, this class will only generate images in memory, and retrieve byte streams. This prevents the class from writing any server files, and potentially overwriting important files or exceeding a disk quota via malicious usage.

Chapter

5

KavaChart Server Objects: Methods and Properties

This chapter discusses the methods available in KavaChart server objects, their organization, and their properties.

KavaChart server objects are located in the Java package "com.ve.kavachart.servlet". These objects are designed to be single-use image generators; you will create an instance of an imaging object, define your chart properties and data, and request an image. The server object will provide the requested image data, and will then be disposed and made available for garbage collection.

What defines a chart image? Every chart image starts out as some chart type, with a set of default values. For example, if you want to create a bar chart, you would start with the KavaChart server object that creates bar charts (com.ve.kavachart.servlet.barApp), which has a default white background, blue bars, and black axes and text.

You would refine this chart by calling the object's "setProperty" method, to attach values to named properties. For example:

```
Object.setProperty("titleString", "My Chart");
```

Or, in a chart tag, your "style" properties file would contain this line:

```
titleString=My Chart
```

This will change the default title (which is no title at all) to "My Chart". Properties are available to modify virtually any aspect of a chart. For example, you can change the background color, assign an image to be used for drawing the bars, set a 3d effect, make a legend visible, change the size and position of the title, add labels to the tops of each bar, and so on.

You can also use properties to define your chart's data. Most charts can take up to 40 series of data. Different kinds of charts use different data values. Our example, a bar chart, uses only Y values. Other charts use X, Y, Y2, Y3, and Y4 values. Data can also be set without using properties, as is discussed below.

Other properties don't affect the look of the chart, but rather they define how the charting object will operate. For example, one property determines what image format will be used for creating output. Another set of properties determines whether the charting object should write the image to a file or simply hold it in memory.

When all the chart properties and data have been defined, the chart object generates image data when one of three methods is called: "getImageBytes()", "getFileName()", or "getLinkMap()". These methods trigger the bulk of the work performed by the chart object:

- The properties are examined and a filename is calculated that identifies this unique chart.
- The chart object checks the server's cache directory (set with the "writeDirectory" property) to see if the file has been previously created.
- If no previous image with this name exist, an internal chart is created. Note: if the file already exists, this is returned immediately to a call to "getFileName()", or the file is read to obtain the bytes requested from a call to "getImageBytes()".
- An internal chart is created, based on the type of server object we're using. In our example, a "BarChart" would be created.
- The properties are examined and the internal chart is modified accordingly.
- An image is created in memory and the internal chart is drawn to the image.
- Pixels are then extracted from this image, and the pixel data is passed through an image encoder for the requested "imageType" property (default is JPEG).
- Finally, the image bytes are either returned directly, or they are written to the caculated filename, and the filename is returned to the "getFileName()" call.

This process happens naturally at the appropriate time when using the chart tag library.

Server Object Methods

Generally, you can create a chart by creating an object instance, calling "setProperty" repeatedly to set up the chart attributes, and then calling "getFileName()" or "getImageBytes()". Since the vast majority of chart object attributes are set by using property pairs, you will only use a few public methods with them. The public methods are listed below.

Method	Effect
Constructor(java.util.Properties) Constructor()	any bean subclass can be instantiated with a java.util.Properties to provide predefined charting properties. The available property names are listed later in the chapter.
public void generate()	Generates an image file or ByteArray based on the current property settings. This is called automatically by getFileName() and getImageBytes().
public String getFileName()	Retrieves the user specified fileName or the default file name. If the image for this chart has not yet been generated, calls generate().
public byte[] getImageBytes()	Retrieves this image as an array of bytes. Generates the image if required. Retrieves the bytes from image cache if available.
public String getParameter(String)	Retrieves a property, or parameter, if set.
public Enumeration getParameterNames()	Retrieves the current list of set properties.
public String getProperty(String)	see getParameter(String).
public void setProperty(String, String)	Sets a specified property to a specified value. Consult the list of servlet properties, general properties, and chart-specific properties for more information.
public void setProperty(String, boolean)	Sets a property to a specified value. The functionality of this method is the same as setting a property with "true" or "false" as a string value.
public void setProperty(String, int)	Sets a property to a specified value. The functionality of this method is the same as setting a property with a string representation of an integer value.
public void setProperty(String, double)	Sets a property to a specified value. The functionality of this method is the same as setting a property with a string representation of a double value.
public void setProperty(String, double[])	Sets a property to a specified value. The functionality of this method is the same as setting a property with a string representation of a list of double values.
public void accumulateProperty(String, String)	Some properties take the form "a,b,c,d". For example, the property "dataset0yValues" might look like this: "432,123,432". In a servlet, ASP or JSP, it's often easier to accumulate these properties in a loop that examines a resultset from a database lookup. To create the list for dataset0yValues, you'd call this method three times: accumulateProperty("dataset0yValues", "432"), accumulateProperty("dataset0yValues", "123"), and so on. *Note: it's even better to create a portable DataProvider implementation in most cases.*
public String getLinkMap()	Retrieves HTML statements for a client-side imagemap that contains tool-tip information for each

	Datum (based on the dwellLabel properties below) and a set of hyperlinks (based on the datasetNLinks and datasetNTargets properties below). This string contains the geometries for each data item on the chart (pie slice, bar, line vertex, etc.) and may be quite long for complex charts.
public void loadProperties(String)	Loads properties from a specified include file. This text file contains a single column of entries with the form "propertyName=value".
public ChartInterface getChart()	Retrieves the internal com.ve.kavachart.chart.Chart object from the charting object for direct customization in java code. Note: this should be called only *after* all properties have been set. See the sections below that discuss how to add data directly without accessing the internal chart.
public void setDataProvider(DataProvider dp)	Installs a DataProvider class to supply this chart with data.
public Dataset getDataset(ChartInterface c, int index)	Method for you to override to provide one data series in com.ve.kavachart.chart.Dataset form. *Deprecated*
public void setUserImagingCodec(UserImagingCodec codec)	Adds a custom class to be used by Bean to produce the output file or bytestream. This can be used to encode charts to PostScript, transparent GIF or other formats not supported by CacheManager. Any custom class should extend com.ve.kavachart.servlet.UserImagingCodec and override either the public byte[] drawChartToOutputStream(ChartInterface), or public byte[] encodeImageBytes(Image) methods. Two such sample classes can be found in the contrib package, they are com.ve.kavachart.contrib.PostScriptEncoder, and com.ve.kavachart.contrib.GifEncoderForKava.

Image Management Properties

The following table describes the built-in image management properties associated with KavaChart server beans. See Chapter 5, "KavaChart on a Server" for more information about how these properties affect the image cache. Set these properties by calling the "setProperty" method.

Property	value type	effect
imageType	String	Output image type. This defaults to PNG generation. Other supported imageTypes include: "flash" or "swf", which produces Macromedia Flash output, "svg", which produces Scalable Vector Graphics, "gifmaker" (KavaChart's sample GIF generator), j_jpeg (a java jpeg generator), j_png (a java PNG generator, recommended), j_bmp (a java .BMP file generator), j_ico (a java based .ICO file generator), j_xbm (a java based X-bitmap generator), j_xpm (a java based X-pixmap generator). Note: if you're using a Java version lower than 1.2 (Microsoft's JVM, for example), use j_png or j_jpeg.
properties	string	a properties file containing default chart parameters. This is a great way to set up a customized look for your chart without constantly passing the same parameters to

		the server chart object. *Note: you can also simply pass the Properties as a part of the chart constructor.*
height	integer	pixel height of generated image
width	integer	pixel width of generated image
writeDirectory	String	KavaChart writes image files into this directory. By default this is set to "public_html/images". This usually means the server will attempt to write images into $SERVER_ROOT/public_html/images/file_name. You can also specify an absolute path, such as /usr/lib/webserver/images for this directory. The write directory must be writable by the servlet engine, and readable by the web server. In a servlet, you can use getServletContext().getRealPath(string), and in a JSP, you can use application.getRealPath(string) to obtain the actual filesystem location. *NOTE: this property is currently disabled in ChartServlet for security reasons.*
debug	String	If “ChartServlet” finds the parameter "debug" it will log messages to the server log about files names, locations, sizes, etc. If one of the charting objects receives this property, it will print the output to standard out. The value of this parameter is ignored.
useCache	boolean	If "true", the chart objects will attempt to use an image in the cache directory matching this set of property definitions. If "false" the image will be generated each time. Note: if you’re using the “DataProvider” interface to supply data, you implement the “getUniqueIdentifier” method to create an identifier string for your data. An identifier string (a checksum, for example) will ensure that the image is regenerated when for each unique set of data. Image caching is enabled by default. If useCache is false and byteStream is true, the bean won't write any output to the server's disk. *NOTE: this property is currently disabled in ChartServlet for security reasons. Without recompilation, ChartServlet will not write files to your server.*
byteStream	boolean	By default ChartServlet writes an image to disk, and then sends a <IMG SRC=...> message to the response OutputStream. If byteStream is set to "true", however, the servlet will instead set the content-type to the appropriate image type and send a stream of bytes. Note, however, that servlet parameters must still be set to generate a meaningful chart. By combining useCache=false and byteStream=true for server chart objects you can avoid using the server's disk entirely. *NOTE: this property is currently disabled in ChartServlet for security reasons*.
fileName	String	An image file name for this chart object. By default, the objects use a Secure Hash Algorithm (SHA) to create a digest of all the parameters in a given chart definition. This digest is used to create a filename that uniquely identifies a chart defined by a given set of parameters. The object’s CacheManager looks for the unique SHA-derived filename in the image cache, and sends that image without regenerating it, if the image exists. Any change in the parameters, even the addition of an unused parameter, will create a new file name, and will cause the image to be regenerated. You can override the default image name with this parameter to force the image to be a specified name *NOTE: this property is currently disabled in ChartServlet for security reasons. ChartServlet must be recompiled before it can write files to your server*.

antialiasOn	true/false	Turns antialiasing on for the resulting chart image.
useCacheCleaner	true/false	Determines whether an instance of CacheCleaner will be created to clean up the cache directory. CacheCleaner is discussed in more detail below.
cacheCleanerDirectory	String	This sets the cache directory to be regulated by CacheCleaner. Note that CacheCleaner cannot tell the difference between chart images and other files so it will delete any files in this directory over the expiration age. CacheCleaner is discussed in more detail below.
cacheCleanerInterval	integer	Determines the sleep interval in minutes for this instance of the CacheCleaner. CacheCleaner is discussed in more detail below.
cacheCleanerExpirationTime	integer	Any file in the cache directory over this age in minutes will be automatically deleted by CacheCleaner. CacheCleaner is discussed in more detail below.

Tooltip and Hyperlink Properties

These properties manage the information returned in a chart object's "getLinkMap()" method. See Chapter 5, "KavaChart on a Server" for more information on how to use hyperlinks and tooltips.

Property	Value type	Effect
mapName	String	ToolTips and hyperlinks use client side imagemaps to link strings and URLs to image geometries. These imagemaps must be named. The default name is "map0". This property lets you override the map name. Not used for SWF or SVG output.
toolTips	true/false	Tells the object whether the client-side MAP should include ALT tags for tool-tip dwell labels.
hasLinkMap	true/false	Tells the object whether the client-side MAP should contain hyperlinks (set using DatasetNLinks).
dwellUseDatasetName	true/false	Tells the servlet whether to use the dataset name in the popup dwell labels
dwellUseLabelString	true/false	Tells the servlet whether to use each datapoint's label as a part of the popup dwell labels.
dwellUseXValue	true/false	Tells the servlet whether to use each datapoint's X value as a part of the popup dwell labels.
dwellUseYValue	true/false	Tells the servlet whether to use each datapoint's Y value as a part of the popup dwell labels.
dwellXString	String	A text string containing the characters "XX" to add descriptive text to the dwell label X value. Example: "Category XX"
dwellYString	String	A text string containing the characters "XX" to add descriptive text to the dwell label Y value. Example: "Unit

		Sales: $XX"
dwellXPercentFormat	true/false	Determines whether the X label will use a percent format
dwellYPercentFormat	true/false	Determines whether the Y label will use a percent format
dwellXCurrencyFormat	true/false	Determines whether the X label will use a localized currency format
dwellYCurrencyFormat	true/false	Determines whether the Y label will use a localized currency format
dwellXLabelPrecision	Integer	Number of digits of precision for dwell label values. For example, if precision is "2", labels will look like this: 123.45 or 123,45.
dwellYLabelPrecision	Integer	Number of digits of precision for dwell label values. For example, if precision is "2", labels will look like this: 123.45

Data Related Properties

Every chart creates a graphical representation of numeric information. Different kinds of charts require different kinds of numeric information, but every chart requires at least some sort of numbers to start with. KavaChart organizes this information into "Datasets", which contain the numbers and text required by your chart.

Some charts have a single dataset (pie charts and speedos), while others may have many datasets (each line on a line chart is a different dataset). Similarly, some datasets contain a lot of information for each observation (a candlestick chart has a time, high, low, open, close, and label value for each price bar), while others contain only a little (a speedo uses only a single value, and a pie uses one value and one label for each slice).

Following mathematical conventions, the most basic numeric unit for each observation in a chart is called a "Y" value. This means that we use "Y" values to define the value for each slice in a pie, or the height of each column, or even the width of a bar in a horizontal bar chart. Y values are required for any chart to create a meaningful visual.

Every chart can also contain a textual label for each Y value. These charts don't always display that label, but it's available. For example, you might assign some labels like "East", "West", "North", and "South" to a bar chart. The labels might not be visible on the chart, but you could use them in a tooltip label for users that want to explore further.

Some charts also use "X" values, which is generally thought of as the "independent", or deterministic part of your observation. For example, if your chart shows how ozone levels compare to temperature, you would assume that temperatures are "independent" of ozone levels, while ozone levels may be "dependent" on temperatures. Temperature would be used as "X" values in this

case. A line chart that plots ozone levels against temperature might have a variety of temperature observations that don't fall into neatly defined categories, but for each temperature observed (X), there would also be an ozone level observation (Y).

Not all charts use X values. In some cases (pie charts) this is obvious. In other cases, it may not be. For example, bar charts don't usually use X values, because bars are generally used to represent categories, rather than a set of independent numeric values. In the case of a bar or column chart, KavaChart will ignore your X values, and use a set of implied X values (0, 1, 2, ...).

More complex charts, such as hi-lo bar charts or financial charts (OHLC, Candlestick) require additional information, which we call "Y2" and "Y3" data. This auxiliary information takes on special meaning depending on the chart that calls for it.

All this X, Y, Y2, and Y3 data is organized into datasets. Every chart can contain up to 40 datasets, with an unlimited number of observations in each dataset. Some charts (speedo and pie, for example) don't use all the data; these charts use the lowest numbered information available. For example, pie charts use dataset 0, and speedos use only observation 0 of dataset 0.

In addition to the numbers and text, each observation can also take a fill color definition, a line color, and a fill style and line style. The dataset that contains the observations also has fill, line, and color information, and a name for the overall dataset. Different charts use all this information in different ways.

For example, a pie chart uses the color definitions for each observation to draw each slice, and individually colored bar charts use this information for each bar's color and for legend icons. Standard bar charts and line charts use the dataset colors and labels for drawing and legends.

Supplying Data with Properties

How do you get all this information into your server object? A quick way is to just use chart properties to add data. Here's an example of JSP with a simple data definition:

```
<%
com.ve.kavachart.servlet.Bean chart =
                    new
com.ve.kavachart.servlet.columnApp();
//add some data here:
chart.setProperty("dataset0yValues", "234,321,234"); %>
<p>
Here's the chart:
<br>
<img src=/<%= chart.getFileName() %>>
```

This JSP uses the property “dataset0yValues” to define the "Y" values for dataset 0. To add another series, we'd just add another property, using

"dataset1yValues". To add another bar to our chart, we'd just add another number to the list "234,321,234".

In this chart we don't need X values, Y2 values or anything else, because we're just dealing with a simple bar chart. If we wanted to add some labels, we could do this:

```
<%
com.ve.kavachart.servlet.Bean chart =
                    new
com.ve.kavachart.servlet.columnApp();
//add some data here:
chart.setProperty("dataset0Labels", "a,b,c");
chart.setProperty("dataset0yValues", "234,321,234"); %>
<p>
Here's the chart:
<br>
<img src=/<%= chart.getFileName() %>>
```

It doesn't matter what order the properties are in. These labels can be used in various ways. For example, right now, the chart will use these labels along the horizontal axis to label each bar. However, if we add the property "labelsOn" and set the value to "true", we'll get labels at the top of each bar. The same properties can be used for pie charts, line charts, or any other kind of chart. We would just assign a pieApp, areaApp, or some other kind of chart to our "chart" variable.

Properties are available for dataset0yValues, dataset0xValues, dataset0y2Values, dataset0y3Values, and dataset0Labels for datasets 0 through 39.

Since your data is probably coming from some kind of object you can iterate (e.g. a resultset, an array, a Vector, etc.), you might want to use "accumulateProperty" instead of "setProperty", like this:

```
for(int i=0;i<myVector.size();i++){
      String val = myVector.elementAt(i).toString();
      chart.accumulateProperty("dataset0yValues", val);
}
```

Most applications will benefit by implementing a "DataProvider" class to supply data directly to charts. This requires a small bit of Java coding, but it's generally well worth the effort to get the portability and organization provided by a DataProvider. DataProviders are discussed below.

Dataset Properties

The table below gives parameter names and usage descriptions. All parameters listed as "dataset0" are valid for datasets 0 through 39. Items described as "lists" expect a comma separated list of values, colors, etc. You can change the delimiter from a comma to another character with the "delimiter" param.

Property Name	Type	Effect
dataset0xValues	list	comma separated list of X values for dataset 0.
dataset0yValues	list	comma separated list of Y values for dataset 0
dataset0y2Values	list	comma separated list of difference values for dataset 0 hilo bars
dataset0xyValues	List	comma separated list of X,Y values for dataset 0.
dataset0dateValues	List	Comma separated list of time/date strings for dataset 0. See also "inputDateFormat".
dataset0y3Values	list	Tertiary observations for charts that require 3 Y values (e.g. hi-lo-close charts)

Discontinuities

One special case deserves notice here. Some charts support the notion of "discontinuities" (disLineApp, disDateLineApp, etc.). In these charts, you want to have a break in the line or some other visual feedback that shows missing data. In this case, you can just use some non-number, like 'x', to indicate a break. KavaChart recognizes this as a missing point and creates the line break as appropriate. Here's an example:

```
<%
com.ve.kavachart.servlet.Bean chart =
                    new
com.ve.kavachart.servlet.disLineApp();
//add some data here:
chart.setProperty("dataset0yValues",
        "2,3,6,x,4,5,2,x,7,8");                                    %>
<p>
Here's                              the                              chart:
<br>
<img src=/<%= chart.getFileName() %>>
```

Time oriented charts

Charts that display time oriented data (dateLineApp, dateAreaApp, etc.) use time stamps as a special kind of numeric value. For these charts, use the property dataset0dateValues, like this:

```
chart = new com.ve.kavachart.applet.dateLineApp();
chart.setProperty("width", "300");
chart.setProperty("height", "200");
chart.setProperty("dataset0yValues", "234,321,234");
chart.setProperty("dataset0dateValues",
                      "01/01/02,02/01/02,03/01/02");
```

This property translates the dates into a form usable by Java classes and places our Y observations at the proper locations along the axis. Unfortunately, our

date definitions are ambiguous here. Did our observations occur on January 1, 2, and 3? Or did they occur on January 1, February 1, and March 1?

Managing Date Formats

To properly use dataset0dateValues, you should also use inputDateFormat:

```
chart = new com.ve.kavachart.applet.dateLineApp();
chart.setProperty("width", "300");
chart.setProperty("height", "200");
chart.setProperty("inputDateFormat", " MM/dd/yy");
chart.setProperty("dataset0yValues", "234,321,234");
chart.setProperty("dataset0dateValues",
                        "01/01/02,02/01/02,03/01/02");
```

The table below describes how to construct an inputDateFormat to match your data generator.

Field	Full Form	Short Form
Year	yyyy (4 digits)	yy (2 digits)
Month	MMM (name)	MM (2 digits), M (1 or 2 digits)
Day of week	EEEE	EE
Day of Month	dd (2 digits)	d (1 or 2 digits)
Hour (1-12)	hh (2 digits)	h (1 or 2 digits)
Hour (0-23)	HH (2 digits)	H (1 or 2 digits)
Hour (0-11)	kk (2 digits)	k (1 or 2 digits)
Hour (1-24)	KK (2 digits)	K (1 or 2 digits)
Minute	mm	None
Second	ss	None
Millisecond	SSS	None
AM/PM	a	None
Time Zone	zzzz	zz
Day of Week in Month	F (e.g. 2nd Tuesday)	None
Day in year	DDD (3 digits)	D (1, 2, or 3 digits)
Era	G (e.g. BC or AD)	None

Tip:

If you're generating dynamic data from a JSP that uses JDBC, you can probably use the property dataset0xValues. Assuming your observation dates are java.sql.Date classes, just use the "getTime()" method to pass the raw numeric information into the chart instead of formatting the output to match a string input format.

Time and date oriented charts have special properties for managing axes, which are listed below.

Using DataProviders

A DataProvider is a Java class that implements the KavaChart Interface "com.ve.kavachart.utility.DataProvider". By implementing this simple interface in your existing data sources, you can easily translate text and numeric information into graphics.

The Interface:

```
public Interface com.ve.kavachart.utility.DataProvider {
    /*
     ** Returns an Enumeration of Dataset classes.
     */
    public Enumeration getDatasets();
    /*
     ** Returns a String that uniquely identifies this data.
     ** Needed to make chart image caching work properly.
     ** Otherwise unnecessary.
     */
    public String getUniqueIdentifier();
}
```

You probably already have some kind of data connection that provides you with data for a table, some summary figures, etc. It's a good idea to use the same connection to implement DataProvider so you don't have to make multiple database connections, or use up otherwise scarce server resources.

DataProviders are portable, usable in Chart Tags, scriptlets, and servlets. They can also be used to model you data directly in the KavaChart Wizard, and they provide the valuable architectural function of separating your data source management from the visual presentation of that data.

Important Dataset Constructors and Methods

Since a DataProvider returns an Enumeration of Dataset classes, it's important to be able to construct these. Fortunately, this class has a range of convenience constructors that make it quite easy to use.

The Dataset class is found in the com.ve.kavachart.chart package. In the constructors below, set "Globals" to "null".

```
public Dataset();

public Dataset(String   name,
      double[]    xArray,
      double[]    yArray,
```

```
      double[]    y2Array,
      double[]    y3Array,
      int         seriesNumber,
      Globals     g);

public Dataset(String   name,
      double[]    xArray,
      double[]    yArray,
      String[]    labels,
      Int         seriesNumber,
      Globals     g);

public Dataset(String name,
      double[]    xArray,
      double[]    yArray,
      Globals     g);

public Dataset(String  name,
      double      yArray[],
      int         setNumber,
      Globals     g);
```

If you're using time oriented charts, time values should be the underlying values (milliseconds since epoch) used by java.util.Date and java.sql.Date. Generally, these are "X" values, and can be obtained by using Date.getTime().

Dataset also has a number of useful methods that make it easy to create meaningful data:

```
public void addDatum(Datum d);

public void addPoint(double x, double y, String label);
```

A Datum class describes an individual observation: a point on a line, a bar in a bar chart, etc. Datum is also found in the com.ve.kavachart.chart package, and has easy to use constructors and methods:

```
public Datum(double x, double y, Globals g);

public Datum(int whichPoint,
      double y,
      String label,
      Globals g);

public Datum(     double  dataX,
      double      dataY,
      double      dataZ,
      String      str,
      int         element,
      Globals     g);
```

Datum methods let you set internal values:

```
public void setX(double x);
public void setY(double y);
```

```
public void setY2(double y2);
public void setY3(double y3);
public void setLabel(String s);
```

A complete DataProvider is installed in a chart bean using the "setDataProvider" method. In a chart tag, a DataProvider is installed into a server attribute (application, page, or session scope), and the attribute name is passed into the tag as the "dataProviderID".

Here's a simple, complete DataProvider:

```
import java.util.*;
import com.ve.kavachart.chart.*;
public void MyDataProvider implements DataProvider{
      public Enumeration getDatasets(){
            Dataset d = new Dataset();
            double yVals = new double[20];
            for(int i=0;i<yVals.length;i++){
                  d.addPoint(i, y, null);
            }
            d.setName("Fake Data!");
            ArrayList al = new ArrayList();
            al.add(d);
            return Collections.enumeration(al);
      }
      public String getUniqueIdentifier(){
            return (new Date()).toString();
      }
}
```

Note that this code returns the value of "Date" to make sure any cached versions of this chart are not used. Now, to install and use this DataProvider in a JSP scriptlet or a servlet, we'd do this:

```
Bean chart = new columnApp(myStyleProperties);
chart.setDataProvider(new MyDataProvider());
String filename = chart.getFileName();
```

In a chart tag, we would do something like this:

```
<%pageContext.setAttribute("data", new MyDataProvider());%>
<chart:streamed chartType=columnApp dataProviderID="data" />
```

A DataProvider gives you much more flexibility in structuring your pages and maintaining your data. Importantly, it also helps you test your data inputs outside an HTTP context.

Within a chart tag, DataProviders can also be filtered, sorted, combined with other sources, etc. using an intuitive set of tags, discussed in another chapter.

One important chart property interacts with your DataProvider:

Parameter	Value Type	Example

stylesFromDataProvider	boolean	`chart.setProperty("stylesFromDataProvider",true);`
		
		`stylesFromDataProvider=true`

Using this property, it's possible to programmatically change your data's style properties (colors, outlining, etc.) based on your incoming data values. By default, style information is not based on the DataProvider, but based on chart properties.

Color and Style Properties

KavaChart server charts support a lengthy list of properties to help you make your charts look exactly the way you want. These properties are used to set colors, fonts, textures, line styles, and the overall layout of your chart.

Color and style propertes take different kinds of values. The table below gives you some examples of what these values should be. Note that all values are set using a String representation, so a Boolean would "true" or "false", not a java.lang.Boolean. Also, all properties are case sensitive for server-side objects. Some applet environments are more lenient about case usage.

Parameter Type	Explanation	Example

Integer	An integer value, like "1", or "7". This is usually used to specify something like a line style or a marker style; one out of a list of several available types.	`Chart.setProperty("legendTexture","1");`
		
		`legendTexture=1`

Double	A real number value, like 0.25. Generally, these values are expressed in terms of a percentage of the overall chart size.	`Chart.setProperty("plotAreaBottom","0.12");`
		
		`plotAreaBottom=0.12`

Font	font parameters include information for the font name, the font size, and the font style. Any valid Java font works here, but we start with a default of TimesRoman 12pt in most cases to ensure that the font is available. The example instructs KavaChart to use 18 point Arial italic fonts for this chart's X axis labels. 0 is plain, 1	`Chart.setProperty("xAxisLabelFont","Arial,18,2");`

	bold, and 2 italic.	
Color	this field expects a color name, or a hexadecimal color definition (in RGB). Valid colors names in these applets include black, white, gray, darkGray, lightGray, red, pink, orange, yellow, green, magenta, cyan, and blue. A valid hex definition for white is "ffffff". You can also use the color "transparent" if you don't want a particular element to be visible.	`Chart.setProperty("titleColor","ffbb00");`
List	These fields are looking for a list of items, separated by a delimiter. The default delimiter is a comma character, but you can change this with the "delimiter" param.	`Chart.setProperty("dataset0Colors","green,red,ff00aa">`
String	A text string	`Chart.setProperty("titleString", "hello, world");`
url	These fields expect some URL specification within your applet's CODEBASE. Relative or absolute URLs are OK.	`Chart.setProperty("backgroundImage","/tmp/pic.jpg");`
Boolean	Either "true" or "false"	`Chart.setProperty("outlineLegend","false");`
Anything	Some parameters can take any value. The applet just wants to know if the parameter has been defined	`Chart.setProperty("3D","yeah, sure");`

General Color and Font Properties

The first properties apply to all charts. These properties define colors, overall layout, titles, and so on.

Parameter	Value Type	Effect
colorPalette	String	Set the overall default color palette for the chart. Default possibilities: web_sanfrancisco, web_minnesota, web_alaska, web_newyork, web_losangeles, web_grays,

		web_seattle, web_newmexico, web_rosemary, web_pastel, web_prague, presentation_cool, presentation_browns, presentation_southwest, presentation_impact, presentation_deep, presentation_oceana, presentation_sophisticated The default is “web_newyork”
colorPaletteDefinition	list	List of color definitions (e.g. 00ff00,green,blue,black)
titleString	String	Chart Title (default none)
titleFont	font	Font name, size, & style for chart title (default TimesRoman, plain, 12 pt)
titleColor	color	color of text in Title (default black)
titleX	double	X location of the title string, if this is not specified the title will be centered.
titleY	double	Y location of the title string.
subTitleString	String	Chart Sub-Title (default none)
subTitleFont	font	Font name, size, & style for chart title (default TimesRoman, plain, 12 pt)
subTitleColor	color	color of text in Title (default black)
subTitleX	double	X location of the subtitle string, if this is not specified the subtitle will be centered.
subTitleY	double	Y location of the subtitle string.
labelsOn	anything	determines whether bar, line, pie, etc., labels will be visible
labelAngle	integer	the number of degrees to rotate datum labels
labelPrecision	integer	the number of digits of precision for datum labels
legendOn	anything	make the legend visible
legendOff	anything	make the legend invisible (default)
legendColor	color	sets the background color of a legend
legendVertical	anything	legend icons in vertical list (default)
legendHorizontal	anything	legend icons in horizontal list
legendLabelFont	font	Font name, size, & style for legend (default TimesRoman, plain, 12 pt)
legendLabelColor	color	color of text in legend (default black)
legendllX	double	X location of lower left legend corner (default 0.2)
legendllY	double	Y location of lower left legend corner (default 0.2)
iconWidth	double	width of legend icon (default 0.07)
iconHeight	double	height of legend icon (default 0.05)
iconGap	double	gap between icon and next legend entry (default 0.01)

legendSecondaryColor	color	The Color to be used as the secondary color for this legends texture/gradient.
legendGradient	integer	Sets the gradient for this legend. Available gradient values are 0 for left/right mirrored, 1 for top/bottom mirrored, 2 for top to bottom, and 3 for left to right
legendTexture	integer	Sets the texture for this legend. Available texture values are 0 for horizontal stripes, 1 for vertical stripes, 2 for diagonal down stripes, 3 for diagonal up stripes, 4 for cross hashing, and -1 to use the legendimage to create the texture.
legendImage	URL (or filename)	image to use for this legend's background (default none). Use this property to define line markers for scatter plots.
legendLineWidth	integer	pixel width of legend outline
legendLineStyle	integer	Sets the line style for this legend's outline. Available values for this parameter are 0 for dashed, 1 for dotted, 2 for dot-dashed, and -1 for solid (default = -1).
plotAreaTop	double	top of the plotting area
plotAreaBottom	double	bottom of the plotting area
plotAreaRight	double	right side of the plotting area
plotAreaLeft	double	left side of the plotting area
plotAreaColor	color	color of plotting area background (default white)
plotAreaSecondaryColor	color	The Color to be used as the secondary color for this plotarea's texture/gradient.
plotAreaGradient	integer	Sets the gradient for this plotarea. Available gradient values are 0 for left/right mirrored, 1 for top/bottom mirrored, 2 for top to bottom, and 3 for left to right
plotAreaTexture	integer	Sets the texture for this plotarea. Available texture values are 0 for horizontal stripes, 1 for vertical stripes, 2 for diagonal down stripes, 3 for diagonal up stripes, 4 for cross hashing, and -1 to use the plotarea image to create the texture.
plotAreaImage	URL (or filename)	image to use for this plotarea's background (default none). Use this property to define line markers for scatter plots.
plotAreaLineWidth	integer	pixel width of plotarea outline
plotAreaLineStyle	integer	Sets the line style for this plotarea's outline. Available values for this parameter are 0 for dashed, 1 for dotted, 2 for dot-dashed, and -1 for solid (default = -1).
backgroundColor	color	color of chart background (default white)
backgroundSecondaryColor	color	The Color to be used as the secondary color for this background's texture/gradient.
backgroundGradient	integer	Sets the gradient for this background. Available gradient values are 0 for left/right mirrored, 1 for top/bottom mirrored, 2 for top to bottom, and 3 for left to right
backgroundTexture	integer	Sets the texture for this background. Available texture values are 0 for horizontal stripes, 1 for vertical stripes, 2 for diagonal down stripes, 3 for diagonal up stripes, 4 for cross hashing, and -1 to use the plotarea image to create the texture.

backgroundImage	URL (or filename)	image to use for this background's background (default none).
backgroundLineWidth	integer	pixel width of background outline
backgroundLineStyle	integer	Sets the line style for this background's outline. Available values for this parameter are 0 for dashed, 1 for dotted, 2 for dot-dashed, and -1 for solid (default = -1).
3D	anything	turns on 3D effects for this chart (default 2D)
2D	anything	turns on 2D effects for this chart (default 2D)
XDepth	integer	number of pixels of offset in X direction for 3D effect (default 15)
YDepth	integer	number of pixels of offset in y direction for 3D effect (default 15)
locale	String	KavaChart automatically localizes your charts for the locale of the Java Virtual Machine that creates the chart. Generally, locale changes are disallowed by servlet SecurityManagers. Valid locales include canada, canada_french, china, chinese, english, france, french, german, germany, italian, italy, japan, japanese, korea, korean, prc, simplified_chinese, taiwan, traditional_chinese, uk, and us. You can also create a locale using two letter language codes and country codes in this format: langageCode_countryCode (for example "en_US" denotes english/U.S.).
delimiter	String	the separator character for list parameters. Default is comma (e.g. "123.432.123").
defaultFont	Font	A new default font for your charts. This parameter overrides the default font setting for KavaChart graphs. This parameter sets a new default for all KavaChart graphics running within the Java Virtual Machine in the current session, so you should use it cautiously. Its primary value is for settings that wish to start with consistent font usage for all charts.
outlineColor	Color	Color to use for outlining bars, plotareas, etc. (Default none). Using this param automatically enables outlining for most objects
outlineDataRepresentation	true/false	If outlineColor is set to some color, you can selectively turn the outlining off for the DataRepresentation (Bars, Pie, Area, etc.) by setting this property to "false". Default is "true".
outlinePlotarea	true/false	If outlineColor is set to some color, you can selectively turn the outlining off for the Plotarea (the region bounded by the x and y axes) by setting this property to "false". Default is "true".
outlineBackground	true/false	If outlineColor is set to some color, you can selectively turn the outlining off for the Background (the total chart image area) by setting this property to "false". Default is "true".
outlineLegend	true/false	If outlineColor is set to some color, you can selectively turn the outlining off for the chart Legend by setting this property to "false". Default is "true".
showVersion	true/false	If this is set to true, the chart will be created with the version number replacing the chart's title.

annotation0LabelString	String	A label for note 0 (unlimited notes available) *Note: a "|" character will break this note into multiple lines.*
annotation0Alignment	above|below|left|right	Where note should appear relative to location
annotation0CoordinateSpace	pixel|axis	Coordinate space for location values
annotation0Xloc	Number	Pixels or axis values
annotation0YLoc	Number	Pixels or axis values
annotation0LabelFont	Font	Font for this note
annotation0LabelColor	Color	Font color for this note
annotation0FillBackground	true|false	Determines whether this note will have an opaque background
annotation0BackgroundColor	Color	Note's background color
annotation0OutlineColor	Color	This note's outline color (if any)

Axis Related Properties

The following tables contain properties for adjusting axes. Line, area, bar, and their derivatives use these properties. Axis properties include individual properties and an option list. The option list groups several properties together.

Axis Option Lists

The option lists include various options for adjusting the look of an X or Y axis. Use these parameters in a list, like this:

```
setProperty("xAxisOptions" "gridOff, tickOff, lineOn");
```

If you're modifying an auxiliary Y axis (such as in a chart that has left and right axes), use the property name auxAxisOptions.

yAxisOptions (xAxisOptions)	Effect
autoScale	automatically create axis scale (default)
noAutoScale	axis scale defined in other properties
rotateTitle	"true" if vertical axis title should be parallel with axis
logScaling	"true" if axis should use log scaling
lineOn	axis line is visible (default)
lineOff	axis line is invisible
tickOn	major tick marks are visible (default)
tickOff	major tick marks are invisible
minTickOn	minor tick marks are visible
minTickOff	minor tick marks are invisible (default)
labelsOn	axis labels are visible (default)
labelsOff	axis labels are invisible
gridOn	grid lines are visible
gridOff	grid lines are invisible (default)

rightAxis	this axis goes on the right
topAxis	this axis goes on the top
bottomAxis	this axis goes on the bottom
leftAxis	this axis goes on the left (default)
percentLabels	this axis should use localized percentage representations (not valid for date and label axes)
currencyLabels	This axis will use a localized currency representation for labels (not valid for date and label axes)

Detailed Axis Properties

If you're modifying an X Axis (usually on the top or bottom of a chart), use xAxis*PropertyName* instead of yAxis*PropertyName*. X Axes are on the left and right for Horizontal Bar Type charts. Speedo and Polar charts have a single Axis, which is a Y Axis.

If you're modifying an Auxiliary Y Axis (charts that have left and right axes, for example), use auxAxis*PropertyName* instead of yAxis*PropertyName*.

Axis Property	Value Type	Effect
yAxisTitle	string	Axis title
yAxisTitleFont	font	Axis title font
yAxisTitleColor	color	Axis title color
yAxisLabelFont	font	use this font for axis labels
yAxisLabelColor	color	axis labels in this color (default black)
xAxisLabels	list	A comma separated list of user-defined labels for this Axis. This is only effective for certain types of chart (BarChart derivatives, LabelLineChart, Area charts) that use a LabelAxis. By default, LabelAxis is only used for X axes.
yAxisLabelAngle	integer	label rotation in degrees (default 0). Note: rotations of 0 and 90 degrees will be the most readable
yAxisLabelFormat	0:default, 1:Comma, 2:European	(default 0) Note: by default, charts will automatically localize formats based on the Java Virtual Machine generating the chart.
yAxisLabelPrecision	integer	Number of digits past the decimal point to display
yAxisLineColor	color	axis line color (default black)
yAxisTickColor	color	axis tick mark color (default black)
yAxisGridColor	color	axis grid line color (default black)
yAxisColor	color	sets axis grids, ticks, lines and labels to the same color
yAxisTickLength	integer	number of pixels long for axis tick marks
yAxisMinTickLength	integer	number of pixels long for axis minor tick marks
yAxisStart	double	starting value on axis. By default, axes automatically determine a starting and ending

		value. By setting this value, you can give the axis a default minimum value. If the Axis is set to noAutoScale, this value will be used directly. Otherwise, this value may be adjusted slightly to yield better looking labels. For example, if you set yAxisStart to 0.01, the chart may decide to round the value down to 0.0 to create even axis increments.
yAxisEnd	double	ending value on axis. By default, axes automatically determine a starting and ending value. By setting this value, you can give the axis a default maximum value. If the Axis is set to noAutoScale, this value will be used directly. Otherwise, this value may be adjusted slightly to yield better looking labels. For example, if you set yAxisStart to 9.99, the chart may decide to round the value up to 10.0 to create even axis increments.
yAxisLabelCount	integer	how many labels on an axis set to noAutoScale
yAxisTickCount	integer	how many tick marks on an axis set to noAutoScale
yAxisMinTickCount	integer	how many minor tick marks on an axis set to noAutoScale
yAxisGridCount	integer	how many grid lines on an axis set to noAutoScale
yAxisGridStyle	integer	the line style of the grid lines for this axis
yAxisGridWidth	integer	the width in pixels of the grid lines for this axis
yaxisThresholdLine0Color	Color	The color of reference line 0 (40 available).
yAxisThresholdLine0LabelColor	Color	The color of the label for reference line 0
yAxisThresholdLine0LabelFont	Font	Font for for reference line 0's label
yaxisThresholdLine0LabelString	String	Optional label for reference line 0
yAxisThresholdLine0LineStyle	Integer	Line style for reference line 0
yAxisThresholdLine0Value	Double	Where on the Y axis should reference line 0 draw.

Tip:

If you want an axis to start at a specific value, but end at some value based on data, just use yAxisStart without including noAutoScale among your yAxisOptions.

Date and Time Axis Properties

The following list contains options for Time/Date X axes, such as those used for dateLineApp and dateAreaApp, as well as financial chart types like stickApp and hiLoCloseApp

DateAxis Properties	Type	Effect
startDate	string	time/date for axis starting value. Note: this string is passed into Java's "Date" class to be translated into a machine

		independent time stamp. Many time-stamp formats will work. If you need to use a specific input format, see the Date Format section above.
endDate	string	time/date for axis ending value
axisDateFormat	string	By default, DateAxis selects an appropriate labelling type based on your time range and your locale. This property lets you override the axis labels to use your specific formatting instructions. See the Date Format table above for more information on how to use the formatting patterns.
axisSecondaryDateFormat	string	Some DateAxes use a primary and secondary format to highlight important boundaries, like years or hours. This parameter lets you set the date or timestamp format for one of these boundaries. See the Date Format section above for more information on how to use the formatting patterns.
scalingType	integer	1 scale by seconds 2 scale by minutes 3 scale by hours 4 scale by days 5 scale by weeks 6 scale by months 7 scale by years
axisTimeZone	string	This determines the timezone used for displaying date data. By default chart objects use the timezone of the server jvm. This may be incorrect in some cases. For example, if my servlet is parsing time data in New York, and I want a user in California to see the data in real-time not New York time, then this parameter can be used to change the way the data is displayed. Timezones can be specified by JDK 1.1 deprecated strings like PST, EST, etc., by Java 2 standards: "America/Los_Angeles", or by the difference from GMT in this syntax: GMT[+|-]hh[[:]mm] (for example Eastern Standard Time would be equivalent to "GMT-5:00").
inputTimeZone	string	This determines the timezone used for parsing date data. By default chart objects use the timezone of the server jvm. This may be incorrect in some cases. For example, if my data is based in New York, my client's applet is parsing time data in California, and I want my user to see the data in real-time, then this parameter can be used to change the way the data is parsed. This is an alternate to inputting the timezone in your date strings. Timezones can be specified by JDK 1.1 deprecated strings like PST, EST, etc., by Java 2 standards: "America/Los_Angeles", or by the difference from GMT in this syntax: GMT[+|-]hh[[:]mm] (for example Eastern Standard Time would be equivalent to "GMT-5:00").

Dataset Related Color and Style Parameters

Dataset colors and styles are very important to KavaChart applets. These colors are used to define the color of bars, pie slices, legend icons, and so on.

Dataset Parameters (available datasets 0 through 39)	Type	Effect
dataset0Name	string	name for display in legend (default "dataset0")
dataset0Color	color	color to use for this dataset (default varies)
dataset0Colors	list of colors	colors to use for pie slices or bars (default varies)

dataset0SecondaryColor	color	The Color to be used as the second color with dataset textures/gradients. The default is transparent.
dataset0SecondaryColors	list of colors	Colors to be used as the second color with dataset textures/gradients. The default is transparent.
dataset0Gradient	integer	Sets the gradient for this dataset. Available gradient values are 0 for left/right mirrored, 1 for top/bottom mirrored, 2 for top to bottom, and 3 for left to right
dataset0Gradients	list of integers	Sets the gradients for this dataset. For available values see datset0Gradient.
dataset0Texture	integer	Sets the texture for this dataset. Available texture values are 0 for horizontal stripes, 1 for vertical stripes, 2 for diagonal down stripes, 3 for diagonal up stripes, 4 for cross hashing, and -1 to use the dataset image to create the texture.
dataset0Textures	list of integers	Sets the textures for this dataset. For available values see datset0Texture.
dataset0Image	Filename or URL	image to use for this dataset's markers (default none). Use this property to define line markers for scatter plots.
dataset0Images	list of filenames	images to use for this chart's markers (default none). Use this property to define individual line markers for scatter plots. These values will also be used as fill images for pie charts or individually colored bar charts.
dataset0MarkerStyle	integer	Specify an internal marker for line charts and scatter plots (0=box, 1=diamond, 2=circle, 3=triangle). Default is -1 (none)
dataset0MarkerStyles	list of integers	Specify internal markers for datsets drawn with different markers at each data point. See datset0MarkerStyle for available marker values.
dataset0MarkerSize	integer	pixel width of internal marker for line charts and scatter plots.
dataset0MarkerSizes	list of integers	pixel widths of internal markers for line charts with individual markers
dataset0LineWidth	integer	pixel width of plot line
dataset0LineStyle	integer	Sets the line style for this line. Available values for this parameter are 0 for dashed, 1 for dotted, 2 for dot-dashed, and -1 for solid (default = -1).
dataset0LabelFont	font	font to use for this dataset's labels (default TimesRoman 12pt)
dataset0LabelColor	color	color to use for this dataset's labels (default black)

Chapter

6

Server Chart Objects

This chapter details the specific chart objects available in KavaChart ProServe. These charts can be used directly in a servlet or scriptlet, or can be named in a Chart Tag.

Each chart type has a few properties that deal with the specifics of that chart type. For example, pie charts have a property that lets you set the starting angle of the pie. This property doesn't make sense for bar charts.

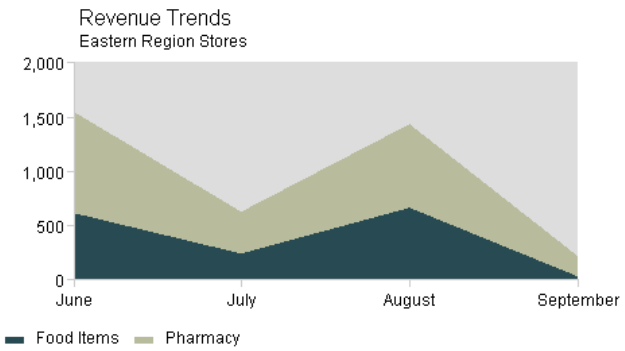
In a chart tag, charts in the javachart.servlet package are identified by the class name. For example, if you wanted to use a column chart, you'd use a tag like this:

```
<chart:streamed chartType="columnApp" … />
```

Area Charts

An area chart uses polygons to describe trends. This type of chart is most appropriate for trends that include cumulative values. For example, an area chart may be most appropriate for displaying revenue trends for several categories. The overall trend appears at the top, while each item's contribution would appear as a layer.

com.ve.kavachart.servlet.areaApp

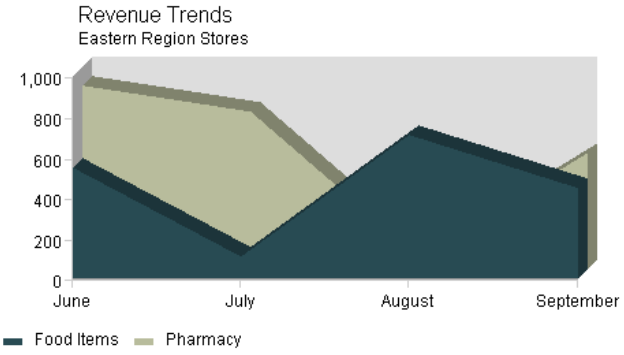


AreaApp ignores your X value specifications and assumes the values are 0, 1, 2, 3, … This ensures that the areas will align properly. Use the xAxisLabels parameter to specify your actual labels.

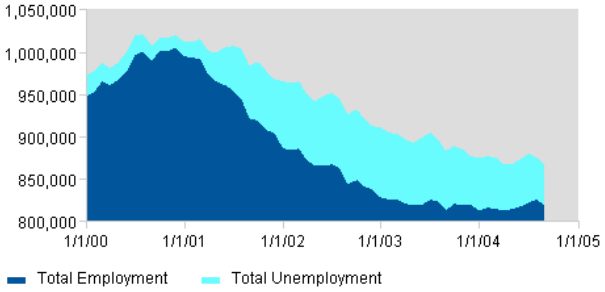
Because area charts are used to display cumulative trends, they don't generally give a clear idea of where individual data points are. For this reason, they're most appropriate for general trends. Also, dwell labels and hyperlink hot spots run from mid-point to mid-point for this type of chart.

It's important for area charts with multiple datasets to use the same X values for every dataset. Otherwise the areas cannot stack properly.

Note that un-stacked, 3D area charts are problematic. Areas can become completely obscured, as in the final observation in the chart below:



com.ve.kavachart.servlet.dateAreaApp



AreaApp ignores your X value specifications and assumes the values are 0, 1, 2, 3, … This ensures that the areas will align properly. Use the xAxisLabels property to specify your actual labels.

DateAreaApp assumes that X values are timestamps as described in the section above on data for time oriented charts. It also uses X axis parameters for time oriented charts.

Because area charts are used to display cumulative trends, they don't generally give a clear idea of where individual data points are. For these charts,

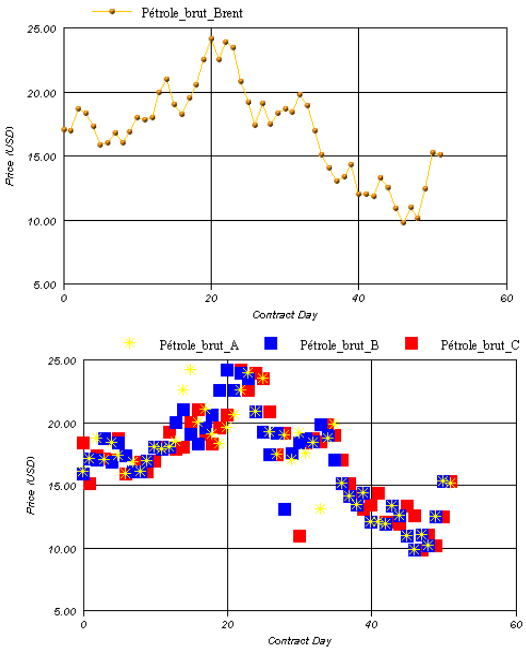
"getLinkMap()" define regions that go from mid-point to mid-point for each observation.

Property	value type	effect
baseline	double	sets the baseline value for this area
stackAreas	true/false	determines whether the areas will be stacked on top of each other (default is true)

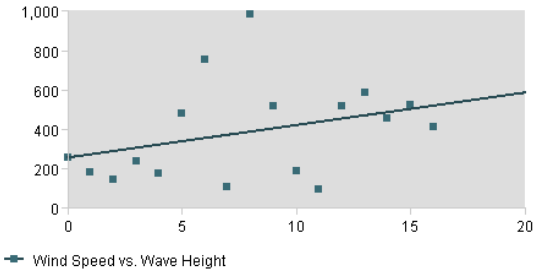
Line and Scatter Charts

These charts include:

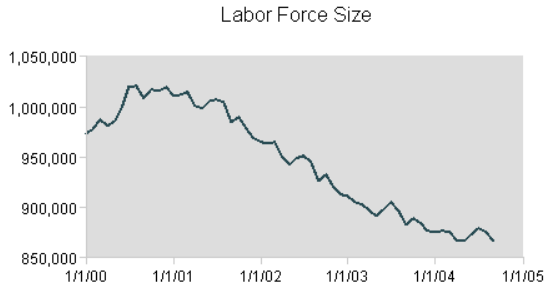
com.ve.kavachart.servlet.lineApp



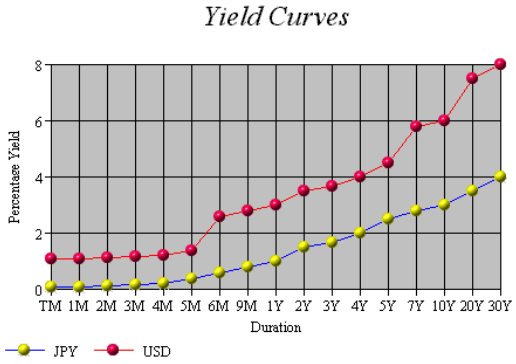
com.ve.kavachart.servlet.regressApp



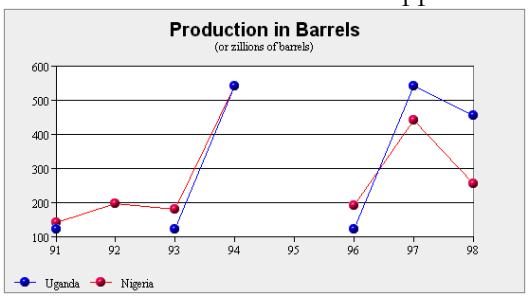
com.ve.kavachart.servlet.dateLineApp



com.ve.kavachart.servlet.labelLineApp



com.ve.kavachart.servlet.disLineApp



com.ve.kavachart.servlet.disLabelLineApp

In general, these charts can be used as conventional line charts, with or without markers at each vertex. Plot lines can be turned off with the "plotLinesOff" property or the dataset0Color property. If markers are turned on with lines turned off, these charts become scatter plots. You can plot some dataset lines and make others invisible by setting dataset0Color to "transparent" for the scatter-only datasets.

Chart objects that begin with "dis", such as disLineApp, support discontinuous data. They will create line breaks where data is missing. See the data section above to understand how to define discontinuities

DateLineApp uses time oriented data, as discussed in the data section above. These charts also recognize properties for formatting time oriented axes, discussed above.

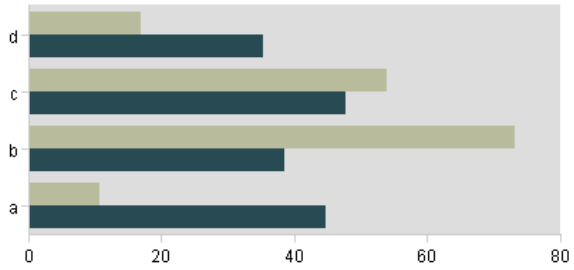
RegressApp performs a simple linear regression calculation on the chart's data values. Markers appear at the actual data points, while the line is drawn according to the regression's prediction. This is a classic "scatter plot" that shows positive, negative, or no correlation, and gives visual feedback about the strength of that correlation.

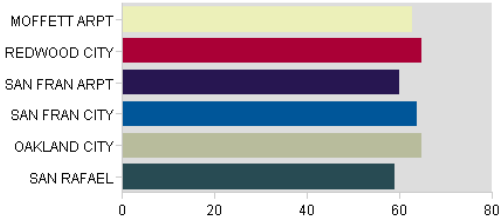
Property	value type	effect
plotLinesOn	anything	plot lines should display (default)
plotLinesOff	anything	Create a scatter plot by making plot lines invisible
individualMarkers	true/false	If markers are used, this parameter determines whether or not the datum markers will be used rather than the dataset marker (default is false).

Bar and Column Charts

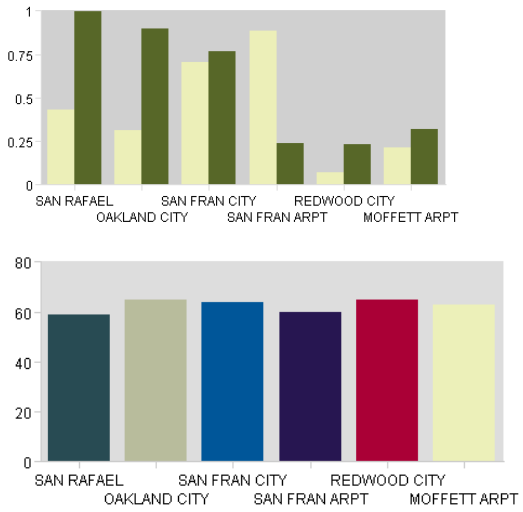
This category includes both charts with vertical and horizontal bars, as well as hi-lo bars. The charts are:

com.ve.kavachart.servlet.barApp

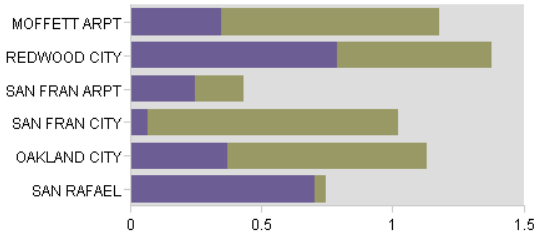




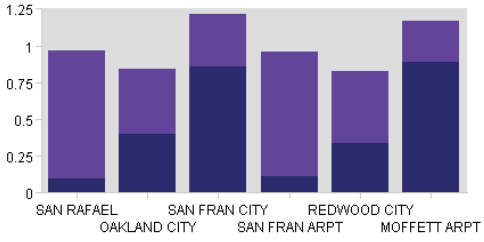
com.ve.kavachart.servlet.columnApp



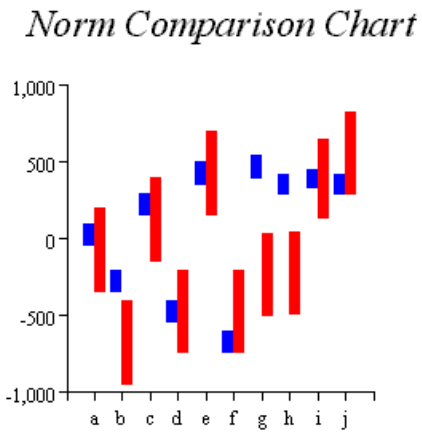
com.ve.kavachart.servlet.stackBarApp



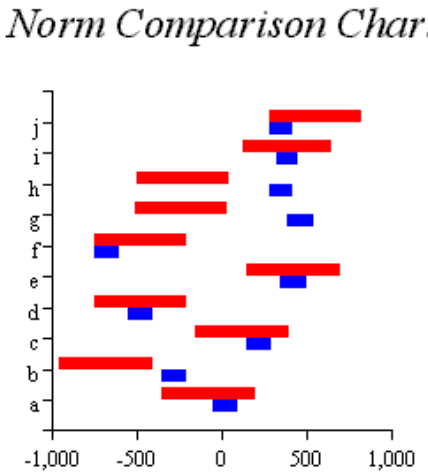
com.ve.kavachart.servlet.stackColumnApp



com.ve.kavachart.servlet.hiLoBarApp



com.ve.kavachart.servlet.hHiLoApp (horizontal hi lo bars)

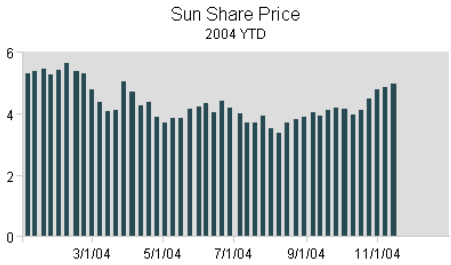


com.ve.kavachart.servlet.dateBarApp

com.ve.kavachart.servlet.dateColumnApp

com.ve.kavachart.servlet.dateStackBarApp

com.ve.kavachart.servlet.dateStackColumnApp



Bar charts have variable bar width, an adjustable baseline, and labels that can be toggled on or off. If you don't include a parameter to define X axis labels, this chart will use item labels (parameter dataset0Labels) beneath each bar. If item labels aren't defined, this chart will display each bar's Y value beneath it.

If you want each bar to have a different color, set the property "individualColors" to true, and define the colors with "dataset0Colors".

StackBarApp and stackColumnApp stack datasets instead of clustering them. This is useful to display a cumulative summary along with the individual data.

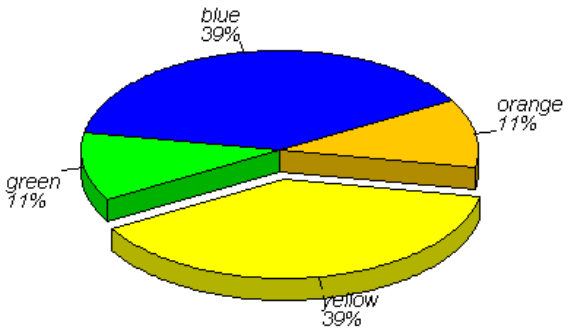
Charts that begin with "date" use time oriented data, as described above. These charts also support the use of time oriented axis formatting parameters, as described above.

If you're in a Java 2 or newer environment, dataset image parameters will cause your bars to be drawn using tiles of the specified image.

Property	value type	effect
barBaseline	double	bars ascend or descend from this value
barClusterWidth	double	This determines how wide each bar should be. If the value is 1.0, bar 1 will touch bar 2. If the value is 0.5, each bar will take 50% of the available space. If you have more than one data series defined, this value describes the total width of a cluster of bars.
individualColors	true/false	In bar/column charts that normally use only the Dataset color for drawing bars this will determine whether datum colors should be used instead (default is false).
useValueLabels	true/false	Determines whether y values or data labels are used for bar labels.
dataset0y2Values	List	This list of numbers is used to add error bars to each bar (Note: error bars also require "X" values to operate
errorBars	true/false	Determines whether error bars should be displayed

Pie Charts

Pie Charts are drawn with com.ve.kavachart.servlet.pieApp.



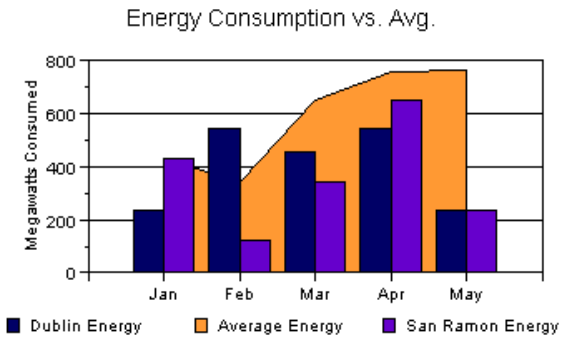
Pie charts can toggle percentage, value, and textual labels. They can also set a beginning angle value, and can set an exploded slice for emphasis. Pie chart colors are defined with the parameter dataset0Colors. Pie charts ignore datasets beyond dataset0.

Pie Chart Properties	value type	effect
explodeSlice	integer	slice number to explode
explodeSlices	list of doubles	This will be list of explosion values for each slice. Explosion values should be between 0 and 1, but generally pretty close to 0. The default value when a slice is exploded with explodeSlice is .05
textLabelsOn	anything	make string labels visible
textLabelsOff	anything	make string labels invisible (default)
valueLabelsOn	anything	make numeric labels visible
valueLabelsOff	anything	make numeric labels invisible (default)
percentLabelsOn	anything	make percentage labels visible (default)
percentLabelsOff	anything	make percentage labels invisible
percentPrecision	integer	the number of digits of precision for Pie percent labels
labelPosition	integer	0: at center of slice, 1: at edge of slice, 2: outside edge of slice with pointer
startDegrees	integer	degrees counterclockwise from 3 o'clock for first slice
xLoc	double	x Location for center of pie (between 0 & 1, default 0.5)
yLoc	double	y Location for center of pie (between 0 & 1, default 0.5)
pieWidth	double	% of window for pie diameter (default .6 = 60%)
pieHeight	double	% of window for pie diameter (default .6 = 60%)
pointerLengths	list	a values to redefine the pointer lengths for external labels. By default, this value is 0.2.
lineColor	Color	redefines the color used for pie slice pointers

Combinations: Bar-Area Chart

BarArea charts layer bars over areas, with shared axes. BarArea charts have variable bar width, and labels that can be toggled on or off. If you don't include a parameter to define X axis labels, this chart will use item labels (param dataset0Labels) beneath each bar. If these labels aren't defined, this chart will display each bar's Y value beneath it. Data series can be assigned to either Bar or Area style charting. Bars draw over areas, and may be stacked or clustered. Areas are always stacked.

com.ve.kavachart.servlet.barAreaApp

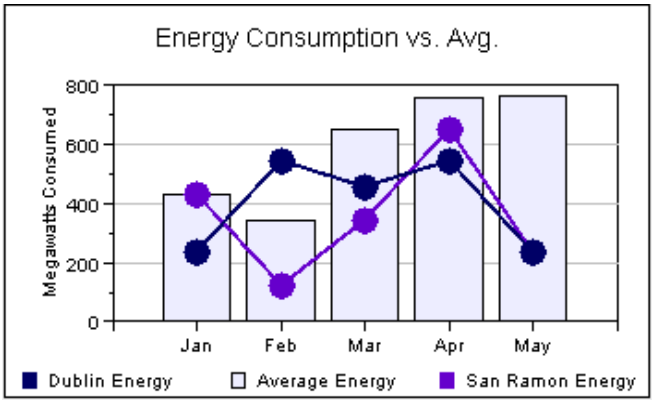


Parameter	value type	effect
datasetNType	Bar|Area	dataset N will be either Bar or Area, based on this value.
stackedBar	true|false	If "true", bars will be stacked, one series upon another.
barBaseline	double	bars ascend or descend from this value
barClusterWidth	double	This determines how wide each bar should be. If the value is 1.0, bar 1 will touch bar 2. If the value is 0.5, each bar will take 50% of the available space. If you have more than one data series defined, this value describes the total width of a cluster of bars.
barLabelsOn	true|false	determines whether labels will be drawn above each bar
barLabelAngle	integer	degrees to rotate bar labels
barLabelPrecision	integer	digits of precision for the bar labels
useValueLabels	true|false	determines whether the bar labels will be dataset labels or y values

Combinations: Bar-Line Chart

Bar-Line charts layer lines over bars, with shared axes. BarLine charts have variable bar width, and labels that can be toggled on or off. If you don't include a parameter to define X axis labels, this chart will use item labels (param dataset0Labels) beneath each bar. If these labels aren't defined, this chart will display each bar's Y value beneath it. Data series can be assigned to either Bar or Line style charting. Lines draw over bars, and bars may be stacked or clustered.

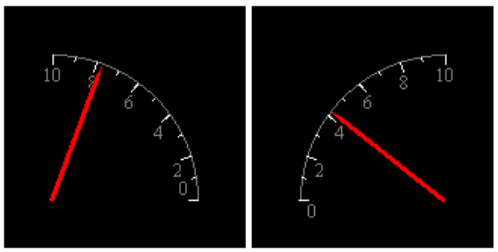
com.ve.kavachart.servlet.barLineApp

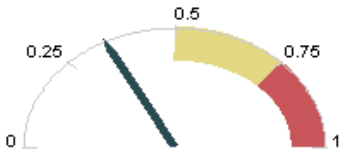


Parameter	value type	
datasetNType	Bar|Line	dataset N will be either Bar or Line, based on this value.
stackedBar	true|false	If "true", bars will be stacked, one series upon another.
barBaseline	double	bars ascend or descend from this value
barClusterWidth	double	This determines how wide each bar should be. If the value is 1.0, bar 1 will touch bar 2. If the value is 0.5, each bar will take 50% of the available space. If you have more than one data series defined, this value describes the total width of a cluster of bars.
barLabelsOn	true|false	determines whether labels will be drawn above each bar
barLabelAngle	integer	degrees to rotate bar labels
barLabelPrecision	integer	digits of precision for the bar labels
useValueLabels	true|false	determines whether the bar labels will be dataset labels or y values

Speedos

These include com.ve.kavachart.servlet.speedoApp and com.ve.kavachart.servlet.hSpeedoApp. The only difference between these two Is that hSpeedoApp adds a history mark in the background; a sort of high water mark.





Speedo charts have adjustable axis locations and styles, as well as adjustable needle styles. This chart can be particularly useful in conjunction with an image background to superimpose a dial and needle on a scanned image of a physical gauge.

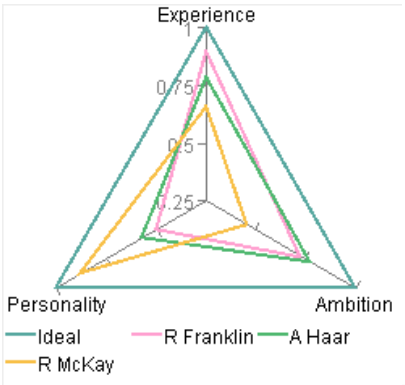
Speedos only use the first value of dataset 0. However, the other values in dataset 0 are considered for building the speedo's scale.

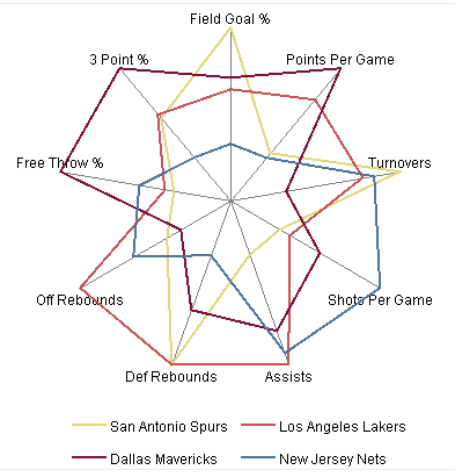
Speedo Chart properties	value type	effect
needleStyle	integer	Kind of needle (default 1) 0 = arrow, 1 = line, 2 = thick arrow, 3 = swept arc
speedoPosition	integer	0 (default) is a mostly complete circle, 1 - 4 are semi circles in various positions, 5-8 are quarter circles in various positions

labelsInside	anything	labels on the inside of the speedo
labelsOutside	anything	labels on the outside of the speedo
watermarkColor	color	for hSpeedoApp, determines the color of the history watermark

Radar Charts

KavaChart "radar charts" or "polar charts" are also called "Kiviat Diagrams". These charts draw multiple spoke axes, with a line for each dataset encircling the center. By default, these charts assume one axis spoke per observation, and they assume that all datasets have the same number of observations.

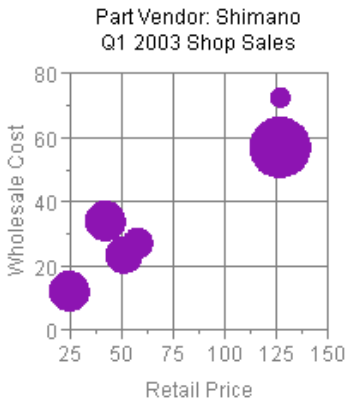




Polar Chart Properties	value type	effect
manualSpoking	true|false	If defined, you are responsible for determining how many "spokes" should be drawn in this chart's axis representation
numSpokes	integer	The number of spokes in this chart's Axis system (default 4)

Bubble Charts

Use com.ve.kavachart.servlet.bubbleApp to build a bubble chart. This chart draws circles at X,Y values specified by dataset0xValues and dataset0yValues. The size of the circle is determined by dataset0y2Values.



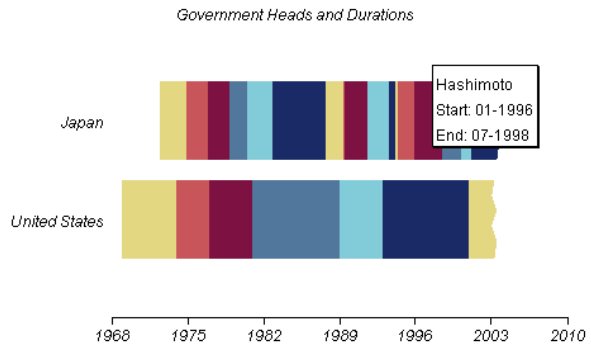
These charts may have filled or hollow circles, crossing X and Y axes, and manual or automatic Z scaling. Z scaling refers to the relative size of the bubbles, based on the overall set of Z (y2) values.

Bubble Chart Properties	value type	effect
zAutoScaleOff	anything	Indicates that you want to set the Z scaling (in terms of a percentage of the Y axis scale.
setZScale	double	Sets the size of bubbles, relative to Y axis units. For example, if the y2 value for a particular bubble is 10 and zScale is set to 2, then the bubble's diameter will be twice as big as a 10 unit increment on the Y axis.
crossAxes	Boolean	Determines whether the X and Y axes should cross. If true, the default crossing value is 0, 0.
xCrossVal	double	Where the Y axis should cross the X axis.
yCrossVal	double	Where the X axis should cross the Y axis.

Gantt Charts

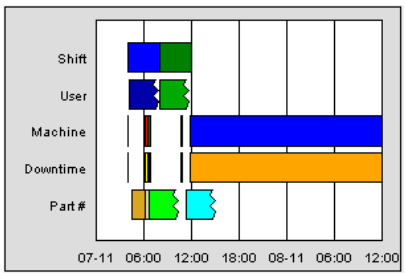
Gantt charts are a specialized chart designed to show when tasks start and end. This sort of chart is particularly useful for resource allocation and project planning, but it can also be used to visually describe the progress of multiple projects or processes.

com.ve.kavachart.servlet.ganttApp



This chart uses the special params dataset0StartDates and dataset0EndDates to describe the start and end of each colored bar on track "0". Each dataset is arrayed along a single track. In the example above, we're using dataset0 and dataset1 to represent United States and Japanese leader's tenure, respectively. The tooltip label shows the start and end value along with the label (leader's name in this case)

A "discontinuity", or invalid value, like "x" in place of a date creates a torn edge, like the end point on the United States bar, when the property "useTearEdge" is set to "true".



Another special property for this chart, minBarWidth, ensures that very narrow bars, like those in the applet above, will remain visible.

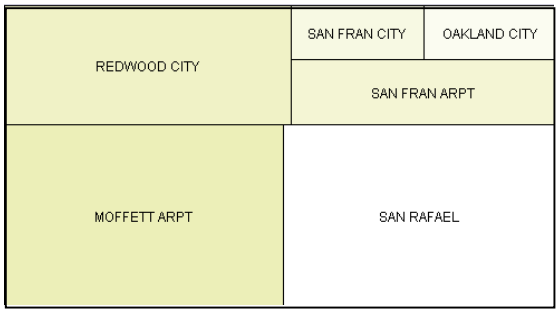
If you're using a DataProvider class to supply data to this kind of chart, start and end values are provided with Y and Y2 values respectively. These values are derived from the long value returned by java.util.Date.getTime(), or java.sql.Date.getTime().

Parameter	value type	effect
dataset0StartDates	list	A list of dates in "inputDateFormat" format, describing the start times/dates for each item in a particular row. Datasets 0 through 39 are available. Dataset names are used to label the vertical axis.
dataset0EndDates	list	A list of dates in "inputDateFormat" format, describing the ends for each bar segment in a particular row. An un-parseable date, like "XX", would be interpreted as an incomplete task.
dwellLabelDateFormat	Date format	A format string to describe start and end dates
dwellStartString	String	This string defines the dwell label string for the start date. This string should have 'XX' characters where the date will occur. Default is "Start XX"
dwellEndString	String	This string defines the dwell label string for the end date. Default is "End XX"
dwellIndefiniteString	String	This string defines the dwell label string for an indefinite start/end. Default is "Indefinite"

Sectormap Charts

Sectormap charts are very efficient visuals for displaying certain kinds of data. The size of each square in a sectormap represents its relative size (Y value) within the dataset, and the color of the rectangle represents another factor, such as price change (X value). Each dataset is bounded by a rectangle that represents the Dataset's overall contribution to Y values for the entire set of datasets.

com.ve.kavachart.servlet.sectorMapApp



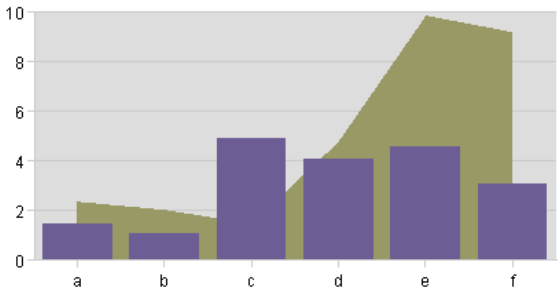
A sectormap could be used to represent financial values in a customer's portfolio, for example, where each data represents a market sector (e.g. finance, transportation, utilities, etc.), and each item in the dataset represents a particular security in that sector. You can tell at a glance how your portfolio is performing, which sectors are doing well in the displayed time period, and which stocks are having the most impact on your portfolio.

Parameter	value type	effect
individualColors	True|false	Determines whether colors should come from "dataset0Colors"
gradientColoring	True|false	Determines whether colors should be auto-graduated from the dataset color to the "secondary color"
sectorSecondaryColor	Color	A second color to be used for gradient coloring
baseColor	Color	A color to be used as a neutral value when "baseValue" is used, giving effectively a 2 dimensional gradient – dataset color to base color to secondary color
baseValue	Double	A value to be used for the baseColor.

Combinations: Bar-Area Chart

BarArea charts layer bars over areas, with shared axes. BarArea charts have variable bar width, and labels that can be toggled on or off. If you don't include a property to define X axis labels, this chart will use item labels (defined with dataset0Labels) beneath each bar. If these labels aren't defined, this chart will display each bar's Y value beneath it. Data series can be assigned to either Bar or Area style charting. Bars draw over areas, and may be stacked or clustered. Areas are always stacked.

com.ve.kavachart.servlet.barAreaApp

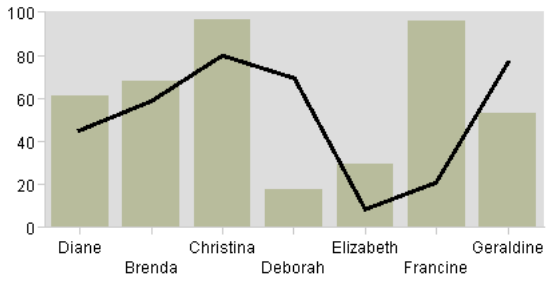


Property	value type	effect
datasetNType	Bar|Area	dataset N will be either Bar or Area, based on this value.
stackedBar	true|false	If "true", bars will be stacked, one series upon another.
barBaseline	double	bars ascend or descend from this value
barClusterWidth	double	This determines how wide each bar should be. If the value is 1.0, bar 1 will touch bar 2. If the value is 0.5, each bar will take 50% of the available space. If you have more than one data series defined, this value describes the total width of a cluster of bars.
barLabelsOn	true|false	determines whether labels will be drawn above each bar
barLabelAngle	integer	degrees to rotate bar labels
barLabelPrecision	integer	digits of precision for the bar labels
useValueLabels	true|false	determines whether the bar labels will be dataset labels or y values

Combinations: Bar-Line Chart

Bar-Line charts layer lines over bars, with shared axes. BarLine charts have variable bar width, and labels that can be toggled on or off. If you don't include a parameter to define X axis labels, this chart will use item labels (param dataset0Labels) beneath each bar. If these labels aren't defined, this chart will display each bar's Y value beneath it. Data series can be assigned to either Bar or Line style charting. Lines draw over bars, and bars may be stacked or clustered.

com.ve.kavachart.servlet.barLineApp

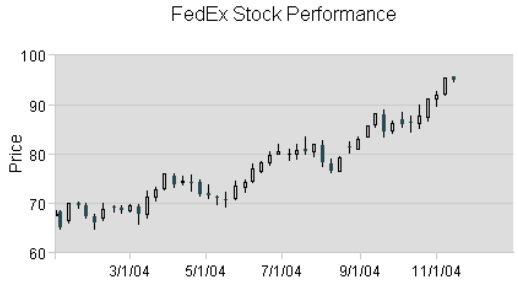


Property	value type	effect
datasetNType	Bar|Line	dataset N will be either Bar or Line, based on this value.
stackedBar	true|false	If "true", bars will be stacked, one series upon another.
barBaseline	double	bars ascend or descend from this value
barClusterWidth	double	This determines how wide each bar should be. If the value is 1.0, bar 1 will touch bar 2. If the value is 0.5, each bar will take 50% of the available space. If you have more than one data series defined, this value describes the total width of a cluster of bars.
barLabelsOn	true|false	determines whether labels will be drawn above each bar
barLabelAngle	integer	degrees to rotate bar labels
barLabelPrecision	integer	digits of precision for the bar labels
useValueLabels	true|false	determines whether the bar labels will be dataset labels or y values

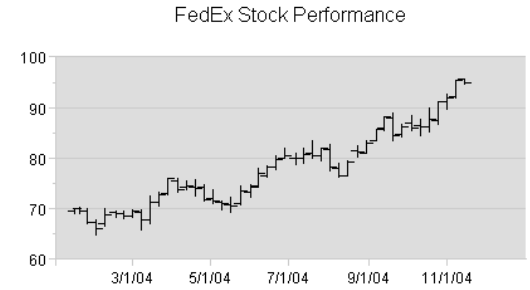
Candlestick and OHLC Charts

This collection includes some standard charts for dealing with financial data: com.ve.kavachart.servlet.candlestickApp and com.ve.kavachart.servlet.hLOCApp use 4 Y values for each observation at a single date or time. These are the high, low, open, and close prices for a particular time period.

com.ve.kavachart.applet.candlestickApp



com.ve.kavachart.applet.hLOCApp



Special properties for these charts:

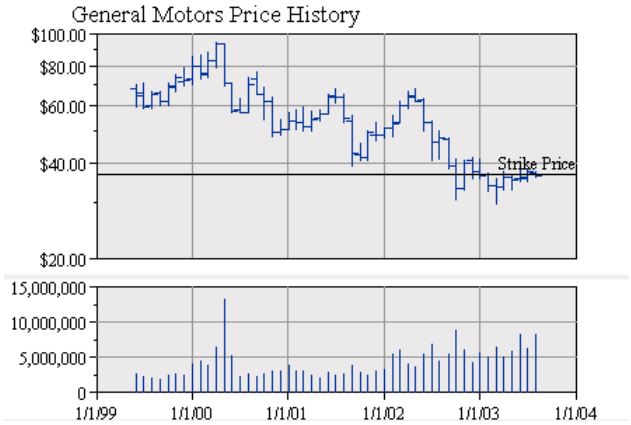
Parameter	value type	effect
dataset0highValues	list	High price at observed dates
dataset0lowValues	list	Low price at observed dates
dataset0openValues	list	Open price at observed dates
dataset0closeValues	list	Close price at observed dates
dataset0dateValues	list	List of dates in "inputDateFormat"
CustomDatasetHandler	URL	A url containing rows of date,open,high,low.close values

com.ve.kavachart.servlet.hiLoCloseApp is very similar to Candlestick and OHLC charts, but it uses 3 Y values for each time period. These represent the high, low, and closing prices for a particular time period. Close data is provided with dataset Y values, high data is Y2 and low data is Y3.

Stick Charts

com.ve.kavachart.servlet.stickApp is similar to a bar chart, but draws a narrow bar, or "stick' at each time period. The width of these bars can be specified in pixels. Multiple datasets do not stack or cluster.

This chart is frequently combined with a hi-lo-close, candlestick or ohlc chart to display price over volume:



Combination Charts

com.ve.kavachart.servlet.finComboApp combines hiLoClose, line, and stick elements into a single chart with multiple windows. The "splitWindows" parameter determines whether all datasets will appear in a single window, or each dataset should appear in a unique window.

The best way to supply data to these charts is through an implementation of com.ve.kavachart.utility.DataProvider. The datasets provided by this DataProvider should supply datasets that contain com.ve.kavachart..parts.CandlestickDatum classes. See the demos for several examples that supply candlestick data. You can also download one of these DataProviders here:

http://www.kavachart.com/sample_classes/examples.zip

Finance charts can also read data from a URL specified through the parameter "customDatasetHandler". The expected input stream has a column of dates or times in the format specified by the "inputDateFormat" parameter, and then a number of columns of Y data. Each dataset consumes the number of columns appropriate for its data type. For example, in a candlestick chart, each dataset uses the first column as the X axis period, and then uses 4 columns for high, low, open, and close data. A stick would use the first column for the date or time, and then use a single column for each dataset's Y (or price, volume, etc.) values.

Property	value type	effect
datasetNType	HLOC|Stick|Line	dataset N will be either Stick, HLOC, or Line, based on this value. (finComboApp only).
splitWindow	true|false	if true (default) each dataset type will be in a a separate window with an independent Y axis. The X axis will be shared among all dataset types.
stickWidth	Integer	Width (in pixels) of stick bars (stickApp only).

Some of the multiple axis combination charts and time oriented charts are frequently used for financial data.

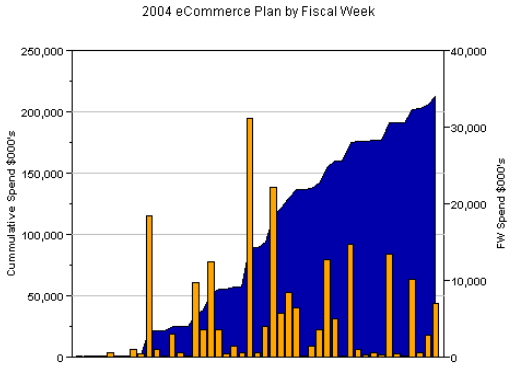
The KavaChart Enterprise Edition also includes a "kcfinance" package, which is specifically designed to support most common finance charts. This package takes some coding to attach data sources properly, but it's much more sophisticated than the more basic server classes at representing financial data. "Kcfinance" is especially well suited for generating images on a server.

It's also worth noting that there are also several finance-oriented charts in the com.ve.kavachart.contrib. package, which is part of the KavaChart Enterprise Edition. These include charts that overlay markers on candlestick charts, box-jenkins statistical charts, and histograms. Also, with a little bit of Java programming, you can combine various KavaChart elements into an endless variety of custom finance charts. These elements are described in more detail in other chapters, as well as documentation on how to put these custom charts into the same applet or servlet framework as standard off-the-shelf KavaChart charts.

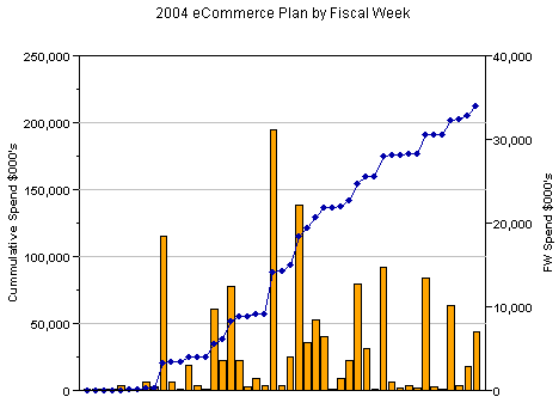
Combinations: Multiple Axis Charts

Many combination charts are more useful if elements are assigned to different Y axes. For example, you might want to compare trends for baseball scores and basketball scores in the same chart. Baseball scores will be much lower, but there still might be some discernable trend. In this case, you could just use twinAxisLineApp to assign baseball scores the the right axis, and basketball scores to the left axis.

twinAxisBarAreaApp: assigns bar data to the left axis and area data to the right (auxAxis).

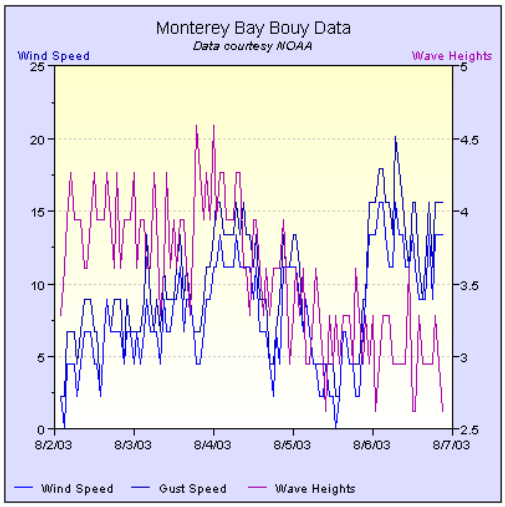


twinAxisBarLineApp: assigns line data to the left axis and bar data to the right (auxAxis).



twinAxisDateComboApp: uses time oriented data, and time oriented axis parameters for the X axis. Datasets can be bar, line, area, or stick, and may be assigned to either left or right axes.

twinAxisDateLineApp: uses time oriented data, and time oriented axis parameters. Datasets are assigned to the left axis by default, and the right (auxAxis) by parameter.



twinAxisLineApp: uses numeric X values. Datasets are assigned to the left axis by default, and the right (auxAxis) by parameter.

twinAxisStackBarLineApp: uses a Line element for the left axis, and a StackBar element for the right axis. Axis assignment is implied by the dataset type.

To change the colors, fonts, title, scaling, etc. for the right axis, use "auxAxis" in place of "yAxis". For example, to set the title, you would use the parameter "auxAxisTitle" for the right, and "yAxisTitle" for the left.

Property	value type	effect
datasetNType	Bar | Line | Area | Stick	This determines the DataRepresentation for datasetN. "Area" is only available for TwinAxisBarAreaApp, Line is not available for TwinAxisBarAreaApp, and so on. Stick is only available for TwinAxisDateComboApp
datasetNonRight	true|false	This determines whether dataset N will be assigned to the standard left axis or the auxilliary right axis. Only applicable to twinAxisDateLineApp, twinAxisLineApp, and twinAxisDateComboApp. Other charts assign one data representation type (e.g. bar) to the primary axis, and the other (e.g. area) to the auxiliary axis.
plotLinesOn	anything	plot lines should display (default). Applicable to all of the Twin

		Axis Charts except twinAxisBarAreaApp.
plotLinesOff	anything	Create a scatter plot by making plot lines invisible. Appicable to all of the Twin Axis Charts except twinAxisBarAreaApp.
auxPlotLinesOn	anything	plot lines should display (default). Applicable to twinAxisDateLineApp, twinAxisLineApp, and twinAxisDateComboApp.
auxPlotLinesOff	anything	Create a scatter plot by making plot lines invisible. Applicable to twinAxisDateLineApp, twinAxisLineApp, and twinAxisDateComboApp.
barBaseline	double	Bars ascend or descend from this value. Also applicable to Sticks in twinAxisDateComboApp.
barClusterWidth	double	This determines how wide each bar should be. If the value is 1.0, bar 1 will touch bar 2. If the value is 0.5, each bar will take 50% of the available space. If you have more than one data series defined, this value describes the total width of a cluster of bars.
barLabelsOn	true|false	determines whether labels will be drawn above each bar
barLabelAngle	integer	degrees to rotate bar labels
barLabelPrecision	integer	digits of precision for the bar labels
useValueLabels	true|false	determines whether the bar labels will be dataset labels or y values
areaBaseline	double	sets the baseline value for this area
auxBarBaseline	double	sets the baseline value for the Sticks assigned to the aux axis in TwinAxisDateComboApp
barWidth	integer	sets the width in pixels of the Sticks in TwinAxisDateComboApp
auxBarWidth	integer	sets the width in pixels for the Sticks assigned to the aux axis in TwinAxisDateComboApp

Using a Properties Object or File

In our scriptlet examples so far, we have always used the "setProperty" method to assign property values. There are other ways to set chart properties, and you can combine the various property setting techniques to optimize your server chart implementation.

Property Files

You can store your chart properties in an external properties file, which the chart object reads during chart generation. This file is specified with the property named "properties".

Putting properties in an external file has some significant benefits. If your charts are very different from the default charts, your JSP or servlet can become cluttered with calls to "setProperty", making your code appear more complicated than it really is. This sort of inline coding is also not particularly portable, except by doing cut-and-paste operations. Putting all the properties in your code also makes you responsible for the overall style and appearance of the chart, and there might be someone in your organization better suited to creating stylish charts.

By putting your properties in an external file, your code can focus on the task of acquiring data and translating it into dynamic page content. The properties file can contain all non-dynamic page content, and can be used for multiple chart types to create uniform colors, fonts, etc.

> **Note:**
>
> If you're using a ChartTag, properties files are used for the "style" attribute. These persist in your application memory unless you override this feature by specifying that you want to reload styles with every image.

You can also use a single properties file for several applications, even though the data acquisition logic may vary dramatically.

A properties file is a text file that looks like this:

```
titleString=Annual Sales
backgroundColor=00ffa6
plotAreaColor=00ff77
dataset0Color=green
yAxisLabelFont=Arial,12,0
```

And so on.

Constructor Properties

You can also construct your chart objects with a pre-existing java.util.Properties object. This approach is very efficient for some application architectures.

For example, if you're building a servlet, you can put the Properties into a class variable that is instantiated one time, when the servlet is instantiated. Even though you will create a new chart object every time the servlet services a request, you can reuse the exisiting Properties object. In this case, your servlet may store the properties in an external file, but that file is read only one time, when the servlet starts. After that, you get the benefits of an external properties file without having to re-read it to service each request.

The online chart wizard exports definitions like this. See the servlet or JSP scriplet output produced by the wizard for specific examples.

Chapter

7

Image Format Recommendations

KavaChart supports most popular image formats. Which one is best for your application? That depends on the application, but every image encoder has pluses and minuses. These are discussed below

GIF

The GIF89a image format is probably the most widely used on the worldwide web. It provides excellent compression for images that have 256 colors or less, and has some nice features, like transparency and animation.

KavaChart uses a built-in GIF generator when the "imageType" property is set to "gif". This image encoder reduces all images to 256 colors by dithering, and lacks support for transparency.

If you require GIF output with no dithering, or GIF output with transparency support, see the chapter on programming server objects for more information about installing custom image encoders.

JPEG

The JPEG image format is ideal for high color images, such as photographs.

Select an imageType of "jpeg" to use a native JPEG encoder that is available on most Java 2 runtime environments. Despite the promise of high performance from native code, in practice, the java based encoders (imageType=j_jpeg) are nearly as fast.

JPEG has some drawbacks when rendering chart output. Because it's designed for photographic images, and uses a lossy compression algorithm, lines (such as axis lines, bar outlines, etc.) may appear blurry. Depending on your situation, however, the image quality might be appropriate.

JPEG also lacks support for transparency. This image format is supported by every web client. Also, this image format requires no special licensing for production use.

PNG

PNG is KavaChart's default image format. Portable Network Graphics (PNG) holds the promise of eventually replacing GIF, and supplanting JPEG for some applications. If you are generating charts for browsers that support PNG, this is an ideal format for KavaChart output.

Unfortunately, some browsers (notably many older Macintosh browsers) lack support for this format, which may limit its usefulness in your production environment.

Select PNG with the property "imageType=j_png".

Flash

KavaChart can produce Macromedia Flash "movies" that represent charts. Flash has a number of advantages over other image formats: It's a "vector" format, which means that the information is rendered on the client browser. Definitions can be scaled (zoomed in and out by a user), and they carry a built-in set of tooltips and hyperlinks with the flash data stream. KavaChart's highlighting and tooltip behavior is much more animated than the default tooltip labels associated with other image types. Flash output is also "antialiased", which removes the jagged edges associated with the edges of pie charts or diagonal lines. Flash output is very highly compressed, so users perception of performance tends to be very good. The Macromedia Flash plug-in is installed by default on most browsers.

On the other hand, some users are in organizations that don't permit the Flash plug-in. Flash output can't be copy/pasted into other applications. Also some users dislike the slight "blurring" of character edges caused by Flash antialiasing.

In a high-volume production environment, Flash has the advantage of being a vector and polygon format; it doesn't require as much image memory on the server as generating true image formats.

Because Flash output is not image output, you'll need to place this output into your own OBJECT and EMBED tags if you're not using the KavaChart tag library. See Macromedia's developer site for more information about the correct syntax for these tags.

Select the Flash format with "imageType=flash" or "imageType=swf".

SVG

Another vector format available through KavaChart is "Scalable Vector Graphics". SVG support is available in some browser installations, although it's not as widespread as Flash or image support. KavaChart's SVG support is very similar its Flash output: highly animated tooltips and highlighting, highly compressed vector output, etc. You can download and install SVG support from Adobe's SVG web pages.

Although some browsers will recognize and handle SVG as a MIME type, you will generally need to use EMBED tags to handle this data type with Adobe's plug-in.

Select SVG format with "imageType=svg".

BMP

The BMP format is supported in most Windows applications, and is a non-lossy image format, so the images are crisp. Unfortunately, charts created in BMP format are not compressed, and image data can be very large. BMP files are typically not appropriate for web based documents, and are not supported in all browsers.

Select BMP format with "imageType=j_bmp".

Other Formats

Other KavaChart formats, such as XBM, XPM, and ICO are not widely used for document images. These formats are lossless (except for color data in XBM), but lack compression. If you require one of these image formats, you will likely understand the implications of its use.

External Formats: PostScript, GIF Transparency

KavaChart server objects have a plug-in architecture that permits you to define your own image or graphics encoder. The KavaChart Enterprise Edition includes code samples that implement two of the most popular image and graphics formats.

The first is a PostScript encoder, based on a public domain PostScript generator. The PostScript generator gives you high-quality, scalable output appropriate for printed documents.

The samples also include an implementation that uses the popular and free Acme GIF encoder to produce transparent GIF output.

To find out more about these external encoders, see the section in the programming chapter for information about installing image encoders.

Chapter

8

JSP ChartTag Overview

ChartTags hide the complexity of managing server chart imaging, making them the preferred method for building multi-tier JSP applications.

Setup

Integrate Taglib Descriptor and web.xml

KavaChart's Tag Library is described by the file kavachart-taglib.tld. This file contains an XML description of the various chart tags and inner tags, including a description of what tag attributes are available, required, and available for runtime evaluation.

Place this file in an appropriate location in your application directory hierarchy, such as WEB-INF/kavachart-taglib.tld.

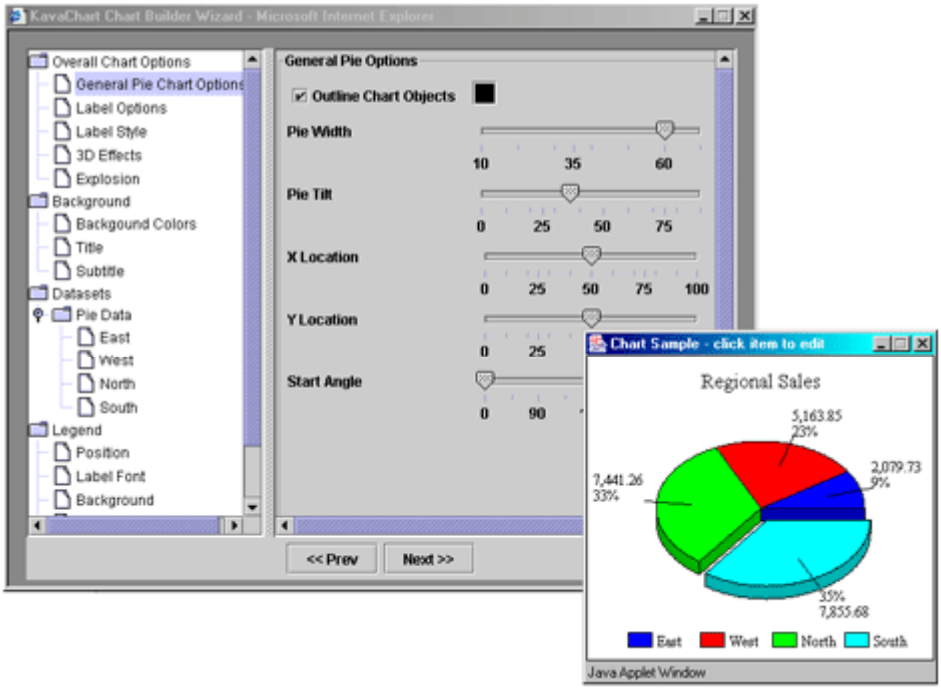
To integrate KavaChart tags into your own server, you'll also need to integrate the KavaChart web.xml into your own web.xml configuration file. KavaChart's web.xml includes <servlet> definitions, <servlet-mapping> definitions, and a <taglib> definition. Remember that XML is order-dependent, so you'll need to place these definitions in various places in your own web.xml file.

Create a Chart Style

Chart styles are stored in properties files. The chart tag reads these properties on startup, and then (by default) stores them in memory for subsequent chart generation requests.

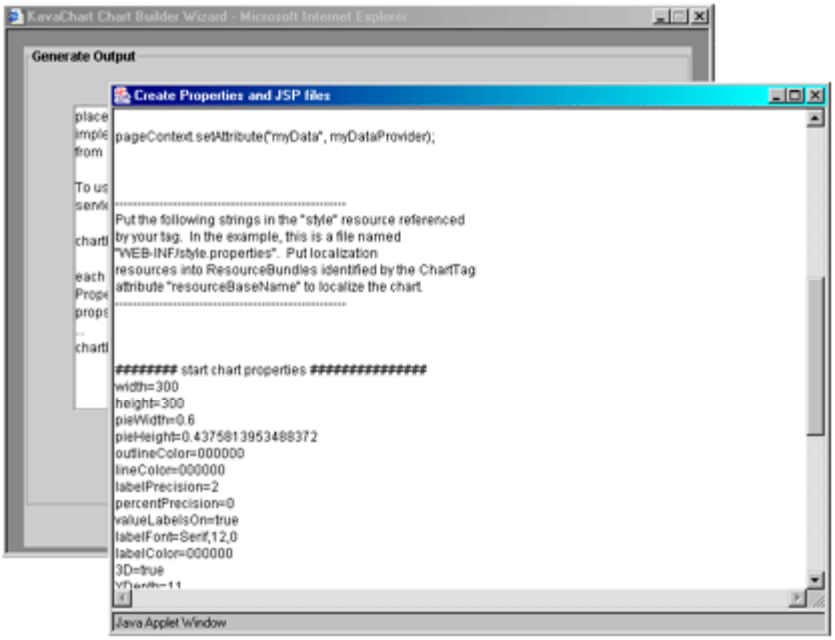
Use the ChartWizard

KavaChart's Chart Wizard generates chart properties for all of KavaChart's chart types. These properties are generally also applicable to custom charts.



The ChartWizard starts with a specific chart type and a pixel size for your target image, and lets you interactively edit your chart style. There are a variety of options in the wizard for simulating your data sources. Since your data sources will most likely provide live data, data definitions are not created as a part of the wizard's output.

After you have edited your chart style and stepped through the wizard's questions about your output targets, you'll end up with a complete chart tag and property sheet to integrate into your JSP.



Copy and paste the properties section of this output into a properties file. Copy and paste the tag definition into your JSP. Make sure you add the "include" statement, as described in the "Edit Your JSP" section of this document. If necessary, these definitions can be edited later with any sort of text editor.

Note that even though this style was created for a particular chart type, you can change the chart type as a part of your chart tag. You might start out with a stacked column chart, for example, and later decide that side-by-side columns better represent your information.

All that's left is to create a DataProvider that links your chart generation tags to a live data source.

Create a DataProvider

A DataProvider is an implementation of the KavaChart Interface "com.ve.kavachart.utility.DataProvider". By implementing this simple interface in your existing data sources, you can easily translate text and numeric information into graphics.

Your DataProvider can be used within the KavaChart Wizard, too. You can even create a graphical user-interface for your DataProvider for use with the Wizard. See Appendix C for more information about this topic.

The Interface:

```
public Interface com.ve.kavachart.utility.DataProvider {
     /*
      ** Returns an Enumeration of Dataset classes.
      */
     public Enumeration getDatasets();
     /*
      ** Returns a String that uniquely identifies this data.
      ** Needed to make chart image caching work properly.
      ** Otherwise unnecessary.
      */
     public String getUniqueIdentifier();
}
```

You probably already have some kind of data connection that provides you with data for a table, some summary figures, etc. It's a good idea to use the same connection to implement DataProvider so you don't have to make multiple database connections, or use up otherwise scarce server resources. You might want to implement DataProvider with your existing DataSource class.

Here's one example, based on a pre-existing database connection:

```
ArrayList chartData;
String query = "SELECT SUM(REVENUES) FROM WIDGETS WHERE QUARTER='Q1'";

public void getSomeData(){

     //-------- a bunch of database connection setup stuff deleted here...
     statement = connection.createStatement();
     myresult = statement.executeQuery(query);
     while (myresult.next()){ //retrieve next row
       int i=0;
       //you're going to do something else interesting with the data here...
       series0yValues.addElement(myresult.getString(i++));
       series0Labels.addElement(myresult.getString(i++));
     }

     //create some KavaChart data - this is one way to do it...
     //first series
     double yVals[] = new double[series0Values.size()];
     String labels[] = new String[series0Labels.size()];
     for(int i=0;i<yVals.length;i++){
       yVals[i] = Double.valueOf(series0Values.elementAt(i)).doubleValue();
       labels[i] = (String)series0Labels.elementAt(i);
     }
     Dataset set1 = new Dataset("series 1", yVals, labels, null);

     //put them into our Vector, declared as a Class variable:
     chartData = new ArrayList();
     chartData.add (set1);
}
```

```
//and finally the DataProvider implementation:
public Enumeration getDatasets(){
     return Collections.enumeration(chartData);
}

//a unique identifier to prevent chart collisions in a multi-threaded
     environment
public String getUniqueIdentifier(){
     return query; //not unique here, but it will be in your class
}
```

This code segment creates a KavaChart Dataset class, and then puts it into an ArrayList. It returns that ArrayList's elements, along with the query string. The query string is simply used as a unique identifier in the event that multiple threads are requesting charts at the same time. It won't appear anywhere in the HTML or image data generated by the chart tag.

As you can see, implementing the Interface consists of creating 1 or more Dataset classes and placing these into a List or other Enumeration. You're free to determine how these classes are constructed, what data to consider in the Dataset, whether labels or X values are relevant, and so on. Numerous Dataset constructors exist. Consult the KavaChart API documentation for more information on constructing Datasets.

Install your DataProvider

The chart tag retrieves the DataProvider from the page, session, or application context. To do this, the DataProvider must be placed into the context at some point during the page generation process.

One way to accomplish this is to create a JavaBean from your DataProvider, and install it as a part of the page definition:

```
<jsp:useBean id="dataId" scope="page"
        class="examples.RandomNumberDataProvider" />
```

This statement, for example, installs RandomNumberDataProvider into the ID "dataId". This class is provided with KavaChart's taglib example classes to provide sample data where no real data exists. It might also be useful in cases where you don't have a real DataProvider ready, but the page designer wants to proceed.

A more likely methodology involves installing the DataProvider with a small scriptlet, or as another part of your overall page generation code. For example, let's say we have a data acquisition class instance "myDataSource" that implements our DataProvider interface. We might create that class like this:

```
RandomNumberDataProvider myDataSource = new RandomNumberDataProvider();
```

And then you would install the DataProvider like this:

```
pageContext.setAttribute("dataId", myDataSource);
```

Since we named our DataProvider "dataId", the chart tag would reference it like this:

```
<chart:streamed
        dataProviderID="dataId"
        style="WEB-INF/myStyle.properties"
        chartType="lineApp" >
</chart:streamed >
```

When the tag is executed, a KavaChart "lineApp" server bean is created, data from the DataProvider "dataId" (actually the class instance "myDataSource") is applied to the bean, styles from "myStyle.properties" are applied, and the image bytes are streamed back into the page.

Installing your DataProvider in the most appropriate context will ensure the most efficient use of that resource. For example, DataProviders placed into the application context will persist for the lifetime of the application, and can be shared by all users who generate charts. This might be appropriate for shared information, like weather data. Page context resources will be discarded after each page is viewed. This is appropriate for charts that have page-unique information, like a pie chart showing a user's financial portfolio distribution.

Since data management is cleanly separated from the presentation layer, the DataProvider can change dramatically without altering the overall page presentation, and vice-versa.

Edit Your JSP

As we discussed above, KavaChart's ChartWizard will generate a chart tag with the appropriate attributes and properties file for the chart you created there. You can also directly edit and create chart tags and property sheets. Here's how to add a chart tag to your JSP:

Add an Include Statement

Add this line to your JSP:

```
<%@ taglib uri="http://www.ve.com/kavachart-taglib" prefix="chart" %>
```

This assumes that your web.xml file includes a definition for the kavachart-taglib.tld taglib descriptor, and that the taglib URI in that description matches the uri above. You can use any prefix you like. We'll use "chart" for our examples here.

Add a Chart Tag

Example:

```
<chart:streamed
      dataProviderID="myData"
```

```
      style="WEB-INF/myStyle.properties"
      chartType="pieApp" >
</chart:streamed >
```

Chart tags use a form like the example above. Each tag starts with a reference to the taglib you're using ("chart" in our case), and then the particular tag you're using ("streamed" in this example).

Chart tags can use several attributes. The three attributes used in this example are the most important:

1. **dataProviderID** supplies the data for this chart. See the section above for more information on how to create a DataProvider and set its ID.

2. **style** specifies the name of a properties file used to modify this chart's appearance. The "Styles" section above describes how to use the KavaChart Chart Wizard to create these properties files.

3. **chartType** specifies the KavaChart server bean to be used to build the chart image. You can use any of KavaChart's server chart beans, or you can build and use your own chart bean using the KavaChart server bean framework.

When the JSP is compiled into a servlet, this tag is translated into Java code. The tag approach optimizes the chart generation process by caching persistent information like style properties or frequently accessed images. The tags also simplify things like localization, data filtering, and so on.

This particular tag is translated into an image tag that retrieves its image data from a helper servlet. Other tags will represent the chart with an applet, or by creating a cache image on the server.

The next chapter discusses tag attributes in more detail.

Variations

KavaChart's tags have a variety of useful attributes and variations that might be useful in you application.

Localization

The chart tags use conventional Java ResourceBundles to achieve localization. Typically a resource bundle is a set of properties files consisting of property=value statements. The default ResourceBundle must be located within your application's CLASSPATH, and must be named resourceBundleBaseName.properties. Localized resource bundles use the convention resourceBundleBaseName_en.properties, where "en" is replaced by

the specific locale ID represented by that bundle. You can localize any property, including fonts, titles, dataset names, even colors. Non-western locales can be supported with ResourceBundle classes rather than text files.

A chart tag attribute lets you specify the ResourceBundle base name.

You can also create ResourceBundle instances during the page creation process and install those resources as an attribute:

```
getPageContext().setAttribute("resBndl", myResourceBundle);
```

The chart tag will retrieve this resource bundle and apply it to your charts if you specify the **resourceBundleID** like this:

```
<chart:streamed
          dataProviderID="dataId"
          style="WEB-INF/myStyle.properties"
          chartType="lineApp"
          resourceBundleID="resBndl" />
```

This approach lets you combine your page localization resources together with your chart localization resources.

An inner tag *(locale)* exists to let you test your chart localization approach without finding a localized client machine.

Data Manipulation

The chart tags include a number of inner tags that can filter the data or produce metadata based on your DataProvider. These tags can be used for data reduction, data analysis, or for creating industry specific chart types, like pareto charts, histograms, percentage change charts, and so on.

The tag collection also includes tags (datasimulation, timedatasimulation) to simulate incoming data. This lets you proceed with the page design process in the absence of a working DataProvider class.

Image vs. Applet

Chart tags have 4 variants: streamed, cached, balanced, and applet. The samples we've shown so far use the "streamed" approach.

The "cached" tag saves an image on your server's filesystem and returns a reference to that image. This can provide significant optimizations if your application generates images with high reuse. For example, if your users are looking at daily inventory levels, your data will change only once per day. A busy server might benefit by re-using chart images, rather than re-generating the chart image for each user.

If you're using a cluster of mirrored servers, the "balanced" tag will distribute the chart generation request to the first available server. You can also use the tag attribute "StreamServletName" to direct all chart generation traffic to a specific server.

Another variant is the "applet" tag. This tag generates an applet definition that will produce your chart image on a client browser. The primary advantage of using applets to generate your chart image is that all imaging activity is distributed to client machines. In the right circumstances, this can save server bandwidth and even enhance page rendering speed. Applets rely on having consistent browser configurations, however, and don't give users the same flexibility to copy and paste images into other applications.

Chapter

7

Chart Tag Details

KavaChart includes a simple taglib implementation to encapsulate the features of KavaChart's server objects. Because ChartTags hide the complexity of managing server chart imaging, they are the preferred method for building multi-tier JSP applications.

Design Goals

KavaChart's Custom Tag Library consists of a small set of easy to use tags that hide the complexity of applying data and style information to server-generated charts. These tags will help you effectively create persistent data stores and style information to make chart image generation as efficient as possible.

In addition, JSPs that contain these chart tags will be more intuitive and easier to work with than extensive scriplet or servlet references for non-programmers and programmers alike. These tags will make your pages more portable and flexible and will promote re-use of your existing data sources.

This library is comprised of three chart generation tags subclasses of com.ve.kavachart.servlet.ChartTag) and a few utility tags, described below.

Taglib Statement

The first step in adding a custom tag to a JSP is an include statement. This statement provides a unique description of the tag library you intend to use. KavaChart's taglib include statement:

```
<%@ taglib uri="http://www.ve.com/kavachart-taglib" prefix="chart" %>
```

While you can use any prefix you want for your chart tags, we'll use "chart" for all our examples here.

The taglib statement assumes that you've configured your web.xml file to point to the "kavachart.tld" tag library descriptor using the URI "http://www.ve.com/kavachart-taglib".

The Chart Tags

While your HTML coders may be familiar with applet definitions, this tag provides you with the opportunity to separate style logic (properties stylesheets) from your data in a simple tag. Chart data and properties are automatically populated without cluttering the logic of your JSP source code.

In an ideal situation, data comes to these chart tags via a DataProvider class. You will have implemented this class to create an Enumeration of Dataset classes and a unique identifier string (such as a database query string, etc.) to keep the chart cache logic in order.

Your page designers will use these tags to combine styles (properties files created with the KavaChart ChartWizard) with your data source. If appropriate, you can apply locales, resource bundles, and other attributes (described below) during your page generation process.

Cached ChartTag

This chart tag generates an encoded chart image and writes it to your server's filesystem. It returns a reference to that image, along with (optionally) an image map used to create tooltip labels.

Usage:

```
<chart:cached attr1="value1" attr2="value2" .../>
```

All this tag's attributes are available for runtime expressions. The tag also has a number of useful inner tags described later in this document.

Streamed ChartTag

This chart tag generates an encoded chart image and saves it to your application memory. The generated page includes an "IMG" tag that points to a servlet used to retrieve the image and delete it from memory. This tag will also (optionally) create an image map used to create tooltip labels to accompany that image.

Usage:

```
<chart:streamed attr1="value1" attr2="value2" .../>
```

All this tag's attributes are available for runtime expressions. The tag also has a number of useful inner tags described later in this document.

Balanced ChartTag

This chart tag constructs an IMG tag that points to a servlet, complete with arguments to produce the chart from that servlet. Since all the chart data is carried with the servlet command, the servlet URL constructed with this tag can be rather unwieldy.

This tag works best in mirrored server clusters, which provide a pool of resources for chart creation. Each server must have the same chart style properties and resource bundles to ensure uniform rendering.

Usage:

```
<chart:balanced attr1="value1" attr2="value2" .../>
```

All this tag's attributes are available for runtime expressions. The tag also has a number of useful inner tags described later in this document.

This tag also includes an "imageType" attribute. If you're using Flash or SVG output, this attribute provides some optimizations.

Applet ChartTag

This chart tag generates an applet definition consistent with the server-side imaging processes. This definition includes data defined by your DataProvider classes, localization from ResourceBundles, and any other modifications performed by the other Chart tags.

Usage:

```
<chart:applet attr1="value1" attr2="value2" .../>
```

This tag lets you switch from server-side imaging to client-side imaging (via an applet) with a minimum of effort.

Table ChartTag

This tag generates an HTML table using your chart data. This definition includes data defined by your DataProvider classes, localization from ResourceBundles, titles and labels from your chart's "style" attribute, and any other modifications performed by the other Chart tags.

Usage:

```
<chart:table attr1="value1" attr2="value2" .../>
```

This tag is a useful companion to charts, and is designed to let you re-use your HTML stylesheets to present chart data in a tabular way.

Special table tag attributes are found below.

ChartTag Attributes

Attributes are added to a ChartTag using a standard syntax. Each attribute is a string that identifies some aspect of the chart or the imaging process. The strings can be generated by runtime expressions.

Style

A string value that defines this chart's property sheet. A property sheet is a file that contains lines in form **`property=value`**. These files can be easily generated with the KavaChart ChartWizard. Because a property file is a simple ASCII file, you can also edit it with a text editor, using the standard KavaChart server bean property strings.

Example:

```
<chart:cached style="WEB-INF/myChart.properties" />
```

reloadStyle

"true" or "false". By default, the chart's property sheet is loaded into an application attribute for efficient re-use. If you want changes to the property sheet to take effect, set "reloadStyle" to "true".

Example:

```
<chart:cached style="WEB-INF/niceChart.properties"
        reloadStyle="true" />
```

chartType

A string value that describes the KavaChart server chart bean you want to use. For example, if you specify "lineApp", the tag will use com.ve.kavachart.servlet.lineApp to create the chart. ChartBean classes that aren't part of the com.ve.kavachart.servlet package must use a fully qualified name, such as "kcfinance.ChartBean".

Example:

```
<chart:cached chartType="pieApp" />
```

useLinkMap

"true" or "false". By default, this attribute is "true" and the tag creates a link map with "ALT" text to describe each point. While this is a nice feature for simple charts, you may want to disable it for charts with a large number of points.

Example:

```
<chart:cached chartType="dateLineApp" dataProviderID="myData"
        useLinkMap="false" />
```

resourceBundleID

An attribute ID for this tag's resource bundle. This permits you to set a resource bundle attribute at any point in the page generation process and use it to redefine a chart's labels, titles, etc. You can even redefine the chart's style based on resource bundle information.

Example:

```
<chart:streamed chartType="pieApp"
        resourceBundleID="localizationResources" />
```

resourceBundleBaseName

A basename for a resource bundle used by this chart. If you don't use a common resource bundle for all your page's attributes, you can still localize the chart automatically using resource bundles.

Typically a resource bundle is a set of properties files consisting of property=value statements. This file must be located within your application's CLASSPATH, and must be named with a *resourceBundleBaseName*.properties. Localized resource

bundles use the convention *resourceBundleBaseName_en*.properties, where "en" is replaced by the specific locale ID represented by that bundle.

It's generally more efficient to put non-Western resources in ResourceBundle classes.

Resource bundle properties files are loaded and unloaded by the ClassLoader, so they are cached efficiently in a platform-dependent way. You may need to restart your server to see changes to your resource bundles take effect.

If your target languages includes non-Western languages, you may need to create ResourceBundle classes to get the correct character information.

Example:

```
<chart:cached chartType="pieApp"
        resourceBundleBaseName="chartResources" />
```

cacheDirectory

(cached chart tag only) Because the cached chart tag must write an image file to your server's filesystem, it's important to specify where on the filesystem you wish to save the image.

The location specified here is relative to your application context, and defaults to /images/KavaChartImages.

Example:

```
<chart:cached chartType="pieApp" cacheDirectory="/images/charts" />
```

codebase

(applet chart tag only) Applets use a CODEBASE attribute to determine security policies and class file locations. This optional attribute defaults to your application's root directory.

You can set this value to point to another directory, or even another server (which would contain your applet's jar file, image resources, etc.). Since the chart's data is embedded into the parameters, you can easily host the applet from another server.

Example:

```
<chart:applet chartType="columnApp" codebase="/images/charts" />
```

archive

(applet chart tag only) Applets use an ARCHIVE attribute to point to a jar archive containing the applet's code resources. This attribute defaults to KavaChart's applet naming convention (e.g. com.ve.kavachart.applet.lineApp is contained in a jar archive named lineApp.jar), but you can override this in the case you have a customized applet.

Example:

```
<chart:applet chartType="custom.FancyChart" archive="Fancy.jar" />
```

streamServletName (streamed and balanced chart tag only) By default the streamed chart tag points to a servlet named /servlet/com.ve.kavachart.servlet.ChartStream to retrieve its image bytes. You may wish to obscure this by changing the servlet-mapping attribute in your web.xml file, and by applying this attribute to your streamed chart tag.

Example:

```
<chart:streamed chartType="custom.FancyChart"
          streamServletName="StreamingChart.do" />
```

Table Tag Attributes

rowwise (table tag only) true|false. Determines whether data should be displayed with datasets in rows, or in columns.

Example:

```
<chart:table rowwise="true" dataProviderID=”foo” />
```

useYVals (table tag only) true|false. Determines whether Y data should be displayed.

Example:

```
<chart:table useYVals="true" dataProviderID=”foo” />
```

useY2Vals (table tag only) true|false. Determines whether Y2 data should be displayed.

Example:

```
<chart:table useY2Vals="true" dataProviderID=”foo” />
```

useXVals (table tag only) true|false. Determines whether X data should be displayed.

Example:

```
<chart:table useXVals="true" dataProviderID=”foo” />
```

useY3Vals (table tag only) true|false. Determines whether Y3 data should be displayed.

Example:

```
<chart:table useY3Vals="true" dataProviderID="foo" />
```

useLabelVals (table tag only) true|false. Determines whether data labels should be displayed.

Example:

```
<chart:table useLabelVals="true" dataProviderID="foo" />
```

useDatasetName (table tag only) true|false. Determines dataset names should be displayed as row or column headers.

Example:

```
<chart:table useDatasetName="true" dataProviderID="foo" />
```

dateFormat (table tag only) e.g. yyyy-MM-dd. Defines a date format for time oriented chart X data.

Example:

```
<chart:table useXVals="true" dateFormat="yyyy" dataProviderID="foo" />
```

tableClass (table tag only) a CSS stylesheet class to be used for this overall table.

Example:

```
<chart:table tableClass="charttable" dataProviderID="foo" />
```

cellClass (table tag only) a CSS stylesheet class to be used for cells in this table.

Example:

```
<chart:table cellClass="chartcells" dataProviderID="foo" />
```

columnHeaderClass (table tag only) a CSS stylesheet class to be used for column headers in this table.

Example:

```
<chart:table columnHeaderClass="ch" dataProviderID="foo" />
```

rowHeaderClass (table tag only) true|false. A CSS stylesheet class to be used for row headers in this table.

Example:

```
<chart:table rowHeaderClass="rh" dataProviderID="foo" />
```

Chapter

8

Inner Tags

KavaChart ProServe includes a collection of helper tags that can simplify otherwise difficult operations. These tags let you override titles, styles, default locales, create meta-data, and so on.

KavaChart's tag library includes a number of useful inner tags that supply debugging or other useful information. These tag statements are used after the start of the tag and before the end of the tag.

Non-Data Tags

Param

Usage:

```
<chart:param name="titleString" value="hello, world" />
```

This inner tag overrides properties set in the style properties file and localized resource bundles. It provides a behavior similar to conventional applet param tags, but can be set with JSP runtime expressions. This inner tag can be set multiple times.

name: (required) Name of property to override.

value: (required) The property's new value.

Example:

```
<chart:streamed chartType="lineApp" style="WEB-
        INF/myChart.properties" >
        <chart:param name="backgroundColor" value="orange" />
        <chart:param name="titleString" value="<%=title%>" />
        <chart:param name="yAxisEnd" value="50." />
</chart:streamed>
```

Locale

Usage:

```
<chart:locale name="localeString" value="BE_fr" />
```

This inner tag overrides locale settings from your page or application context. It's useful for testing resource bundles in an ad-hoc way.

localeString: (required) Named locale, using the convention of country_variant

Example:

```
<chart:streamed chartType="pieApp" style="WEB-INF/ch.props" >
        <chart:locale localeString="ES_es" />
</chart:streamed>
```

Data Manipulation Tags

Datafilter

This inner tag constrains the data provided by a referenced DataProvider class. All attributes are settable by runtime expressions, and all attributes are optional.

Usage:

```
<chart:datafilter startDataset="0" endDataset="7" .../>
```

dataProviderID: This is original DataProvider class to be filtered. Set this attribute at any point in your page generation process, and the datafilter tag will limit the emitted data to the attributes below.

startObservation: This is the index (0 based) for the first point from the chart's DataProvider to appear on the chart.

endObservation: This is the index for the last point from the chart's DataProvider to appear on the chart.

startDataset: This is the index for the first Dataset from the chart's DataProvider to appear on the chart.

endDataset: This is the index for the last Dataset from the chart's DataProvider to appear on the chart.

Example:

```
<chart:streamed chartType="columnApp" >
      <chart:datafilter dataProviderID="moneyChartData"
            startObservation="20"
            endObservation="120"
            startDataset="2"
            endDataset="2" />
</chart:streamed>
```

Datasimulation

Usage:

```
<chart:datasimulation numDatasets="2" numObservations="25" />
```

This inner tag provides datasets with random numbers to permit your page designers to proceed with page development in the absence of a working DataProvider class.

numDatasets: (required) positive integer describing the number of datasets provided to this chart.

numObservations: (required) positive integer describing the number of points in each dataset provided to this chart. In essence, this tag installs itself as the chart tag's DataProvider, regardless of what other attributes may be set.

Example:

```
<chart:streamed chartType="columnApp" >
      <chart:datasimulation
            numDatasets="3"
            numObservations="10" />
</chart:streamed>
```

Timedatasimulation

Usage:

```
<chart:timedatasimulation numDatasets="2" numObservations="25" />
```

This inner tag provides datasets with random numbers to permit your page designers to proceed with page development in the absence of a working DataProvider class.

The X values in this DataProvider are timestamps, starting with now, and increasing by a specified number of days for each observation. The default is one day per point.

This simulator provides extended data suitable for charts that use CandlestickDatum, such as candelstick charts, hi-lo-open charts, and so on.

numDatasets: (required) positive integer describing the number of datasets provided to this chart.

numObservations: (required) positive integer describing the number of points in each dataset provided to this chart. In essence, this tag installs itself as the chart tag's DataProvider, regardless of what other attributes may be set.

dayIncrement: (required) positive integer describing the number of days between each point in this DataProvider's datasets.

Example:

```
<chart:streamed chartType="candlestickApp" >
      <chart:timedatasimulation
            dayIncrement="7"
            numDatasets="1"
```

```
                numObservations="120" />
</chart:streamed>
```

dataarithmetic Usage:

```
<chart:dataarithmetic adder="2" multiplier="2.5" />
```

This inner tag manipulates the data provided by a referenced DataProvider class, scaling each value by a multiplier, and then adding to each value as specified. All attributes are settable by runtime expressions, and all attributes

dataProviderID: the original DataProvider to be filtered. (note: you can also supply a DataProvider by another inner tag, such as when you want to accumulate data, create a histogram, etc.).

multiplier: This floating point value is used to scale each observation's Y values (e.g. Y, Y2, Y3, or high/low/open/close). Default multiplier = 1.

adder: This floating point value is added to the result of each multiplication. Default adder = 0.

Example:

```
<chart:streamed chartType="columnApp" >
      <chart:dataarithmetic dataProviderID="smogLevels"
            adder="50"
            multiplier="100" />
</chart:streamed>
```

datatransposer Usage:

```
<chart:datatransposer … />
```

This inner tag changes row-wise observations into column-wise observations, or vice-versa. It's typically used to regroup bars differently. For example, a default bar (column) chart with multiple datasets creates clusters of bars; the first cluster is each dataset's first observation, the second cluster is each dataset's second observation, and so on.

The datatransposer tag would alter that data so that the first cluster consists of all the first dataset's observations, the second cluster consists of the second dataset's observations, and so on.

dataProviderID: the original DataProvider to be filtered. (note: you can also supply a DataProvider by another inner tag, such as when you want to accumulate data, create a histogram, etc.).

Example:

```
<chart:streamed chartType="columnApp" >
      <chart:datatransposer dataProviderID="smogLevels"/>
</chart:streamed>
```

dataaccumulator

Usage:

```
<chart:dataaccumulator dataProviderId="source1,source2"/>
```

An inner tag to combine several data sources into a single chart. This tag is particularly useful when a page needs to display data in a variety of charts. For example, a page may use one data source for each of two pie charts, and then combine the data sources into a single bar chart.

dataProviderID: A comma-delimited list of DataProviders to be combined.

Example:

```
<chart:streamed chartType="columnApp" >
      <chart:dataacumulator
dataProviderID="salesData,expenseData" />
</chart:streamed>
```

Alternate Example:

```
<chart:streamed chartType="columnApp" >
      <chart:dataacumulator>
            <datawebservice
serviceURL="http://bigcompany.com:9011/salesXML?now" />
            <datawebservice
serviceURL="http://bigcompany.com:9011/salesXML?then" />
      </chart:dataacumulator>
</chart:streamed>
```

datahistogram

Usage:

```
<chart:datahistogram ... />
```

An inner tag to generate histogram metadata from incoming data sources. Data is automatically arranged into a reasonable number of "bins", with ranges used to label the bins. This tag is typically used with a column or bar chart.

dataProviderID: the original DataProvider to be filtered..

binSize: the number of items to be allocated to each histogram bin.

labelPrecision: A decimal precision to be used for labelling each bin

Example:

```
<chart:streamed chartType="columnApp" >
      <chart:datahistogram
dataProviderID="dailyTemperatures" />
</chart:streamed>
```

datapercentchange

Usage:

```
<chart:percentchange … />
```

An inner tag to generate metadata that describes the percentage change of incoming data source Y values. This type of chart is typically used with time-series data to understand how values have changed over time. It is common in financial services to use this technique to compare two stocks, for example.

dataProviderID: the original DataProvider to be analyzed.

Example:

```
<chart:streamed chartType="dateLineApp" >
  <chart:datapercentchange dataProviderID="stockPrices" />
</chart:streamed>
```

Alternate Example:

```
<chart:streamed chartType="dateLineApp" >
 <chart:datapercentchange>
      <chart:dataaccumulator dataProviderID="IBMstockData,FDXStockData" />
 </chart:datapercentchange>
</chart:streamed>
```

dataregressionfeed

Usage:

```
<chart:dataregressionfeed … />
```

Linear regression is typically used to understand correlation between two variables. In practice, these variables often exist as the dependent (Y) values of two different datasets. This tag uses Y values from two datasets to create X and Y metadata for an output dataset. The output data is then directed to a regressionApp chart or scatter plot to visualize the correlation between these variables.

dataProviderID: the original data to be analyzed.

Example:

```
<chart:streamed chartType="regressionApp" >
      <chart:dataregressionfeed
dataProviderID="windAndTemperatureData" />
</chart:streamed>
```

Alternate Example:

```
<chart:streamed chartType="regressionApp" >
      <chart:dataregressionfeed>
            <chart:dataaccumulator
dataProviderID="windData,temperatureData" />
      </chart:dataregressionfeed>
</chart:streamed>
```

datasorter

Usage:

```
<chart:datasorter … />
```

A tag to sort incoming data sources. Combined with a column chart, this can turn data into a Pareto analysis chart. You can also use this chart to visually understand other aspects of incoming data sources.

dataProviderID: the original data to be sorted.

sortBy: X|Y|Y2|Y3|label. Defaults to "Y".

Example:

```
<chart:streamed chartType="columnApp" >
      <chart:datasorter dataProviderID="salaryData" />
</chart:streamed>
```

Alternate Example:

```
<chart:streamed chartType="regressionApp" >
      <chart:dataregressionfeed>
            <chart:dataaccumulator
dataProviderID="windData,temperatureData" />
      </chart:dataregressionfeed>
</chart:streamed>
```

DataProvider Interface

One of the keys to implementing an efficient and flexible architecture using KavaChart ChartTags is creating DataProvider classes to supply data to ChartTags.

Typically, you will implement DataProvider in a class that contains data you've retrieved from a database to create charts, tables, and other data representations. Each DataProvider supplies a specific view on your data. Page-scope DataProviders are used for things like individual financial data, which change with each page view. Application scope DataProviders can be created for data that doesn't vary by page view.

Put your DataProvider into a page, application or other scoped attribute, and provide a reference to that class's ID in your ChartTag dataProviderID attribute. The ChartTag will use the DataProvider during the chart construction process to supply that chart's Dataset information. A "UniqueIdentifier" makes sure charts with identical styles but different data aren't confused by the caching mechanisms.

```
public Interface com.ve.kavachart.utility.DataProvider {
       /*
        ** Returns an Enumeration of Dataset classes.
        */
       public Enumeration getDatasets();
       /*
        ** Returns a String that uniquely identifies this data.
        ** Needed to make chart image caching work properly.
        ** Otherwise unnecessary.
        */
       public String getUniqueIdentifier();
}
```

Important Dataset Constructors and Methods

Since a DataProvider returns an Enumeration of Dataset classes, it's important to be able to construct these. Fortunately, this class has a range of convenience constructors that make it quite easy to use.

The Dataset class is found in the com.ve.kavachart.chart package. In the constructors below, set "Globals" to "null".

```
public Dataset();

public Dataset(String   name,
      double[]    xArray,
      double[]    yArray,
      double[]    y2Array,
      double[]    y3Array,
      int         seriesNumber,
      Globals     g);
```

```
public Dataset(String   name,
      double[]    xArray,
      double[]    yArray,
      String[]    labels,
      Int         seriesNumber,
      Globals     g);

public Dataset(String name,
      double[]    xArray,
      double[]    yArray,
      Globals     g);

public Dataset(String  name,
      double      yArray[],
      int         setNumber,
      Globals     g);
```

If you're using time oriented charts, time values should be the underlying values (milliseconds since epoch) used by java.util.Date and java.sql.Date. Generally, these are "X" values, and can be obtained by using Date.getTime().

Dataset also has a number of useful methods that make it easy to create meaningful data:

```
public void addDatum(Datum d);

public void addPoint(double x, double y, String label);
```

A Datum class describes an individual observation: a point on a line, a bar in a bar chart, etc. Datum is also found in the com.ve.kavachart.chart package, and has easy to use constructors and methods:

```
public Datum(double x, double y, Globals g);

public Datum(int whichPoint,
      double y,
      String label,
      Globals g);

public Datum(     double  dataX,
      double      dataY,
      double      dataZ,
      String      str,
      int         element,
      Globals     g);
```

Datum methods let you set internal values:

```
public void setX(double x);
public void setY(double y);
public void setY2(double y2);
public void setY3(double y3);
public void setLabel(String s);
```

A complete DataProvider is installed in a chart bean using the "setDataProvider" method. In a chart tag, a DataProvider is installed into a server attribute (application, page, or session scope), and the attribute name is passed into the tag as the "dataProviderID".

Here's a simple, complete DataProvider:

```
import java.util.*;
import com.ve.kavachart.chart.*;
public void MyDataProvider implements DataProvider{
      public Enumeration getDatasets(){
            Dataset d = new Dataset();
            double yVals = new double[20];
            for(int i=0;i<yVals.length;i++){
                  d.addPoint(i, y, null);
            }
            d.setName("Fake Data!");
            ArrayList al = new ArrayList();
            al.add(d);
            return Collections.enumeration(al);
      }
      public String getUniqueIdentifier(){
            return (new Date()).toString();
      }
}
```

Note that this code returns the value of "Date" to make sure any cached versions of this chart are not used. Now, to install and use this DataProvider in a JSP scriptlet or a servlet, we'd do this:

```
Bean chart = new columnApp(myStyleProperties);
chart.setDataProvider(new MyDataProvider());
String filename = chart.getFileName();
```

In a chart tag, we would do something like this:

```
<%pageContext.setAttribute("data", new MyDataProvider());%>
<chart:streamed chartType=columnApp dataProviderID="data" />
```

A DataProvider gives you much more flexibility in structuring your pages and maintaining your data. Importantly, it also helps you test your data inputs outside an HTTP context.

Within a chart tag, DataProviders can also be filtered, sorted, combined with other sources, etc. using an intuitive set of tags, discussed in another chapter.

Consult the com.ve.kavachart.chart.Dataset documentation for more information on constructing Dataset classes. Convenient constructor signatures are available for most situations.

Web.xml

Your server recognizes a tag library implementation by references to it in your web.xml file. The KavaChart tag library descriptor is a file named "kavachart-taglib.tld" referenced in this file. Consult your server documentation for specific information on web.xml usage, but generally, you'll install the KavaChart ChartTags in these steps:

- Add "kcServlet.jar" to your CLASSPATH (usually WEB-INF/lib)

- Integrate the sample WEB-INF/web.xml (from kavachart.war) into your own. In particular, you'll need the "ChartStream" servlet definition and mapping for cached charts, and you'll need the taglib reference. Remember that XML entries are order-dependent, so your <servlet-mapping>, etc. tags must be placed in the right order throughout your web.xml file.

Add kavachart-taglib.tld (the taglib descriptor) to the location specified in your web.xml file.

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 to create images and fonts. 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

Note:

If you're using JDK 1.4 or newer with a Unix server, set the System property "java.awt.headless" to "true" to make sure you don't need any special system resources to create graphics. You can set this in your server startup script (add "-Djava.awt.headless=true" to the command line), or as a temporary workaround, you can add the line "System.getProperties().setProperty('java.awt.headless', 'true');" to a JSP.

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. If you're running a version of Java earlier than 1.4 with a Unix server, however, your server 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

ColdFusion MX Server Setup

KavaChart ProServe makes a powerful companion to ColdFusion and FlashMX, but this server requires some special setup, and ColdFusion tags are a little different from onventional usage.

The ColdFusion MX server acts as a J2EE container for the ColdFusion application, but it doesn't support easy drop-in of .war or .ear files to set up new applications. To integrate KavaChart with this server, do the following steps:

Installation and Setup

- Extract "kavachart.war" into a temporary directory to get the necessary parts.
- In that directory, you'll find a WEB-INF folder with many of the necessary pieces: integrate WEB-INF/web.xml with the ColdFusion WEB-INF/web.xml. To do this, make sure you keep the tags all together – ordering matters (add ours to the end of the default), and keep the tags together, and so on. Make sure you include the KavaChart taglib section.
- Copy WEB-INF/lib/kcServlet.jar into the ColdFusion MX WEB-INF/lib directory.
- Copy WEB-INF/jsp/kavachart-taglib.tld into the ColdFusion MX WEB-INF directory.
- To get the KavaChart samples working, copy the WEB-INF/classes/examples folder, with its contents to the ColdFusion WEB-INF/classes folder.
- Copy WEB-INF/paramchart.properties into the ColdFusion WEB-INF folder.

- Copy tag-samples/simplechart.jsp into wwwroot. Modify this file so the "taglib uri" statement points to the KavaChart taglib definition: "/WEB-INF/kavachart-taglib.tld".

Run The Example

Now you should be able to run this example and produce graphical output. (http://localhost:8080/simplechart.jsp). In summary, we did the following: a) added web.xml and kavachart-taglib.tld information, b) provided a stylesheet for simplechart.jsp, and b) made sure simplechart.jsp points to the correct tag library definition.

This sample uses the KavaChart "streamed" tag, which is appropriate for most installations. If you prefer to use the "cached" tag, you'll also need to create an image cache directory. For the samples, create "wwwroot/images/KavaChartImages".

When copying other examples from the kavachart.war contents, make sure you check the "taglib uri". JSP scriptlet examples should also work, although these are more difficult to combine with CF tags.

The first step to using KavaChart tags is to import the tag library descriptor:

```
<cfimport taglib-"/WEB-INF/kavachart-taglib.tld"
prefix="chart">
```

This tells ColdFusion that all tags with a prefix of "chart" will be processed by KavaChart, after ColdFusion has processed CF data tags.

KavaChart's tag library supports a "param" syntax nearly identical to KavaChart applets. Although it's more efficient and neater to place your persistent style attributes (background colors, margin sizes, etc.) into a properties file referenced by the tag "style" attribute, you can set any param/property like this:

```
<chart:param name="dataset0yValues" value="123,234,345" />
```

Dynamic values like this are generally represented as either strings or comma - separated values. A chart that uses ColdFusion to populate these values might look like this:

```
<chart:streamed
      style="/WEB-INF/paramchart.properties"
      chartType="barApp" >

      <cfoutput>
            <chart:param name="dataset0yValues" value="#ValueList(get_depts_by_year.dptCount)" />
            <chart:param name="dataset0Labels" value="#ValueList(get_depts_by_year.dptName") />
            <chart:param name="dataset0Labels" value="#ValueList(get_depts_by_year.year)" />
      </cfoutput>

</chart:streamed>
```

In this example, "get_depts_by_year" is the name of a CFQUERY defined elsewhere on the page. Like "simplechart.jsp", this .cfm page uses "paramchart.properties" to define the basic look of the chart and it uses a bar chart as the visual representation. The result of this output will be an image in PNG (the imageType param in paramchart.properties), placed in the page in the overall context of this chart tag.

Other important tag options are alternate image types (flash, svg, jpeg, etc.), the "reloadStyle" attribute, which lets you modify the style properties file while you're developing, and "cached" and "balanced" tags, which behave the same way, but implement automatic server-side caching and load balancing, respectively.

Appendix

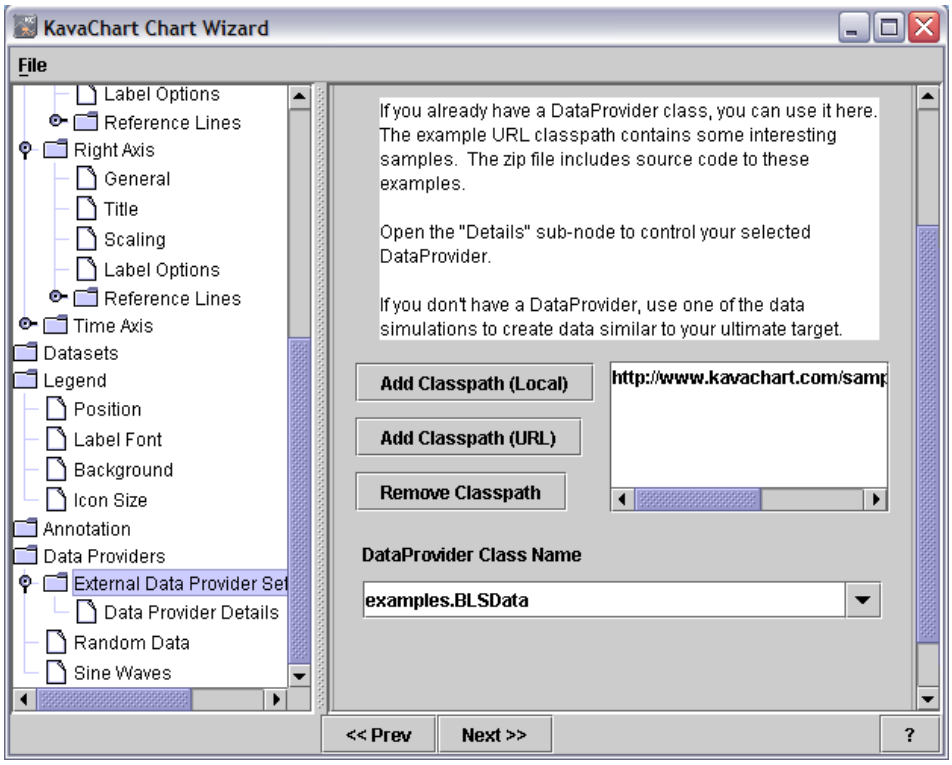
C

DataProvider GUIs

One of the most effective ways to use KavaChart is to combine your own DataProvider classes with the Chart Wizard using a graphical user interface.

DataProviders in the Wizard

When run as a Java WebStart application, the Chart Wizard lets you set CLASSPATHs from URLs or your local filesystem, and to use DataProvider classes in that CLASSPATH for chart data.

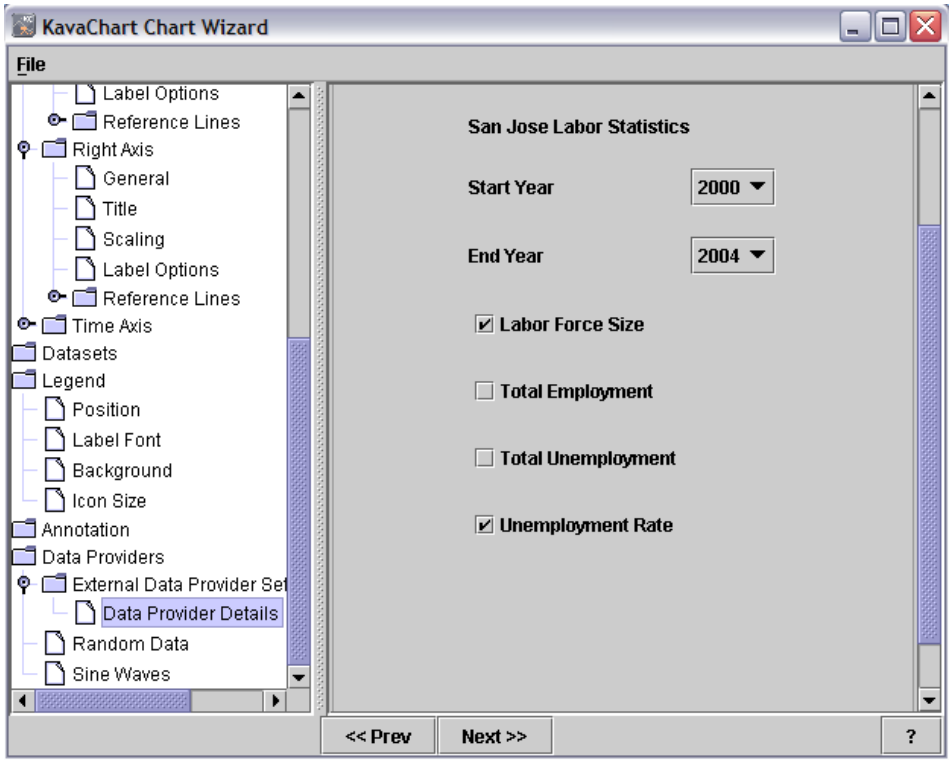


When you load a zip file, jar file, or identify a directory at the top of a class hierarchy, the wizard looks for a file named "DataProviders.list". This is an

ASCII file containing a list of fully qualified DataProvider class names. The list is used to populate the ComboList at the bottom of this editor.

After selecting your DataProvider from the CoomboList (or adding it to the list manually), you can run the DataProvider class by selecting "Data Provider Details". The "Update" button at the bottom of this page causes the wizard to run your DataProviders "getDatasets()" method to replace the existing chart data.

If your DataProvider is also a subclass of javax.swing.Jpanel, you can also implement your own graphical user interface, which is displayed by the wizard when you select "Data Provider Details". Here's an example, provided in the sample classes at http://www.kavachart/com/sample_classes/examples.zip:



Note this is the default sample URL CLASSPATH in the wizard. It's worth viewing from time to time, since we supply new examples periodically.

Although you can't see it in this sample, there's an "Update" button at the bottom of the panel. This is supplied by the editor, and will trigger data acquisition through the DataProvider.

Appendix

D

Installing A License Key

This section describes how to make KavaChart ProServe operate in fully licensed mode.

KavaChart ProServe operates in "demo mode" until you install a license key. This mode draws information about Visual Engineering over your chart images, and modifies some hyperlinks to point to the ve.com web site.

Obtain a license key

Once you've purchased a KavaChart license, you'll receive a licence key file. This binary file can be downloaded from our web site (www.ve.com) by going to the "MyKavaChart" section. License keys are available that will work with a server on any IP address (distribution license key), or that will work only with specific IP addresses (deployment license key). An incorrect deployment key will cause KavaChart ProServe to operate in demo mode.

Put the key in your CLASSPATH

Installing a license key is a simple matter of placing the file into your CLASSPATH. Most servers have multiple locations that are included in a web application's CLASSPATH. For example, on Apache Tomcat, you can place this file into your application's WEB-INF/classes directory, or you can put it into the "shared/classes" directory.

This file is loaded by the Java ClassLoader as a resource, so it can be handled like any other resource.

Index