JClass Chart™
Programmer's Guide

Version 6.3
for Java 2 (JDK 1.3.1 and higher)

The Best Java Charting Solution

**QUEST
SOFTWARE**™

Table of Contents

Part II: Reference Appendices

Preface

Introducing JClass Chart

JClass Chart is a charting/graphing component written entirely in Java. The chart component displays data graphically in a window and can interact with a user.

The chart component can be used easily by all types of Java programmers:

■ Component users, setting JClass Chart properties programmatically.

■ OO developers, instantiating and extending JClass Chart objects.

■ JavaBean developers, setting JClass Chart properties using a third-party Integrated Development Environment (IDE).

You can freely distribute Java applets and applications containing JClass components according to the terms of the License Agreement that appears during the installation.

Feature Overview

You can set the properties of JClass Chart objects to determine how the chart will look and behave. You can control:

■ Chart type (Plot, Scatter Plot, Area, Stacking Area, Bar, Stacking Bar, Pie, Hi-Lo, Hi-Lo-Open-Close, Candle, Polar, Radar, and Area Radar).

■ Header and footer positioning, border style, text, font, and color.

■ Number of data views, each having its own data, chart type, axes, and chart styles.

■ Flexible data loading from applets, files, URLs, input streams, and databases.

■ Chart styles: line color, fill color, point size, point style, and point color.

■ Legend positioning, orientation, border style, anchor, font, and color.

■ Chart positioning, border style, color, width, height, and 3D effect (Bar, Stacking Bar, and Pie charts only).

■ Axis labelling using Point labels, Series labels, Value labels, or Time labels.

■ Number of X- or Y-axes, each having its own minimum and maximum, axis numbering method, numbering and ticking increment, grid increment, font, origin, axis direction, and precision.

■ Control of user interaction with components including picking, mapping, Chart Customizer, rotation, scaling, and translation.

■ Chart labels that can appear anywhere on the chart, including automatic dwell labels for each point on the chart.

JClass Chart is compatible with JDK 1.4. If you are using JDK 1.4 and experience drawing problems, you may want to upgrade to the latest drivers for your video card from your video card vendor.

Assumptions

This manual assumes that you have some experience with the Java programming language. You should have a basic understanding of object-oriented programming and Java programming concepts such as classes, methods, and packages before proceeding with this manual. See Related Documents later in this section of the manual for additional sources of Java-related information.

Typographical Conventions Used in this Manual

Typewriter Font	■ Java language source code and examples of file contents.
	■ JClass Chart and Java classes, objects, methods, properties, constants, and events.
	■ HTML documents, tags, and attributes.
	■ Commands that you enter on the screen.
Italic Text	■ Pathnames, filenames, URLs, programs, and method parameters.
	■ New terms as they are introduced, and to emphasize important words.
	■ Figure and table titles.
	■ The names of other documents referenced in this manual, such as *Java in a Nutshell.*
Bold	■ Keyboard key names and menu references.

Overview of Manual

Part I – Using JClass Chart describes programming with JClass Chart.

Chapter 1, JClass Chart Basics, provides a programmer's overview of JClass Chart. It covers class hierarchy, object containment, terminology, programming basics, and specific issues to be aware of before using JClass Chart.

Chapter 2, Chart Types and Special Chart Properties, covers the special features of chief JClass Chart charting types.

Chapter 3, SimpleChart Bean Tutorial, introduces basic Bean concepts, and guides you through developing a chart application in an IDE or BeanBox.

Chapter 4, Bean Reference, is a guide to the different JClass Chart Beans. It illustrates all of the properties available, including the different data loading methods.

Chapter 5, MultiChart, is a user's guide for MultiChart, an advanced charting Bean.

Chapter 6, Chart Programming Tutorial, is designed to introduce you to JClass Chart programming, by compiling and running an example program. It includes examples of common chart programming tasks.

Chapter 7, Axis Controls, covers JClass Chart properties used when first setting up your chart, concentrating on axis properties.

Chapter 8, Data Sources, shows how to use different pre-built data sources and outlines how to use the data source toolkit to create your own.

Chapter 9, Text and Style Elements, covers JClass Chart properties used to customize the appearance of a chart, including header/footer, legend, and chart styles.

Chapter 10, Advanced Chart Programming, looks at programming more advanced aspects of the chart.

Part II – Reference Appendices – contains detailed technical reference information.

Appendix A, JClass Chart Property Listing, summarizes the properties contained in all of the JClass Chart objects.

Appendix B, Distributing Applets and Applications, is an overview of how to deploy applets and applications.

Appendix C, HTML Property Reference, lists the syntax of JClass Chart properties when specified in an HTML file.

Appendix D, Porting JClass 3.6.x Applications, comprises the key changes to version 4.0 and the recommended porting strategy.

API Reference

The API reference documentation (Javadoc) is installed automatically when you install JClass Chart and is found in the *JCLASS_HOME/docs/api/* directory.

Licensing

In order to use JClass Chart, you need a valid license. Complete details about licensing are outlined in the *JClass DesktopViews Installation Guide*, which is automatically installed when you install JClass Chart.

Related Documents

The following is a sample of useful references to Java and JavaBeans programming:

■   *"Java Platform Documentation"* at *http://java.sun.com/docs/index.html* and the *"Java Tutorial"* at *http://java.sun.com/docs/books/tutorial/index.html* from Sun Microsystems

■   For an introduction to creating enhanced user interfaces, see *"Creating a GUI with JFC/Swing"* at *http://java.sun.com/docs/books/tutorial/uiswing/index.html*

■   *"Java in a Nutshell, 2nd Edition"* from O'Reilly & Associates Inc. See the O'Reilly Java Resource Center at *http://java.oreilly.com*

■   Resources for using JavaBeans at *http://java.sun.com/beans/resources.html*

These documents are not required to develop applications using JClass Chart, but they can provide useful background information on various aspects of the Java programming language.

About Quest

Quest Software, Inc. (NASDAQ: QSFT) is a leading provider of application management solutions. Quest provides customers with Application Confidence[sm] by delivering reliable software products to develop, deploy, manage and maintain enterprise applications without expensive downtime or business interruption. Targeting high availability, monitoring, database management and Microsoft infrastructure management, Quest products increase the performance and uptime of business-critical applications and enable IT professionals to achieve more with fewer resources. Headquartered in Irvine, Calif., Quest Software has offices around the globe and more than 18,000 global customers, including 75% of the Fortune 500. For more information on Quest Software, visit *www.quest.com*.

Contacting Quest Software

E-mail	*sales@quest.com*
Address	Quest Software, Inc. World Headquarters 8001 Irvine Center Drive Irvine, CA 92618 USA
Web site	*www.quest.com*
Phone	949.754.8000 (United States and Canada)

Please refer to our Web site for regional and international office information.

Customer Support

Quest Software's world-class support team is dedicated to ensuring successful product installation and use for all Quest Software solutions.

SupportLink	*www.quest.com/support*
E-mail	*support@quest.com*

You can use SupportLink to do the following:

■ Create, update, or view support requests

■ Search the knowledge base, a searchable collection of information including program samples and problem/resolution documents

■ Access FAQs

■ Download patches

■ Access product documentation, API reference, and demos and examples

Please note that many of the initial questions you may have will concern basic installation or configuration issues. Consult this product's *readme* file and the *JClass DesktopViews Installation Guide* (available in HTML and PDF formats) for help with these types of problems.

To Contact JClass Support

Any request for support *must* include your JClass product serial number. Supplying the following information will help us serve you better:

■ Your name, email address, telephone number, company name, and country

- The product name, version and serial number
- The JDK (and IDE, if applicable) that you are using
- The type and version of the operating system you are using
- Your development environment and its version
- A full description of the problem, including any error messages and the steps required to duplicate it

JClass Direct Technical Support	
JClass Support Email	*support@quest.com*
Telephone	949-754-8000
Fax	949-754-8999
European Customers Contact Information	Telephone: +31 (0)20 510-6700 Fax: +31 (0)20 470-0326

Product Feedback and Announcements

We are interested in hearing about how you use JClass Chart, any problems you encounter, or any additional features you would find helpful. The majority of enhancements to JClass products are the result of customer requests.

Please send your comments to:
Quest Software
8001 Irvine Center Drive
Irvine, CA 92618

Telephone: 949-754-8000
Fax: 949-754-8999

Part I

Using JClass Chart

1

JClass Chart Basics

This chapter covers concepts and vocabulary used in JClass Chart programming, and provides an overview of the JClass Chart class hierarchy.

1.1   Chart Areas

The following illustration shows the terms used to describe chart areas:
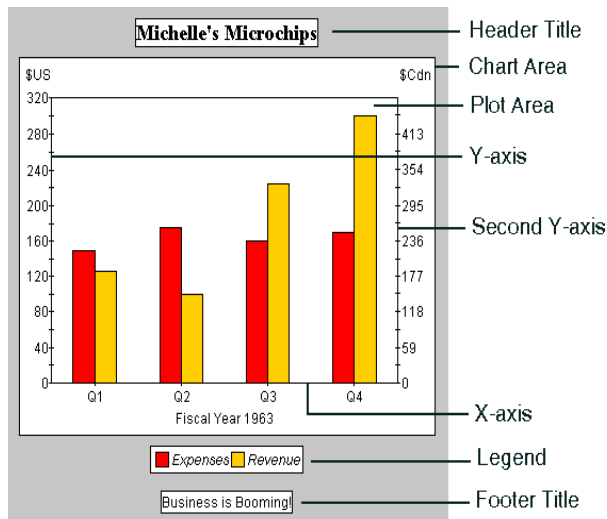
Figure 1   Elements contained in a typical chart.

1.2   Chart Types

JClass Chart can display data as one of 13 basic chart types: Plot, Scatter Plot, Area,
Stacking Area, Bar, Stacking Bar, Pie, Hi-Lo, Hi-Lo-Open-Close, Candle, Polar, Radar,
and Area Radar.

It is also possible to simulate more specialized types of charts using one of these basic
types.

Use the `ChartType` property to set the chart type for one `ChartDataView`. Each data view
managed by the chart has its own chart type. The following table lists basic information
about each chart type, including the enumeration that sets that type and the data layouts it
can display (see the next section for an introduction to data).
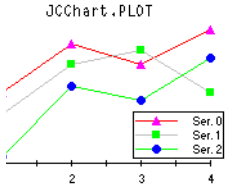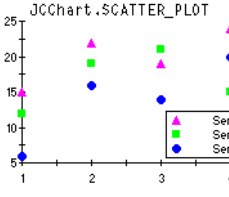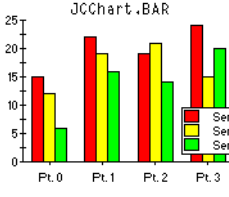
Chart Type	Single X-series	Multiple X-series	Notes
	✔	✔	**Plot** Draws each series as connected points of data. ■ Series appearance determined by chart style line color, symbol shape, size, and color properties.
	✔	✔	**Scatter Plot** Draws each series as unconnected points of data. ■ Series appearance determined by chart style symbol shape, size, and color properties.
	✔		**Bar** Draws each series as a bar in a cluster. The number of clusters is the number of points in the data. Each cluster displays the *n*th point in each series. ■ X-axis generally annotated using Point labels. ■ Series appearance determined by chart style fill color and image properties. ■ 3D effect available using depth, elevation, and rotation properties.
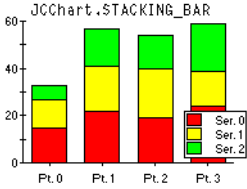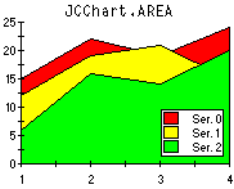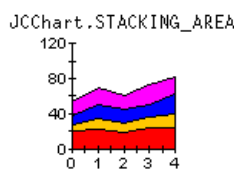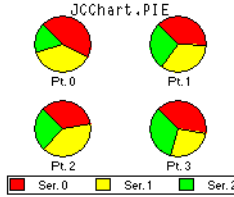
Chart Type	Single X-series	Multiple X-series	Notes
	✔		**Stacking Bar** Draws each series as a portion of a stacked bar cluster, the number of clusters being the number of data points. Each cluster displays the *n*th point in each series. Negative Y-values are stacked below the X-axis. ■ X-axis generally annotated using Point labels. ■ Series appearance determined by chart style fill color property. ■ 3D effect available using depth, elevation, and rotation properties.
	✔		**Area** Draws each series as connected points of data, filled below the points. Each series is layered over the preceding series. ■ Series appearance determined by chart style fill color property.
	✔		**Stacking Area** Draws each series as connected points of data, filled below the points. Places each Y-series on top of the last one to show the area relationships between each series and the total. ■ Series appearance determined by chart style fill color property.
	✔		**Pie** Draws each series as a slice of a pie. The number of pies is the number of points in the data (values below a certain threshold can be grouped into an *other* slice). Each pie displays the *n*th point in each series. ■ Pies are annotated with Point labels only. ■ Series appearance determined by chart style fill color property. ■ 3D effect available using depth and elevation properties.
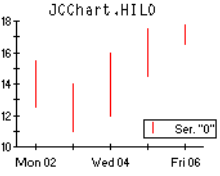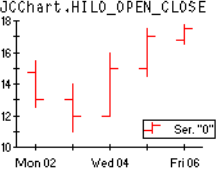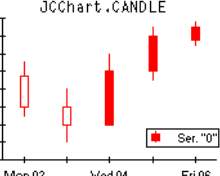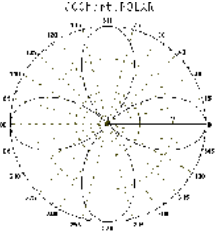
Chart Type	Single X-series	Multiple X-series	Notes
JCChart.HILO chart image	✔		**Hi-Lo** Draws *two* series together as a "high-low" bar. The points in each series define one portion of the bar: 1st series – points are the "high" value 2nd series – points are the "low" value ■ Appearance determined by chart style line color property in the **first** series of each pair.
JCChart.HILO_OPEN_CLOSE chart image	✔		**Hi-Lo-Open-Close** Similar to Hi-Lo, but draws *four* series together as a "high-low-open-close" bar. The additional series' points make up the other components of the bar: 3rd series – points are the "open" value 4th series – points are the "close" value ■ Appearance determined by chart style line color and symbol size properties in the **first** series of each set.
JCChart.CANDLE chart image	✔		**Candle** A special type of Hi-Lo-Open-Close chart; draws four series together as a "candle" bar. ■ *Simple* candle appearance determined by chart style line color, fill color, and symbol size properties in the **first** series of each set. ■ *Complex* candle appearance determined by different chart style properties from **each** series of each set.
JCChart.POLAR chart image	✔	✔	**Polar** Draws each series as connected points of data on a polar coordinate system (*theta*,*r*). X-values represent the amount of rotation and Y-values are the distance from the origin. ■ When using Array data, X-values are shared across series. ■ X-axis bounds cannot be set; Y-axis bounds cannot be set inside the data extents. ■ Appearance determined by ChartStyles' line and symbol properties of each series.

Chart Type	Single X-series	Multiple X-series	Notes
	✔		**Radar** Draws each series as connected points along radar "sticks" spaced equally apart. The *n*th stick charts the Y-value of the *n*th point in each series. ■ X-axis annotated with Point-labels or integer values. ■ Appearance determined by ChartStyles' line and symbol properties of each series.
	✔		**Area Radar** Draws each series as connected points of data, filled inside the points. The points are the same as they would be for a Radar chart. Each series is drawn "on top" of the preceding series. ■ X-axis annotated with Point-labels or integer values. ■ Appearance determined by ChartStyles' fill and line properties.

1.3 Loading Data

Data is loaded into a chart by attaching one or more chart data sources to it. A chartable data source is an object that takes real-world data and puts it into a form that JClass Chart can use. Once your data source is attached, you can chart the data in a variety of ways.

Several stock (built-in) data sources are provided with JClass Chart, enabling you to read data from an input stream, a file, a URL, databases, and HTML applet <PARAM> tags. Loading data from a database is called 'data binding'. You can also create your own data sources. See the Data Sources, in Chapter 8 for more information on loading data, data binding, and creating your own data sources.

1.4 Setting and Getting Object Properties

There are four ways to set (and retrieve) JClass Chart properties:

■ By calling property set and get methods in a Java program.

■ By specifying applet properties in an HTML file.

■ By using a Java IDE at design-time (JavaBeans).

■ By using the Chart Customizer at run-time.

Each method changes the same chart property. This manual therefore uses *properties* to discuss how features work, rather than using the method, Customizer tab, or HTML parameter you might use to set that property.

Note: In most cases, you need to understand the chart's object containment hierarchy to access its properties. Use the Objects contained in a chart – traverse contained objects to access properties. diagram to determine how to access the properties of an object.

1.4.1 Setting Properties with Java Code

Every JClass Chart property has a `set` and `get` method associated with it. For example, to retrieve the value of the `AnnotationMethod` property of the first X-axis, the `getAnnotationMethod()` method is called:

```
method = c.getChartArea().getXAxis(0).getAnnotationMethod();
```

To set the `AnnotationMethod` property of the same axis:

```
c.getChartArea().getXAxis(0).setAnnotationMethod(
    JCAxis.POINT_LABELS);
```

These statements navigate the objects contained in the chart by retrieving the values of successive properties, which are contained objects. In the code above, the value of the `ChartArea` property is a `JCChartArea` object. The chart area has an `XAxis` property, the value of which is a collection of `JCAxis` objects. The axis also has the desired `AnnotationMethod` property.

For detailed information on the properties available for each object, consult the online API reference documentation. The API is automatically installed when you install JClass and is found in the *JCLASS_HOME/docs/api/* directory.

1.4.2 Setting Applet Properties in an HTML File

Another way to set chart properties, particularly appropriate for applets, is in an HTML file. Applets built with JClass Chart automatically parse applet `<PARAM>` tags and set the chart properties defined in the file. (A pre-built applet called *JCChartApplet.class* is provided with JClass Chart.) Even standalone Java applications can save the values of chart properties to an HTML file, which can serve as a useful debugging tool.

Using HTML to set properties has the following benefits:

■ Speed – see the effect of different property values quickly without recompiling.

■ Flexibility – use a single applet class to create many different kinds of charts simply by varying HTML properties; end-users can modify HTML properties to suit their own needs.

Chart properties are coded in HTML as applet `<PARAM>` tags. The `NAME` element of the `<PARAM>` tag specifies the property name; the `VALUE` element specifies the property value to set.

This line of code

```
<PARAM name="chart.dataFile" value="sample_1.dat">
```

in the following example HTML file supplies the chart's data in the applet.

```
<HTML>
<HEAD>
<TITLE>Sample Plot Chart</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFCC">
<FONT FACE="ARIAL,VERDANA,HELVETICA" SIZE="-1">
<CENTER><H2>Sample Plot Chart</H2></CENTER>
<P>
<HR COLOR=CC3333>
<P>
<BLOCKQUOTE>
Simple plot chart example.
</BLOCKQUOTE>
<CENTER>
<P>
<APPLET CODEBASE="../../.." WIDTH=400 HEIGHT=300
CODE="com/klg/jclass/chart/applet/JCChartApplet.class">
<PARAM name="chart.dataFile" value="sample_1.dat">
<PARAM name="chart.data.chartType" value="Plot">
<PARAM name="chart.data.series1.label" value="Ser. 1">
<PARAM name="chart.data.series1.symbol.shape" value="triangle">
<PARAM name="chart.data.series2.label" value="Ser. 2">
<PARAM name="chart.data.series2.symbol.shape" value="box">
<PARAM name="chart.data.series3.label" value="Ser. 3">
<PARAM name="chart.data.series3.symbol.shape" value="dot">
<PARAM name="chart.legend.visible" value="true">
<PARAM name="chart.legend.borderType" value="plain">
<PARAM name="chart.yaxis.min" value="5">
<PARAM name="chart.yaxis.max" value="25">
<PARAM name="chart.yaxis.precision" value="0">
<PARAM name="chart.yaxis.tickSpacing" value="2.5">
<PARAM name="chart.xaxis.precision" value="0">
</APPLET>
<P>
<B><I><A HREF="../index.html">More Applet Examples...</A></I></B>
</CENTER>
<!-- copyright information added -->
<P>
<HR COLOR=CC3333>
<P>
<P><FONT FACE="ARIAL,VERDANA,HELVETICA" SIZE=-2><A
HREF="http://www.quest.com/corporate/copyright.html">Copyright&#169;
</A>
2002 Quest Software</FONT></FONT>
</BODY>
</HTML>
```
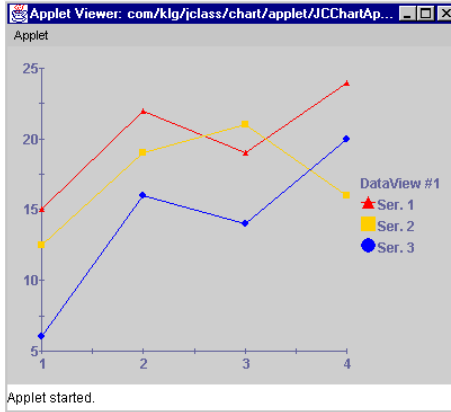
The easiest way to create a set of HTML properties is to use the JClass Chart Customizer to save the property values to an HTML file. For more details, see the The Chart Customizer section in this chapter.

Another way to load and save JClass Chart HTML properties is to use the convenience methods in JCChartFactory. For example, to create a chart from the above HTML file stored in *chart.html*, the following code can be used:

```
LoadProperties loadProps = new LoadProperties();
String inFile = "chart.html";
String chartName = "myChart";
JCChart chart = null;
try {
    chart = JCChartFactory.makeChartFromFile(inFile, loadProps, chartName,
                                             JCChartFactory.HTML);
}
catch (JCIOException e) {
    System.out.println("Error accessing external file:" + e.getMessage());
}
catch (JCParseException e) {
    System.out.println("Error parsing file:" + e.getMessage());
}
catch (IOException e) {
    System.out.println("Error reading " + inFile + ":" + e.getMessage());
}
```

where

- ■ chartName is the name of the chart, and is the name with which each of the properties begins. If there is more than one chart in the HTML file, only the parameters beginning with that name are assigned to the chart. If there is only one set of chart parameters stored in the file, the name can be dropped and an empty String passed as the third parameter.

- ■ LoadProperties is a class containing properties that specify how to load the chart (see LoadProperties for more details).

When a file is read in, all HTML tags, other than the PARAM tags, are ignored. This means that chart information can be read in from a file that contains only PARAM tags. A full listing of the syntax of JClass Chart properties when used in HTML files can be found in Appendix C, HTML Property Reference. Many example HTML files are located in the *JCLASS_HOME/examples/chart/applet* directory.

LoadProperties

The LoadProperties class contains properties that tell JCChart what to do when loading the chart from an HTML file. This class is responsible for the following:

- Telling the chart how to access data files and image files, given the filename. Inside an HTML file, the file access can be specified as one of the following:

  - DEFAULT_ACCESS: The default access is ABSOLUTE.

  - ABSOLUTE: Interprets the filename as an absolute name.

  - RESOLVING_CLASS: Specifies a Class object to use as a resolving class for loading the file. This resolving Class must be set on the LoadProperties object. The ClassLoader of the resolving class is used to resolve the name through a call to getResource(*filename*). In the resolution process, if the filename starts with "/", it is unchanged; otherwise, the package name of the resolving Class is added to the front of the filename after converting "." to "/".

  - URL: Interprets the filename as a URL.

  - RELATIVE_URL: Interpret the filename as URL after adding a relativeURLPrefix to the beginning of it. One must set the relativeURLPrefix on the LoadProperties object (it defaults to the empty String).

- Identifying what to do when there is an error that resulted from reading a data or image source specified within the HTML file. Normally, JCChart throws a JCIOException when this happens. However, you can tell JCChart to ignore these exceptions and continue loading the chart by setting the ignoreExternalResourceExceptions property to true.

- Allowing the user to specify an object to be passed to an external java class when the external-java-code tag is used.

Other JCChartFactory Methods

Other JCChartFactory methods that create or update JCChart's are:

- makeChartFromStream()
- makeChartFromReader()
- makeChartFromString()
- updateChartFromFile()
- updateChartFromStream()
- updateChartFromReader()
- updateChartFromString()

Each of these methods throws a JCParseException if JCChart fails to parse the HTML file.

1.4.3 Saving a JCChart Instance to HTML

The JCChartFactory class also has methods that save a JCChart instance to a stream, file, or String. For example, to save a chart to the file *chart.out.html*, use the following code:

```
String outFile     = "chart.out.html";
String outDataFile = "chart.dat";
ChartDataView dv = chart.getDataView(0);
if (dv != null) {
    OutputDataProperties outProps = dv.getOutputDataProperties();
    if (outProps == null) {
        outProps = new OutputDataProperties();
    }
    outProps.setOutputFileName(outDataFile);
    outProps.setPropertyName("file:///C:/jclass_home/" + outDataFile);
    outProps.setSaveType(OutputDataProperties.DATA_FILE_TXT);
    outProps.setFileAccess(OutputDataProperties.URL);
    dv.setOutputDataProperties(outProps);
}
try {
    JCChartFactory.saveChartToFile(chart, outFile,
                                    JCChartFactory.HTML);
}
catch (IOException e) {
    System.out.println("Error writing to " + outFile + ":" +
                        e.getMessage());
}
```

In the above code, the instance of OutputDataProperties serves two purposes. First, it causes the chart's HTML output to specify how the data for the given ChartDataView should be read in when it is loaded into a JCChart. This is done by specifying three properties:

■ propertyName, which is the name of the data input source. The way this name is interpreted depends on the saveType and fileAccess mechanism.

■ saveType, which is one of:

   ■ NO_DATA: This ChartDataView will have no data when the HTML file is loaded into JCChart. Nothing concerning data is written out.

   ■ EMBED_DATA: The data is embedded directly into the HTML file in text format. No external data file is needed.

   ■ DATA_FILE_TXT: The ChartDataView gets its data from a file and it is specified in text format (see Section 8.9, Data Formats, for the proper format).

   ■ DATA_FILE_XML: The ChartDataView gets its data from a file and it is specified in XML format (see Section 8.8, Loading Data from an XML Source, for the proper format).

■ `fileAccess`, mechanism determines how the `propertyName` is interpreted and dictates which way the data file is accessed when loaded into a `JCChart`. See the `fileAccess` types in LoadProperties.

Second, it tells `JCChart` whether or not it should save the data to a file. If the `outputFileName` property is non-null and the save type is either `DATA_FILE_TXT` or `DATA_FILE_XML`, the data is saved to the specified file based on the save type. Note that `outputFileName` is an absolute file name.

In the previous example, the data is written as text to *chart.dat*. In the output file *chart.out.html*, the data is specified to be read in from the URL as follows:

```
<param name=dataFile value="file:///C:/jclass_home/chart.dat">
<param name=dataFileType value="Text">
<param name=dataFileAccess value="Url">
```

Note: Images are saved in a similar fashion to data files, except that an instance of `OutputProperties` is used instead of `OutputDataProperties`.

Other methods that save a `JCChart` are:

■ `saveChartToStream()`

■ `saveChartToString()`

1.4.4 Updating Charts with Data

`JCChartFactory` also has a method, `updateChartWithData()`, that updates a chart with a new data set. In the given data view, the old data set is replaced by the new data set, with information provided by a file or an input stream. The method definition is:

```
public static void
updateChartWithData(JCChart chart, int dataType, Object data,
                        int dataViewIndex, LoadProperties)
```

where

■ `chart` is the chart to update.

■ `dataType` is the type of data (either `DATA_FILE_TEXT` or `DATA_FILE_XML`).

■ `data` is the data object (either a `Reader`, a filename String or an `InputStream`).

■ `dataViewIndex` is the index of the `ChartDataView` on which the data is to be set (there is also a method where the `ChartDataView` can be specified by name).

■ `LoadProperties` is a class containing properties that specify how to load the chart (see LoadProperties for more details).

This method creates a data source from the data object, and sets this new data source on the appropriate `ChartDataView` on the chart.

1.4.5 Setting Properties with a Java IDE at Design-Time

A JClass Chart Bean can be used with a Java Integrated Development Environment (IDE), and its properties can be manipulated at design-time. Consult your IDE's documentation for details on how to load third-party Bean components into the IDE.

You can also refer to the JClass and Your IDE chapter in the *JClass DesktopViews Installation Guide*.

Most IDEs list a component's properties in a property sheet or dialog. Simply find the property you want to set in this list and edit its value. Again, consult your IDE's documentation for complete details.

1.4.6 Setting Properties Interactively at Run-Time

If enabled by the developer, end-users can manipulate property values on a chart running in your application. Clicking a mouse button launches the JClass Chart Customizer. The user can navigate through the tabbed dialogs and edit the properties displayed.

For details on enabling and using the Customizer, see The Chart Customizer later in this chapter.

1.5 Other Programming Basics

Working with Object Collections
Many chart objects are organized into collections. For example, the chart axes are organized into the XAxis collection and the YAxis collection. In Beans terminology, these objects are held in indexed properties.

To access a particular element of a collection, specify the index that uniquely identifies this element. For example, the following code changes the maximum value of the first X-axis to 25.1:

```
c.getChartArea().getAxis(0).setMax(25.1);
```

Note that the index zero refers to the first element of a collection. Also, note that by default, JCChartArea contains one element in XAxis and one in YAxis.

Also note that for a Polar, Radar, and Area Radar chart, there can be only one Y-axis and one X-axis.

Calling Methods
To call a JClass Chart method, access the object that defines the method. For example, the following statement uses the coordToDataCoord() method, defined by the ChartDataView collection, to convert the location of a mouse click event in pixels to their equivalent in data coordinates:

```
JCDataCoord dc = c.getDataView(0).coordToDataCoord(10,15);
```

Details on each method can be found in the API documentation for each class.

1.6   JClass Chart Inheritance Hierarchy

The following provides an overview of class inheritance of JClass Chart.

Class Inheritance Hierarchy

Visuals:

Data Source:
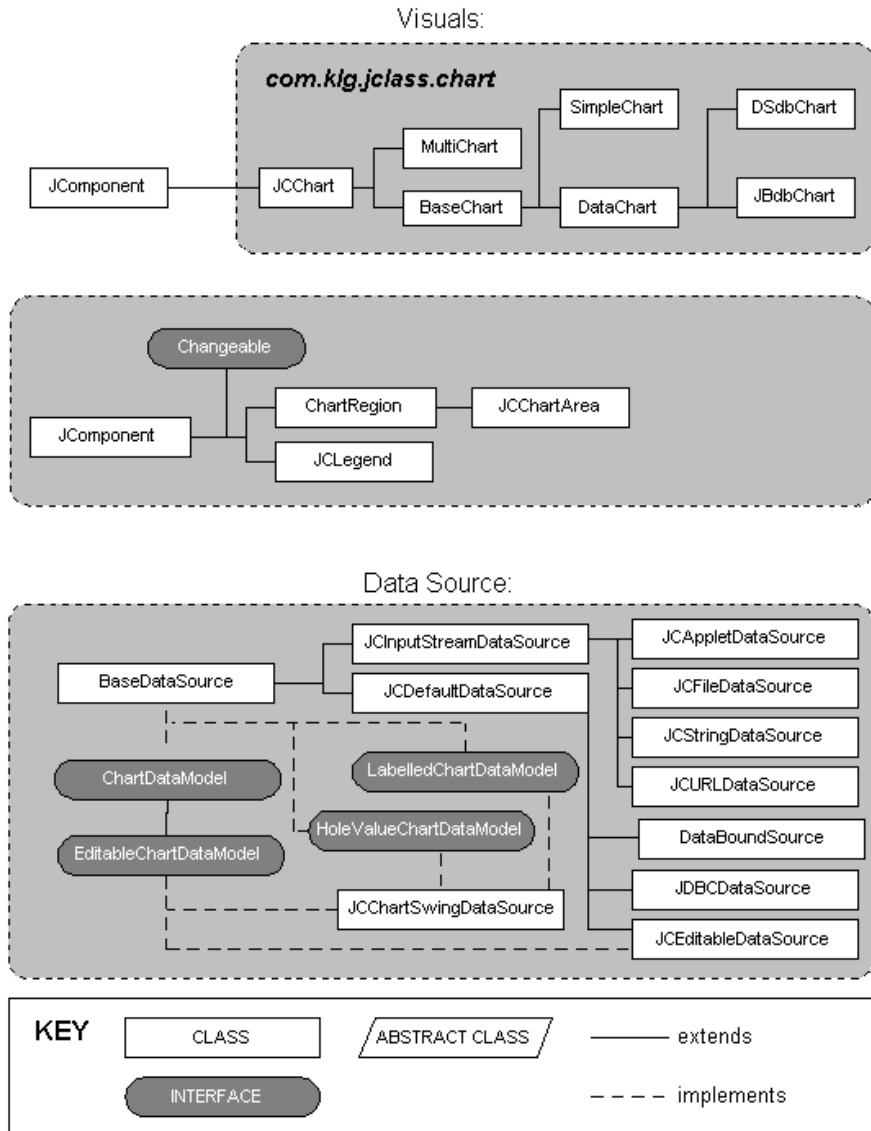
Figure 2   Class hierarchy of the `com.klg.jclass.chart` *package.*

1.7   JClass Chart Object Containment

When you create (or instantiate) a new chart, several other objects are also created. These objects are contained in and are part of the chart. Chart programmers need to traverse these objects to access the properties of a contained object. The following diagram shows the object containment for JClass Chart.
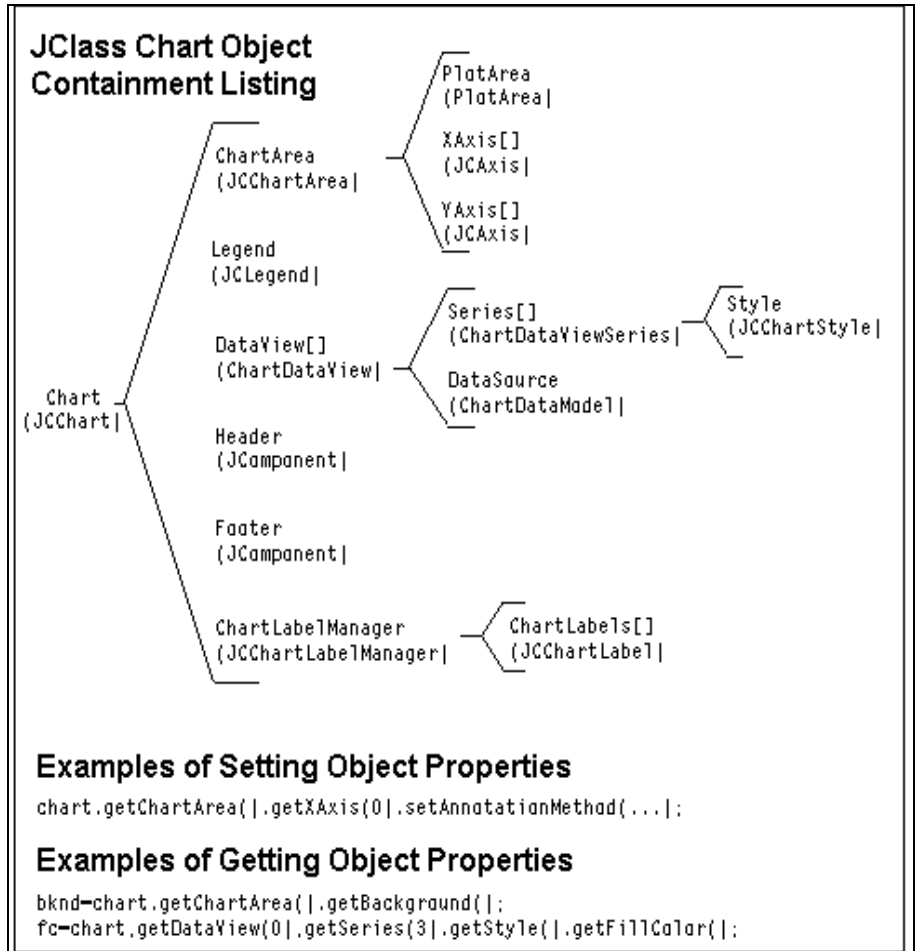
Figure 3   Objects contained in a chart – traverse contained objects to access properties.

JCChart (the top-level object) manages header and footer JComponent objects, a legend (JCLegend), and the chart area (JCChartArea). The chart also contains a collection of data view (ChartDataView) objects and can contain the ChartLabelManager (JCChartLabelManager) which manages a collection of chart label (JCChartLabel) objects.

The chart area contains most of the chart's actual properties because it is responsible for charting the data. It also contains and manages a collection of X-axis (JCAxis) objects and Y-axis (JCAxis) objects (one of each by default).

The data view collection contains objects and properties (like the chart type) that are tied to the data being charted. Each data view contains a collection of series (ChartDataViewSeries) objects, one for each series of data points, used to store the visual display style of each series (JCChartStyle).

Note that chart does not own the data itself, but instead merely views on the data. Each data view also contains a data source (ChartDataModel) object. The data is owned by the DataSource object. This is an object that your application creates and manages separately from the chart. For more information on JClass Chart's data source model, see Data Sources.

1.8   The Chart Customizer

The JClass Chart Customizer enables developers (or end-users if enabled by your program) to view and customize the properties of the chart as it runs.
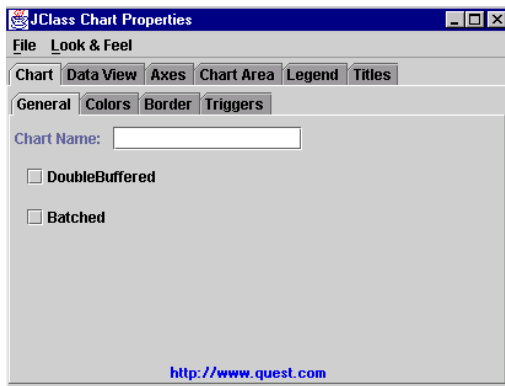
Figure 4   The JClass Chart Customizer.

The Customizer can save developers a lot of time. Charts can be prototyped and shown to potential end-users without having to write any code. Developers can experiment with combinations of property settings, seeing results immediately in the context of a running application, greatly aiding chart debugging.

1.8.1 Displaying the Chart Customizer at Run-Time

By default, the Customizer is disabled at run-time. To enable it, you need to set the chart's `AllowUserChanges` and `Trigger` properties, for example:

```
chart.setAllowUserChanges(true);
chart.setTrigger(0, new EventTrigger(InputEvent.META_MASK,
    EventTrigger.CUSTOMIZE);
```

To display the Customizer once it has been enabled, move the mouse over the chart and click the *secondary* mouse button; that is, the button on your system that displays popup menus, for example:

■ Windows – Right mouse button

■ UNIX – Middle mouse button

1.8.2 Editing and Viewing Properties

1. Select the tab that corresponds to the chart element that you want to edit. Tabs contain one or more inner tabs that group related properties together. Select inner tabs to narrow down the type of property you want to edit.

2. If you are editing an indexed property, select the specific object to edit from the lists displayed in the tabs. The fields in the tab update to display the current property values.

3. Select a property and edit its value.
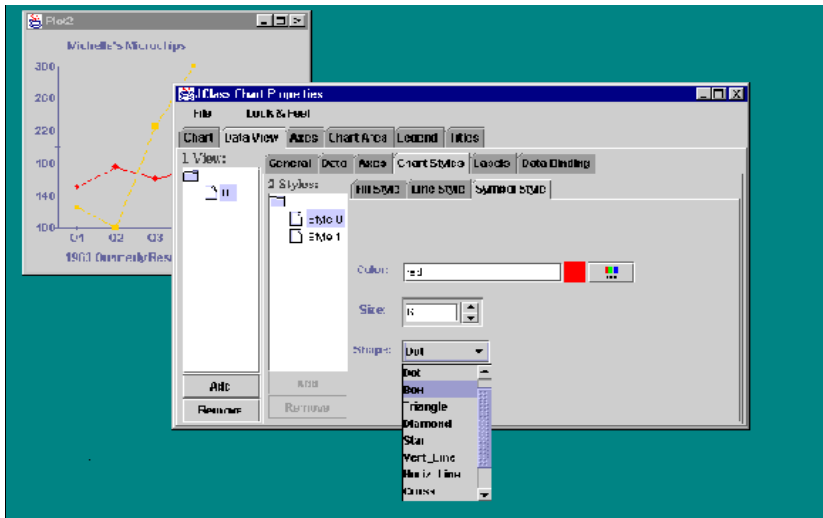
Figure 5   Editing a sample chart with the Customizer.

As you change property values, the changes are immediately applied to the chart and displayed. You can make further changes without leaving the Customizer. However, once you have changed a property the only way to "undo" the change is to manually change the property back to its previous value.

To close the Customizer, close its window (the actual steps differ for each platform).

1.9   Internationalization

Internationalization is the process of making software that is ready for adaptation to various languages and regions without engineering changes. JClass products have been internationalized.

Localization is the process of making internationalized software run appropriately in a particular environment. All Strings used by JClass that need to be localized (that is, Strings that are seen by a typical user) have been internationalized and are ready for localization. Thus, while localization stubs are in place for JClass, this step must be implemented by the developer of the localized software. These Strings are in resource bundles in every package that requires them. Therefore, the developer of the localized software who has purchased source code should augment all .java files within the */resources/* directory with the .java file specific for the relevant region; for example, for France, *LocaleInfo.java* becomes *LocaleInfo_fr.java*, and needs to contain the translated French versions of the Strings in the source *LocaleInfo.java* file. (Usually the file is called *LocaleInfo.java*, but can also have another name, such as *LocaleBeanInfo.java* or *BeanLocaleInfo.java*.)

Essentially, developers of the localized software create their own resource bundles for their own locale. Developers should check every package for a */resources/* directory; if one is found, then the *.java* files in it will need to be localized.

For more information on internationalization, go to:
http://java.sun.com/j2se/1.4.2/docs/guide/intl/index.html.

2

Chart Types and
Special Chart Properties

In this chapter, chart types are discussed and special features of chief JClass Chart charting types are outlined.

2.1 Chart Type: Polar Charts
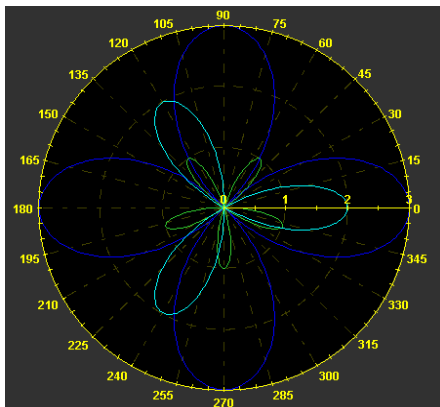
A polar chart draws the X- and Y-coordinates in each series as (*theta,r*), where *theta* is amount of rotation from the X-origin and *r* is the distance from the Y-origin. *theta* may be specified in degrees (default), radians, or gradians. Because the X-axis is a circle, the X-axis maximum and minimum values are fixed.

Using ChartStyles, you can customize the line and symbol properties of each series.

2.1.1 Background Information for the Polar Charts

In order to work efficiently with Polar charts, you should understand the following basic concepts.

Theta

Theta (θ), which is the angle from the X-axis origin, is measured in a counterclockwise direction. In cartesian (rectangular) X- and Y-plots, theta "translates" to the X-axis.

r value

r represents the distance from the Y-axis origin. In cartesian (rectangular) X- and Y-plots, r "translates" to the Y-axis. Multiple r values are allowed.
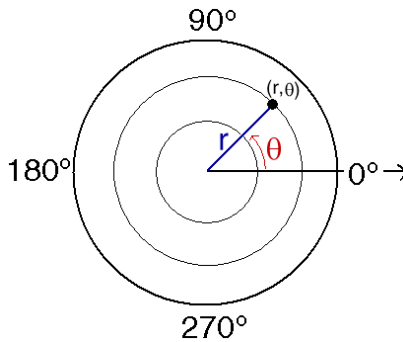
Angles

Angles can be measured in degrees, radians, or gradians.

X and Y Values in Polar Charts
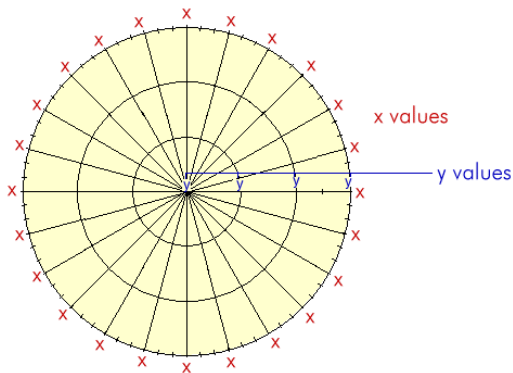
2.1.2 Setting the Origin

All angles are relative to the origin base angle.

The position of the X-axis origin is determined by the origin base angle. The **OriginBase** property is a value between 0 and 360 degrees (if the angle unit is degrees).

In the Property Editor, the **OriginBase** property is located on the **Polar/Radar** inner tab, located in the **DataView** tab's **General** tab.

The Y-axis angle is the angle that the Y-axis makes with the origin base.

The origin base angle is set to 0° by default. The Y-axis angle is set to 0° to the origin base by default.
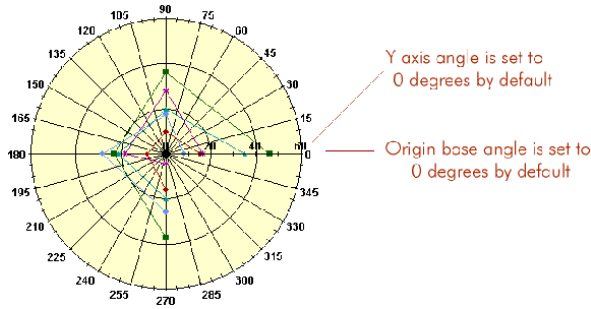
You can change the origin base angle, the Y-axis angle, or both.
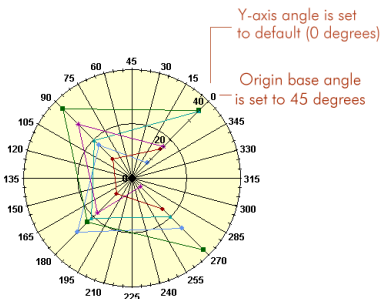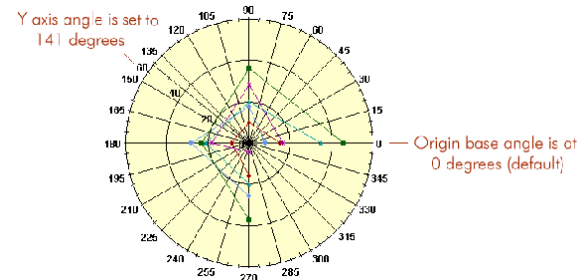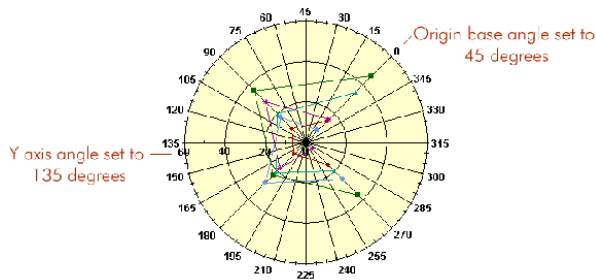
2.1.3 Data Format

The data format for Polar charts is either:

■   general – (*x*,*y*) for every series; or

■   array – only one X-value.

The X-array contains the theta values; the Y-array contains the r values. For array data, the X-array represents a fixed theta value for each point.

For more information on general and array data, please see the discussion in Loading Data From a File, in Chapter 6.
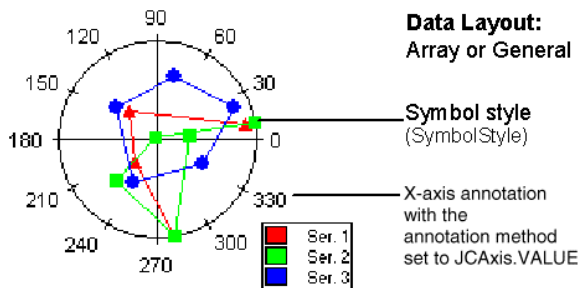
2.1.4 PolarChartDraw class

The `PolarChartDraw` class (which extends `ChartDraw`) is a drawable object for Polar charts. This object is used for rendering a Polar chart based on data contained in the `dataObject`.

The default constructor is `PolarChartDraw()`.

There are two key methods in this class:

■   `recalc()` – recalculates the extents of related objects

■   `draw()` – draws related objects and takes as its parameter the graphics context to use for drawing

2.1.5 Full or Half-Range X-Axis

Use the **HalfRange** property to determine whether the X-axis is displayed as one full range from 0 to 360 degrees (**HalfRange** is `false`) or two half-ranges: from –180 degrees to zero degrees to 180 degrees (**HalfRange** is `true`). In interval notation the range would

be [0,360) when **HalfRange** is `false` and (−180, 180] when **HalfRange** is `true`. The default value for the **HalfRange** property is `false`.
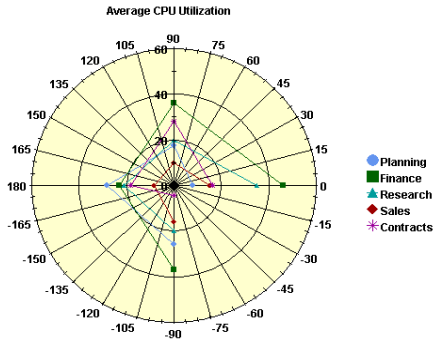
Figure 6   Half-range is On.
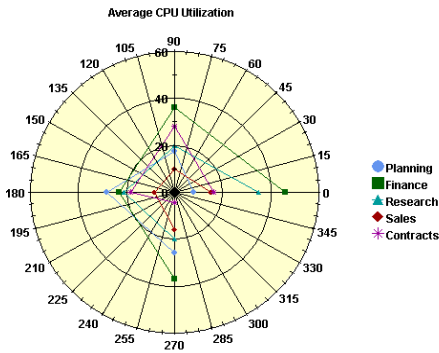
Figure 7   Half-range is Off.

This property is exclusive to Polar charts.

The **HalfRange** property is located on the **Polar/Radar** inner tab on the Property Editor, in the **DataView** tab's **General** tab.

2.1.6   Allowing Negative Values

Polar charts do not allow negative values for the Y-axis unless the Y-axis is reversed. A negative radius is interpreted as a positive radius rotated 180 degrees.
Thus (*theta*, *r*) = (*theta* +180, −*r*)

2.1.7   Gridlines

Polar charts allow for gridlines to be turned on and off.

Use the `JCAXIS.setGridVisible()` method to show or hide gridlines. The default is off.

For Polar charts, Y-gridlines will be circular while X-gridlines will be radial lines from the center to the outside of the plot.

2.2   Chart Type: Radar Charts
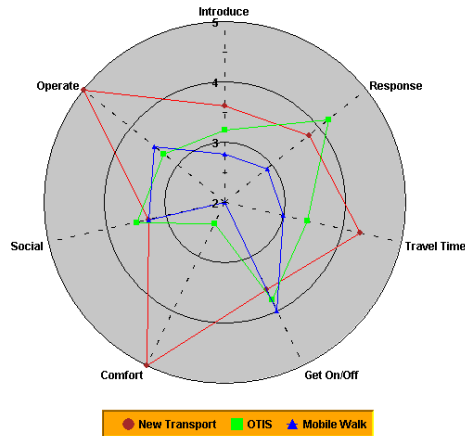
A Radar chart plots data as a function of distance from a central point. A line connects the data points for each series, forming a polygon around the chart center.

A Radar chart draws the Y-value in each data set along a radar line (the X-value is ignored). If the data set has *n* points, then the chart plane is divided into *n* equal angle segments, and a radar line is drawn (representing each point) at *360/n* degree increments. By default, the radar line representing the first point is drawn horizontally (at 0 degrees).

Radar charts permit easy visualization of symmetry or uniformity of data, and are useful for comparing several attributes of multiple items. Although Radar charts look as if they have multiple Y-axes, they have only one; hence, you cannot change the scale of just one spoke.

Using ChartStyles, you can customize the line and symbol properties of each series.

2.2.1 Background Information for Radar Charts

An example of the X- and Y-values of a Radar chart is shown below; in this case, there are seven X-values and three series of Y-values.
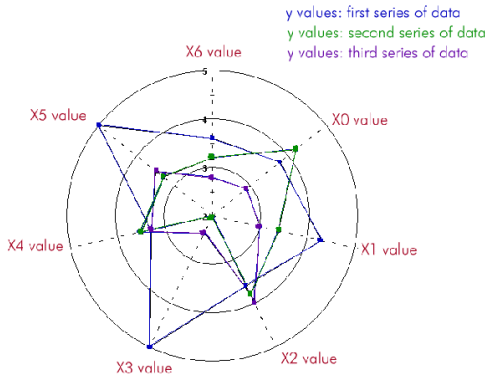
2.2.2 Data Format

A Radar chart uses only array data. For more information on array data, please see the discussion in Loading Data From a File, in Chapter 6.
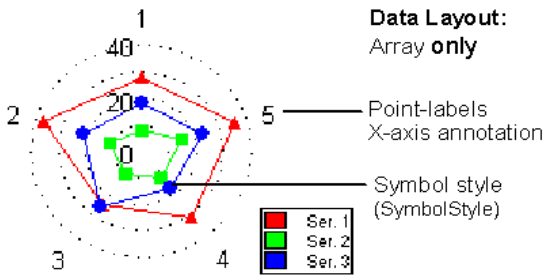
2.2.3 RadarChartDraw Class

The `RadarChartDraw` class (which extends `PolarChartDraw`) is a drawable object for radar charts. This object is used for rendering a radar chart based on data contained in the `dataObject`.

The default constructor is `RadarChartDraw()`.

There are two key methods in this class:

- `recalc()` – recalculates the extents of related objects
- `draw()` – draws related objects and takes as its parameter the graphics context to use for drawing

2.2.4   Gridlines

Radar lines are represented by the X-axis gridlines. You may choose normal gridlines (circular) or "webbed" gridlines. As with other chart types, gridlines may be displayed or hidden (default is hidden).
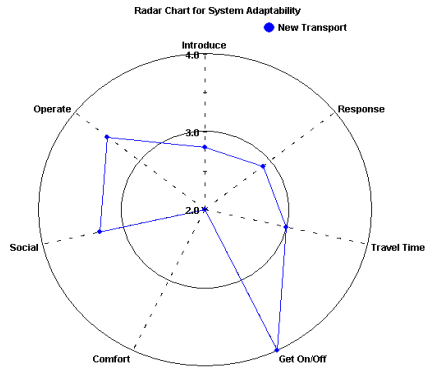
Figure 8   Circular Gridlines.
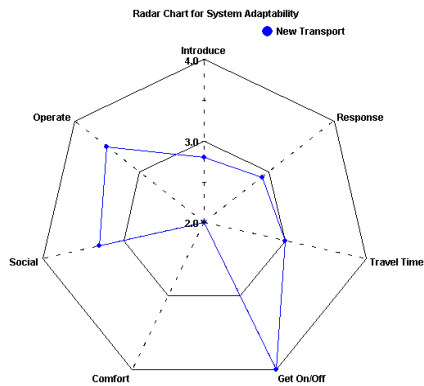
Figure 9   Webbed Gridlines.

2.3   Chart Type: Area Radar Charts
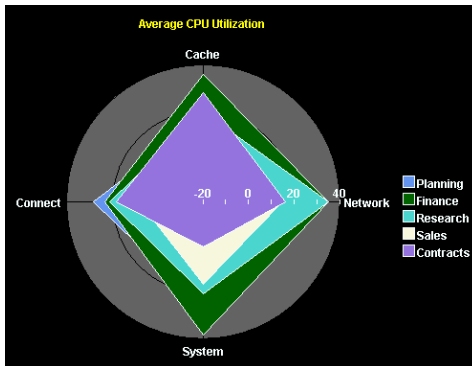
An area radar chart draws the Y-value in each data set along a radar line (the X-value is ignored). If the data set has *n* points, the chart plane is divided into *n* equal angle segments, and a radar line is drawn (representing each point) at *360/n* degree increments. Each series is drawn "on top" of the preceding series.

Area radar charts are the same as Radar charts, except that the area between the origin and the points is filled.

Using ChartStyles, you can customize the fill and line properties of each series.

2.3.1   Background Information for Area Radar Charts

An example of the X- and Y-values of an Area Radar chart is shown below; in this case, there are seven X-values and three series of Y-values.
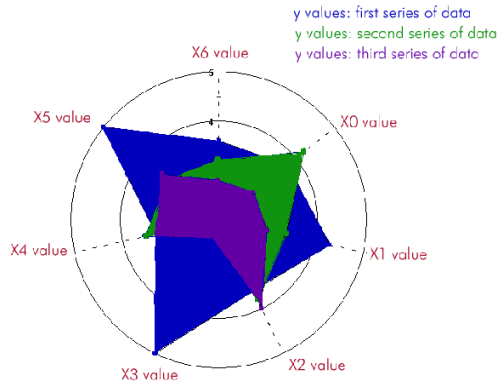
2.3.2   Data Format

An Area Radar chart uses only array data. For more information on array data, please see the discussion in Loading Data From a File, in Chapter 6.
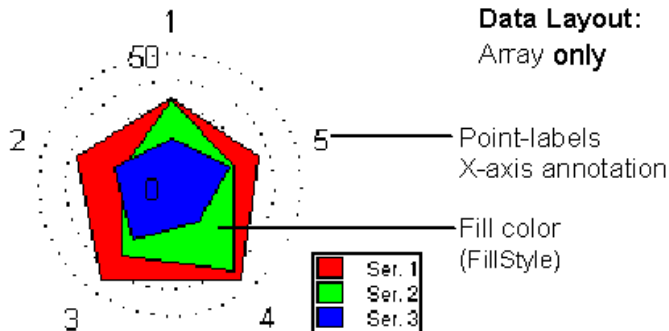
2.3.3   AreaRadarChartDraw Class

The `AreaRadarChartDraw` class (which extends `RadarChartDraw`) is a drawable object for Area Radar charts. This object is used for rendering an Area Radar chart based on data contained in the `dataObject`.

The default constructor is `AreaRadarChartDraw()`.

2.3.4   Gridlines

Radar lines are represented by the X-axis gridlines. You may choose normal gridlines (circular) or "webbed" gridlines. As with other chart types, gridlines may be displayed or hidden (default is hidden).
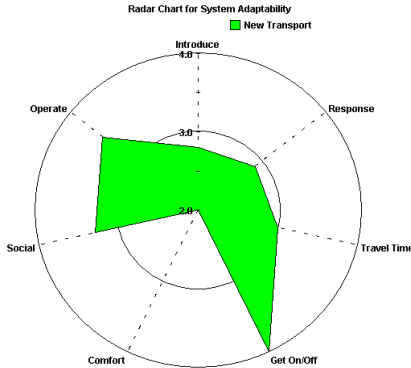
Figure 10   Circular Gridlines.
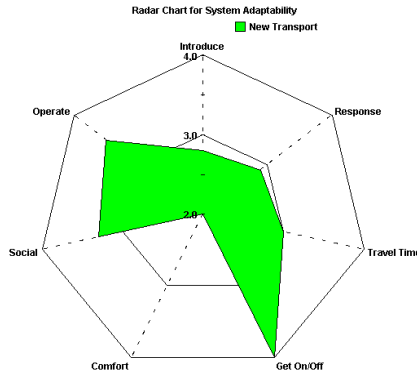
Figure 11   Webbed Gridlines.

2.4   JCPolarRadarChartFormat Class

The `JCPolarRadarChartFormat` class provides methods to get or set properties specific to Polar, Radar, or Area Radar charts.

Origin Base

The origin base is the angle at which the theta axis origin is displayed. A value of 0 degrees corresponds to the 3 o'clock position.

Set or get the origin base using the following public methods:

- ■ `public void setOriginBase(int units, double angle);`
- ■ `public double getOriginBase(int units);`

The `units` parameter can have values of `JCChartUtil.DEGREES`, `JCChartUtil.RADIANS`, or `JCChartUtil.GRADS`.

Alternatively, you can call the following methods without specifying an angle unit to get or set the origin base. In this case, the angle units are assumed to be the current value of the chart area's `angleUnit` property:

- ■ `public void setOriginBase(double angle);`
- ■ `public double getOriginBase();`

Y-Axis Angle

The Y-axis angle is the angle at which the Y-axis is displayed relative to the theta axis origin. Set or get the Y-axis angle using the following public methods:

- ■ `public void setYAxisAngle(int units, double angle);`
- ■ `public double getYAxisAngle(int units);`

Alternatively, you can call the following methods without specifying an angle unit to get or set the Y-axis angle. In this case, the angle units are assumed to be the current value of the chart area's `angleUnit` property:

- ■ `public void setYAxisAngle(double angle);`
- ■ `public double getYAxisAngle();`

Half-Range Flag

If the half-range flag is set, the theta axis labels range from –180 to 180 degrees. Set or get the half-range flag using the following methods:

- ■ `public void setHalfRange(boolean fHalfRange);`
- ■ `public boolean isHalfRange();`

RadarCircularGrid

The `isRadarCircularGrid` property is specific to Radar and Area Radar charts. If the circular grid flag is set, Y-gridlines will be circular; otherwise, the Y-grid will be webbed. Set or get the `isRadarCircularGrid` property using the following methods:

- ■ `public void setRadarCircularGrid(boolean fCircular);`
- ■ `public boolean isRadarCircularGrid();`

2.5   Special Bar Chart Properties

Bar charts display each point as one bar in a *cluster*. There are several properties defined
in `JCBarChartFormat` that control exactly how the bars are spaced and displayed. Use the
`getChartFormat(JCChart.BAR)()` method to retrieve and set these properties.

Cluster Overlap
Use the bar `ClusterOverlap` property to set the amount that bars in a cluster overlap each
other. The default value is 0. The value represents the percentage of bar overlap. Negative
values add space between bars and positive values cause bars to overlap. Valid values are
between -100 and 100. The syntax is as follows:

`((JCBarChartFormat)dataView.getChartFormat()).setClusterOverlap(50)`
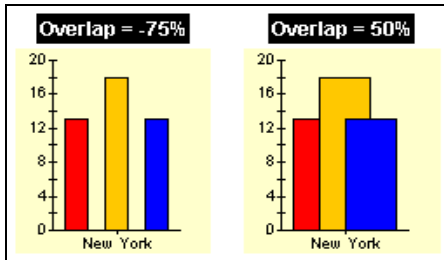
Figure 12   Negative and positive bar cluster overlap.

Cluster Width
Use the bar `ClusterWidth` property to set the space used by each bar cluster. The default
value is 80. The value represents the percentage available space, with valid values
between 0 and 100. The syntax is as follows:

`((JCBarChartFormat)dataView.getChartFormat()).setClusterWidth(100)`
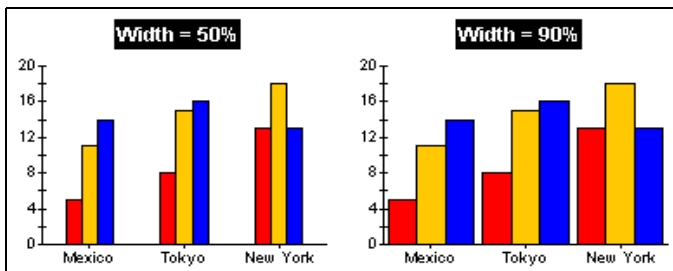
Figure 13   Setting different bar cluster widths.

100-Percent Stacking Bar Charts
The Y-axes of stacking bar charts can display a percentage interpretation of the bar data
using the `100Percent` property. When set to `true`, each stacked bar's total Y-values

represents 100%. The Y-value of each bar is interpreted as its percentage of the total. This property has no effect on bar charts. The syntax is as follows:

```
((JCBarChartFormat)dataView.getChartFormat()).set100Percent(true)
```

2.6   Special Pie Chart Properties

Pie charts are quite different from the other chart types. They do not have the concept of a two-dimensional grid or axes. They also introduce a special category called *"Other"*, into which all data values below a certain threshold can be grouped.

You can customize your pie charts with the properties of `JCPieChartFormat`. The following code snippet shows the syntax for setting `JCPieChartFormat` properties:

```
JCPieChartFormat pcf = (JCPieChartFormat) arr.getChartFormat();
pcf.setOtherLabel("Other Bands");
pcf.setThresholdValue(10.0);
pcf.setThresholdMethod(JCPieChartFormat.PIE_PERCENTILE);
pcf.setSortOrder(JCPieChartFormat.DATA_ORDER);
pcf.setStartAngle(90.0);
```

2.6.1   Building the "Other" Slice

Pie charts are often more effective if unimportant values are grouped into an "Other" category. Use the `ThresholdMethod` property to select the grouping method to use. `SLICE_CUTOFF` is useful when you know the data value that should be grouped into the

"Other" slice. PIE_PERCENTILE is useful when you want a certain percentage of the pie to be devoted to the "Other" slice.
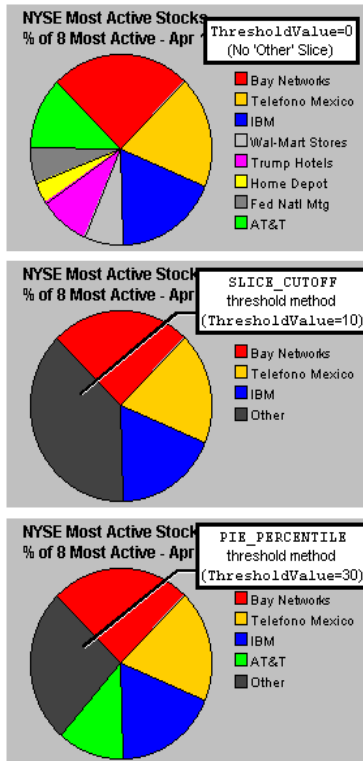
Figure 14   Three JClass Charts illustrating how the "Other" slice can be used.

Use the MinSlices property to fine-tune the number of slices displayed before the "Other" slice. For example, when set to 5, the chart tries to display 5 slices in total. This means that, if there is an "Other" slice, the chart will display 4 slices and the "Other" slice; if there is no "Other" slice, the chart will display 5 or more slices.

2.6.2   "Other" Slice Style and Label

The OtherStyle property allows access to the ChartStyle used to render the "Other" slice. Use FillStyle's Pattern and Color properties to define the appearance of the Other slice.

Use the OtherLabel property to change the label of the "Other" slice.

2.6.3   Pie Ordering

Use the `SortOrder` property to specify whether to display slices largest-to-smallest, smallest-to-largest, or the order they appear in the data.

2.6.4   Start Angle

The position in the pie chart where the first pie slice is drawn can be specified with the `StartAngle` property. A value of zero degrees represents a horizontal line from the center of the pie to the right-hand side of the pie chart; a value of 90 degrees represents a vertical line from the center of the pie to the top-most point of the pie chart; a value of 180 degrees represents a horizontal line from the center of the pie to the left-hand side of the pie chart; and so on. Slices are drawn clockwise from the specified angle. Values must lie in the range from zero to 360 degrees. The default value is 135 degrees.

2.6.5   Exploded Pie Slices

It is possible to have individual slices of a pie "explode" (that is, detach from the rest of the pie). Exploded slices can be used in both 2D and 3D pie charts.

Two properties of `JCPieChartFormat` are responsible for this function: `ExplodeList` and `ExplodeOffset`.

`ExplodeList` specifies a list of exploded pie slices in the pie charts. It takes *pts* as a parameter, which is composed of an array of `Point` objects. Each point object contains the data point index (pie number) in the X-value and the series number (slice index) in the Y-value, specifying the pie slice to explode. To explode the "other" slice, the series number should be `OTHER_SLICE`. If null, no slices are exploded.

`ExplodeOffset` specifies the distance a slice is exploded from the center of a pie chart. It takes `off` as a parameter, which is the explode offset value.

The following code sample shows how `ExplodeList` and `ExplodeOffset` can be used to set the list of exploded slices.

```
Point[] exList = new Point[3];
exList[0] = new Point(0, 0);
exList[1] = new Point(1, 5);
exList[2] = new Point(2, JCPieChartFormat.OTHER_SLICE);
pcf.setExplodeList(exList);
pcf.setExplodeOffset(10);
```

The following code sample shows how to set up a pick listener such that when a user clicks on an individual pie slice, that slice explodes (and then implodes if the user clicks on it again):

```
public void pick(JCPickEvent e)
{
    JCDataIndex di = e.getPickResult();
    if (di == null) return;
    Object obj = di.getObject();
```

```
            ChartDataView vw = di.getDataView();
            ChartDataViewSeries srs = di.getSeries();
            int slice = di.getSeriesIndex();
            int pt = di.getPoint();
            int dist = di.getDistance();
            if (vw != null && slice != -1) {
                JCPieChartFormat pcf = (JCPieChartFormat)vw.getChartFormat();
                Point[] exList = pcf.getExplodeList();
                if (exList == null) return;
                // implode existing exploded slices
                for (int i = 0; i < exList.length; i++) {
                    if ((exList[i].x == pt) && (exList[i].y == slice)) {
                        Point[] newList = new Point[exList.length - 1];
                        for (int j = 0; j < i; j++)
                            newList[j] = exList[j];
                        for (int j = i; j < newList.length; j++)
                            newList[j] = exList[j + 1];
                        pcf.setExplodeList(newList);
                        return;
                    }
                }
                // explode new slice
                Point[] newList = new Point[exList.length + 1];
                for (int j = 0; j < exList.length; j++)
                    newList[j] = exList[j];
                newList[exList.length] = new Point(pt, slice);
                pcf.setExplodeList(newList);
            }
        }
    }
```

The full code for this program can be found in *JCLASS_HOME/examples/chart/ interactions/.* For more information on `pick`, see Using Pick and Unpick, in Chapter 10.

Saving and Loading Exploding Pie Slices

Exploded pie slice properties can be saved or loaded to or from HTML. This is done by passing `JCPieChartFormat`'s `setExplodeList()` method an array of `Point` objects which correspond to the exploded series and points. For each `Point` object in the array, the *X* value represents the pie (or point) number, while the *Y* value represents the slice (or series) number. To specify all of the points or series, use the `ALL` integer; to specify that the "other" slice should be exploded, use `other` as the *Y* value.

In HTML, the code should resemble the following:

```
<APPLET CODEBASE="../../" ARCHIVE="lib/jcchart.jar" WIDTH=450 HEIGHT=300
CODE="com/klg/jclass/chart/applet/JCChartApplet.class">
<PARAM name="dataFile" value="sample_1.dat">
<PARAM name="data.chartType" value="PIE">
<PARAM name="data.pie.explodeList" value="0,all|all,1|3,3|4,other">
</APPLET>
```

where all slices on the first pie are exploded (`0,ALL`), the slices corresponding to the first dataseries are exploded on all pies (`ALL,1`), the slice corresponding to the third dataseries is exploded on the fourth pie (`3,3`), and the fifth pie's "other" slice is also exploded

(4,other). Note that the pie (or point) number starts at 0; therefore, the first pie is 0, the second is 1, and so on.

2.7   Special Area Chart Properties

Similar to the stacking bar type, a stacking area chart is provided in JClass Chart. To see an example of a stacking area chart, launch the Area demo from *JCLASS_HOME/demos/chart/area/.*

Stacking Area Charts

A stacking area chart places each Y-series on top of the last. This shows the area relationships between each series and the total. The following example shows the same set of data as displayed by stacking area and area types:
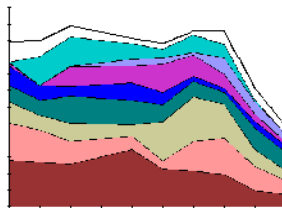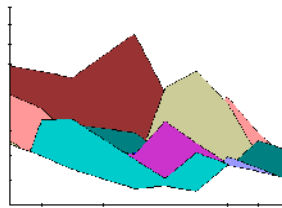
Stacking Area Chart   **Area Chart**

To create a stacking area chart, set the `ChartType` property to `JCChart.STACKING_AREA`, as follows:

```
dataView.setChartType(JCChart.STACKING_AREA);
```

100-Percent Stacking Area Charts

When `100Percent` property is set to `true`, the Y-axes display as an area percentage of the total. The top of the chart is 100% (the total of all Y-values).
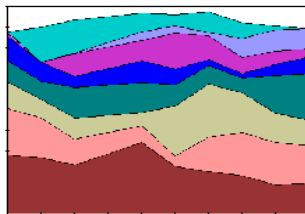
100-Percent Stacking Area Chart

Use the following syntax to display data in 100-Percent mode:

```
((JCAreaChartFormat)dataView.getChartFormat()).set100Percent(true)
```

2.8   Hi-Lo, Hi-Lo-Open-Close, and Candle Charts

JClass Chart's Hi-Lo, Hi-Lo-Open-Close, and Candle financial chart types use the Y-values in multiple series to construct each "bar". Hi-Lo charts use every *two* series and Hi-Lo-Open-Close and candle charts use every *four* series. Each series defines a specific portion of the bar:

■   First series – High value

■   Second series – Low value

■   Third series (if needed) – Open value

■   Fourth series (if needed) – Close value
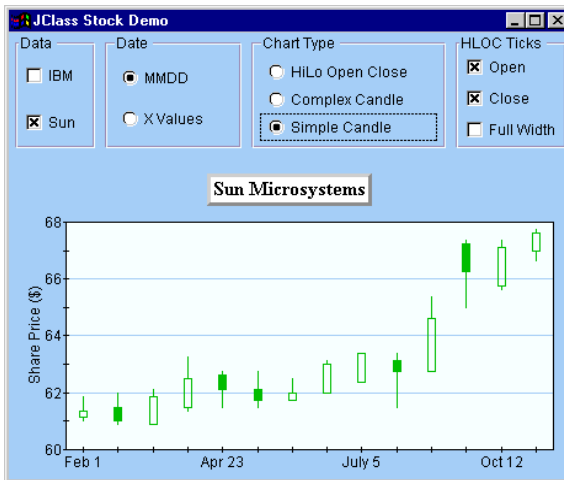
Figure 15   Simple Candle chart displayed by stock demo.

It is useful to think of each group of series as one "logical series". But note that most JClass Chart properties or methods that use a series (such as chart labels attached by DataIndex) use the *actual* series index.

Hi-Lo-Open-Close Charts

When the chart type is `JCChart.HILO_OPEN_CLOSE`, several properties defined in `JCHLOCChartFormat` control how open and close ticks are displayed:

ShowingOpen	Displays or hides open tick marks
ShowingClose	Displays or hides close tick marks
OpenCloseFullWidth	Displays open/close ticks across both sides of the bar. This is useful for creating error bar charts.

Customizing Chart Styles

Because these chart types use multiple series for each "row" of Hi-Lo or Candle bars, it is difficult to determine which chart style specifies the display attributes of a particular row of bars. To make programming the chart styles of financial charts easier, JClass Chart provides several methods that retrieve and set the style for a *logical* series. These methods are defined in the JCHiloChartFormat, JCHLOCChartFormat, and JCCandleChartFormat classes. Each get method returns the JCChartStyle object used for the logical series you specify. You can customize the properties in this returned object and then use the appropriate set method to apply them to the same logical series in the chart.

Most of the financial chart types use only one or two JCChartStyle properties. The following table lists the properties used by each chart type (see Chart Styles, in Chapter 9 for more information on chart styles):

	LineColor	SymbolSize
Hi-Lo	✔	
Hi-Lo-Open-Close	✔	✔
Candle (simple)	✔	✔
Candle (complex)	see below	

For every financial chart type, except complex candle, the actual chart style used is that of the *first* series.

Simple and Complex Candle Charts

You can choose between a simple and complex candle chart display using the Complex property defined in JCCandleChartFormat.

When set to false, the chart style from just *one* series (the first) determines the appearance of the candle. The table above shows the properties used. A rising stock price is indicated by making the candle transparent. A falling stock price displays in the color specified by FillColor.

Complex candle charts (Complex is true), use elements of the chart styles of all *four* series, providing complete control over every visual aspect of the candles. The convenience methods defined in JCCandleChartFormat make it easy to retrieve/set the style that controls the appearance of a particular aspect of the candles.

The following lists the JCChartStyle properties that control each aspect of a complex candle, along with which of the four chart styles is used:

■ Hi-Lo line – LineColor property (first chart style).

■ Rising price candle color and width – FillColor and SymbolSize properties (second chart style).

- Falling price candle color and width – `FillColor` and `SymbolSize` properties (third chart style).
- Candle outline – `LineColor` property (fourth chart style).

Example Code
The following code sets the rising and falling candle styles of a complex candle chart:

```
JCChartStyle chartStyle;
JCCandleChartFormat candleFormat;
```

```
// Set candle to complex type so we can change colors
 candleFormat=(JCCandleChartFormat)chart.getDataView(1).getChartFormat();
candleFormat.setComplex(true);

// Change rising candle color
chartStyle = candleFormat.getRisingCandleStyle(0);
chartStyle.setLineColor(Color.green);
chartStyle.setFillColor(Color.red);

// Change falling candle color
chartStyle = candleFormat.getFallingCandleStyle(0);
chartStyle.setLineColor(Color.green);
chartStyle.setFillColor(Color.yellow);
```

Two demo programs included with JClass Chart illustrate creating financial charts: the stock demo, located in *JCLASS_HOME/demos/chart/stock/*, and the financial demo, located in *JCLASS_HOME/demos/chart/financial.*

3

SimpleChart Bean Tutorial

3.1   Introduction to JavaBeans

JClass Chart components are JavaBean-compliant. The JavaBeans specification makes it very easy for a Java Integrated Development Environment (IDE) to "discover" the set of properties belonging to an object. The developer can then manipulate the properties of the object easily through the graphical interface of the IDE when constructing a program.

The three main characteristics of a Bean are:

■   the set of properties it exposes

■   the set of methods it allows other components to call; and

■   the set of events it fires

Properties control the appearance and behavior of the Bean. Bean methods can also be called from other components. Beans fire events to notify other components that an action has happened.

3.1.1   Properties

"Properties" are the named method attributes of a class that can affect its appearance or behavior. Properties that are readable have a "get" (or "is" for booleans) method, which enables the developer to read a property's value, and those properties that are writable have a "set" method, which enables a property's value to be changed.

For example, the `JCAxis` object in JClass Chart has a property called `AnnotationMethod`. This property is used to control how an axis is labelled. To set the property value, the `setAnnotationMethod()` method is used. To get the property value, the `getAnnotationMethod()` method is used.

For complete details on how JClass Chart's object properties are organized, see JClass Chart Object Containment and Setting and Getting Object Properties, in Chapter 1.

Setting Bean Properties at Design-Time

One of the features of any JavaBean component is that it can be manipulated interactively in a visual design tool (such as a commercial Java IDE) to set the initial property values when the application starts. Consult the IDE documentation for details on how to load third-party Bean components into the IDE.

You can also refer to the JClass and Your IDE chapter in the *JClass DesktopViews Installation Guide.*

Most IDEs list a component's properties in a property sheet or dialog. Simply find the property you want to set in this list and edit its value. Again, consult the IDE's documentation for complete details.

3.2   SimpleChart Bean Tutorial

This tutorial guides you through the development of an application that uses `SimpleChart` to chart the financial information of "Michelle's Microchips". It is a good starting point for learning basic JClass Chart features. To explore more advanced features of JClass Chart, however, we recommend that you use the MultiChart Bean.

The tutorial does not cover all of the properties available in `SimpleChart`. For a complete reference, see Bean Reference, in Chapter 4. The screen captures have all been taken from Sun's BeanBox and will differ slightly from your IDE's appearance.

3.2.1   Steps in this Tutorial

This tutorial has eight steps:

1. Create a new application in your IDE and add a container.
2. Put a `SimpleChart` object into the container.
3. Load the data for Michelle's Microchips.
4. Add a header, footer, and legend.
5. Add point labels to the X-axis.
6. Change the background color to white.
7. Set the chart type to bar, and add 3D effects.
8. Compile and run the application.

Step 1: Create the 'Michelle' Application

Create a new application in your IDE and add a container to hold a `SimpleChart` object. In most IDEs this will be a panel. See your IDE's documentation for instructions on creating a basic application and adding a container.

Step 2: Put a Chart Object into the Container

With the container displayed in design mode, click the SimpleChart icon and place a
SimpleChart object into the container's area. See your IDE's documentation for details
on placing objects into a container. The SimpleChart icon looks like this:

In your container object, you should now see a basic chart area with an X- and
Y-axis, like this:
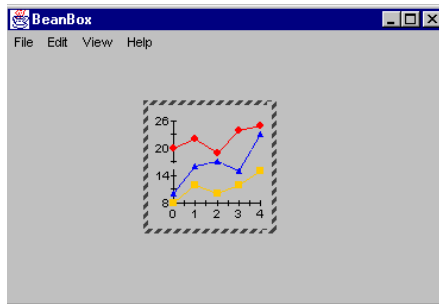
If you open your property list (the window that displays the Bean's properties) with the
SimpleChart area selected, you should see the property editors that are available in
SimpleChart.

Step 3: Load Data from a File

This tutorial uses data from a file named *chart2.dat* contained in the
JCLASS_HOME/examples/chart/intro/chart2.dat directory. To load *chart2.dat* into
SimpleChart, bring up the custom data source editor by clicking on the data property:

The data source editor provides two methods for loading data: editing data in the text
area, or loading data from a file. For Michelle's Microchips, click the **Load data from a**

file radio button. Then, enter the full path name of *chart2.dat* in the **File Location** field. After you click **Done**, you should see the data displayed in the chart area as follows:
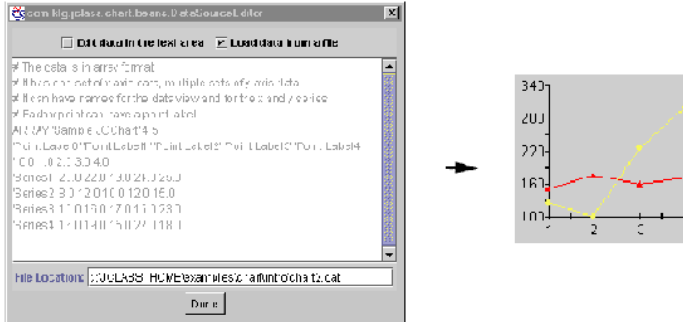
What's in chart2.dat?

chart2.dat has financial information for Michelle's Microchips, formatted for the file data source method of data loading. SimpleChart accepts only *.dat* files, or modifications to the default data in the editor. For more information on creating a file data source, see Loading Data from a File, in Chapter 8.

The content of *chart2.dat* is:

```
ARRAY '' 2 4
'Q1' 'Q2' 'Q3' 'Q4'
'' 1.0  2.0  3.0  4.0
'Expenses' 150.0 175.0 160.0 170.0
'Revenue' 125.0 100.0 225.0 300.0
```

JClass Chart also has other Beans which allow you to chart data from a database easily. See Bean Reference, in Chapter 4, for more information.

Step 4: Add a Header, Footer, and Legend

Enter "Michelle's Microchips" in the headerText property editor and "1963 Quarterly Results" in the footerText property editor:
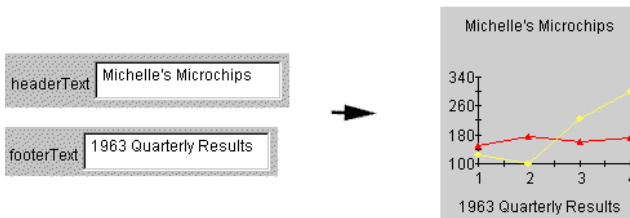
To add the legend, set the `legendVisible` property to `true`. The legend text is taken from information in the data source. Notice how the plot area is resized to accommodate the legend. You may have to resize your chart area to accommodate the changes:
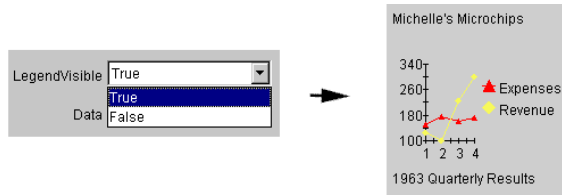
For more information on legend properties, see Legends, in Chapter 4.

Step 5: Add Point Labels to the X-axis
By default, `SimpleChart` annotates the axes with values. You can change the annotation to show point labels or time labels.

For Michelle's Microchips, change the X-axis annotation from values to point labels. Do this by setting the `xAxisAnnotationMethod` property to `Point_Labels`:
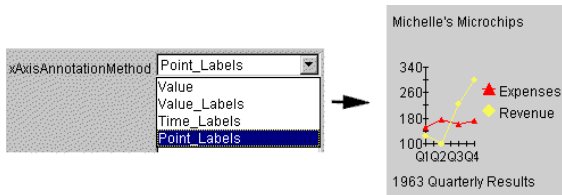
You should now see "Q1", "Q2", "Q3", and "Q4" on the X-axis. These labels are contained in the *chart2.dat* file, and come up automatically when `Point_Labels` is selected. For more information on axis annotation, see Axis Properties, in Chapter 4.

Step 6: Change the Background Color
To change the background color to white, click the `background` property to bring up your color editor:

The custom color editor used by your IDE will differ from the BeanBox. Select pure white from the options on your color editor:
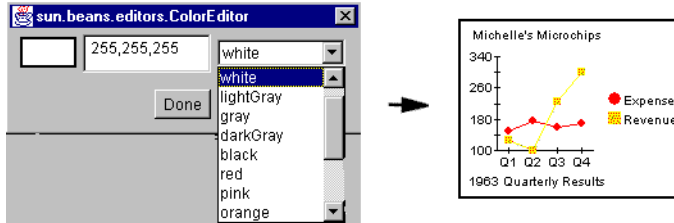
Step 7: Change to Bar Chart and add 3D Effects

You can select from 13 chart types using the `chartType` property editor (see Chart Types, in Chapter 4, for a complete list). For Michelle's Microchips, select the `BAR` type:
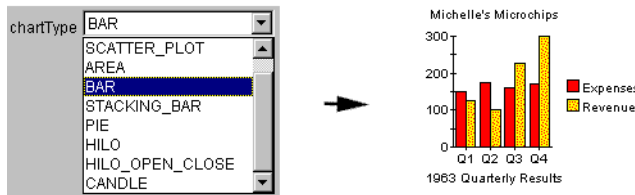
To add three-dimensional visuals to your chart, click the `view3D` property to bring up the **View3DEditor**:

There are two main settings in the **View3DEditor** (below): depth, and combined elevation and rotation. They are both set either by dragging the box in the editor with a mouse or by typing in the value in the editable box next to these settings.

First, drag the square with your mouse until you have an Elevation of 45 and a Rotation of 45, or simply type "45" in the editable box next to these settings. Second, check the

Change Depth box, and drag the red square until it has a depth of 31, or simply type "31" in the editable box next to Depth. Click **Done** to set the changes:
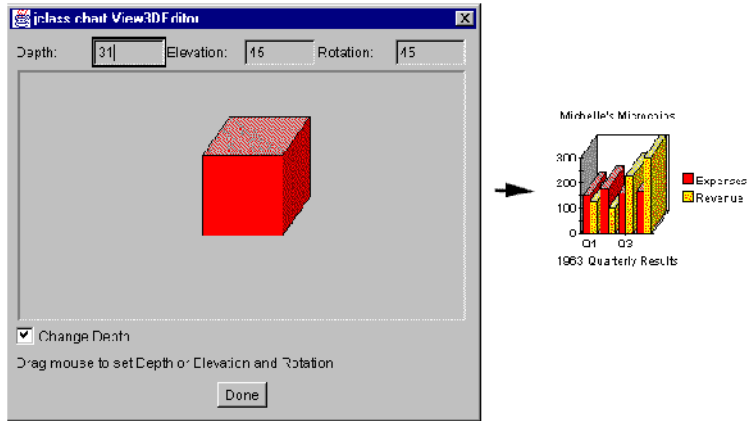
Step 8: Compile and Run the Application

For the last step, compile and run the application. See your IDE's documentation for details. When you run the application, you should have a window with a chart, displaying Michelle's Microchips' financial information.

The following example illustrates how the application appears when run:

4
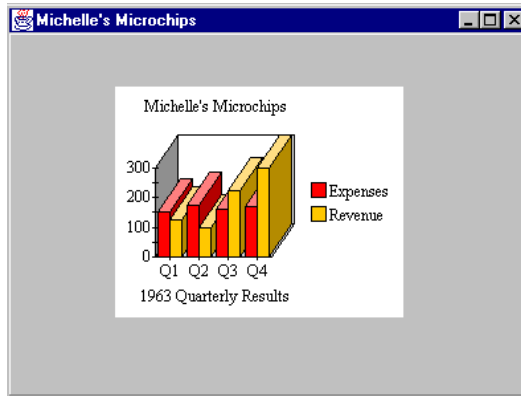
Bean Reference

Choosing the Right Bean ■ *Standard Bean Properties* ■ *Data-Loading Methods*

This chapter is a reference for JClass Chart Beans and their properties. For basic Bean concepts and a tutorial, see the SimpleChart Bean Tutorial, in Chapter 3.

4.1 Choosing the Right Bean

When creating new applications in an IDE, you can use `MultiChart`, `SimpleChart`, or one of the data-binding Beans. Unless you are binding to a database, we recommend using `MultiChart`, both for learning JClass Chart's features and creating new applications.

The MultiChart Bean
`MultiChart` is JClass Chart's most powerful Bean. It contains a richer set of features than previous Beans, highlighting the superiority of JClass Chart as a charting application tool. Among its features are the ability to handle multiple data sources and multiple axes. For more information, see MultiChart, in Chapter 5.

SimpleChart
`SimpleChart` was designed for quick chart development in any IDE environment. It exposes the most commonly used charting properties, and presents them in easy-to-use property editors. `SimpleChart` can load data from a file or a design-time editor.

`SimpleChart` and the data- binding Beans share a common set of properties that are covered in this chapter. `SimpleChart` and the data-binding Beans only differ in how they load data. Therefore, this chapter is divided into Standard Bean Properties and Data-Loading Methods.

Data-Binding Beans
If you want to load data from a database, you will have to use one of the data-binding Beans. In order to chart data from a database, your application must be able to establish a connection, perform necessary queries on the data, and then put the data into a chartable format. This type of database connectivity is often called 'data binding'.

There are data-binding Beans for JBuilder and for JClass DataSource.

Once you have set up your data handling for a specific Bean, you can then use the Standard Bean Properties to customize your chart.

4.1.1 JClass Chart Beans

The following table shows all of the available JClass Beans and their uses:

JClass Chart Bean	Description
MultiChart	The most powerful charting Bean. ■ Charts data from two data sources and plots them against multiple axes. ■ Data sources can be a file, or data entered at design-time. Also supports using Swing `TableModel` objects as data sources. ■ Compatible with all IDEs. See MultiChart, in Chapter 5, for complete details.
SimpleChart	Charts data from a file or data entered at design-time. Also supports a Swing `TableModel` object as a data source. Compatible with all IDEs.
DsdbChart	Binds a chart to JClass DataSource and charts data from a database. Compatible with all IDEs and the BeanBox (requires JClass DataSource Component).
JBdbChart	Binds a chart to a JBuilder DataSet and charts data from a database (requires Borland JBuilder 3.0+).

4.1.2 JClass Chart Beans and JCChart

All JClass Chart Beans are subclasses of the main chart object, `JCChart`. This means that the entire JClass Chart API is available to any developer using any of the Beans.

4.2 Standard Bean Properties

`SimpleChart` and the data-binding Beans (`VBdbChart`, `JBdbChart`, and `DSdbChart`) have a set of standard properties that allow you to control the appearance and behavior of your charts.

They only differ in the way they retrieve data. This section covers the standard properties. See Data-Loading Methods, in Chapter 4, for information on data management properties for the different Beans.

4.2.1 Axis Properties

JClass Chart Beans set up basic axis properties for you automatically, and adjust these properties to your data. You can also customize your axes with the axes property editors. You have control over the following axis properties:

■ Axis Titles

- Annotation Method
- Axis Number Intervals
- Axis Range
- Axis Grids
- Axis Hiding
- Logarithmic Notation
- Axis Orientation

Axis Titles

Enter X- and Y-axis titles in the `xAxisTitleText` and `yAxisTitleText` property editors:
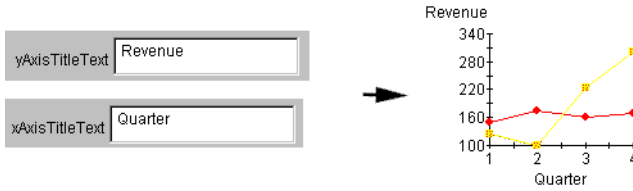
Annotation Method

Set the annotation method for the axes using the `xAnnotationMethod` and `yAnnotationMethod` editors. By default, `Value` annotation is used for both:
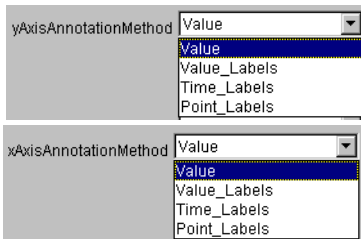
`Value_Labels` notation can only be added programmatically or by using HTML parameters; therefore, it is not very useful for Bean programming. The following examples show the three applicable annotation methods as applied to the X-axis:
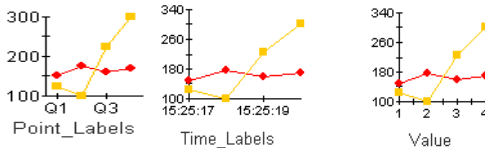
Axis Number Intervals

To specify the number interval on the axes, enter the interval into the `yAxisNumSpacing` or `xAxisNumSpacing` property editors:

Axis Range

The axis number range is determined by the minimum and maximum values of the axes. By default, these values are set automatically, based on the available data. You can specify

the range by using the `xAxisMinMax` and `yAxisMinMax` property editors. Enter the minimum value on the left of the comma, and the maximum on the right:

Logarithmic Notation
You can specify that one or both of the axes are logarithmic by setting the `xAxisLogarithmic` or `yAxisLogarithmic` properties to `true`:
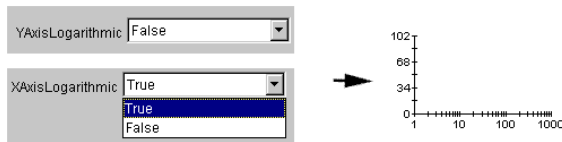
Hiding Axes
By default, both the X- and Y-axes are displayed. You can hide them by setting the `xAxisVisible` or `yAxisVisible` properties to `false`. The following example hides the Y-axis:
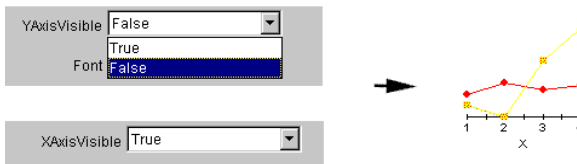
Showing Grids
Display gridlines for one or both axes by setting the `xAxisGridVisible` or `yAxisGridVisible` properties to `true`. By default, the grids are hidden. The following example sets both axes to display gridlines:

Axis Orientation
Axis orientation determines how the axes are positioned on the chart. By default, the axes are positioned with the Y-axis left/vertical and the X-axis right/horizontal. Use the axis

orientation custom editor to change how your axes are oriented. To launch the custom editor, click the `axisOrientation` property:

The axis orientation editor will illustrate the eight combinations. Select the desired orientation and click **Done**.
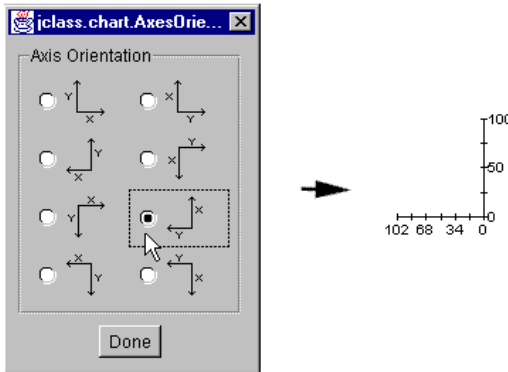
4.2.2   Chart Types

By default, JClass Chart Beans use the Plot chart type to display data. To change to another type, use the `chartType` property editor. The following example selects the PIE type:
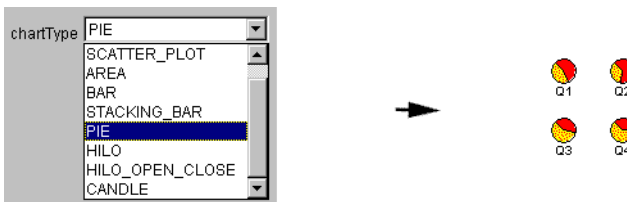
Data Interpretation
The following examples show how data is displayed by the different chart types:

| Area | Bar | Candle | HiLo |

Hilo_Open_Close **Pie** **Plot** **Scatter_Plot**

Stacking_Area **Stacking_Bar** **Polar** **Area Radar**

Radar
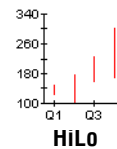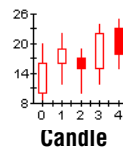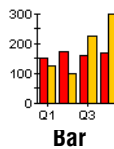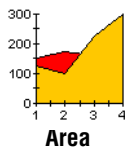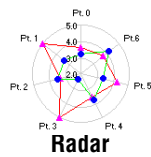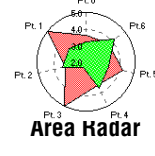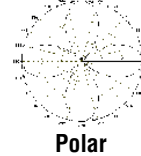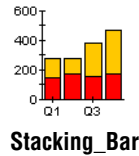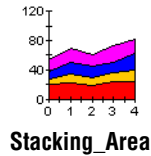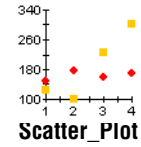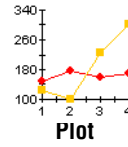
4.2.3   Display Properties

Font

Set the size and style of text on your chart by clicking the font property:

font Abcde...

The font you choose will apply to all text on the chart simultaneously with the exception of the header and footer. Note that the font editor that appears in your IDE may be different from the example below. The following example sets the font to **Courier, Bold, 24 point**, with the BeanBox font editor:
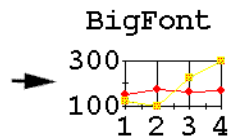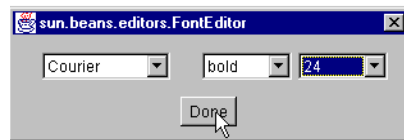
Note: The different font properties all work in the same way. Font affects all text on the chart area and legend. Header font affects the header, and Footer Font affects the footer.

Foreground and Background Colors

Click the `foreground` and `background` properties to set the foreground and background colors of your chart. A color editor will appear. By default, the colors are black foreground and light-gray background:
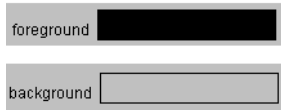
Most IDEs have their own color editors that differ from the BeanBox. The following example sets the background color to `red`:
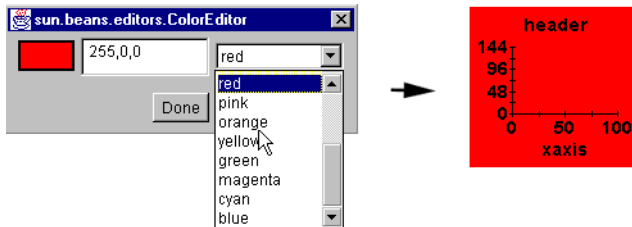
3D Effects

To add 3D effects to your chart, click the `View3D` property:

This will bring up the **View3DEditor**. There are two main settings in the **View3DEditor**: depth, and combined elevation and rotation.

You can add 3D effects either by typing a value in the editable box next to the Depth, Elevation, and Rotation settings, or by dragging the red square in the editor until it has the desired Elevation and Rotation. Then, check the **Change Depth** option box, and drag the red square until it has the Depth you want; alternatively, simply type in the value in the editable box next to this setting.

The degree of depth, elevation, and rotation is displayed in numbers at the top of the editor. Click **Done** to set the changes:
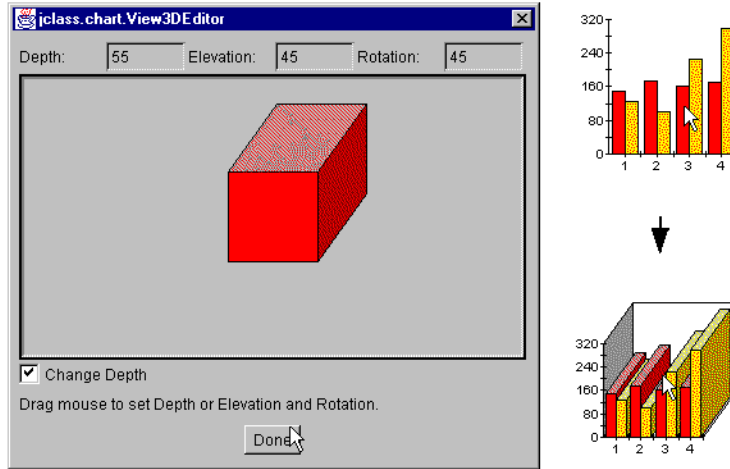
4.2.4   Headers and Footers

Add a header, footer, or both with the `headerText` and `footerText` property editors. The following example sets both:
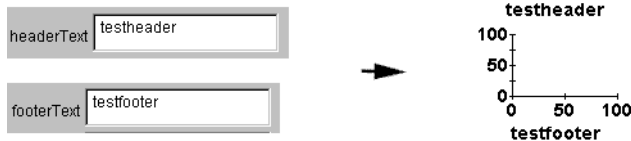
The font characteristics of the header and footer are determined by the Header Font and Footer Font properties. See Section 4.2.3, Display Properties, for more details.

4.2.5   Legends

You can add a legend, position it, and select its layout. The legend is set up from information in the data source. For information on how to set up legend items in the data source, see Data Formats, in Chapter 8.

Showing the Legend

To show the legend, set the legendVisible property to true:
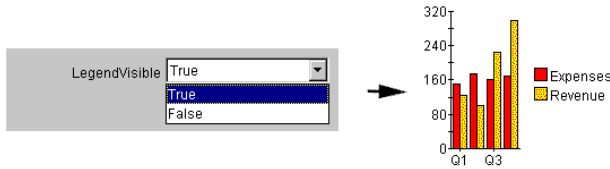
Legend Placement

Specify where the legend will be anchored in the chart area by selecting a compass direction from the legendAnchor property options. By default, legends are anchored on the East. The following example anchors the legend North:
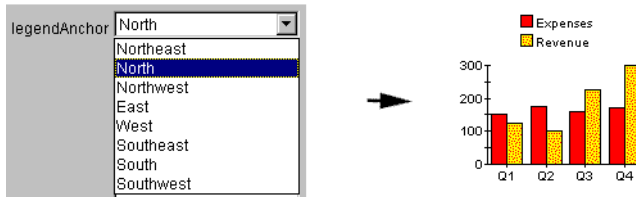
Legend Layout

Legend items can be laid out vertically or horizontally. By default the legend has a vertical layout. To specify a horizontal layout, set the legendOrientation property to Horizontal:
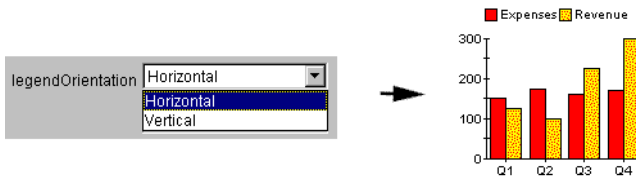
4.3   Data-Loading Methods

This section covers the data-loading methods of SimpleChart and the data-binding Beans. For MultiChart data-loading details, see MultiChart, in Chapter 5. Select the Bean that

best matches your data needs and follow the instructions on loading the data for that Bean:

JClass Chart Bean	Data Source & IDE Compatibility
SimpleChart	■ Formatted file or design-time editor. ■ Also supports using a Swing `TableModel` object as the data source. ■ All IDEs.
DSdbChart	■ Data binding. ■ All IDEs (requires JClass DataSource component).
JBdbChart	■ Data binding. ■ Borland JBuilder 3.0+.

If you are using an IDE other than Borland JBuilder, and you want to connect to a database, you will have to use JClass DataSource (see below). JBuilder users may still want to use the JClass DataSource for data-binding instead of their IDE-specific solutions.

JClass DataSource

JClass DataSource is a full data-binding solution. It is a robust, hierarchical, multiple-platform data source that you can use to bind and query any JDBC-compatible database. It can also bind to platform-specific data solutions in JBuilder.

JClass DataSource is available only in the JClass DesktopViews suite (which also contains JClass Chart, JClass Chart 3D, JClass Elements, JClass Field, JClass HiGrid, JClass JarMaster, JClass LiveTable, and JClass PageLayout). Visit *http://www.quest.com* for information and downloads.

4.3.1   SimpleChart: Loading Data from a File

There are two ways of loading data with the `SimpleChart` Bean: from a *.dat* file, or by entering data directly into the custom editor. Both methods are managed by the **DataSourceEditor**. To bring up the **DataSourceEditor**, click on the `data` property:

data

The DataSource Editor will appear (see below).

Loading Data from a *.dat* File

To load data from a file, click **Load data from a file**, enter the name of the file in the **File Location** field, and click **Done**:
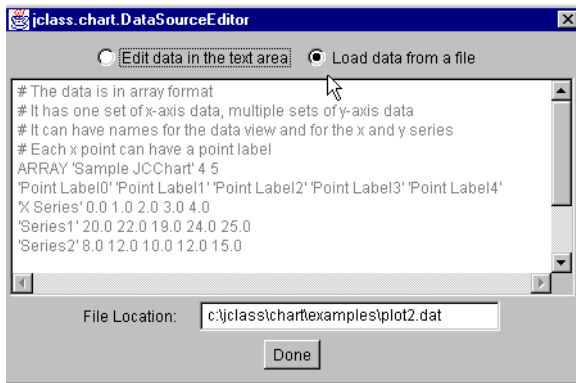
Specify the full path of the file. The file must be pre-formatted to the JClass Chart Standard (see Data Sources). Sample data files are located in the *JCLASS_HOME/examples/intro/chart2.dat* directory.

Editing the Default Data

You can use the data provided in the editor as is, or you can modify it. To use existing data, just check the **Edit data in the text area** radio button, and click **Done**. Change data by deleting and inserting text in the area provided. Be careful to preserve the punctuation surrounding the original text:
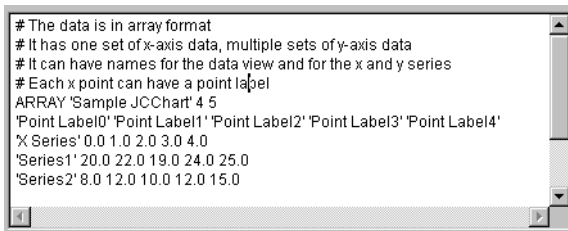
The chart below shows how the default data appears as a plot. Notice where the different elements are positioned. Each point on the X-axis is labelled with the names specified in the default data. The name of each series of y-values appears in the legend. The name of the data view is positioned directly above the legend.

In order for the default data to display this way, you must first set the xAxisAnnotation property to Point_Labels, and the legendVisible property to true.
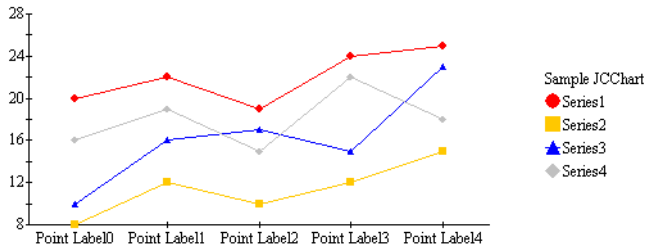
4.3.2　SimpleChart: Using Swing TableModel Data Objects

Your (Swing) application may have the data you want to chart contained in a Swing TableModel-type data object. You can use this object as your data source instead of using the JClass Chart built-in data sources if your IDE supports a TableModel editor.

Use the SwingDataModel property to specify an already-created Swing TableModel object to use as the chart's data source.

4.3.3　Data Binding in Borland JBuilder

Binding a chart to a database in JBuilder involves adding a database connection and query functionality with JBuilder Components, and then using JBdbChart to connect to the dataset and chart the data. This section walks through these steps.

Database connection and querying are handled by JBuilder components. Our coverage of these components is only intended as a guide. For detailed information on JBuilder database connectivity, consult your JBuilder documentation.

Before proceeding, make sure you have:

■ Borland JBuilder 3.0+.

■ JBdbChart Bean loaded in your JBuilder Palette. For details on how to load a Bean, see the *JClass DesktopViews Installation Guide* (available in HTML and PDF formats) or your JBuilder documentation.

■ Database set up properly.

■ Basic SQL command knowledge.

Step 1: Connect to a Database

Use JBuilder's `Database` Bean to add a database connection. The icon is found under the **Data Express** tab.

Add an instance to your frame. Then, use the `connection` property to specify the URL of the database that you want to use.

Step 2: Query the Data

To query the database, add an instance of JBuilder's `QueryDataSet` to your frame. This Bean is found under the **Data Express** tab.

Select columns that you may want to chart with the `query` property editor. Each column will represent a series of data, or point labels. For example, to select all of the columns from a table named MotorVehicle_Sales, you would type a statement similar to:

```
select * from MotorVehicle_Sales
```

You can include all columns at this step, and then use JBdbChart to choose which ones to display later.

Step 3: Connect the Chart to the DataSet

With the database connection established and the query created, you can now use `JBdbChart` to connect to the JBuilder DataSet and chart the data. `JBdbChart`'s data binding properties are `dataSet`, and `DataBindingMetaData`.

Insert a `JBdbChart` into your frame.

Select a query from the `dataSet` property's pull down menu. If the database connection and query are set up properly with JBuilder components, there should be one or more queries in the list.
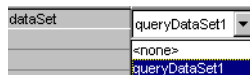
You can now select the columns and range of data that will be displayed. Columns that contain numeric data are considered 'data series', and can be plotted on a chart. Columns

that have non-numeric data can be used for point labels on the X-axis. Click the `dataBindingConfig` property to bring up the custom editor:

This editor allows you to set the columns and the data range of the chart. Click on column names to select them (when they are highlighted, they are selected).
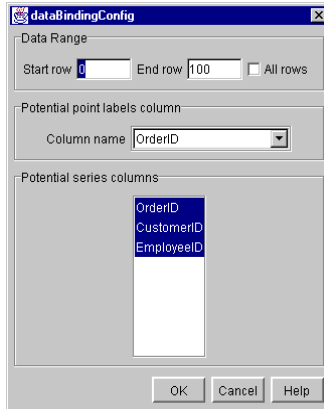
The **Potential series columns** are numeric. The **Potential point label column** is non-numeric.

You can either set the range to all data by checking the **All rows** box, or you can specify a range using the **Start row** and **End Row** fields.

In order to display the point labels on the X-axis, you have to set the `xAxisAnnotationMethod` property to `Point_Labels`. For more information, see Section 4.2.1, Axis Properties.

You should see your data in the design frame:
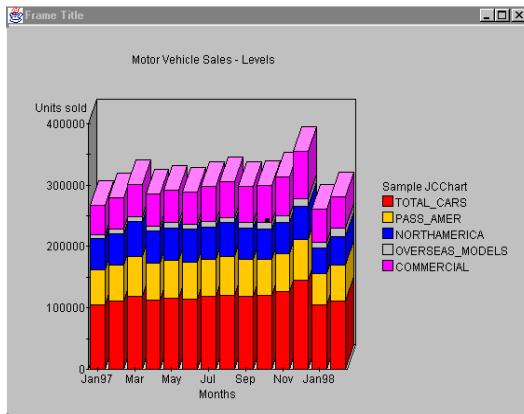
With your connection established, you can then use the Standard Bean Properties to customize and enhance your chart. In the example above, a header, footer, axis title, legend, point labels, and 3D effects have been added.

4.3.4   Data Binding with JClass DataSource

The JClass DataSource manages all connection and query functionality for data binding. After establishing a connection and query with JClass DataSource, you then bind DSdbChart to JClass DataSource to chart the data.

The JClass DataSource package contains a number of Beans used for binding to databases, including JCTreeData and JCData. This section will illustrate the process with the JCData Bean. DSdbChart uses the same method to connect to either Bean. Consult your JClass DataSource documentation for details on their features and how to use them.

To use this solution, you require the following:

■   Sun's BeanBox or any IDE.

■   JClass DataSource (available only in JClass DesktopViews. Visit *http://www.quest.com* for information and downloads).

■   DSdbChart loaded into the BeanBox or IDE. For details on how to load a Bean, see the *JClass DesktopViews Installation Guide* (available in HTML and PDF formats) or your JClass DataSource documentation.

■   If you are using Windows, you will need to establish an ODBC database connection. Set this in **Control Panel > ODBC**. If you are using Windows 2000, establish an ODBC database connection via **Control Panel > Administrative Tools > Data Sources** (**ODBC**). For more information on running JClass DataSource examples, please see the *readme* file.

The following steps guide you through using `DSdbChart` to connect to JClass DataSource. They are: connect to a database, query the data, and connect `DSdbChart` to the JClass DataSource.
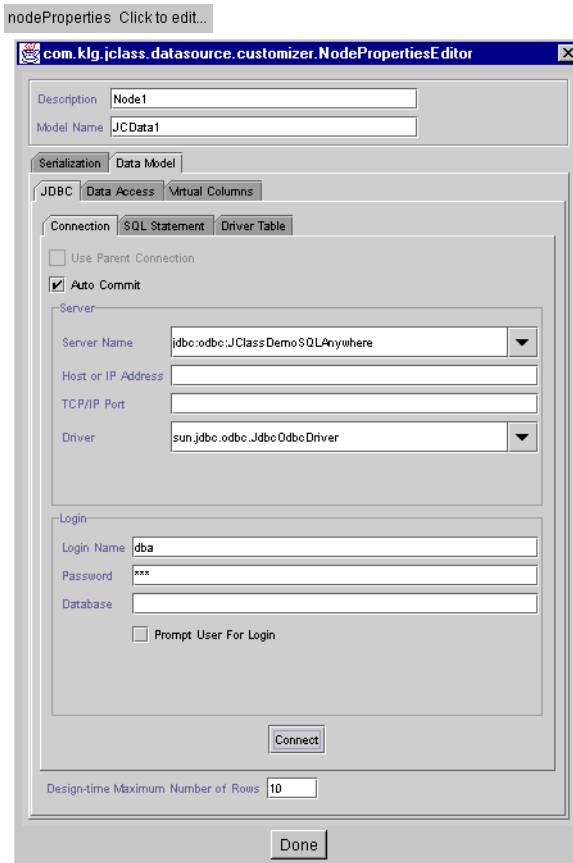
Step 1: Connect to a Database

Add a `JCData` instance to your design area. The icon looks like this:

Click the `nodeProperties` property to bring up the NodePropertiesEditor.

This editor manages all of the connection and query settings. The first thing you have to do is set up a serialization file under the **Serialization** tab. This file saves information and settings about the connection. You can then proceed to set up a connection and query.

To set up a database connection, go to the **DataModel > JDBC > Connection** tab, and specify the *Server Name* and *Driver* for the database you want to connect to. Test the connection. If there are error messages, consult your JClass DataSource documentation.

When your connection is successful, you can then proceed to set up a query.

Step 2: Query the Data

Click the **Data Model > JDBC > SQL Statement** tab to show the query options:

You can create your whole SQL query using mouse clicks. First, add a table, and then create a query by selecting columns. When you are all finished, click **Set/Modify**, and then **Done**.

Step 3: Connect a Chart to JClass DataSource

With your database connection established, you can then bind a chart to the data. This is done using the `dataBinding` and `DataBindingMetaData` property editors.

First, add `DSdbChart` to your design area. The icon looks like this:

Click the `dataBinding` property to bring up the DataBindingEditor.

If the connection in JClass DataSource is properly established, you should see one or more data sources to select from:
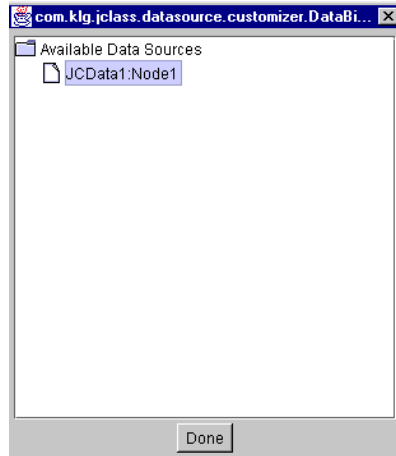
Select a source and click **Done**.

You can now select the columns and range of rows to be displayed in the chart. To do this, click the `DataBindingConfig` property to bring up the `DataBindingConfig` custom editor:
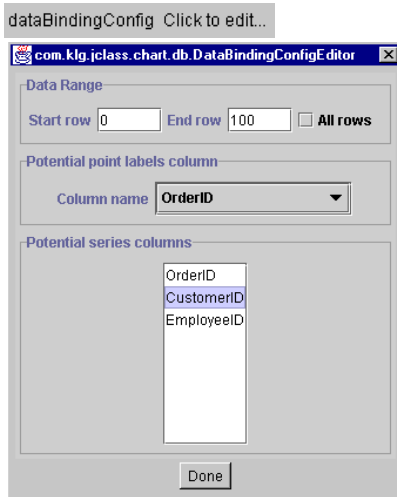
There are two lists of columns:

■ a *Potential point labels column* – a combo box containing the columns that can be used for the X-axis point labels.

■ a *Potential series column* – a list comprising the numeric columns that can be used as the Y-series.

In order to display the point labels on the X-axis, set the `xAxisAnnotationMethod` property to `Point_Labels`. For more information, see Section 4.2.1, Axis Properties.

You can either set the range to all data by checking the **All rows** box, or you can specify a range using the **Start Point** and **End Point** fields.

When you click **Done**, you should see the data displayed in the design area of the Beanbox or IDE. Your data binding is complete.

5

MultiChart

Introduction to MultiChart ■ *Getting Started with MultiChart* ■ *MultiChart Property Reference*

5.1 Introduction to MultiChart

`MultiChart` is the next generation charting Bean from JClass Chart. It contains a richer set of features than previous Beans, highlighting the superiority of JClass Chart as a charting application tool.

The `MultiChart` icon:

Highlights of the MultiChart Bean

■ Handles multiple data sources.

■ Plots data against multiple X- and Y-axes.

■ Fully customizable axes.

■ Extensive control of font, colors, borders, and styles for each chart element.

5.1.1　Multiple Axes

`MultiChart` can have two X- and two Y-axes, as in the example below:
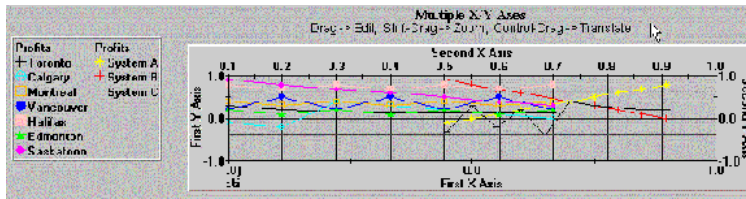
Setting Properties on Multiple Axes

Axis properties can be set for each axis individually. At the top of each axis editor you will see four radio buttons:

When a radio button is selected, all that follows below will apply to that axis.

5.1.2　Multiple Data Views

`MultiChart` allows you to load data from two different sources at the same time. When loading data from two different sources, they are each assigned to a separate data view.

By default, both data views are showing, but you can hide or reveal data views depending on your application's needs. Both sets of data can be mapped to the same set of X- and Y-axes, or, mapped to different axes.

Note: Radar, area radar, and pie charts do not support multiple data views. For more information on Data Views, see Data Views, in Chapter 8.

5.1.3　Intelligent Defaults

`MultiChart` has a sophisticated set of dynamic default settings in the custom property editors. You can override these defaults to suit your needs. When you override a default value in a text editor, it becomes static, and will not automatically adjust anymore.

Returning to Default Values

If you want to return to default settings in the custom editors after overriding them, all you have to do is delete the contents of the changed field, and leave it blank. The next time you bring the editor you will see that the automatic values have returned.

5.2   Getting Started with MultiChart

`MultiChart` has a sophisticated set of dynamic default settings that adjust to your data and other settings. This means that you only have to make a minimum of settings to have a respectable chart. The following list describes the most common start-up tasks and the editors used for them:

- **Load Data.** To load data in the chart, use the DataSource editor. This editor allows you to load data from one or two sources. There is also a default set of data built-in that you can use to experiment with. Alternately, you can use a Swing `TableModel` data object as the chart's data source using the `SwingDataModel` property.

- **Select Chart Types.** For each data view, you can select a chart type and the axes that the data will be plotted against with the DataChart editor.

- **Set BackGround Color.** Use ChartAppearance to set the color of the chart background.

- **Set Axis Annotation.** By default, `MultiChart` uses values to annotate the axes. You can also use value labels, point labels, or time labels by setting the annotation type with the AxisAnnotation editor.

- **Add a Legend.** Add a legend by checking the Visible box in the LegendAppearance editor.

- **Add a Header and Footer.** To add a header, use HeaderText to add the text, and then select the Visible check box in HeaderAppearance. The footer is the same, but uses the FooterText, and FooterAppearance editors

- **Add Extra Axes.** By default a standard X-Y axis set is displayed. If you require, you can display a second X- or Y-axis. Display them with the AxisMisc editor's **Visible** property. Then use the many axis editors, such as AxisPlacement, to set up and align the axes.

5.3   MultiChart Property Reference

The following property reference section covers all of `MultiChart's` features.

5.3.1   Axis Controls

This group of editors sets up the axes. `MultiChart` has a sophisticated set of automatic default values that adjust to your data. This makes chart development fast and easy. `MultiChart` is also extremely flexible, and every aspect of the axes can be adjusted.

AxisAnnotation

With the `AxisAnnotation` editor, you can set the annotation type for each axis, and control how they look. Axis annotations are numbers or text that appear along the axes.

Options in the **Method** menu are: `Value`, `Time_Labels`, `Point_Labels`, and `Value_Labels`.
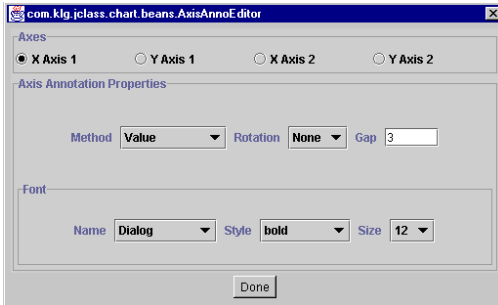
For each of the labelling methods, there is a corresponding editor that gives you more control over the behavior and appearance. For Value, use AxisScale, for `Point_Labels`, use AxisPointLabels, for `Time_Labels`, use AxisTimeLabels, and for `Value_Labels`, use AxisValueLabels.

The following examples illustrate the different label types:
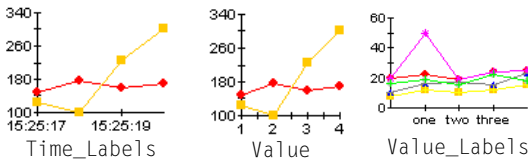
With the **Rotation** property, you can rotate the labels on the axis. The following example shows Value_Labels, rotated by 270 degrees and with bold, 12pt font:
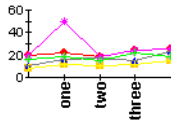
Gap controls the space between annotations, in pixels. If, for example, you used point labels, you could use the **Gap** property to make sure they have enough room to display properly.

AxisGrid

Use the `AxisGrid` editor to set up gridlines on each of the axes. There are also controls for color, line spacing, and line width of the gridlines.
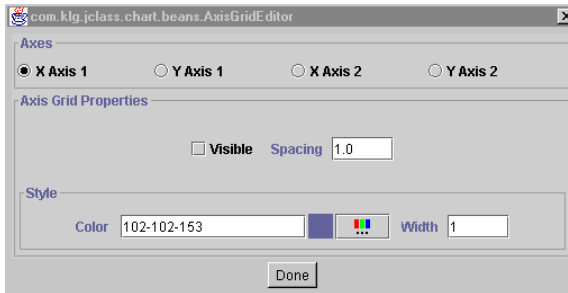
The following example sets X Axis 1 grid and Y Axis 1 grid to **Visible** with **Spacing** = 1 and **Width** = 1 for the X Axis, and with **Spacing** = 1 and **Width** = 10 for the Y Axis:
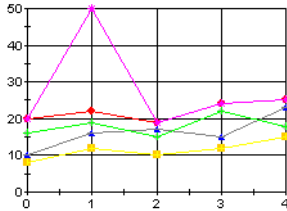
AxisOrigin

The `AxisOrigin` editor allows you to specify an origin by coordinates, or by choosing an option from a pull down menu. By default, axes origins are set automatically, based on the available data.

To place the origin, you can select one of the locations from the pull-down menu, such as Min or Max. If you want to set the origin to a specific value on the axis, select Value_Anchored from the menu and then enter the value in the **Origin** field:
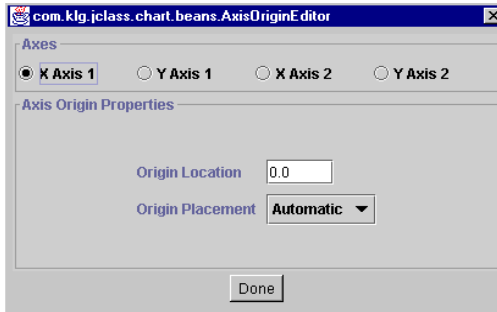
The following example anchors the origin of Y Axis 1 at 20 (default data):
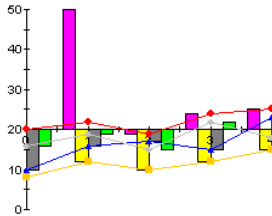
Note that, by default, X Axis 1 is placed at the origin of Y Axis 1. To override this default, use the AxisPlacement editor.

AxisPlacement

Axis placement determines the placement of an axis in relation to another. By default, this is set automatically by MultiChart, based on the given data. Sometimes, however, you may want to locate an axis in a different location.
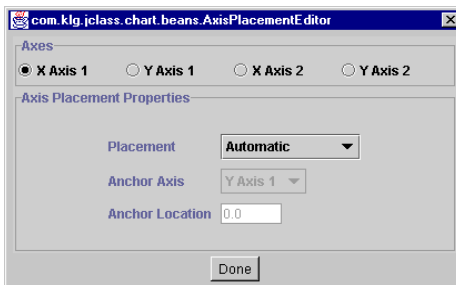
Using the **Placement** field, select the type of placement for the axis selected. Placement options include: Min, Max, Automatic. Origin, and Value_Anchored.

The **Axis** field selects the anchor-axis that you want to place the current axis against (for example, place X Axis 1 in relation to Y Axis 2). If you select None as an Axis, MultiChart will use the default axis.

To place the axis at a specific value along another axis, select Value_Anchored from the pull-down menu, and enter the value in the **Location** field.

The following example shows X Axis 1, with a **Placement** of Max in relation to Y Axis 1:
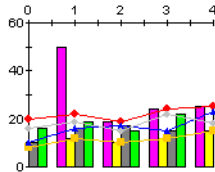
AxisMisc

Use AxisMisc to show or hide any of the axes. It also allows you to make any axis logarithmic. The Editable property, when selected, allows zooming, editing, and translation for the selected axis. For more information on interactive events, see Section 5.3.6, Event Controls.
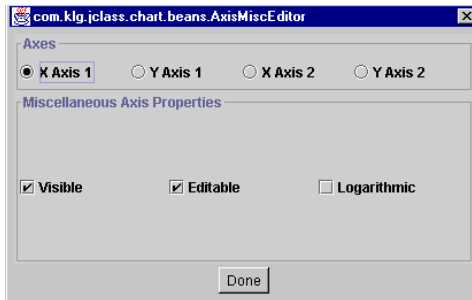
The following example hides X Axis 1 from view by deselecting Visible.
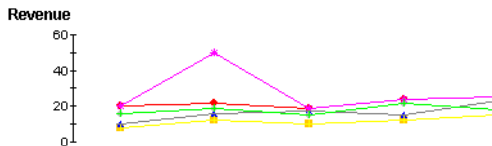
AxisPointLabels

Use the AxisPointLabels editor to create point labels (applies to X1 and X2 axes only). Point labels label specific points of data on the X-axes.

The editor reads data from the data source associated with the selected axis and provides a list of point labels. To change the text in these labels, change the text alongside the

point. Note that the format is "point value then comma then the name of the label". For example,

```
3.0, PointLabel3
```

In order for the labels to appear on the chart, you also have to set the annotation method to Point_Labels in the `AxisAnnotation` editor. See below for an example.
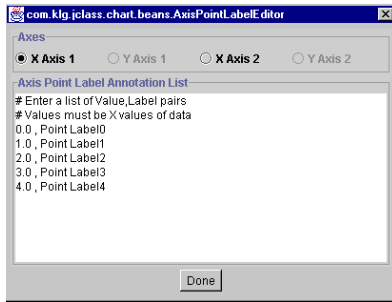
The following example shows how the default data's point labels appear on X Axis 1:
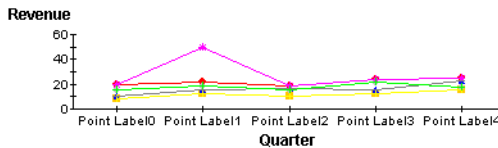
Note that if you are mapping multiple data sources against a single axis, then you will want to use value labels instead, as the `AxisPointLabels` editor only uses points from the first data source associated with the selected axis.

AxisRelationships

The `AxisRelationship` editor allows you to create a mathematical relationship between two axes. For example, if you want to create a thermometer chart with Celsius values on the left and the Fahrenheit values on the right, you could create a Celsius axis, and then base the Fahrenheit axis values on it.

There are three properties included in this calculation: **Originator**, **Multiplier**, and **Constant**. The calculation is based on the formula:

New Axis Value = Constant + Multiplier X Originator.
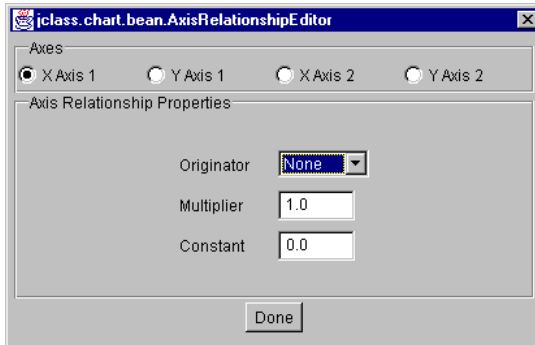
To use this editor, first click on the radio button next to the Axis that you want to alter. Next, select an axis from the **Originator** menu that your calculation will be based on, and then enter a value in the **Multiplier** field that represents the relationship. The **Constant** value is optional; its default value is 0.0.

AxisScale

The AxisScale editor controls the range on each axis, the interval of the numbering, and **Tick Spacing**. It is used primarily for the Value method of axis annotation (see the AxisAnnotation). **Precision** determines the numeric precision of the axis numbering.

The effect of Precision depends on whether it is positive or negative:

- *Positive* values add that number of places after the decimal place. For example, a value of 2 displays an annotation of 10 as "10.00".

- *Negative* values indicate the minimum number of zeros to use before the decimal place. For example, a value of –2 displays annotation in multiples of 100.

The default value of Precision is calculated from the data supplied.

The Min and Max fields determine the range of data that is displayed on the chart. There are intelligent defaults in this editor that adjust to your data and other chart settings. You can override these settings with the fields provided.
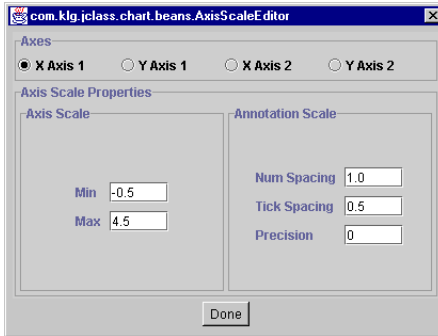
AxisTimeLabels

The `AxisTimeLabel` editor allows you to control how the time labels appear. When you select the annotation method with `AxisAnnotations`, you can select time labels, which represent the values on the axis as units of time.

Time Base determines the date and time that the labelling starts from (default is current time/date). **Time Unit** is the unit of time the labels use, such as year, month, day, minute, second, and so on. The default time unit is minutes.

Time Format allows you to customize the text in the time labels with a set of formatting codes. See Axis Labelling and Annotation Methods, in Chapter 7, for a list of these codes.
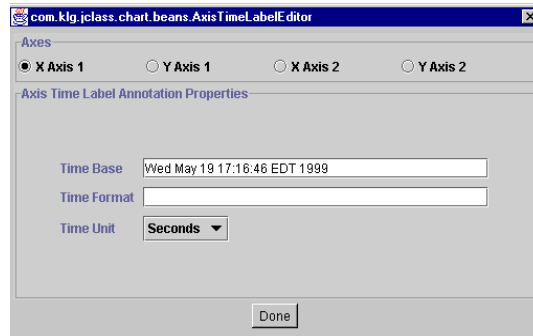
The following example uses time labelling on X Axis 1, with seconds as the time unit:
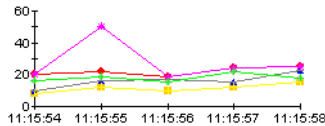
AxisTitle

Using the `AxisTitle` editor, you can add axis titles to each axis. There are also settings for the font, point, rotation, and placement of the title.
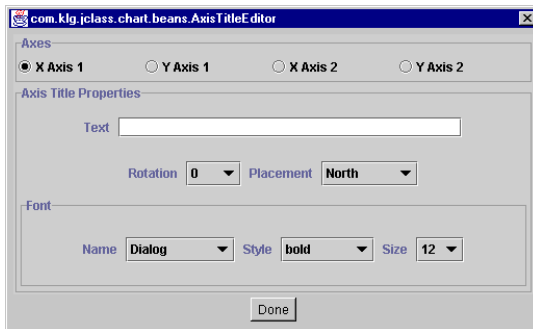
In the **Placement** field's pull-down menu are a list of compass directions for title placement. Not all options are available to X- and Y-axes. If you select a placement, and it returns to the previous selection, that placement is not available for that axis. The

following image shows the effects of adding titles to X Axis 1 and Y Axis 1, and setting the font to bold, with a size of 12:
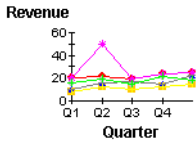
AxisValueLabels

Use the `AxisValueLabel` editor to enter value labels for the axes. Value labels appear along the axis at specified values. You also have to set the annotation method to **Value_Labels**, in the `AxisAnnotation` editor before the labels will display.
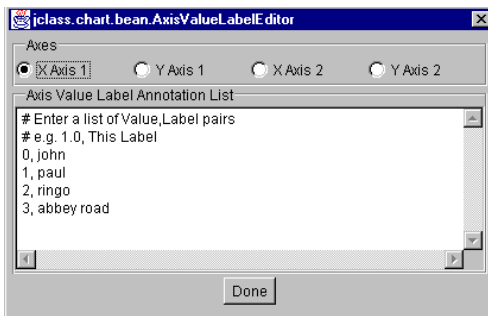
To add value labels, enter the value, followed by a comma and a label (see above). The following example shows how the labels in the editor above appear on X Axis 1.
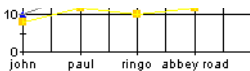
5.3.2   Headers, Footers, and Legends

FooterText

The `FooterText` editor allows you to enter text that will appear at the bottom of the chart area. You can also select a font, font style, and size for the footer.
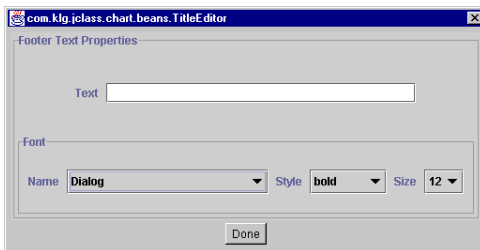
Note that the footer will not display unless you check the **Visible** box, in the FooterAppearance editor (this editor also controls footer opacity, background, and foreground).
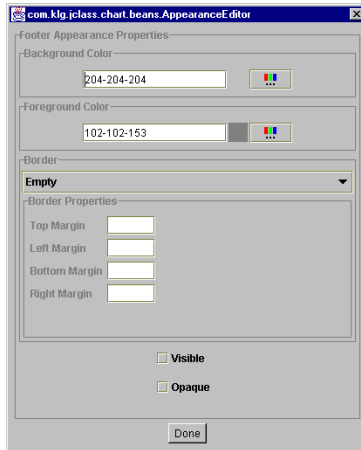
The following example shows how a 'pointless footer' appears on the chart area:
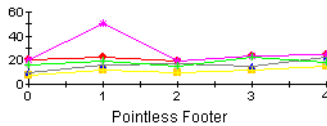
HeaderText

The HeaderText editor allows you to enter header text, that will appear at the top of the chart area. You can also select a font, font style, and size of the header.
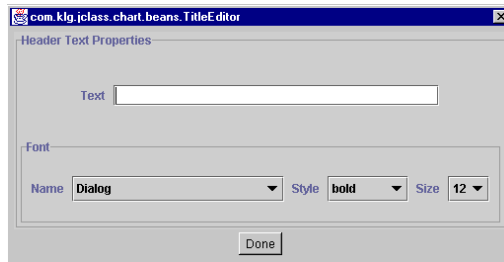
Note that the header will not display unless you check the **Visible** box, in the HeaderAppearance editor (which also controls header opacity, background and foreground).
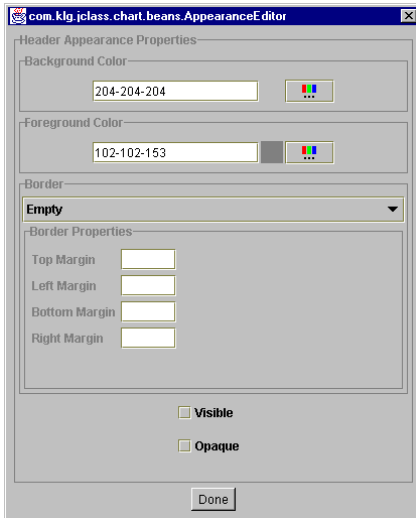
The following example shows how a 'pointless header' displays on the chart:
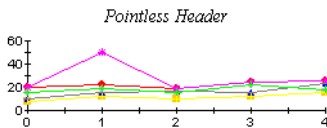
LegendLayout

The LegendLayout editor controls the layout of the legends. **Orientation** determines how the legend items are placed in the legend (either vertically or horizontally). The **Anchor** property positions the entire legend on the chart, based on compass directions.

In order for the legend to display on your chart, the **Visible** checkbox in the LegendAppearance editor must be selected.
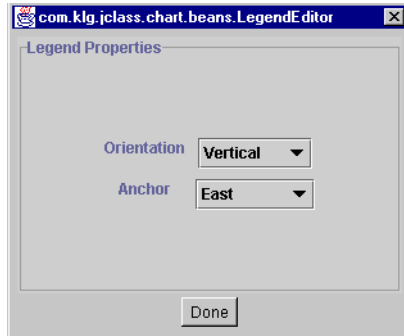
Below are two examples of legend layout:

The example on the left uses the default settings with **Anchor** = East and **Orientation** = Vertical. In the example on the right, **Anchor** = North and **Orientation** = Horizontal.

5.3.3   Data Source and Data View Controls

This group of editors manages the properties that control the data source, and the views on the data. MultiChart can load data from two different sources. Each of the data sources is assigned to a data view.

DataChart

The `DataChart` editor allows you to select the chart type of each data view, and which axes each data view will be mapped against.
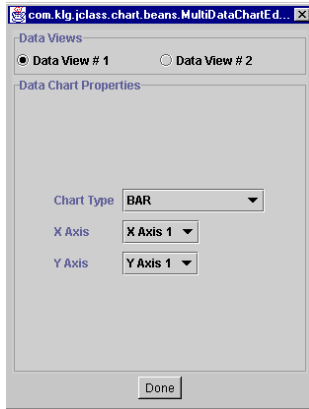
The ChartType property selects from the following chart types:
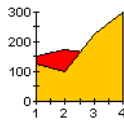
Area
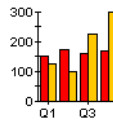
Bar
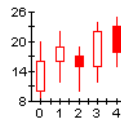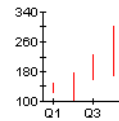
Candle

HiLo

Hilo_Open_Close

Pie
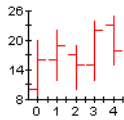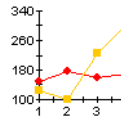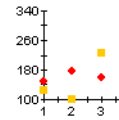
Plot

Scatter_Plot
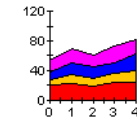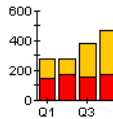
Stacking_Area

Stacking_Bar
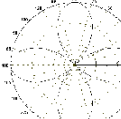
Polar
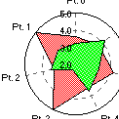
Area Radar

Radar

DataMisc

The `DataMisc` editor controls several aspects of the data views.
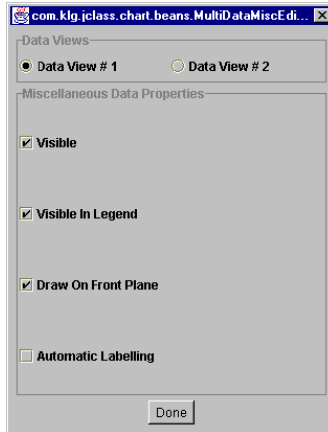
With the **Visible** property, you can show or hide each data view from the display area. **Visible In Legend** will show/hide a data view from the legend (but the data will still be charted).

Automatic Labelling attaches a dwell label to every data point in the chart. A dwell label is an interactive label that shows the value of a point, bar or slice, when a user's mouse moves over it. In the example below, '225' appears on top of the green bar as the cursor passes over it, indicating that the value of the bar is 225.
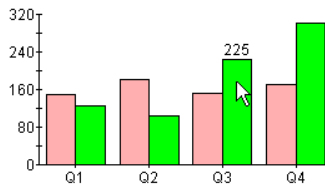
When **Draw on Front Plane** is selected, the data view will be mapped on the front plane of a three dimensional chart space. Applies only in cases where there are multiple data series, displayed on multiple axes, using 3D effects.

DataSource

There are three ways of loading data with the MultiChart Bean. Two are handled by this property: from a *.dat* file, or by entering data directly into the custom editor. Both methods are managed by the DataSource editor.

The third method is to use a Swing TableModel-type data object as a data source, instead of using the JClass Chart built-in data source. See SwingDataModel below for details.

The first step is to select a data view with one of the radio buttons. Then, follow the procedure below for each data view.

To load data from a file into a data view, click **Load data from a file**, enter the name of the file in the **File Location** field, and click **Done**:
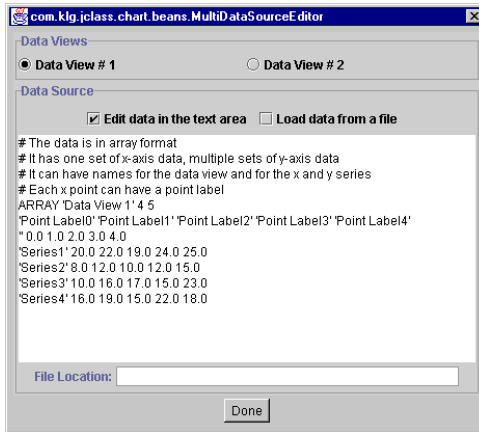
Specify the full path of the file. The file must be pre-formatted to the JClass Chart Standard (see Data Sources, in Chapter 8). Sample data files are located in the *JCLASS_HOME/jclass/chart/examples* directory.

You can use the data provided in the editor, as is, or you can modify it. To use existing data, just check the **Edit data in the text area** radio button, and click **Done**. Change data by deleting and inserting text in the area provided. Be careful to preserve the punctuation surrounding the original text:
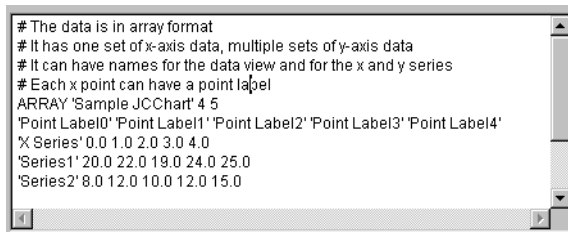
The chart below shows how the default data for Data View 1 appears as a plot. Notice where the different elements are positioned. Each point on the X-axis is labelled with the names specified in the default data. The name of each series of y-values appears in the legend. The name of the data view is positioned directly above the legend.

In order for the default data to display this way, you must first set the xAxisAnnotation property to Point_Labels, and the legendVisible property to true.
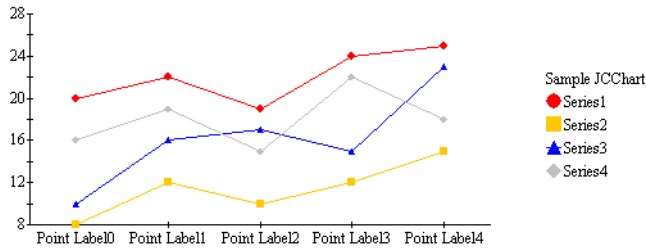
SwingDataModel

Instead of using the chart's internal data source, you can use a Swing TableModel-type data object that you have already created for your application, if your IDE supports an editor for TableModel. This saves reformatting your data to match the format used by JClass Chart.

Use the SwingDataModel1 property to specify an already-created Swing TableModel object to use as the data source for the first data view. Use SwingDataModel2 to specify a TableModel object to use for the chart's second data view.

5.3.4   Appearance Controls

This group of editors allows you to control the look of specific chart subcomponents. You can control font, borders, background, and foreground for the chart, chart area, plot area,

header, footer, and legend. The following diagram illustrates the different chart subcomponents.
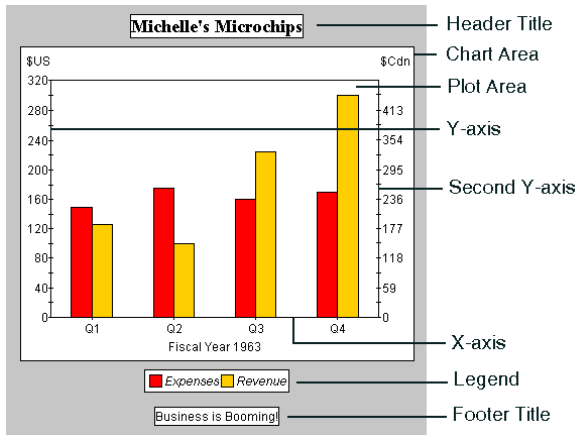
All of the editors have the same basic functionality that apply to a specific chart subcomponent, as follows:
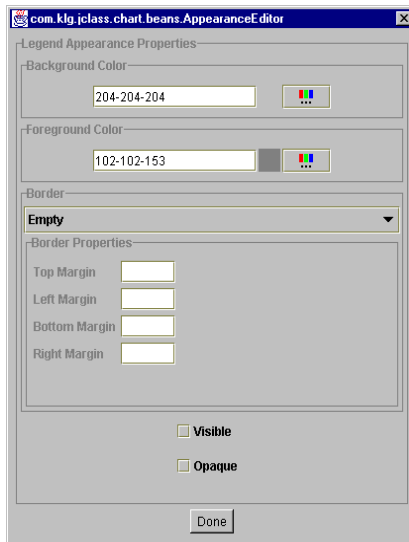
Small differences in each editor will be discussed below. Note that for most of the appearance editors, there are corresponding editors for controlling other properties of that chart element.

ChartAppearance
The `ChartAppearance` editor sets the foreground/background border, and opaque values for the chart. This editor affects the areas behind all other chart elements.

ChartAreaAppearance
The `ChartAreaAppearance` editor sets the foreground/background border, visible, and opacity values for the chart area (see diagram above).

FooterAppearance
The `FooterAppearance` editor sets the foreground/background border, visible, and opaque values for the footer. When **Visible** is checked, the footer will be displayed in the chart. By default the footer is not showing. The `FooterAppearance` editor works in conjunction with the FooterText editor, which is used to enter the footer text.

HeaderAppearance
The `HeaderAppearance` editor sets the foreground/background border, visible, and opaque values for the header. When **Visible** is checked, a header will be displayed in the chart. By default the header is not showing. This editor works in conjunction with the HeaderText editor.

LegendAppearance
The `LegendAppearance` editor sets the foreground/background border, visible, and opaque values for the legend and determines if it is displayed. By default, the legend will not appear. When **Visible** is checked, a legend will be displayed in the chart.

The content of the legend comes from the information in the data source. In order to change the contents of the legend, you have to change what is in the data source. For information on how to set up legend items in the data source, see Data Formats, in Chapter 8.

Other legend settings are found in the LegendLayout editor.

PlotAreaAppearance
The `PlotAreaAppearance` editor sets the foreground and background for the plot area, and allows you to add an **Axis Bounding Box**. A bounding box is a graphical feature that closes off the axes, thus forming a square.

Font
The `Font` editor sets the font defaults for your chart.

The font you choose will apply to all text on the chart simultaneously. The following example sets the font to **Courier, Bold, 24 point:**
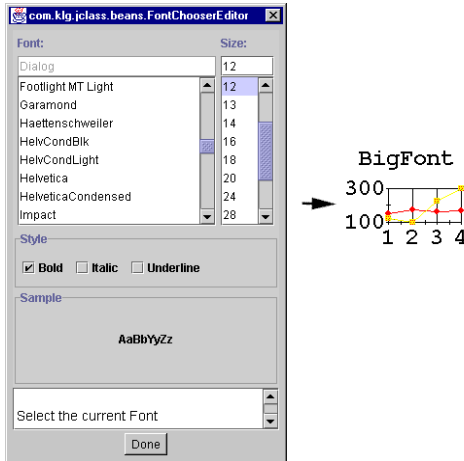
This font editor sets up a default font for the chart (not including the header and footer). You can, however, change font for selected elements using custom editors for each property. For example, the HeaderText, FooterText, and AxisAnnotation editors allow you to override the default font settings.

5.3.5 View3D

To add 3D effects to your chart, click the `View3D` property.

First drag the red square in the editor until it has the desired Elevation and Rotation. Then, check the **Change Depth** option box, and drag the red square until it has the

Depth you want to see on your chart. The degree of depth, elevation and rotation is displayed in numbers at the top of the editor. Click **Done** to set the changes:
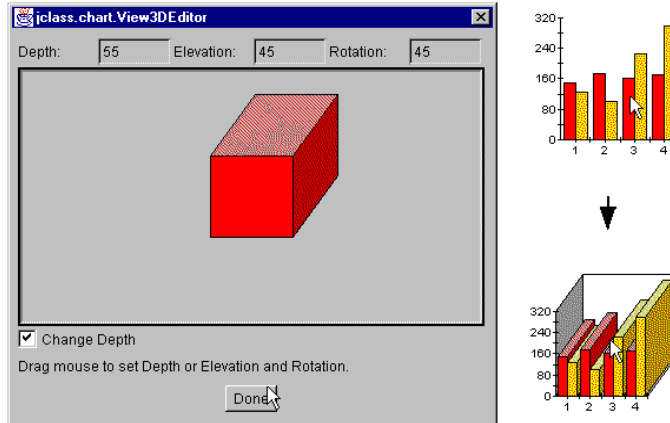
5.3.6   Event Controls

TriggerList

The TriggerList editor sets up what user events the chart will handle, either from a mouse, or mouse-keyboard combination.
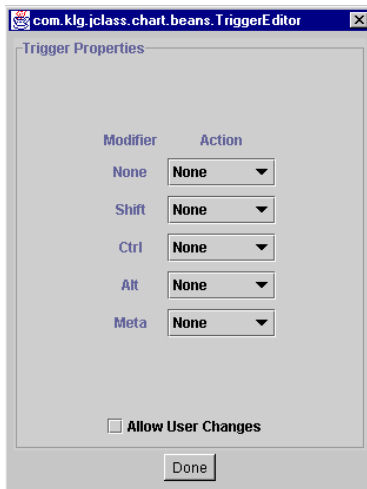
Actions are the available event mechanisms, such as Zoom, Rotate, Depth, Customize, Pick, and Translate. By setting up these triggers, the end-user can examine data more closely or visually isolate part of the chart. The following list describes these interactions:

- ■ **Translate** allows moving of the chart.
- ■ **Zoom** allows zooming into or out from the chart.
- ■ **Rotate** allows rotation (only for bar or pie charts displaying a 3D effect).
- ■ **Depth** allows adding depth cues to the chart (only for bar or pie charts displaying a 3D effect).
- ■ **Customize** allows the user to launch the chart Customizer. To use this feature, you must also check the **Allow User Changes** box.
- ■ **Pick** allows you to set up pick events. The `pick` method is used to retrieve an *x,y* coordinate in a Chart from user input and then translate that into the data point nearest to it. This feature requires some non-bean programming. See Using Pick and Unpick, in Chapter 10, for more details.

A **Modifier** is a keyboard event that can 'modify' a mouse click.

It is also possible in most cases for the user to reset the chart to its original display parameters. The interactions described here affect the chart displayed inside the `ChartArea`; other chart elements, like the header, are not affected.

6

Chart Programming Tutorial

6.1   Introduction

This tutorial shows you how to start using JClass Chart by compiling and running an example program. It is different from the SimpleChart Bean tutorial because it focuses on programmatic use of JClass Chart. For a Bean tutorial, see the SimpleChart Bean Tutorial, in Chapter 3. This program, *Plot1.java*, will graph the 1963 Quarterly Expenses and Revenues for "Michelle's Microchips", a small company a little ahead of its time.

The following table shows the data to be displayed:

	Q1	Q2	Q3	Q4
Expenses	150.0	175.0	160.0	170.0
Revenue	125.0	100.0	225.0	300.0

6.2 A Basic Plot Chart

When *Plot1.java* is compiled and run, the window shown below is displayed:
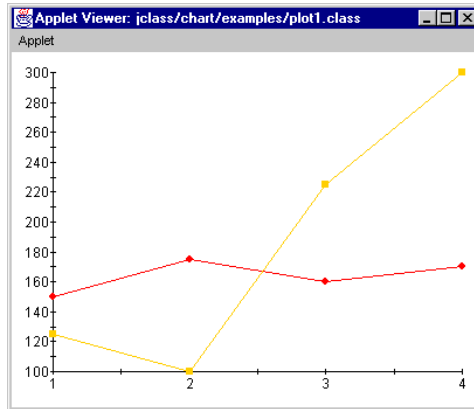
Figure 16   The Plot1.java program displayed.

The following listing displays the program *Plot1.java*. This is a minimal Java program that creates a new chart component and loads data into it from a file. It can be run as an applet or a standalone application.

Line	Source
1	`package examples.chart.intro;`
2	
3	`import java.awt.GridLayout;`
4	`import javax.swing.JPanel;`
5	`import com.klg.jclass.chart.JCChart;`
6	`import com.klg.jclass.chart.ChartDataView;`
7	`import com.klg.jclass.chart.data.JCFileDataSource;`
8	`import com.klg.jclass.util.swing.JCExitFrame;`
9	
10	`import demos.common.FileUtil;`
11	
12	`/**`
13	`* Basic example of Chart use.  Load data from`

Line	Source
14	`* a file and displays it as a simple plot chart.`
15	`*/`
16	`public class Plot1 extends JPanel {`
17	
18	`/**`
19	`* Default constructor for this class. Loads data and`
20	`* sets up chart.`
21	`*/`
22	`public Plot1() {`
23	`    setLayout(new GridLayout(1,1));`
24	
25	`    // Create new chart instance.`
26	`    JCChart chart = new JCChart();`
27	`    // Load data for chart`
28	`    try {`
29	`      // Use JCFileDataSource to load data from specified file`
30	`      String fname = FileUtil.getFullFileName(`
31	`        "examples.chart.intro","chart1.dat");`
32	`      chart.getDataView(0).setDataSource(new JCFileDataSource`
33	`        (fname));`
34	`    }`
35	`    catch (Exception e) {`
36	`      e.printStackTrace(System.out);`
37	`    }`
38	`    // Add chart to panel for display.`
39	`    add(chart);`
40	`}`
41	

Line	Source
42	`public static void main(String args[]) {`
43	`    JCExitFrame f = new JCExitFrame("Plot1");`
44	`    Plot1 p = new Plot1();`
45	`    f.getContentPane().add(p);`
46	`    f.setSize(200, 200);`
47	`    f.setVisible(true);`
48	`}`
49	
50	`}`
51	

Most of the code in *Plot1.java* should be familiar to Java programmers. The first few lines (3–10) import the classes necessary to run *Plot1.java*. In addition to the standard AWT `GridLayout` class and Swing `JPanel` class, three classes in the `jclass.chart` package are needed: `JCChart` (the main chart class), `ChartDataView` (the data view object), and `JCFileDataSource` (a stock data source). This example also makes use of the `JCExitFrame` from JClass Elements, which is a part of the JClass DesktopViews suite. Line 16 provides the class definition for this program, a subclass of `JPanel`.

Lines 22–40 define the constructor. The `Layout` property on line 23 lays out a simple grid structure to display the components it holds. A new chart is then instantiated on line 26. Lines 30-31 load data from a file named *chart1.dat* into a new data source object (`JCFileDataSource`) and tell the chart to display this data.

Lines 42-48 define the `main()` method needed when the program is run as a standalone Java application.

6.3   Loading Data From a File

A common task in any JClass Chart program is to load the chart data into a format that the chart can use. JClass Chart uses a "model view/control" (MVC) architecture to handle data in a flexible and efficient manner. The data itself is stored in a object that implements the ChartDataModel interface created and controlled by your application. The chart has a `ChartDataView` object that controls a view on this data source, providing properties that control which data source to use, and how to display the data.

JClass Chart includes several stock (built-in) data sources that you can use (or you can define your own). This program uses the data source that reads data from a file: `JCFileDataSource`. With this understanding we can look more closely at lines 32-33:

```
chart.getDataView(0).setDataSource(new JCFileDataSource
    (fname));
```

Two things are happening here: a new `JCFileDataSource` object is instantiated, with the name of the data file passed as a parameter in the constructor, and the `DataSource` property of the chart's first (default) data view is being set to use this data source.

The following shows the contents of the *chart1.dat* file:

```
ARRAY 2 4
# X-values
1.0 2.0 3.0 4.0
# Y-values
150.0 175.0 160.0 170.0
# Y-values set 2
125.0 100.0 225.0 300.0
```

This file is in the format understood by `JCFileDataSource`. Lines beginning with a '#' symbol are treated as comments. The first line tells the `FileDataSource` object that the data that follows is in Array layout and is made up of two series containing four points each. The X-values are used by all series.

There are two types of data: Array and General. Use Array layout when the series of Y-values share common X-values. Use General when the Y-values do not share common X-values, or when all series do not have the same number of values.

Note that for data arrays in Polar charts, (*x*, *y*) coordinates in each data set will be interpreted as (*theta*, *r*). For array data, the X-array will represent a fixed theta value for each point.

In Radar and Area Radar charts, only array data can be used. (*x*, *y*) points will be interpreted in the same way as for Polar charts (above), except that the theta (that is, x) values will be ignored. The circle will be split into `nPoints` segments with `nSeries` points drawn on each radar line.

For complete details on using data with JClass Chart, please see Data Sources, in Chapter 8.

6.4 Adding Header, Footer, and Labels

The plot displayed by *Plot1.java* is not very useful to an end-user. There is no header, footer, or legend, and the X-axis numbering is not very meaningful.

The chart below displays various changes that can be made to a chart to make it more useful. The changes made to this chart are listed below. Full source code can be found in the *plot2.java* program, located in the *JCLASS_HOME/examples/chart/intro* directory.
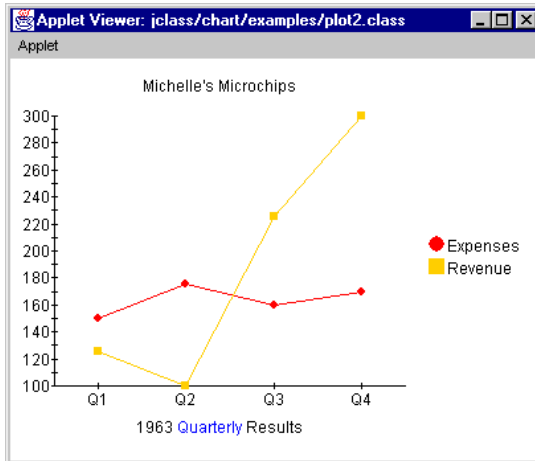
Figure 17   The program created by Plot2.java.

JClass Chart will always try to produce a reasonable chart display, even if very few properties have been specified. JClass Chart will use intelligent defaults for all unspecified properties.

All properties for a particular chart may be specified when the chart is created. Properties may also be changed as the program runs by calling the property's set method. A programmer can also ask for the current value of any property by using the property's get method.

Adding Headers and Footers

To display a header or footer, we need to set properties of the Header and Footer objects contained in the chart. For example, the following code sets the Text and Visible properties for the footer:

```
// Make footer visible
   chart.getFooter().setVisible(true);
   // By default, footer is a JLabel - set its Text property
   ((JLabel)chart.getFooter()).setText("1963 Quarterly Results");
```

Visible displays the header/footer. Text specifies the text displayed in the header/footer.

By default, headers and footers are JLabels, although they can be any Swing JComponent. JLabels support the use of HTML tags. The use of HTML tags overrides the default Font and Color properties of the label.

Please note that HTML labels may not work with PDF, PS, or PCL encoding.

Adding a Legend and Labelling Points

A legend clarifies the chart by showing an identifying label for each series in the chart. We would also like to display more meaningful labels for the points along the X-axis. Both types of information can be easily specified in the data file itself. The following lists *chart2.dat*, a modified version of the previous data file that includes series labels (for the legend), and point labels (for the X-axis):

```
ARRAY '' 2 4
# Point Labels
'Q1' 'Q2' 'Q3' 'Q4'
# X-values, with a blank series label ('') -- a blank series
# label is required if the Y-values have series labels
'' 1.0  2.0  3.0  4.0
# Y-values, with Series label (in this case, Expenses)
'Expenses' 150.0 175.0 160.0 170.0
# Y-values set 2, with Series label (in this case, Revenue)
'Revenue' 125.0 100.0 225.0 300.0
```

Lines beginning with a '#' symbol are treated as comments.

As noted in the comments within the above code, if series labels are being used for the Y-values, then the X-data must be preceded by a blank series label (''). This blank label will not show up on the chart. The third line specifies the point labels (for instance, "Q1"). Subsequent lines of data begin with a Y-data series label ("Expenses" and "Revenue").

This data file now provides the labels that we want to use, but to actually display them in the chart, we need to set the Legend object's Visible property and change the AnnotationMethod property of the X-axis to annotate the axis with the point labels in the data.

These and the previous changes are combined; now the chart is created with code that looks like this:

```
// Create new chart instance.
chart = new JCChart();
    // Load data for chart
    try {
        // Use JCFileDataSource to load data from specified file
        String fname = FileUtil.getFullFileName("examples.chart.intro",
            "chart2.dat");
        chart.getDataView(0).setDataSource(new
            JCFileDataSource(fname));
    }
    catch (Exception e) {
        e.printStackTrace(System.out);
    }
    // Make header visible, and add some text
    chart.getHeader().setVisible(true);
    // By default, header is a JLabel   -- set its Text property
    ((JLabel)chart.getHeader()).setText("Michelle's Microchips");
    // Make footer visible
    chart.getFooter().setVisible(true);
    // By default, footer is a JLabel -- set its Text property
    ((JLabel)chart.getFooter()).setText("1963 Quarterly Results");

    // Make legend visible
    chart.getLegend().setVisible(true);

    // Make X-axis use point labels instead of default value labels.
    chart.getChartArea().getXAxis(0).setAnnotationMethod
        (JCAxis.POINT_LABELS);

    // Add chart to panel for display.
    add(chart);
```

Because we are accessing a variable defined in JCAxis, we need to add that to the classes imported by the program:

```
 import jclass.chart.JCAxis;
```

In the line that sets the annotation method, notice that XAxis is a collection of JCAxis objects. A single chart can display several X- and Y-axes.

6.5 Changing to a Bar Chart
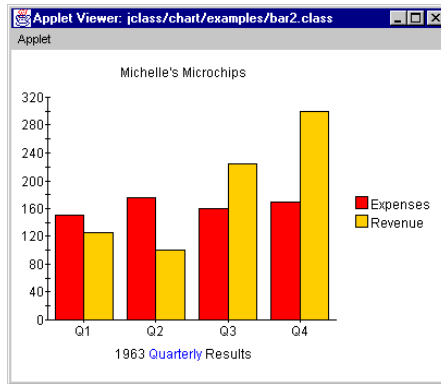
Figure 18   The bar2.java program displayed.

A powerful feature of JClass Chart is the ability to change the chart type independently of any other property. (Although there are interdependencies between some properties, most properties are completely orthogonal.) For example, to change the Plot2 chart to a bar chart, the following code can be used:

```
c.getDataView(0).setChartType(JCChart.BAR);
```

This sets the ChartType property of the data view. Alternately, you can set the chart type when you instantiate a new chart, for example:

```
JCChart c = new JCChart(JCChart.BAR);
```

The full code for this program (*Bar2.java*) can be found in with the other examples.

JClass Chart can display data as one of 13 different chart types. For more information on chart types, see Chart Types and Special Chart Properties, in Chapter 2.

6.6 Inverting Chart Orientation

Most graphs display the X-axis horizontally and the Y-axis vertically. It is often appropriate, however, to invert the sense of the X- and Y-axis. This is easy to do, using the Inverted property of the data view object.

In a plot, inverting causes the Y-values to be plotted against the horizontal axis, and the X-values to be plotted against the vertical. In a bar chart, it causes the bars to be displayed horizontally instead of vertically.

When programming JClass Chart, try not to assume that the X-axis is always the horizontal axis. Determining which axis is vertical and which horizontal depends on the value of the Inverted property.

To invert, set the data view object's `Inverted` property to `true`. By default it is `false`.

```
c.getDataView(0).setInverted(true);
```

The following shows the windows created by *Plot2.java* and *Bar2.java* when inverted:
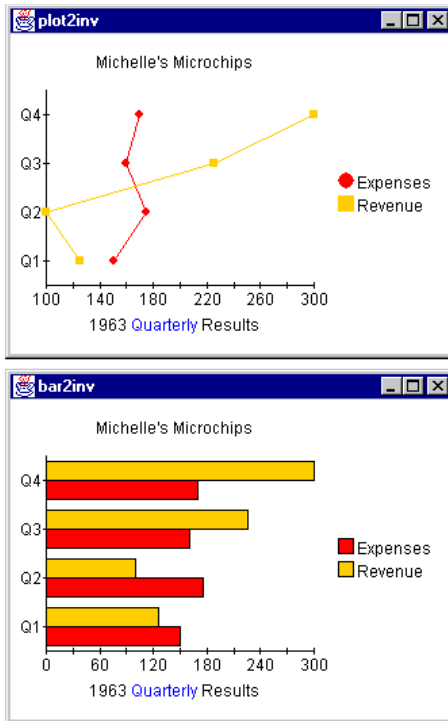
Figure 19   Plot2 and Bar2 windows with Inverted set to true.

Full code for these examples is in the *JCLASS_HOME/examples/chart/intro* directory.

6.7   Bar3d and 3d Effect

Chart 3D effects can be added to bar and stacking bar charts. Three properties affect the display of 3D information: Depth, Elevation, and Rotation. Modifying these properties will alter the 3D effects displayed. Depth and at least one of Elevation or Rotation must be non-zero to see any 3D effects. The properties can be set as follows:

```
chart.getChartArea().setElevation(20);
chart.getChartArea().setRotation(30);
chart.getChartArea().setDepth(10);
```

Function call Header for the function	Description

`setDepth()`
 	
`public void setDepth(`
`    int newDepth)`
	Controls the apparent depth of the chart; the parameter `newDepth` represents the depth as a percentage of the width; valid values are 0 to 500.

`setElevation()`
 	
`public void setElevation(`
`    int newElevation)`
	Controls the distance above the X-axis for the 3D effect; the parameter `newElevation` is the number of degrees above the X-axis that the chart is to be positioned; valid values are between -45 and 45.

`setRotation()`
 	
`public void setRotation(`
`    int newRotation)`
	Controls the position of the eye relative to the Y-axis for the 3D effect; the parameter `newRotation` is the number of degrees to the right of the Y-axis the chart is to be positioned; valid values are between -45 and 45.

6.8   End-User Interaction

More than simply a display tool, JClass Chart is an interactive component. Programmers can explicitly add functions that enable an end-user to directly interact with a chart. The following end-user interactions are possible:

■   Translation – users can move a graph or a series of graphs along the X- and/or Y- axes.

■   Rotate – users can change the vantage point of a chart type, to better view information contained with a JClass Chart component.

■   Zoom – users can zoom in or out of a JClass Chart component to better view information contained within it.

■   Depth – users can change the apparent depth of a 3D chart.

■   Edit – users can change the placement of data points within a chart.

■   Customize – users can alter the other display features of a chart, (such as color, label names, or the numerical value of data points) that comprise a chart display.

■   Pick – users can determine the position of data points displayed on a chart.

6.9 Get Started Programming with JClass Chart

The following suggestions should help you become productive with JClass Chart as quickly as possible:

■ Check out the sample code – the example and demo programs included with JClass Chart are useful in showing what JClass Chart can do, and how to do it. Run them and examine the source code. They can all be found in the *JCLASS_HOME/demos/chart* and *JCLASS_HOME/examples/chart* directories.

■ Browse the JClass Chart API documentation – complete reference documentation on the API is available online in HTML format. The properties, methods, and events for each component are documented.

7

Axis Controls

JClass Chart can automatically set properties based on the data, so axis numbering and data display usually do not need much customizing. You can however, control any aspect of the chart axes, depending on your requirements. This chapter covers the different axis controls available. If you are developing your chart application using one of the JClass Chart Beans, please refer to Bean Reference, in Chapter 4.

7.1 Creating a New Chart in a Nutshell

1. If one exists, use an existing chart as a starting point for the new one. The sample charts provided in *JCLASS_HOME/examples/chart/* are a good starting point. Load a chart description resembling the new chart.

2. Load your data into the chart.

3. Set the chart type.

4. Annotate and format the axes and data if necessary, described as follows:

   ■ Axis annotation (`Values` [default], `ValueLabels`, `PointLabels`, `TimeLabels`)

   ■ Positioning Axis Annotations

   ■ Chart Orientation and Axis Direction

   ■ Setting Axis Bounds

   ■ Customizing Origins

   ■ Logarithmic Axes

   ■ Titling Axes and Rotating Axis Elements

   ■ Adding Gridlines

   ■ Adding a Second Axis

7.2   Axis Labelling and Annotation Methods

There are several ways to annotate the chart's axes, each suited to specific situations.

- The chart can automatically generate numeric annotation appropriate to the data it is displaying.
- You can provide a label for each point in the chart (X-axis only).
- You can provide a label for specific values along the axis.
- The chart can automatically generate time-based annotations.

Please note that none of the axis properties discussed in this section apply to Pie charts because Pie charts do not have axes. To annotate a Pie Chart, use Chart Labels; for more information, please see Chart Labels, in Chapter 9.

Whichever annotation method you choose, the chart makes considerable effort to produce the most natural annotation possible, even as the data changes. You can fine-tune this process using axis annotation properties.

User-set annotations support the use of HTML tags. The use of HTML tags overrides the default `Font` and `Color` properties of the label.

Please note that HTML labels may not work with PDF, PS, or PCL encoding.

7.2.1   Choosing Annotation Method

A variety of properties combine to determine the annotation that appears on the axes. The `JCAxis AnnotationMethod` property specifies the method used to annotate the axis. The valid annotation methods are:

`JCAxis.VALUE` (default)	The chart chooses appropriate axis annotation automatically (with possible callbacks to a label generator), based on the chart type and the data itself.
`JCAxis. POINT_LABELS` (X-axis only)	The chart spaces the points based on the X-values and annotates them with text you specify (in the data source) for each point.
`JCAxis. VALUE_LABELS`	The chart annotates the axis with text you define for specific X- or Y-axis coordinates.
`JCAxis. TIME_LABELS`	The chart interprets the X- or Y-values as units of time, automatically choosing time/date annotation based on the starting point and format you specify. Not for Polar, Radar, or Area Radar charts.

Notes:

- Point labels annotation (`JCAxis.POINT_LABELS`) is only valid for an X-axis when it has been added to the X-axis collection in `JCChartArea`. This means that a new `JCAxis` instance that has not yet been added to `JCChartArea` will not be considered an X-axis.

- The spokes of Area Radar and Radar charts are automatically labelled "0", "1", "2", and so forth, unless the X-annotation method is `JCAxis.POINT_LABELS`.

- `NumSpacing` has no effect on value labels.

- For Polar charts, the default annotation for `JCAxis.VALUE` depends on the angle units specified. If it is radians, the symbol for pi will not be used (it will be represented by 3.14 instead). Also, the X-axis will always be linear; that is, setting the logarithmic properties to `true` will be ignored.

The following topics discuss setting up and fine-tuning each type of annotation.

7.2.2 Values Annotation

`Values` annotation produces numeric labelling along an axis, based on the data itself. The chart can produce very natural-looking axis numbering automatically, but you can fine-tune the properties that control this process.

Axis Annotation Increments, Numbering, and Precision

When a `JCAxis` is instantiated, a pair of `JCAnno` objects representing default labels and ticks are automatically created and set on the axis. Those default `JCAnno` objects may be modified, deleted, or augmented with other `JCAnno` objects.

The following describes the different properties that can be set on a `JCAnno` object in order to fully customize the labels and tick marks:

Property	Function
startValue	Sets the value at which the annotation begins.
stopValue	Sets the value where the annotation ends.
incrementValue	Sets the increment between annotation along an axis.
innerExtent	Defines the space, in pixels, that tick marks extend into the plot area.
outerExtent	Defines the space, in pixels, that tick marks extend out of the plot area.
tickColor	Determines the color of the tick marks.
drawTicks	Determines whether or not the tick marks defined by `JCAnno` are drawn.
labelExtent	Defines the distance, in pixels, of the labels from the axis.
labelColor	Determines the color of the labels.

Property	Function
precision	Sets the number of decimal places to use when displaying a chart label number. The effect depends on whether it is positive or negative:
	■ *Positive* values add that number of places after the decimal place. For example, a value of 2 displays an annotation of 10 as "10.00".
	■ *Negative* values indicate the minimum number of zeros to use before the decimal place. For example, a value of -2 displays annotation in multiples of 100.
drawLabels	Determines whether or not the labels defined by JCAnno are drawn.

When the annotation method for an axis is VALUE_LABELS, POINT_LABELS, or TIME_LABELS, the labels are either user-supplied or internally generated without the use of JCAnno objects. The boolean UseAnnoTicks property of a JCAxis determines how tick marks are drawn in those cases. If UseAnnoTicks is true, tick marks are drawn at the labels. If the value is false, ticks defined by JCAnno objects are drawn instead.

Using multiple JCAnno objects, an axis can be drawn with major and minor ticks. Labels can be turned on or off for the individual tick series, as can the actual tick marks, enabling further flexibility.
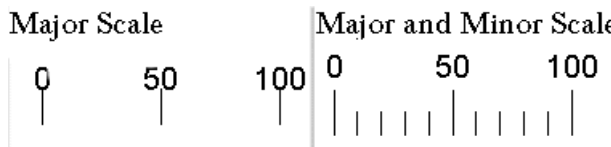
Figure 20 Different tick styles that can be applied to a chart axis.

Please refer to the AnnoGrid.java example included in the *examples/chart/intro* package to view different tick marks in a JClass Chart example.

7.2.3 PointLabels Annotation

PointLabels annotation displays defined labels along an X-axis. This is useful for annotating the X-axis of any chart for which all series share common X-values. PointLabels are most useful with bar, stacking bar, and pie charts. It is possible to add,

remove, and edit `PointLabels`. In JClass Chart, `PointLabels` are typically defined with the data.
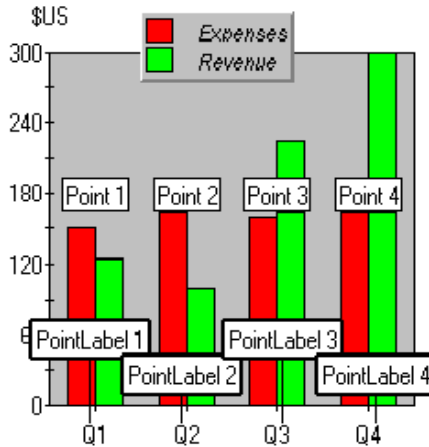
Figure 21   PointLabels X-axis annotation.

`PointLabels` are a collection of labels. The first label applies to the first point, the second label applies to the second point, and so on.

The labels can also be supplied by setting the `PointLabels` property of the `ChartDataView` object for this chart. For example, the following code specifies labels for each of the three points on the X-axis:

```
String[] labels = {"Q1", "Q2", "Q3, "Q4"};
c.getChartArea().getXAxis(0).setAnnotationMethod(JCAxis.POINT_LABELS);
ChartDataView cd = c.getDataView(0);
ArrayList pLabels = new ArrayList();
for (int i = 0; i < labels.length; i++) {
    pLabels.add(labels[i]);
}
cd.setPointLabels(pLabels);
```

For Polar, Radar, and Area Radar charts, if the X-axis annotation is `POINT_LABELS` and the data is of type array, then a point label is drawn at the outside of the X-axis for each point. (Series labels are used in the legend as usual.)

7.2.4   ValueLabels Annotation

`ValueLabels` annotation displays labels at the axis coordinate specified. This is useful for displaying special text at a specific axis coordinate, or when a type of annotation that the chart does not support is needed, such as scientific notation. You can set the axis

coordinate and the text to display for each ValueLabel, and also add and remove
individual ValueLabels.

Figure 22   Using ValueLabels to annotate axes.

Every label displayed on the axis is one ValueLabel. Each ValueLabel has a Value
property and a Label property.

If the AnnotationMethod property is set to JCAxis.VALUE_LABELS, the chart places labels
at explicit locations along an axis. The ValueLabels property of JCAxis, which is a
ValueLabels collection, supplies this list of Strings and their locations. For example, the
following code sets value labels at the locations 10, 20, and 30:

```
String[] labels = {"Sales", "Beta Testing", "Documentation",
                   "Alpha Testing", "Programming",
                   "Production Definition"};
JCAxis y = c.getChartArea().getYAxis(0);
y.setAnnotationMethod(JCAxis.VALUE_LABELS);
JCValueLabel[] valueLabels = new JCValueLabel[labels.length];
for (int i = 0; i < labels.length; i++) {
     valueLabels[i] = new JCValueLabel(10.0 * (i + 1), labels[i], y);
}
y.setValueLabels(valueLabels);
```

The ValueLabels collection can be indexed either by subscript or by value:

```
JCAxis x = c.getChartArea().getXAxis(0);
// The following retrieves the second value label
JCValueLabel v1 = x.getValueLabels(1);
// The following retrieves the closest label to chart coordinate 2.0
JCValueLabel v2 = x.getValueLabel(2.0);
```

7.2.5   TimeLabels Annotation

TimeLabels annotation interprets the value data as units of time. The chart calculates and displays a time-axis based on the starting point and format specified. A time-axis is useful for charts that measure something in seconds, minutes, hours, days, weeks, months, or years.
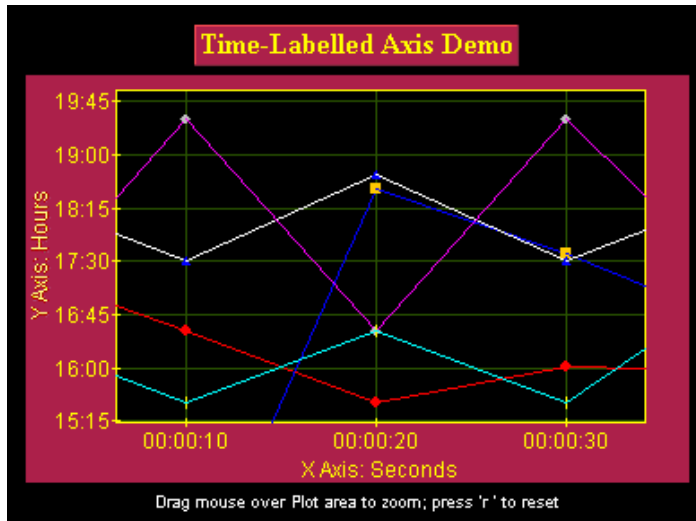
Figure 23   TimeLabels annotating X- and Y-axes.

Four properties are used to control the display and behavior of TimeLabels:

- ■   AnnotationMethod (set to JCAxis.TIME_LABELS to use this annotation method)
- ■   TimeUnit
- ■   TimeBase
- ■   TimeFormat

Time Unit

Use the TimeUnit property to specify how to interpret the values in the data. Select either JCAxis.SECONDS, JCAxis.MINUTES, JCAxis.HOURS, JCAxis.WEEKS, JCAxis.MONTHS, or JCAxis.YEARS. For example, when set to JCAxis.YEARS, values that range from 5 to 15 become a time-axis spanning 10 years. By default, TimeUnit is set to JCAxis.SECONDS.

Time Base

Use the TimeBase property to set the date and time that the time-axis starts from. Use the Java Date class (java.util.Date) to specify the TimeBase. The default for TimeBase is the current time.

For example, the following statement sets the starting point to January 15, 1985:

```
c.getChartArea().getXAxis(0).setTimeBase(new Date(85,0,15));
```

Time Format

Use the `TimeFormat` property to specify the text to display at each annotation point. The `TimeFormatIsDefault` property allows the chart to automatically determine an appropriate format based on the `TimeUnit` property and the data, so it is often unnecessary to customize the format.

`TimeFormat` specifies a *time format*. You build a time format using the Java time format codes from the `java.text.SimpleDateFormat` class. The chart displays only the parts of the date/time specified by `TimeFormat`. The format codes are based on the default Java formatting provided by java.text.

Symbol	Meaning	Presentation	Example
G	era designator		AD
y	year	Number	1997
M	month in year	Text & Number	July 07
d	day in month	Number	10
h	hour in am/pm (1 ~12)	Number	12
H	hour in day (0~23)	Number	0
m	minute in hour	Number	30
s	second in minute	Number	55
S	millisecond	Number	978
E	day in week	Text	Tuesday
D	day in year	Number	189
F	day of week in month	Number	2nd Wed in July
w	week in year	Number	27
W	week in month	Number	2
a	am/pm marker	Text	PM
k	hour in day (1~24)	Number	24
K	hour in am/pm (0~11)	Number	0
z	time zone	Text	Pacific Standard Time
'	escape for text	delimiter	

Symbol	Meaning	Presentation	Example
”	single quote	Literal	

The default for `TimeFormat` is the same as the default used by Java's `SimpleDateFormat` class (located in the `java.text` package).

Using Date Methods

The `dateToValue()` method converts a Java date value into its corresponding axis value (a floating-point value). The `valueToDate()` method converts a value along an axis to the date that it represents. Note that the axis must already be set as a time label axis.

Here is a code example showing the `dateToValue()` method converting a date (in this case, February 2, 1999) to a Y-axis value, and showing the `valueToDate()` method converting a Y-axis value (in this case, 3.0) to the date that it represents.

```
JCAxis y = chart.getChartArea().getYAxis(0);
Date d = y.valueToDate(3.0);
double val = y.dateToValue(new Date(99,1,2));
```

7.2.6 Custom Axes Labels

JClass Chart will label axes by default. However, you can also generate custom labels for the axes by implementing the `JCLabelGenerator` interface. This interface has one method – `makeLabel()` – that is called when a label is required at a particular value.

Note that the spokes of Radar and Area Radar charts will be automatically labelled "0", "1", "2", and so forth, unless the X-annotation method is `JCAxis.POINT_LABELS`.

To generate custom axes labels, the axis' `AnnotationMethod` property, which determines how the axis is labelled, must be set to `VALUE`. Also, the `setLabelGenerator()` method must be called with the class that implements the `JCLabelGenerator` interface.

The number of labels, that is, the number of times `makeLabel()` is called, depends on the `NumSpacing` parameter of the axis. Not all labels will be displayed if there is not enough room.

The `makeLabel()` method takes two parameters: `value` (the axis value to be labelled) and `precision` (the numeric precision to be used).

- In the usual case, the `makeLabel()` method returns a `String`, and that `String` will be used as the axis label at `value`.
- If the `makeLabel()` method returns a `ChartText` object, then that `ChartText` object will be used as the axis label at `value`.
- If an object other than `String` or `ChartText` is returned, the `String` derived from calling that object's `toString()` method will be used as the axis label at `value`.

Here is a code example showing how to customize the labels for a linear axis by implementing the `JCLabelGenerator` interface. In this case, Roman numeral labels are going to be generated (instead of the usual Arabic labels) for the numbers 1 through 10.

```
class MyLabelGenerator implements JCLabelGenerator
{
    public Object makeLabel(double value, int precision) {
        int intvalue = (int) value;
        String s = null;
        switch (intvalue) {
            case 1 :
                s = "I";
                break;
            case 2 :
                s = "II";
                break;
            case 3 :
                s = "III";
                break;
            case 4 :
                s = "IV";
                break;
            case 5 :
                s = "V";
                break;
            case 6 :
                s = "VI";
                break;
            case 7 :
                s = "VII";
                break;
            case 8 :
                s = "VIII";
                break;
            case 9 :
                s = "IX";
                break;
            case 10 :
                s = "X";
                break;
            default :
                s = "";
                break;
        }
        return s;
    }
}
```

Note that the user will need to specify the label generator as follows:

```
axis.setLabelGenerator(new MyLabelGenerator());
```

Also note that JClass Chart calls the `makeLabel()` method for each *needed* label (recall that each axis requests needed labels based on its `NumSpacing`, `Min`, and `Max` properties). Thus, if JClass Chart needs *n* labels, the `makeLabel()` method is called *n* times.

7.3   Positioning Axes

Use the `Placement` property to make a specific axis placement or use the
`PlacementIsDefault` property to specify whether the chart is meant to determine axis
placement. When making a specific axis placement, the axis may be placed against its
partner axis at the partner axis' minimum value, maximum value, origin value, or a user-
specified value.

For example,

```
axis.setPlacement(JCAxis.MIN);
```

will place the axis against its partner axis' minimum value, and

```
axis.setPlacement(otherAxis, 5.0)
```

will place the axis against `otherAxis` at the value 5.0.

Note: When `Placement` is set to Origin, changing the axis origin will move the placed
axis to the new origin value.
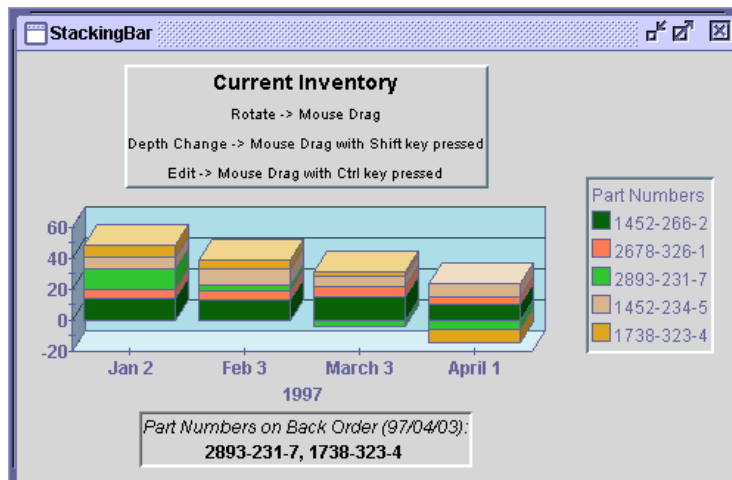
Figure 24   An example of axes positioning; the X-axis is placed against the Y-axis' minimum value.

Polar Charts – Special Minimum and Maximum Values

Note that for Polar charts, the X-axis max and min values are fixed, and these fixed values
change depending on the angle unit type. The Y-axis max and min values are adjustable,
but are constrained to avoid data clipping. The Y-axis min will never be less than zero
(unless the Y-axis is reversed). (*theta*, −*r*) will be interpreted as (*theta+180*, *r*). The Y-axis
min will always be at the center unless the axis is reversed, in which case the Y-axis max
will be at the center.

Radar and Area Radar Charts – Minimum Values
The minimum value for a Y-axis in Radar and Area Radar charts can be negative.

7.4   Chart Orientation and Axis Direction

A typical chart draws the X-axis horizontally from left-to-right, and the Y-axes vertically from bottom-to-top. You can reverse the orientation of the entire chart, and/or the direction of each axis.

7.4.1   Inverting Chart Orientation

Use the `ChartDataView` object's `Inverted` property to change the chart orientation. When set to `true`, the X-axis is drawn vertically and the Y-axis horizontally for the data view. Any properties set on the X-axis then apply to the vertical axis, and Y-axis properties apply to the horizontal axis.

Note: To switch the orientation of charts with *multiple* data views, you must set the `Inverted` property of each `ChartDataView` object.
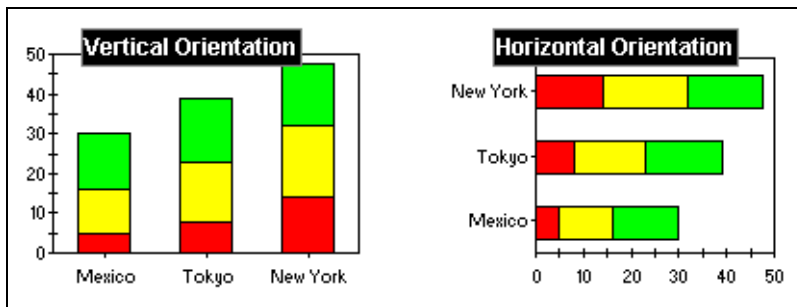
Figure 25   Normal and inverted orientation.

7.4.2   Changing Axis Direction

Use the `Reversed` property of `JCAxis` to reverse the direction of an axis. By default, `Reversed` is set to `false`.
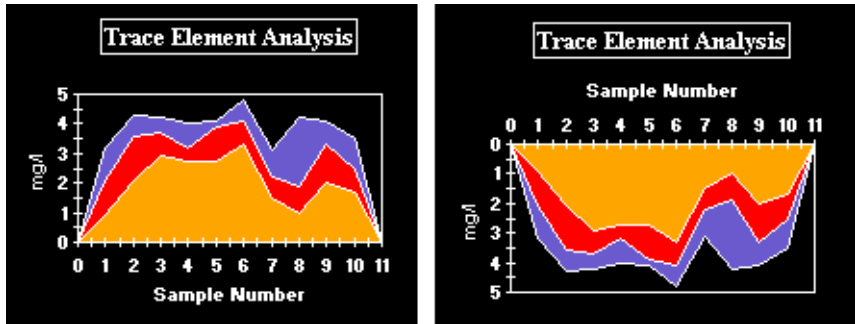
Figure 26   Two charts depicting a normal and reversed Y- axis.

For Polar charts, data points with positive X-values will be displayed in a counterclockwise direction starting at the origin base. When the `XAxis.reversed` flag is `true`, positive X-values will be displayed clockwise.

7.5   Setting Axis Bounds

Normally a graph displays all of the data it contains. There are situations where only part of the data is to be displayed. This can be accomplished by fixing axis bounds.

Min and Max

Use the `Min` and `Max` properties of `JCAxis` to frame a chart at specific axis values. The `MinIsDefault` and `MaxIsDefault` properties allow the chart to automatically determine axis bounds based on the data bounds.

7.6   Customizing Origins

The chart can choose appropriate origins for the axes automatically, based on the data. It
is also possible to customize how the chart determines the origin, or to directly specify the
coordinates of the origin.
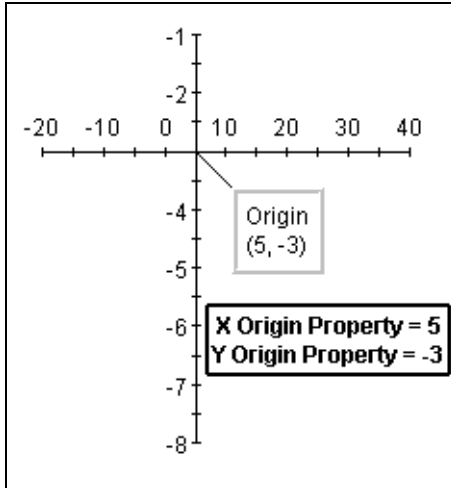
Figure 27   Defining origins for X- and Y-axes.

Origin Placement

The easiest way to customize an origin is by controlling its placement, using the Axes'
`OriginPlacement` property. It has four possible values: `AUTOMATIC`, `ZERO`, `MIN`, and `MAX`.
When set to `AUTOMATIC`, the origin is placed at the axis minimum or at zero, if the data
contains positive and negative values or is a bar chart. `ZERO` places the origin at zero. `MIN`
places the origin at the minimum value on the axis. `MAX` places the origin at the maximum
value on axis.

Origin Coordinates

When the origin of a coordinate must be set to a value different from the default (0,0), use
the `Axes'` `Origin` property. The `OriginIsDefault` property allows the chart to
automatically determine the origin coordinate based on the data.

Note: When an origin coordinate is explicitly set or fixed, the chart ignores the
`OriginPlacement` property.

7.7   Logarithmic Axes

Axis annotation is normally interpreted and drawn in a *linear* fashion. It is also possible to set any axis to be interpreted *logarithmically* (log base 10), as shown in the following image. Logarithmic axes are useful for charting certain types of scientific data.
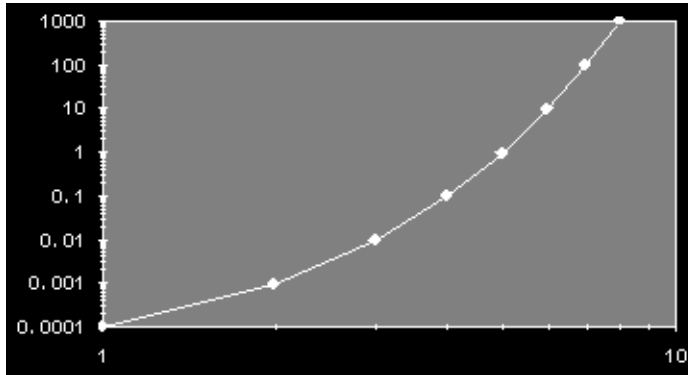
Figure 28   Logarithmic X- and Y-axes.

Because of the nature of logarithmic axes, they impose the following restrictions on the chart:

- any data that is less than or equal to zero is not graphed (it is treated as a *data hole*), since a logarithmic axis only handles data values that are greater than zero. For the same reason, axis and data minimum/maximum bounds and origin properties cannot be set to zero or less.

- axis numbering increment, ticking increment, and precision properties have no effect when the axis is logarithmic.

- the X-axis of bar and stacking bar charts cannot be logarithmic.

- the annotation method for the X-axis cannot be `PointLabels` or `TimeLabels`.

Specifying a Logarithmic Axis

Use the `Logarithmic` property of `JCAxis` to make an axis logarithmic.

Note: Pie charts are not affected by logarithmic axes.

7.8 Titling Axes and Rotating Axis Elements

Adding a title to an axis clarifies what is charted along that axis. You can add a title to any axis, and also rotate the title or the annotation along the axis, as shown below.
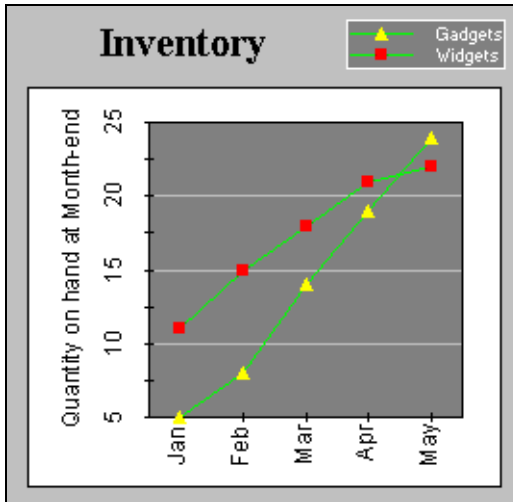
Figure 29   Rotated axis title and annotation.

Adding an Axis Title

Use the `Title` property to add a title to an axis. It sets the `JCAxisTitle` object associated with the `JCAxis`. `JCAxisTitle` controls the appearance of the axis title. `JCAxisTitle`'s `Text` property specifies the title text.

Axis Title Rotation

Use the `Rotation` property of `JCAxisTitle` to set the rotation of the title. Valid values are defined in `ChartText`: `DEG_0` (no rotation), `DEG_90` (90 degrees counterclockwise), `DEG_180` (180 degrees), and `DEG_270` (270 degrees).

Rotating Axis Annotation

Use the `AnnotationRotation` property of `JCAxis` to rotate the axis annotation to either 90, 180, or 270 degrees clockwise from the horizontal position. 90-degree rotation usually looks best on a right-hand side axis.

This property can also be used to rotate the annotation at any other specified angle, if it is set to `AnnotationRotation.ROTATION_OTHER`. The new angle will be determined by the `AnnotationRotationAngle` property's value. By default, the angle is 0.0 degrees.

It is important to know that some fonts may not draw properly at an angle; therefore, they might not be visually appealing. If you are using rotated labels, your font choice should be made with care.

Note: In some cases, rotated labels will overlap. When labels overlap, the `visible` property for the higher indexed label is cleared, and only the lower indexed label is visible.

7.9   Adding Gridlines

Displaying a grid on a chart can make it easier to see the exact value of data points. The spacing between lines on the grid can be defined to determine how a grid is displayed.
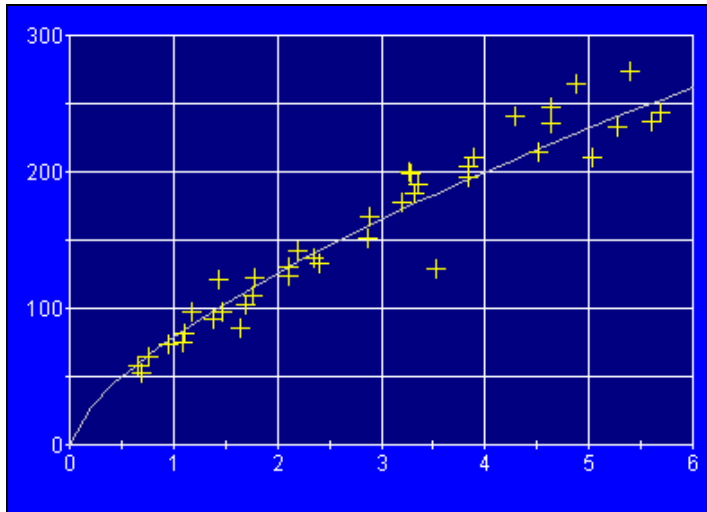
Figure 30   JClass Chart illustrating the effects of gridlines.

Horizontal gridlines are a property of the Y-axis. Vertical gridlines are a property of the X-axis. Set `GridVisible` to `true` to display gridlines.

Note that for Polar charts, Y-gridlines will be circular while X-gridlines will be radial lines from the center to the outside of the plot. For both Radar and Area Radar charts, radar lines are represented by the X-axis gridlines. You may choose normal gridlines (circular) or "webbed" gridlines. For the Y-axis, you may also have gridlines on (default is off).

Grid Spacing

Use the `GridSpacing` property to customize the grid spacing for an axis. The `GridSpacingIsDefault` property allows the chart to space the grid automatically, drawing a gridline wherever there is annotation. By default, gridlines will correspond with axis annotations.

Grid Appearance

Use the grid `GridStyle` properties to customize the line pattern, thickness, and color of the gridlines. The following code fragment provides a sample of `GridStyle` and `GridVisible` used within a program:

```
otherXAxis.setGridVisible(true);
otherXAxis.getGridStyle().getLineStyle().setColor(Color.green);
otherYAxis.setGridVisible(true);
otherYAxis.getGridStyle().getLineStyle().setColor(Color.green);
```

7.10   Adding a Second Axis

There are two ways to create a second Y-axis on a chart. The simplest way is to define a numeric relationship between the two Y-axes, as shown in the following illustration. Use this to display a different scale or interpretation of the same graph data.

Note that for Polar, Radar, and Area Radar charts, there is no second Y-axis.

Defining Axis Multiplier

Use the `Multiplier` property to define the multiplication factor for the second axis. This property is used to generate axis values based on the first axis. The multiplication factor can be positive or negative.

Using a Constant Value

Use the `Constant` axis property to define a value to be added to or subtracted from the axis values generated by `Multiplier`.
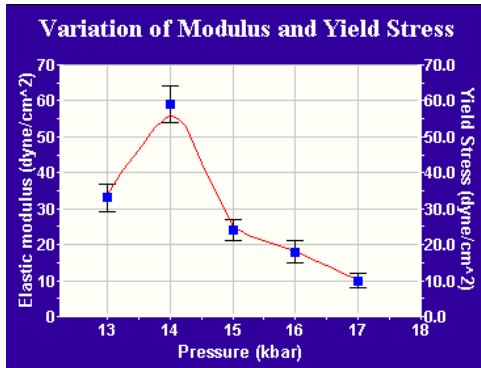
Figure 31   Chart containing multiple Y-axes.

In some cases, it may be desirable to show two sets of data in the same chart that are plotted against different axes. JClass Chart supports this by allowing each `DataView` to specify its own `XAxis` and `YAxis`. For example, consider a case in which a second data set

d2 is to be plotted against its own Y-axis. A `JCAxis` instance must be created and added to the `JCChartArea`, as shown:

```
// Create a Y-axis and set it vertical
otherYAxis = new JCAxis();
otherYAxis.setVertical(true);

// Add it to the list of Y-axes in the chart area
c.getChartArea().setYAxis(1, otherYAxis);
// Add it to the data view
d2.setYAxis(otherYAxis);
```

Hiding the Second Axis

Set the `Visible` property to `false` to remove it from display. By default, it is set to `true`.

Other Second-Axis Properties

All axes have the same features. Any property can be set on any axis.

8

Data Sources

8.1   Overview

Data is loaded into a chart by attaching one or more chart data sources to it. A chartable data source is an object that takes real-world data and puts it into a form that JClass Chart can use. Once your data source is attached, you can chart the data in a variety of ways.

The design of JClass Chart makes it possible to chart data from virtually any real-world source. There is a toolkit you can use to create custom chartable objects (data sources) for your real-world data.

Creating your own data sources can be time consuming. For that reason, JClass Chart provides pre-built chartable data sources for most common real-world data: files, URLs, applets, Strings, and databases.

This chapter describes how to use the pre-built data sources and how to create your own.

8.2   Data Views

DataSources are added to JClass Chart through Data Views, which are encapsulated by the `ChartDataView` object. `ChartDataView` organizes data as a collection of `ChartDataViewSeries` objects, one `ChartDataViewSeries` for each series of data points.

In most cases, your charts will require only one Data View. However, JClass Chart allows you to load data from multiple data sources at the same time, assigning each source to a separate Data View. By default, all Data Views are showing, but each may be hidden or revealed depending on the needs of your application. Data Views may be mapped to the same set of X- and Y-axes, or to different axes.

Note: Radar, area radar, and pie charts do not support multiple Data Views.

8.3   Pre-Built Chart DataSources

The pre-built DataSources for JClass Chart are located in the
`com.klg.jclass.chart.data` package. Their names and descriptions follow.

DataSource name	Description
`BaseDataSource`	A very simple container for chart data.
`JCAppletDataSource`	Used to load data from an applet parameter tag.
`JCChartSwingDataSource`	Used to extract data from a Swing `TableModel`.
`JCDefaultDataSource`	An extension of `BasicDataSource`.
`JCEditableDataSource`	An editable version of `JCDefaultDataSource`.
`JCFileDataSource`	Used to load data from a file.
`JCInputStreamDataSource`	Used to load data from any stream.
`JCStringDataSource`	Used to load data from a String.
`JCURLDataSource`	Used to load data from a URL.
`JDBCDataSource`	Used to load data from a JDBC Result Set.

8.4   Loading Data from a File

An easy way to bring data into a chart is to load it from a formatted file using
`JCFileDataSource`. To load data this way, you create a data file that follows JClass Chart's
standard format, as outlined in Section 8.9, Data Formats.

Then, you instantiate a `JCFileDataSource object` and attach it to a view in your chart
application. The following example shows how to instantiate and attach a
`JCFileDataSource`:

```
chart.getDataView(0).setDataSource(new JCFileDataSource("file.dat"));
```

8.5   Loading DataSource from a URL

You can chart data from a URL address using `JCURLDataSource`. To load data this way,
you create a data file that follows JClass Chart's standard format, as outlined in Section
8.9, Data Formats.

Then, you instantiate `JCURLDataSource` and attach it to a view in your chart. The following example uses data from a file named *plot1.dat*:

```
chart.getDataView(0).setDataSource(new
    JCURLDataSource(getDocumentBase(), "plot1.dat"));
```

Parameter options for JCURLDataSource:
The following are valid parameter combinations for `JCURLDataSource`:

■ URL

■ base, file

■ host, file

host: The WWW hostname.
file: The fully qualified name of the file on the server.
URL: The URL address of a data file, eg, *http://www.quest.com/datafile.dat*.
base: A URL object representing the directory where the file is located.

In the example above, the first parameter passed is `getDocumentBase()`, a method that returns the path where the current applet is located.

8.6 Loading Data from an Applet

You can chart data from an applet using `JCAppletDataSource`.

To prepare the data, put it into the standard format, (see Data Formats), and insert it into the HTML file that calls your applet. The HTML syntax is as follows:

```
<Applet>
...
<PARAM NAME=Your_Data_Name VALUE=" ....formatted data... ">
...
</Applet>
```

"Your_Data_Name" is used by your applet to select the right set of information. Use the same name in the applet and the HTML source. If a name is not provided "data" is assumed.

With your data in the HTML file, instantiate an `JCAppletDataSource` and attach it to a view in your chart as follows:

```
chart.getDataView(0).setDataSource(new JCAppletDataSource(applet,
    "Your_Data_Name"));
```

You can also chart data from an HTML file. For a listing of the syntax of JClass Chart properties when specified in an HTML file, please see Appendix C.

Example of Data in an HTML file

```
<APPLET CODEBASE="../../../.."
CODE="jclass/chart/demos/labels/labels.class"

<PARAM NAME=data VALUE="

 ARRAY 'Oblivion Inc. 1996 Results' 2 4
            'Q1'    'Q2'    'Q3'    'Q4'
 'Quarter'    1      2       3       4
 'Expenses' 150.2  182.1   152.1   170.6
 'Revenue ' 125.5  102.7   225.0   300.9
">
</APPLET>
```

8.7   Loading Data from a Swing TableModel

The `JCChartSwingDataSource` class enables you to use any type of Swing `TableModel` data object for the chart. `TableModel` is typically used for Swing JTable components, so your application may already have created this type of data object.

`JCChartSwingDataSource` "wraps" around a `TableModel` object, so that the data appears to the chart in the format it understands.

This data source is available through the `SwingDataModel` property in the SimpleChart and MultiChart Beans. To use it, prepare your data in a Swing `TableModel` object and set the `SwingDataModel` property to that object.

8.8   Loading Data from an XML Source

8.8.1   XML Primer

XML – eXtensible Markup Language – is a scaled-down version of SGML (Standard Generalized Markup Language), the standard for creating a document structure. XML was designed especially for Web documents, and allows designers to create customized tags ("extensible"), thereby enabling common information formats for sharing both the format and the data on the Internet, intranets, et cetera.

XML is similar to HTML in that both contain markup tags to describe the contents of a page or file. But HTML describes the content of a Web page (mainly text and graphic images) only in terms of how it is to be displayed and interacted with. XML, however, describes the content in terms of what data is being described. This means that an XML file can be used in various ways. For instance, an XML file can be utilized as a convenient way to exchange data across heterogeneous systems. As another example, an XML file can be processed (for example, via XSLT [Extensible Stylesheet Language Transformations]) in order to be visually displayed to the user by transforming it into HTML.

Please note that in XML, certain special characters need to be "escaped" if you want them to be displayed. For example, you cannot simply put and ampersand (&) or a greater than sign (>) into a block of text; these special characters are represented as `&` and `>` respectively. For details on this topic, please see *http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/sax/4_refs.html#chars*.

Further Information About XML

Here are links to more information on XML.

http://www.w3.org/XML/ – another W3C site; contains exhaustive information on standards.

http://www.ucc.ie/xml – an extensive FAQ devoted to XML

http://www.java.sun.com/docs/index.html – Sun's XML site

8.8.2   Using XML in JClass

In order to work with XML in your programs or even to compile the JClass XML examples, you will need to have *jaxp.jar* and *crimson.jar* in your CLASSPATH; these files are distributed with JClass Chart – you can find them in *JCLASS_HOME/lib/*.

JClass Chart can accept XML data formatted to the specifications outlined in `com.klg.jclass.chart.data.JCXMLDataInterpreter`. This public class extends `JCDataInterpreter` and implements an interpreter for the JClass Chart XML data format. `JCXMLDataInterpreter` relies on an input stream reader to populate the specified `BaseDataSource` class.

Data can be specified either by series or by point. This is fully explained below.

Examples of XML in JClass

For XML data source examples, see the XMLArray, XMLArrayTrans, and XMLGeneral examples in *JCLASS_HOME/examples/chart/datasource*. These use the *array.xml, arraytrans.xml*, and *general.xml* data files, respectively.

Interpreter

The interpreter, which converts incoming data to the internal format used by JClass Chart, must be explicitly set by the user when loading XML-formatted data. The interpreter to use for this purpose is `com.klg.jclass.chart.data.JCXMLDataInterpreter`.

Many constructors in the various data sources in JClass Chart take the abstract class `JCDataInterpreter`, which is extended by `JCXMLDataInterpreter`. It is possible for the user to create a custom data format and a custom data interpreter by extending `JCDataInterpreter`.

Here are a few code examples that load XML data using JClass Chart's XML interpreter, `JCXMLDataInterpreter`:

```
ChartDataModel cdm = new JCFileDataSource(fileName,
    new JCXMLDataInterpreter());
ChartDataModel cdm = new JCURLDataSource(codeBase, fileName,
    new JCXMLDataInterpreter());
ChartDataModel cdm = new JCStringDataSource(string,
    new JCXMLDataInterpreter());
```

8.8.3   Specifying Data by Series

When "specifying by series", there can be any number of `<data-series>` tags. Within each `<data-series>` tag, there can be an optional `<data-series-label>` tag. Within each `<data-series>` tag, there can be any number of `<x-data>` tags (these tags represent the X-values for that series). If there are no `<x-data>` tags in any `<data-series>` tag, a single X-array is generated, starting at 1 and proceeding in increments of 1.

If only one series has `<x-data>` tags, then that list of X-data is used for all series. If more than one series has `<x-data>` tags, those tags are used only for the series in which they are located.

Within each `<data-series>` tag, there must be at least one `<y-data>` tag (generally there will be many). `<y-data>` tags represent the Y-values for that series.

If the number of X-values and Y-values do not match within one series, the one with the fewer number of values is padded out with `Hole` values.

Here is an example of an XML data file specifying data by series.

```
<?xml version="1.0"?>
<!DOCTYPE chart-data SYSTEM "JCChartData.dtd">
<chart-data Name="My Chart" Hole="MAX">
    <data-point-label>Point Label 1</data-point-label>
    <data-point-label>Point Label 2</data-point-label>
    <data-point-label>Point Label 3</data-point-label>
    <data-point-label>Point Label 4</data-point-label>
    <data-series>
        <data-series-label>Y Axis #1 Data</data-series-label>
        <x-data>1</x-data>
        <x-data>2</x-data>
        <x-data>3</x-data>
        <x-data>4</x-data>
        <y-data>1</y-data>
        <y-data>2</y-data>
        <y-data>3</y-data>
        <y-data>4</y-data>
    </data-series>
    <data-series>
        <data-series-label>Y Axis #2 Data</data-series-label>
        <y-data>1</y-data>
        <y-data>4</y-data>
        <y-data>9</y-data>
        <y-data>16</y-data>
    </data-series>
</chart-data>
```

This format is similar to both the array and the general formats of the default chart data source.

8.8.4 Specifying Data by Point

In the "specifying by point" format, there can be any number of <data-point> tags. Within each <data-point> tag, there can be one optional <data-point-label> tag. Within each <data-point> tag, there can be one optional <x-data> tag (these tags represent the X-value of that point). If there are no <x-data> tags in any of the <data-point> tags, X-values are generated, starting at 1 and then increasing in increments of 1.

If some <data-point> tags have <x-data> tags but others do not, the missing ones will be replaced with Hole values.

Within each <data-point> tag, there must be at least one <y-data> tag (in general, there will be many). <y-data> tags represent the Y-values of each series at this point.

There should always be the same number of <y-data> tags within each <data-point> tag. If there are not, then the largest number of <y-data> tags in any one <data-point> tag is used as the number of series, and the other lists of Y-values will be padded with Hole values.

Here is an example of an XML data file specifying data by point.

```
<?xml version="1.0"?>
<!DOCTYPE chart-data SYSTEM "JCChartData.dtd">
<chart-data Name="MyChart">
<data-series-label>Y Data</data-series-label>
<data-series-label>Y 2 Data</data-series-label>
<data-point>
    <data-point-label>Point Label 1</data-point-label>
    <x-data>1</x-data>
    <y-data>1</y-data>
    <y-data>1</y-data>
</data-point>
<data-point>
    <data-point-label>Point Label 2</data-point-label>
    <x-data>2</x-data>
    <y-data>2</y-data>
    <y-data>4</y-data>
</data-point>
<data-point>
    <data-point-label>Point Label 3</data-point-label>
    <x-data>3</x-data>
    <y-data>3</y-data>
    <y-data>9</y-data>
</data-point>
<data-point>
    <data-point-label>Point Label 4</data-point-label>
    <x-data>4</x-data>
    <y-data>4</y-data>
    <y-data>16</y-data>
</data-point>
</chart-data>
```

This format is similar to the transposed array format of the default chart data source.

8.8.5   Labels and Other Parameters

<data-point-label> and <data-series-label> tags

<data-point-label> and <data-series-label> tags are optional with both the specifying by series or specifying by point methods. If there are more point labels than data points, or more series labels than data series, the extra labels are ignored. If there are more data points than point labels, or more data series than series labels, then the list is padded with blank labels. If there are no point labels or no series labels at all, the chart default is used – no point labels and series labels containing "Series 1", "Series 2", et cetera.

Name and Hole parameters

The *Name* and *Hole* parameters of the JCChartData tag are also optional. *Name* can be any String. *Hole* can be a value, the String MIN (meaning Double.MIN_VALUE) or the String MAX (meaning Double.MAX_VALUE). To represent virtual hole values in an X-data or y-data tag,

use the word `Hole`. Any X-data or y-data tag can contain a value, the String `MIN`, the String `MAX`, or the String `Hole`.

See the "Specifying Data by Series" and "Specifying Data by Point" sections to view these elements in code samples.

8.9   Data Formats

`JCFileDataSource`, `JCURLDataSource`, `JCInputStreamDataSource`, `JCStringDataSource`, and `JCAppletDataSource` all require that data be pre-formatted. The following table illustrates the formatting requirements of data for pre-built data sources. There are two main ways to format data: Array and General.

Array-formatted data shares a single series of X-data among one or more series of Y-data. General-formatted data specifies a series of X-data for every series of Y-data.

Array format is the recommended standard, because it works well with all of the chart types. **General Format may not display data properly in Stacking Bar, Stacking Area, Pie Charts, and Bar Charts.**

Note that for data arrays in Polar charts, (*x*, *y*) coordinates in each data set will be interpreted as (*theta*, *r*). For array data, the X-array will represent a fixed theta value for each point.

In Radar and Area Radar charts, only array data can be used. (*x*, *y*) points will be interpreted in the same way as for Polar charts (above), except that the theta (that is, x) values will be ignored. The circle will be split into `nPoints` segments with `nSeries` points drawn on each radar line.

General format is intended for use in cases where you want to display multiple X-axis values on the same chart.

The following table shows four formatted data examples. An explanation of each element follows.

8.9.1 Formatted Data Examples

Array Data Format (Recommended)

```
ARRAY 2 3                       # 2 series of 3 points
HOLE 10000                      # Use only if custom hole value needed
'Point 0' 'Point 1' 'Point 2'   # Optional Point-labels
# X-values common to all points
          1.0    2.0    3.0
# Y-values
'Series 0'  50.0  75.0  60.0    # Series-label is optional
'Series 1'  25.0  10.0  50.0
```

Transposed Array Data Format (same data as previous)

```
ARRAY 2 3 T                     # 2 series of 3 points, Transposed
HOLE 10000
   ''                 'Series 0' 'Series 1'   # Optional Series-labels
#           X-values  Y0-values  Y1-values
'Point 0'   1.0       50.0       25.0      # Point-labels are optional
'Point 1'   2.0       75.0       10.0
'Point 2'   3.0       60.0       50.0
```

General Data Format (Use if X-data is different for each series)

```
GENERAL 2 4             # 2 series, max 4 points in each
HOLE -10000             # Use only if custom hole value needed
'Series 0' 2            # 2 points, optional series label
  1.0  3.0              #   X-values
 50.0 60.0              #   Y-values
'Series 1' 4            # 4 points
  2.0   2.5   3.5   5.0 #   X-values
 45.0  60.0  HOLE  70.0 #   Y-values, including data hole
```

Transposed General Data Format (same data as previous)

```
GENERAL 2 4 T           # 2 series, max 4 points in each, Transposed
HOLE -10000
'Series 0' 2            # 2 points, optional series label
# X    Y
 1.0 50.0
 3.0 60.0
'Series 1' 4            # 4 points
# X    Y
 2.0 45.0
 2.5 60.0
 3.5 HOLE
 5.0 70.0
```

8.9.2 Explanation of Format Elements

Initialization – Data Layout, Data Size, Hole Value

The first (non-comment) line must begin with either "ARRAY" or "GENERAL" followed by

---

two integers specifying the number of series and the number of points in each series. For example:

```
# This is an Array data file containing 2 series of 4 points
ARRAY 2 4
```

The only difference with General data is that the second integer specifies the *maximum* number of points possible for each series:

```
# A General data file, 5 series, maximum 10 points
GENERAL 5 10
```

The second line can *optionally* specify a data hole value. A hole value is the number that is interpreted by the chart as missing data. There should be only one hole value per `ChartDataView` class. Use a hole value if you know that a particular value in the data should be ignored in the chart:

```
HOLE 10000
```

You can also indicate that any particular point is a hole by specifying the word "HOLE" for that X- or Y-value. For example:

```
50.0 75.0 HOLE 70.0
```

Note: If the hole value is later changed in the data view, values in the X- and Y-data previously set with hole values will not change their values and will now draw.

Adding Comments
You can use comments throughout the data file to make it easier for people to understand. Any text on a line following a "#" symbol are treated as comments and are ignored.

Point Labels
The third line can *optionally* specify text labels for each data point, which can be used to annotate the X-axis. Point-labels are generally only useful with Array data; if specified for General data they apply to the first series. The following shows how to specify Point-labels:

```
'Point 1' 'Point 2' 'Point 3'   # Optional Point-labels
```

The Data – Array layout
The rest of the file contains the data to be charted. Array layout uses the first line of data as X-values that are common to all points. Subsequent lines specify the Y-values for each data series:

```
1.0 2.0 3.0 4.0             # X-values
150.0 175.0 160.0 170.0    # Y-values, series 0
125.0 100.0 225.0 300.0    # Y-values, series 1
# Y-values continue, until end of data
```

The Data – General layout
General layout provides more flexibility. For each series, the first line of data specifies the number of points in the series (this cannot be greater than the maximum number of

points defined earlier). The second line specifies the X-values for that series; the third line specifies the Y-values:

```
4                    # Series 0, 4 points
50.0 75.0 60.0 70.0  # X-values
25.0 10.0 25.0 30.0  # Y-values
# Next series follows, until end of data
```

Series Labels

You can *optionally* specify text labels for each series, which can be displayed in the legend. Series labels are enclosed in single-quotes. In Array data, the label appears at the start of each line of Y-values, for example:

```
'Series label' 150.0 175.0 160.0 170.0    # Y-values, series 0
```

In General data, the label appears at the start of the line defining the number of points in that series, for example:

```
'Series label' 4     # Series 0, 4 points
50.0 75.0 60.0 70.0  # X-values
25.0 10.0 25.0 30.0  # Y-values
```

Transposed Data

JClass Chart can also interpret transposed data, where the meaning of the data series and points is switched. Note that transposing data also transposes series and point labels. To indicate that the data is transposed, add a "T" to the first line specifying the data layout and size. The following illustrates how data is interpreted when transposed:

```
ARRAY 2 3 T
# X-values   Y0-values   Y1-values
1.0          150.0       125.0
2.0          175.0       100.0
3.0          160.0       225.0
```

8.10   Data Binding: Specifying Data from Databases

In order to chart data from a database, your application must be able to establish a connection, perform necessary queries on the data, and then put the data into a chartable format.

This type of database connectivity is often called 'data binding' and components that can be connected to a database are considered 'data bound'. JClass Chart is a data bound component.

Perhaps the easiest way to bind a chart to a database is to use one of the data binding Beans (DSdbChart or JBdbChart) in an IDE or the BeanBox. There are Beans for connecting to a database using Borland JBuilder and the JClass DataSource. See the Bean Reference for complete details on using these Beans in an IDE.

More complex chart features, however, can only be accessed programmatically. To do data binding programmatically, you can use one of the solutions listed in the table below:

Class	Use with:
JCChart	■ JDBCDataSource ■ An application that provides connection to database and passes an SQL result set to JCDBCDataSource
DSdbChart	■ JClass DataSource component
JBdbChart	■ Borland JBuilder 3.0+ components

The following sections provide a brief outline of these different data binding methods.

8.10.1   Data Binding using JDBCDataSource

JDBCDataSource is not a full data binding solution. It is a data source that you can use to chart data from an SQL Result Set. It does not perform any binding operations such as connecting to or querying the database. You will have to provide that functionality.

To use it, you just attach an instance of JDBCDataSource to your chart and pass it a Result Set from your application, as follows:

```
chart.getDataView(0).setDataSource(new JDBCDataSource(resultSet));
```

8.10.2   Data Binding with JBuilder

`JBdbChart` allows you to bind to JBuilder's DataSet, for a full data binding solution. The following example illustrates how to connect to the necessary JBuilder components:

```
package examples.chart.db.jbuilder;

import java.awt.*;
import javax.swing.JFrame;
import com.borland.dx.sql.dataset.*;

import com.klg.jclass.chart.db.jbuilder.*;
import com.klg.jclass.chart.db.DataBindingConfigWrapper;

/**
 * This file was generated using JBuilder data binding. It is intended
 * to demonstrate the code generated when using JBuilder's
 * QueryDataSet and JBdbChart.
 *
 * (Code has been reindented to conform to Quest Software coding standard.)
 */
public class JBuilderDBChart extends JFrame {

Database database1              = new Database();
QueryDataSet queryDataSet1      = new QueryDataSet();
JBdbChart jBdbChart1            = new JBdbChart();

public JBuilderDBChart()
{
    try {
        jbInit();
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}

private void jbInit() throws Exception
{
    QueryDescriptor qd =
        new QueryDescriptor(database1,
            "SELECT OrderDetails.OrderDetailID,OrderDetails.OrderID," +
            "OrderDetails.ProductID,OrderDetails.DateSold," +
            "OrderDetails.Quantity,OrderDetails.UnitPrice," +
            "OrderDetails.SalesTax,OrderDetails.LineTotal " +
            "FROM OrderDetails",
            null, true, Load.ALL);
    queryDataSet1.setQuery(qd);
    ConnectionDescriptor cd=
        new ConnectionDescriptor("jdbc:odbc:JClassDemo", "dba", "sql",
                                 false, sun.jdbc.odbc.JdbcOdbcDriver");
    database1.setConnection(cd);
    jBdbChart1.setDataSet(queryDataSet1);
    DataBindingConfigWrapper cw=
        new DataBindingConfigWrapper(false, 0, 100, "OrderDetailID",
                                     new String[]{"UnitPrice",
```

```
                                                      "SalesTax"});
    jBdbChart1.setDataBindingConfig(cw);
    this.getContentPane().add(jBdbChart1, BorderLayout.NORTH);
}
public static void main(String args[])
{
    JBuilderDBChart f = new JBuilderDBChart();
    f.pack();
    f.show();
}
}
```

8.10.3   Data Binding with JClass DataSource

JClass DataSource is a full data binding solution. It is a robust hierarchical, multiple-platform data source that you can use to bind and query any JDBC compatible database. It can also bind to platform-specific data solutions in JBuilder.

JClass DataSource is available only in JClass DesktopViews (which also contains JClass Chart, JClass Chart 3D, JClass Elements, JClass Field, JClass HiGrid, JClass JarMaster, JClass DataSource, and JClass PageLayout). Visit *http://www.quest.com* for information and downloads.

To bind a chart to a database through JClass DataSource, use `DSdbChart`.

The following example illustrates the main parts of binding with `DSdbChart`:

```java
package examples.chart.db.datasource;
//JDK specific
import java.awt.BorderLayout;
import java.awt.Event;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

import javax.swing.JPanel;
import javax.swing.JFrame;

//JClass datasource specific
import com.klg.jclass.datasource.TreeData;
import com.klg.jclass.datasource.swing.DSdbJNavigator;
import examples.datasource.jdbc.DemoData;

//JClass Chart specific
import com.klg.jclass.chart.JCAxis;
import com.klg.jclass.chart.EventTrigger;
import com.klg.jclass.chart.db.datasource.DSdbChart;

import com.klg.jclass.util.swing.JCExitFrame;

public class DataBoundChart extends JPanel {

protected DSdbChart chart = null;
protected DSdbJNavigator navigator = null;
protected TreeData treeData = null;
protected int currentRow = 0;

public DataBoundChart() {
    setLayout(new BorderLayout());

    // Create DataSource data-bound Chart instance
    chart = new DSdbChart();
    // Chart formatting
    makeAFancyChart();

    // Create DataSource instance
     treeData = new DemoData();

    // Connect Chart instance to DataSource instance
    chart.setDataSource(treeData, "Orders|OrderDetails");
    // Select point label column from DataSource meta data
    chart.setPointLabelsColumn("OrderDetailID");
    chart.setName("Order Details");

    navigator = new DSdbJNavigator();
    navigator.setDataBinding(treeData, "Orders");

    add(navigator, BorderLayout.SOUTH);
    add(chart, BorderLayout.CENTER);
}

/**
```

```
 * Setting some of the chart parameters to make it look fancy
 */
protected void makeAFancyChart()
{
    JCAxis xAxis = chart.getChartArea().getXAxis(0);
    xAxis.setAnnotationMethod(JCAxis.POINT_LABELS);
    chart.getLegend().setVisible(true);
    chart.setForeground(java.awt.Color.yellow);
    chart.setBackground(java.awt.Color.gray);
    chart.getDataView(0).setChartType(DSdbChart.STACKING_AREA);
    chart.getHeader().setVisible(true);
    chart.getFooter().setVisible(true);

    String name = "com.klg.jclass.chart.customizer.ChartCustomizer";
    chart.setCustomizerName(name);
    chart.setAllowUserChanges(true);
    chart.setTrigger(0, new EventTrigger(Event.META_MASK,
                                         EventTrigger.CUSTOMIZE));
}

/**
 * main function
 */
public static void main(String[] args)
{
    DataBoundChart dbChart = new DataBoundChart();
    JCExitFrame frame = new JCExitFrame("This is a data bound chart");

    frame.getContentPane().add(dbChart);
    frame.pack();
    frame.setSize(500, 400);
    frame.show();
}

}
```

8.11   Making Your Own Chart Data Source

8.11.1   The Simplest Chart Data Source Possible

In order for a data source object to work with JClass Chart, it must implement the
ChartDataModel interface. The EditableChartDataModel interface is an extension of
ChartDataModel and can be used when you want to allow the data source to be editable.
The LabelledChartDataModel and the HoleValueChartDataModel interfaces can be used
in conjunction with ChartDataModel to extend the functionality of ChartDataModel to
allow for label values (via the LabelledChartDataModel interface) and hole values (via the
HoleValueChartDataModel interface).

The `ChartDataModel` interface is intended for use with existing data objects. It allows Chart to ask the data source for the number of data series, and the X-values and y-values for each data series. The interface looks like this:

```
public double[] getXSeries(int index);
public double[] getYSeries(int index);
public int getNumSeries();
```

Basically, JClass Chart organizes data based on data series. Each series has X-values and y values, returned by `getXSeries()` and `getYSeries()`, respectively. It is expected that, for a given series index, the X-series and Y-series will be the same length.

If the X-data is the same for all Y-data, then the same X-series can be returned for each value. JClass Chart will automatically re-use the array.

As an example, consider `SimplestDataSource` in the *examples.chart.datasource* example:

```
/**
 * This example shows the simplest possible chart data source.
 * The data source contains two data series, held in "xvalues"
 * and "yvalues" below.
 */
public class SimplestDataSource extends JPanel implements ChartDataModel {

// X values for chart.
protected double xvalues[] = { 1, 2, 3, 4 };
// Y values.
protected double yvalues[][] = { {20, 10, 30, 25}, {30, 22, 10, 40}};


/**
 * Retrieves the specified x-value series
 * In this example, the same x values are used regardless of
 * the index.
 * @param index data series index
 * @return array of double values representing x-value data
 */
public double[] getXSeries(int index) {
    return xvalues;
}
/**
 * Retrieves the specified y-value series
 * In this example, yvalues contains the y data.
 * @param index data series index
 * @return array of double values representing x-value data
 */
public double[] getYSeries(int index) {
    return yvalues[index];
}

/**
 * Retrieves the number of data series.
 * In this example, there are only two data
 * series.
 */
public int getNumSeries() {
    return yvalues.length;
}
```

There are two series in this example. The X-data is repeated for both series, and is stored in an array of doubles (xvalues). The Y-data is stored in an array of arrays of doubles (yvalues). Each sub-array is the same length as xvalues.

Note: You can run this example from
JCLASS_HOME/Examples/Chart/DataSource/SimplestDataSource.

8.11.2   LabelledChartDataModel – Labelling Your Chart

Sometimes it is important to label each data series and each point in a graph.
This information can be added to a data source using the `LabelledChartDataModel`
interface.

The `LabelledChartDataModel` interface allows specification of series and point labels for
your data. It is an optional part of the chart data model, but is very commonly used:

```
public int getNumSeries();
public String[] getPointLabels();
public String[] getSeriesLabels();
public String getDataSourceName();
```

The `getPointLabels()` call returns the point labels for all points in the chart. The size of
the String array should correspond with the number of items in the XSeries and YSeries
arrays.

The `getSeriesLabels()` call returns the series labels for the chart. The size of the String
array should correspond to the value returned by `getNumSeries()`. Series labels appear in
the legend.

The `getDataSourceName()` returns the name of the data source. This appears in the chart
as the title of the legend.

As an example, consider `LabelledDataSource` in *JCLASS_HOME/examples/chart/datasource/.*

```
/**
 * This example shows how to add point and series labelling
 * to a data source. It extends SimplestDataSource and
 * implements the LabelledChartDataModel interface to add
 * this information. The result can be seen on the X-axis
 * (point labels representing quarters) and in the legend
 * (title, series names).
 */
public class LabelledDataSource extends SimplestDataSource implements
LabelledChartDataModel {

// Point labels
protected String pointLabels[] = { "Q1", "Q2", "Q3", "Q4" };

// Series labels
protected String seriesLabels[] = { "West", "East" };

/*
 * Retrieves the labels to be used for each point in a
 * particular data series.
 * @return array of point labels
 */
public String[] getPointLabels() {
    return pointLabels;
}
/**
 * Retrieves the labels to be used for each data series
 */
public String[] getSeriesLabels() {
    return seriesLabels;
}

/**
 * Retrieves the name for the data source
 */
public String getDataSourceName() {
    return "Sales By Region";
}
```

As noted, this data source extends `SimplestDataSource`, adding in the required methods for returning point labels – `getPointLabels()` – and series labels –`getSeriesLabels()`.

Note that the number of items in the array returned by `getSeriesLabels()` should equal the number returned by `getNumSeries()`.

Note that the number of items in the array returned by `getPointLabels()` should equal the number of items in the array returned by `getXSeries()` and `getYSeries()`. (In cases where the X-data is unique for each series and each series has a possibly different number of points, the point labels are applied to the first series.)

Note: You can run this example from *JCLASS_HOME/Examples/Chart/DataSource/LabelledDataSource.*

8.11.3   EditableChartDataModel – Modifying Your Data

If you want to allow users to modify data using the edit trigger in JClass Chart, your data source must implement `EditableChartDataModel`. The `EditableChartDataModel` interface extends `ChartDataModel`, adding a single method that allows Chart to modify data in the data source:

```
public boolean setDataItem(int seriesIndex, int pointIndex,
   double newValue);
```

The `seriesIndex` and `pointIndex` values are used to save the data sent in `newValue`. Note that `EditableChartDataModel` only allows for Y-values to be changed. In other words, `newValue` is a Y-value!

As an example, consider `EditableDataSource` in *JCLASS_HOME/examples/chart/datasource/*.

```
/**
 * This example shows how to make a data source editable
 * by adding the EditableChartDataModel interface to
 * the object.
 */
public class EditableDataSource extends LabelledDataSource implements
EditableChartDataModel {

/**
 * Change the specified y data value.
 * In this example, the series and point indices index
 * into the yvalues array originally defined in SimplestDataSource.
 *
 * @param seriesIndex series index for the point to be changed.
 * @param pointIndex point index for the point to be changed.
 * @param newValue new y value for the specified point
 * @return boolean value indicating whether the new value was
 * accepted. "true" means value was accepted.
 */
public boolean setDataItem(int seriesIndex, int pointIndex, double
                          newValue) {
   if (newValue < 0) return false;
   yvalues[seriesIndex][pointIndex] = newValue;
   return true;
}
```

In this example, the value is saved back into the yvalues array from `SimplestDataSource`, using the `seriesIndex` and `pointIndex` values to index to the appropriate array member.

This example extends `LabelledDataSource`, adding the `setDataItem()` method to allow chart to modify the data in the data source.

Note: You can run this example from *JCLASS_HOME/Examples/Chart/DataSource/SimplestDataSource.*

8.11.4   HoleValueChartDataModel – Specifying Hole Values

If you want to supply a specific hole value along with your data, your data source must implement the HoleValueChartDataModel interface.

As noted in Section 8.9.2, Explanation of Format Elements, a hole value is a particular value in the data that should be ignored by the chart. There should be only one hole value per data source.

The HoleValueChartDataModel interface has one method, getHoleValue(). This method retrieves the hole value for the data source.

Note: The default hole value is Double.MAX_VALUE.

8.12   Making an Updating Chart Data Source

Quite often, the data shown in JClass Chart is dynamic. This kind of data requires creation of an updating data source. An updating data source is capable of informing chart that a portion of the data has been changed. Chart can then act on the change.

JClass Chart uses the standard AWT/Swing event/listener mechanism for passing changes between the chart data source and JClass Chart. At a very high level, JClass Chart is a listener to data source events that are fired by the data source.

8.12.1   Chart Data Source Support Classes

There are a number of data source related support classes included with JClass Chart. These classes make it easier to build updating data sources.

ChartDataEvent and ChartDataListener

The ChartDataListener interface is implemented by objects interested in receiving ChartDataEvents. Most often, the only ChartDataListener is JClass Chart itself. ChartDataEvent and ChartDataListener give data sources away to send update messages to Chart.

The ChartDataListener interface has only one method:

```
public void chartDataChange(ChartDataEvent e);
```

This method is used by the data source to inform the listener of a change. *In most systems, only JClass Chart need implement this interface.*

The ChartDataEvent object has three immutable properties: Type, SeriesIndex, and PointIndex. SeriesIndex and PointIndex are used to specify the data affected by the posted change. If all data is affected, the enum values ALL_SERIES and ALL_POINTS can be used.

Type is used to specify the kind of update:

Message	Meaning
ADD_SERIES	A new data series has been added to the end of the existing series in the data source.
APPEND_DATA	Used in conjunction with the FastUpdate feature, this tells the listener that data has been added to the existing series. Please see FastUpdate, in Chapter 10, for full details.
CHANGE_CHART_TYPE	A request from the data source to change the chart type. The chart type is held inside seriesIndex.
INSERT_SERIES	A new data series has been added; seriesIndex indicates where the series should be added.
RELOAD	The data has completely changed; the difference here is that the dimensions of the data source (that is, number of data series and number of points) has not changed.
RELOAD_ALL_POINT_LABELS	Tells the listener to reload all point labels.
RELOAD_ALL_SERIES_LABLES	Tells the listener to reload all series labels.
RELOAD_DATA_SOURCE_NAME	Tells the listener the data source name has changed.
RELOAD_POINT	Single data value has changed, as specified by seriesIndex and pointIndex.
RELOAD_POINT_LABEL	Tells the listener to reload the point label specified by pointIndex.
RELOAD_SERIES	An entire data series has changed, as specified by seriesIndex (pointIndex ignored).
RELOAD_SERIES_LABEL	Tells the listener to reload the series label specified by seriesIndex.
REMOVE_SERIES	Removes the series at seriesIndex.
RESET	The data source has completely changed.

ChartDataManageable and ChartDataManager

This interface is used by a data source to tell Chart that it will be sending ChartDataEvents to Chart. Without this interface, there is no way for Chart to know that it has to attach itself as a ChartDataListener to the data source.

The only method in ChartDataManageable returns a ChartDataManager:

```
public abstract ChartDataManager getChartDataManager();
```

A `ChartDataManager` is an object that knows how to register and deregister `ChartDataListeners`. Chart uses this object to register itself as a listener to events from the data source.

The quickest way to get a data source set up is to extend or use `ChartDataSupport`.

ChartDataSupport

`ChartDataSupport` provides a default implementation of `ChartDataManager`. It will manage a list of `ChartDataListeners`. It also provides two convenience methods for firing events to the listeners:

```
public void fireChartDataEvent(int type, int seriesIndex, int
    pointIndex)
public void fireChartDataEvent(ChartDataEvent evt)
```

The first method listed above is the most convenient. Given a `ChartDataEvent Type`, `SeriesIndex`, and `PointIndex`, it constructs and fires a `ChartDataEvent` to all listeners. The second method requires that you construct the `ChartDataEvent` yourself.

Creating an Updating Data Source

If your datasource either extends or contains `ChartDataSupport`, sending updates from the data source to the chart is easy. Simple call `fireChartDataEvent()` with the event you wish to send.

```
fireChartDataEvent(ChartDataEvent.RESET, 0, 0);
```

To have JClass Chart automatically added as a listener, your data source needs to implement the `ChartDataManageable` interface and to return the `ChartDataSupport` instance in the `getChartDataManager()` method.

Chart Data Source Hierarchy

Data Source:
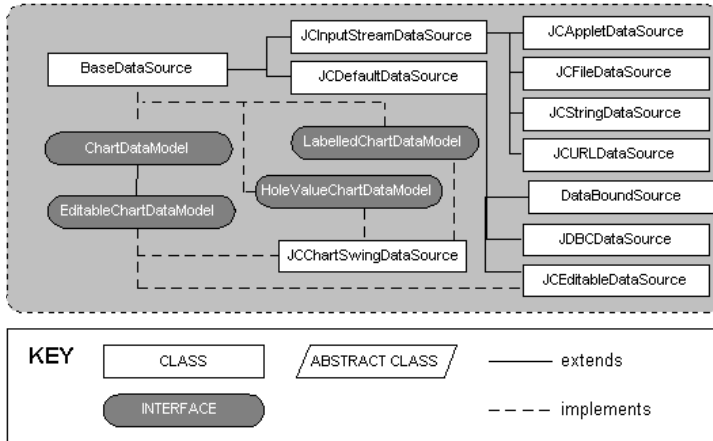
9

Text and Style Elements

This chapter describes the different formatting elements available within JClass Chart, and how they can be used. If you are developing your chart application using one of the JClass Chart Beans, please refer to Bean Reference, in Chapter 4.

9.1   Header and Footer Titles

A chart can have two titles, called the header and footer. By default they are `JLabel` instances and behave accordingly (A `JLabel` class is a Swing class.) A `JLabel` object can display text, an image, or both.

You can specify where in the label's display area the label's contents are aligned by setting the vertical and horizontal alignment. By default, labels are vertically centered in their display area. Text-only labels are left-aligned by default. Image-only labels are horizontally centered by default.

A title consists of one or more lines of text with an optional border, both of which you can customize. You can also set the text alignment, positioning, colors, and font used for the header or footer.

See "How to Use Labels" in the Java Tutorial for further documentation.

9.2   Legends

A legend shows the visual attributes (or `ChartStyle`) used for each series in the chart, with text that labels the series. You can customize the series label and positioning. The legend is a `JComponent`, and all properties such as border, colors, font, and so on, apply.
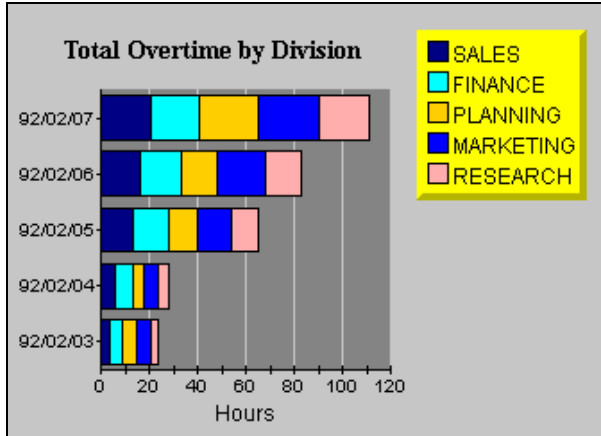
Figure 32   Vertically oriented legend anchored NorthEast.

New Location for the Legend Classes

In order to make the legend classes more accessible to the JClass products, all four legend classes – `JCLegend`, `JCGridLegend`, `JCMultiColLegend`, and `JCLegendItem` – have been moved from *com.klg.jclass.chart* to *com.klg.jclass.util.legend.*

How will this affect you?

For most users, all you will need to do when converting from 4.0.x to 4.5 and higher, is to change the import statements to import the legend classes (`JCLegend`, `JCGridLegend`, `JCMultiColLegend`, and `JCLegendItem`) from the *com.klg.jclass.util.legend* package.

This converting can be done either by hand or via running the provided porting script (please see the second bullet). Each method will yield the same result.

■   **Convert the import statements by hand** by changing these import statements

```
import com.klg.jclass.chart.JCLegend;
import com.klg.jclass.chart.JCGridLegend;
import com.klg.jclass.chart.JCMultiColLegend;
import com.klg.jclass.chart.JCLegendItem;
import com.klg.jclass.chart.*;
```

to

```
import com.klg.jclass.util.legend.JCLegend;
import com.klg.jclass.util.legend.JCGridLegend;
import com.klg.jclass.util.legend.JCMultiColLegend;
import com.klg.jclass.util.legend.JCLegendItem;
import com.klg.jclass.chart.*; import com.klg.jclass.util.legend.*;
```

■ **Convert the import statements via running the porting script**. This Perl script changes the above import statements automatically. The porting script is named *legend4to45.pl* and is provided at *JCLASS_HOME/bin/*. You must have Perl installed on your system for the script to work.

Here is an example of how one would run the script:

```
perl legend4to45.pl filename
```

where *filename* is the name of the file you want to convert from 4.0.x to 4.5 or higher.

As noted, for most users the above changes to the import statements are all that is required when converting from 4.0.x to 4.5 or higher. However, for users who are already overriding JCLegend and implementing custom layouts, converting may require dealing with changes to JClass Chart's JCLegend and JCLegendItem classes. These items are **not** covered by the porting script so will **need to be done manually**.

1. These fields have been added to JCLegendItem:

■ int drawType (determines drawing type; takes as its parameter one of JCLegend.NONE, JCLegend.BOX, JCLegend.IMAGE, JCLegend.IMAGE_OUTLINED, JCLegend.CUSTOM_SYMBOL, or JCLegend.CUSTOM_ALL)

■ Object itemInfo (refers to data related to this legend item – in JClass Chart, this is a JCDataIndex object containing the data view and series to which this legend item is related)

2. The new Object itemInfo field *replaces* these three fields:

■ ChartDataView view (the view associated with the ChartDataView)

■ ChartDataViewSeries series (the series associated with the ChartDataViewSeries)

■ int seriesIndex (the series index associated with the ChartDataViewSeries)

3. JCLegend's drawLegendItem(Graphics gc, JCChart chart, Font useFont, JCLegendItem thisItem) has been changed to:

■ drawLegendItem(Graphics gc, Font useFont, JCLegendItem thisItem)

Legend Text and Orientation

The legend displays the text contained in the Label property of each Series in a DataView. The VisibleInLegend property of the series determines whether the Series will appear in the Legend.

`SeriesLabels` support the use of HTML tags. The use of HTML tags overrides the default `Font` and `Color` properties of the label. Please note that HTML labels may not work with PDF, PS, or PCL encoding.

Use the legend `Orientation` property to lay out the legend horizontally or vertically.

Legend Positioning

Use the legend `Anchor` property to specify where to position the legend relative to the `ChartArea`. You can select from eight compass points around the `ChartArea`.

See Section 9.9, Positioning Chart Elements, for more information.

9.2.1　Customizing Legends

JClass provides two types of legend objects: `JCGridLegend` (the default) for a single-column layout and `JCMultiColLegend` for a multiple-column layout. If these legends do not provide the desired functionality, the user can customize the legend using the JCLegend Toolkit.

Single-Column Legends

The classic single-column legend layout is provided by `JCGridLegend`. This is the default layout in JClass Chart.

Multi-Column Legends

Multi-column legend layout is available using `JCMultiColumnLegend`. To designate this layout, follow these steps:

1. Create an instance.
2. Set the number of rows and columns.
3. Set the legend property of the JClass Chart to this instance to create a multi-column legend.

Multi-Column Legends example

```
JCMultiColLegend mcl = new JCMultiColLegend();
mcl.setNumColumns(2);
c.setLegend(mcl);
```

This example will create a legend for the current chart that has two columns. The number of rows depends on the number of items in the legend. To fix the number of rows, use `setNumRows()`. Both the number of rows and the number of columns are variable by default.

To reset the number of rows and columns to a variable state after they have been fixed, call the appropriate set method with a negative value. If both the `NumRows` and `NumColumns` properties are set to fixed values, the legend will be of that exact size and will ignore any extra items.

JCLegend Toolkit

The JCLegend Toolkit allows you the freedom to design your own legend implementations. The options range from simple changes, such as affecting the order of the items in the legend, to providing more complex layouts.

The JCLegend Toolkit consists of a `JCLegend` class that can be subclassed to provide legend layout rules and two interfaces: `JCLegendPopulator` and `JCLegendRenderer`. `JCLegendPopulator` is implemented by classes wishing to populate a legend with data, and `JCLegendRenderer` is implemented by a class that wishes to help render the legend's elements according to the user's instructions. Examples of how to use the JCLegend Toolkit are provided in *JCLASS_HOME/examples/chart/legend/* .

`JCChartLegendManager` is the class used by JClass Chart to implement both the `JCLegendPopulator` and `JCLegendRenderer` interfaces, and to provide a built-in mechanism for itemizing range objects in a legend.

Custom Legends – Layout

JClass provides a Legend Toolkit that allows creation of custom legend implementations. `JCLegend` is an abstract class that can be subclassed by users wishing to customize the legend layout or other legend behavior.

To provide a custom layout, override the method:

```
public abstract Dimension layoutLegend(List itemList, boolean
      vertical, Font useFont)
```

The `itemlist` argument is a `List` containing a `Vector` for each data view contained in the chart. Each of these sub-vectors contains one `JCLegendItem` instance for each series in the data view and one instance for the data view title.

The `vertical` argument is `true` if the orientation of the legend is vertical, and `false` if the orientation of the legend is horizontal.

The `useFont` argument contains the default font to use for the legend.

Each item in the legend consists of a text portion and a symbol portion. For example, in a Plot Chart, the text portion is the name of the series, and is preceded by the symbol used to mark a point on the chart. For the title of the data view, the text portion is the name of the data view and there is no symbol.

`JCLegendItem` is a class that encapsulates an item in the legend with the properties.

Property name	Description
`Point pos;`	Position of this legend item within the legend.
`Point symbolPos;`	Position of the symbol within the legend item.
`Point textPos;`	Position of the text portion within the legend item.
`Dimension dim;`	Full size of the legend item.

Property name	Description
`Dimension symbolDim;`	Size of the symbol; provided by `JCLegend`.
`Dimension textDim;`	Size of the text portion; provided by `JCLegend`.
`Rectangle pickRectangle;`	The rectangle to use for pick operations; optional.
`int drawType;`	Determines drawing type; one of `JCLegend.NONE`, `JCLegend.BOX`, `JCLegend.IMAGE`, `JCLegend.IMAGE_OUTLINED`, `JCLegend.CUSTOM_SYMBOL`, or `JCLegend.CUSTOM_ALL`.
`Object itemInfo`	Data related to this legend item. In JClass Chart, this is a `JCDataIndex` object containing the data view and series to which the legend item is related.
`Object symbol;`	The symbol if other than the default type; usually null (means `drawLegendItem` decides).
`Object contents;`	The text portion; a `String` or `JCString`.

When the `itemList` is passed to `layoutLegend`, it has been filled in with `JCLegendItem` instances representing each data series and data view title. These instances will have the `symbolDim`, `textDim`, `symbol`, `contents`, `itemInfo`, and `drawType` already filled in.

The value of `drawType` will determine whether a particular default symbol type will be drawn or whether user-provided drawing methods will be called.

The `layoutLegend()` method is expected to calculate and fill in the pos, symbolPos, textPos, and dim fields. Additionally, the method must return a `Dimension` object containing the overall size of the legend. Optionally, it may also calculate the `pickRectangle` member of the `JCLegendItem` class. The `pickRectangle` is used in pick operations to specify the region in the legend that is associated with the series that this legend item represents. If left null, a default `pickRectangle` will be calculated using the dim and pos members.

Any of the public methods in the `JCLegend` class may be overridden by a user requiring custom behavior. One such method is:

```
public int getSymbolSize()
```

`getSymbolSize()` returns the size of the legend-calculated symbols to be drawn in the legend. Default `JCLegend` behavior sets the symbol size to be equal to the ascent of the default font that is used to draw the legend text. It is overridable by users who wish to use a different symbol size. One possible implementation is to use a symbol size identical to that which appears on the actual chart.

Custom Legends – Population

JCLegendPopulator is an interface that can be implemented by any user desiring to populate the legend with custom items. This interface comprises two methods that need to be implemented:

```
public List getLegendItems(FontMetrics fm)
public boolean isTitleItem(JCLegendItem item)
```

getLegendItems() should return a List object containing any number of Vector objects where each Vector object represents one column in the legend. Each Vector object contains the JCLegendItem objects for that column. In JClass Chart, each column generally represents one data view.

isTitleItem() should return true or false, depending on whether the passed JCLegendItem object represents a title for the column. This is used to determine whether a symbol is drawn for a particular legend item.

If implemented, the legend should be notified of the new populator with the setLegendPopulator() method of JCLegend.

Custom Legends – Rendering

JCLegendRenderer is an interface that can be implemented by any user desiring to custom render legend items. This interface consists of four methods that need to be implemented:

```
public void drawLegendItem(Graphics gc, Font useFont,
        JCLegendItem thisItem)
public void drawLegendItemSymbol(Graphics gc, Font useFont,
        JCLegendItem thisItem)
public Color getOutlineColor(JCLegendItem thisItem)
public void setFillGraphics(Graphics gc, JCLegendItem thisItem)
```

JCLegendRenderer also has the capacity to implement custom text objects for drawing, and is called when the legend cannot interpret an object placed in the contents field of the JCLegendItem. This interface consists of one method that needs to be implemented:

```
void drawLegendItemText (Graphics gc, Font useFont, JCLegendItem
        thisItem);
```

drawLegendItem() provides a way for a user to define a custom drawing routine for an entire legend item. It is called when a legend item's draw type has been set to JCLegend.CUSTOM_ALL.

drawLegendItemSymbol() provides a way for a user to define a custom drawing routine for a legend item's symbol. It is called when a legend item's draw type has been set to JCLegend.CUSTOM_SYMBOL.

getOutlineColor() should return the outline color to be used to draw the legend item's symbol. If null is returned, the legend's foreground color will be used.
getOutlineColor() is called when a legend item's draw type has been set to either JCLegend.BOX or JCLegend.IMAGE_OUTLINED.

`setFillGraphics()` should set the appropriate fill properties on the provided `Graphics` object for drawing the provided legend item. `setFillGraphics()` is called when the legend item's draw type has been set to `JCLegend.BOX`.

If implemented, the legend should be notified of the new renderer with the `setLegendRenderer()` method of `JCLegend`.

Examples of Simple Custom Legends

The easiest way to perform simple legend customizations is to extend an existing legend. This is clearly demonstrated in the Reversed Legend example in *JCLASS_HOME/examples/chart/legend/.* This example overrides the `JCChartLegendManager` class (the class that implements the `JCLegendPopulator` and `JCLegendRenderer` interfaces in JClass Chart) to reverse the order of the legend items. This class overrides the `getLegendItems()` method, first calling the superclass' method to get the list of legend items and then rearranging the order before returning the newly reversed list of legend items.
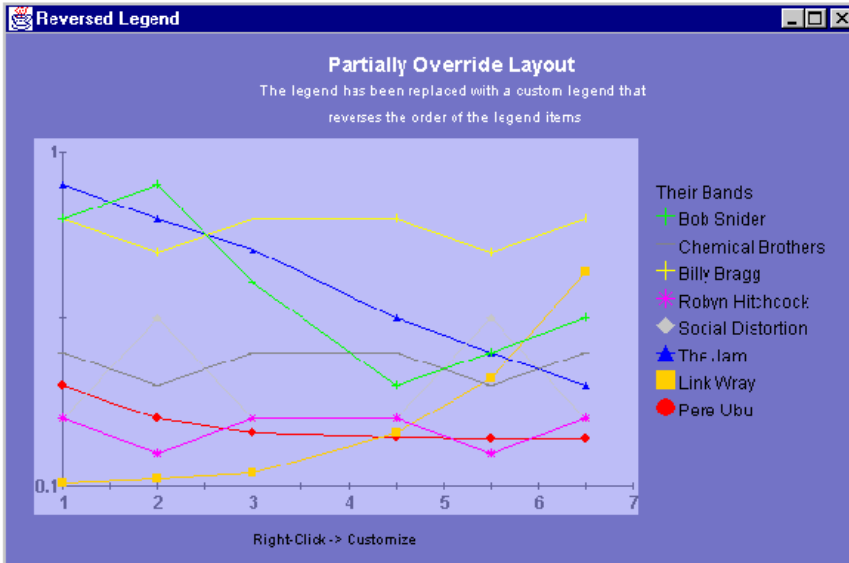
Figure 33   The Reversed Legend example, which extends `JCChartLabelManager` *to reverse the order of the legend items.*

Here's the pertinent code:

```
public ReverseLegend() {

        setLayout(new GridLayout(1,1));

        // replace standard legend with custom legend that reverses
        // the order of the legend items
        JCChart c = new JCChart(JCChart.PLOT);
...
        RevLegendManager legMan = new RevLegendManager(c);
        c.getLegend().setLegendPopulator(legMan);
        c.getLegend().setLegendRenderer(legMan);
        c.getLegend().setVisible(true);
...
}

/** RevLegendManager overrides the standard legend representation
 *  to reverse the drawing order of the legend items. It does this by
 *  overriding getLegendItems() method of the JCChartLabelManager
 *  class to reverse the order of the items in the legend
 *  vector.
 */


class RevLegendManager extends JCChartLegendManager
{

RevLegendManager(JCChart chart)
{
        super(chart);
}
/** Override getLegendItems(). Reverse order of items in legend
 * vector.
 */
public List getLegendItems(FontMetrics fm)
{
        // get the list of legend items from the superclass
        List itemList = super.getLegendItems(fm);

        // reverse the list
        for (int i = 0; i < itemList.size(); i++) {
            List viewItems = (List) itemList.get(i);

            List reverseView = new Vector();
            for (int j = viewItems.size() - 1; j >= 0; j--) {
                JCLegendItem thisItem = (JCLegendItem) viewItems.get(j);

                // reverse items in list, but keep the title at the top.
                if (isTitleItem(thisItem)) {
                    reverseView.add(0, thisItem);
                } else {
                    reverseView.add(thisItem);
                }
            }
            itemList.set(i, reverseView);
```

```
        }
        // now that we've set up the list correctly, let the superclass
        // position it
        return itemList;
    }

}
```

The Separator Legend example in *JCLASS_HOME/examples/chart/legend/* shows how to place a separator between the data view title and the series beneath it. Similar to the Reversed Legend example, the Separator Legend example overrides the JCChartLegendManager class.

In the Separator Legend example, a new JCLegendItem is inserted into the list after the data view title item as part of the layoutLegend() method. This new JCLegendItem has only its textDimension filled in with the size of the separator, but the actual contents field remains null – which is how one recognizes the separator when it is time to draw it.

The drawType field of the JCLegendItem is set to JCLegend.CUSTOM_ALL to ensure that the drawLegendItem() method will be called. Finally, the example returns the item list with the newly added item and lets the superclass do the positioning and sizing calculations.

The drawLegendItem() method is also overridden so that the separator can be drawn. Before drawing, however, it is first determined whether the provided legend item is, indeed, the separator created above.
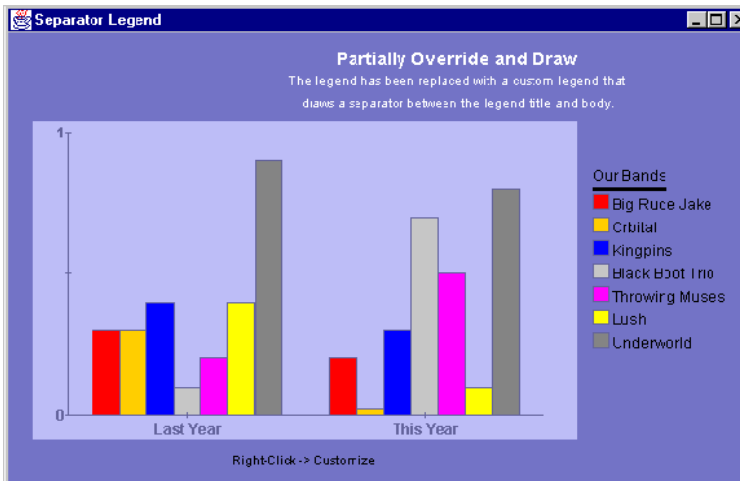
Figure 34   The Separator Legend example places a separator between the data view title and the series beneath it, and extends JCChartLabelManager.

Here's the relevant code:

```
public SeparatorLegend() {

       setLayout(new GridLayout(1,1));

          // replace standard legend with custom legend that draws a
          // separator between the title and the body
        JCChart c = new JCChart(JCChart.BAR);
...
          SepLegendManager sepMan = new SepLegendManager(c);
          c.getLegend().setLegendPopulator(sepMan);
          c.getLegend().setLegendRenderer(sepMan);
       c.getLegend().setVisible(true);
...
}

/** sepLegendManager overrides the standard legend populator and
 * renderer implementations to draw a separator between the legend
 * title and body. It does this by overriding the
 * JCChartLegendManager's getLegendItem() method (to insert an item
 * to take the place of a separator) and drawLegendItem() (to draw
 * the separator) methods.
 */
public class SepLegendManager extends JCChartLegendManager
{

public SepLegendManager(JCChart chart)
{
       super(chart);
}

/** Override getLegendItems() to insert separator item into
 * legend vector.
 */
public List getLegendItems(FontMetrics fm)
{
       // get the list of legend items from the superclass
       List itemList = super.getLegendItems(fm);

       // go through the list to find the spot for the separator
       for (int i = 0; i < itemList.size(); i++) {
           List viewItems = (List) itemList.get(i);

           for (int j = 0; j < viewItems.size(); j++) {
               JCLegendItem thisItem = (JCLegendItem) viewItems.get(j);

               // Insert separator item after title item
               // our separator is identified by having null contents
               // but an existing text dimension. Make the separator as
               // wide as the text portion of the title.

               if (isTitleItem(thisItem)) {
                   JCLegendItem newItem = new JCLegendItem();
                   boolean vertical = chart.getLegend().getOrientation() ==
                       JCLegend.VERTICAL;
```

```
                    if (vertical) {
                        newItem.textDim = new Dimension(thisItem.textDim.
                            width, 3);
                    } else {
                        newItem.textDim = new Dimension(3,
                                                thisItem.textDim.height);
                    }
                    // make sure to set draw type as CUSTOM_ALL so that
                    // drawLegendItem() will be called.
                    newItem.drawType = JCLegend.CUSTOM_ALL;
                    viewItems.add(j+1, newItem);
                    break;
                }
            }
        }

        // now that the list is set up, let the superclass worry about
        // positioning everything
        return itemList;
    }

    /** Override drawLegendItem() to draw the separator item
     * when encountered.
     */
    public void drawLegendItem(Graphics gc, Font useFont,
                                                JCLegendItem thisItem)
    {
            // if our separator, draw it
            if (thisItem.contents == null && thisItem.textDim != null) {
            if (gc.getColor() != getForeground())
                    gc.setColor(getForeground());

                gc.fillRect(thisItem.pos.x + thisItem.textPos.x,
                    thisItem.pos.y + thisItem.textPos.y,
                    thisItem.textDim.width,
                    thisItem.textDim.height);
        }
    }
}
}
```

Remember to use the `setLegendPopulator()` and `setLegendRenderer()` methods of the `JCLegend` class to notify the legend of the new class.

Examples of Complex Legends

More complex customizations are also possible. Legends that require full-scale changes to the rules of layout can override the `JCLegend` class and create their own implementation. Have a look at *JCLASS_HOME/examples/chart/legend/FlowLegend* for an example of a custom legend layout.

9.3   Chart Labels

Chart labels allow you to add more information to your chart. There are static labels that display continuously and interactive labels that pop-up when a cursor moves over a data item. Labels can be attached to different parts of a chart: absolute coordinates, coordinates in the plotting area, or a specific data item. To see a wide range of label uses, browse the demos in the *JCLASS_HOME/demos/chart/labels/* directory.

9.3.1   Label Implementation

JClass Chart contains a list of labels, managed by the `ChartLabelManager`. This property is initially null. By calling `getChartLabelManager()`, JClass Chart will create a manager class with an empty list of labels. When you create a label, you must add it to the manager with `addChartlabel()`. Labels are instances of the `JCChartLabel` class.

9.3.2   Adding Labels to a Chart

Labels are added to a chart in two ways: with the `AutoLabels` property of `ChartDataView`, or by attaching an instance of `JCChartLabel` to a chart element.

Individual labels are attached in three ways: to coordinates on the chart area (`ATTACH_COORD`), coordinates on the plot area (`ATTACH_DATACOORD`), or to a data item (`ATTACH_DATAINDEX`). Interactive labels must use the `ATTACH_DATAINDEX` method.

Each label on the chart below uses a different attachment method. The "Point(100,50)" label, is attached to coordinates originating from the top left corner of the chart area. "Value(2,220)" is attached to axes coordinates, and "Data(Set0,Point2)" is attached to a specific data item.
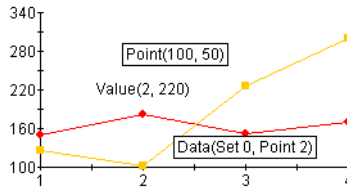
Attaching a Label to a Data Item

To attach a label to a point, bar or slice, set the `AttachMethod` property to `ATTACH_DATAINDEX`. The labels move with the data element; the labels also move when the chart is resized. Note that the points and series are zero-based. The following example puts a label on a chart next to the fourth data point in the second data series.

```
cl = new JCChartLabel("Fourth data point");
cl.setDataIndex(new JCDataIndex(view, series, 1, 3));
cl.setAttachMethod(JCChartLabel.ATTACH_DATAINDEX);
cl.setAnchor(JCChartLabel.AUTO);
chart.getChartLabelManager().addChartLabel(cl)
```

Attaching a Label to Chart Area Coordinates

To attach a label to a point on the chart area, set the `AttachMethod` property to `ATTACH_COORD`. The coordinate origin for this method is the top left corner of the chart area.

```
JCChartLabel cl = new JCChartLabel("Point( 100.50 )");
cl.setAttachMethod( JCChartLabel.ATTACH_COORD );
cl.setCoord( new Point( 100, 50 ) );
chart.getChartLabelManager().addChartLabel(cl)
```

Attaching a Label to Plot Area Coordinates

To attach a label to coordinates on the plot area, set the `AttachMethod` property to `ATTACH_DATACOORD`. The plot area is defined by the chart's X- and Y-axes. The following example places a label in the plot area at X-value 2.5, y-value 160.

```
JCChartLabel cl = new JCChartLabel("Attached to the data coordinate",
false);
cl.setDataCoord( new JCDataCoord( 2.5, 160 ) );
cl.setAnchor( JCChartLabel.NORTH );
cl.setAttachMethod( JCChartLabel.ATTACH_DATACOORD );
cl.setBorderType( Border.ETCHED_OUT );
cl.setBorderWidth( 5 );
chart.getChartLabelManager().addChartLabel(cl)
```

9.3.3   Interactive Labels

You can have labels pop-up as a cursor dwells over a point, bar or slice (a **dwell label**). This allows you to create an interactive chart where information is hidden until the user wants to see it. The `AutoLabel` property will set up a complete series of **dwell labels** for your chart. In the example below, '225' appears on top of the bar as the cursor passes over it, to indicate the value of the bar.
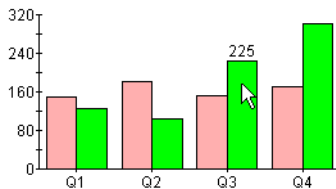
Automatically Generated Labels

The `AutoLabel` property of `ChartDataView` will generate a complete series of dwell labels if set to `true`. It attaches dwell labels to every data index. The following code adds automatic dwell labels to the data:

```
chart.getDataView(0).setAutoLabel(true);
```

Adding Individual Dwell Labels

Attaching an individual dwell label follows the same procedure as attaching a static label to a data item, except that the `DwellLabel` property is set to `true`:

```
cl.setDwellLabel(true);
```

A dwell label can only be used when the `AttachMethod` property is set to `ATTACH_DATAINDEX`.

9.3.4   Adding and Formatting Label Text

`JCChartLabel` is just a holder for any `JComponent`. By default it is a `JLabel` instance, and text can be set the same way you would set text on a `JLabel`. You can access the component portion of the chart label with the `getComponent()` method.

`JLabels` support the use of HTML tags. The use of HTML tags overrides the default `Font` and `Color` properties of the label. Please note that HTML labels may not work with PDF, PS, or PCL encoding.

Adding Label Text
You can add text to a label by passing it to the constructor, or by using the `Text` property. To add text to a label when it is constructed, include the text in the constructor's argument, as follows:

```
JCChartLabel cl = new JCChartLabel("I'm a Label", false);
```

To add text using the `Text` property, use the `setText` method, as follows:

```
((JLabel)cl.getComponent()).setText("I'm a Label");
```

Formatting Label Text
```
Font f = new Font("timesroman", Font.BOLD, 24);
cl.getComponent(),setFont(f)
```

`JComponent` properties such as fonts, borders, colors, an so on, are set in the same manner.

9.3.5   Positioning Labels

The `Anchor` property determines the position of the label, relative to the point of attachment. Valid positions include: `NORTH`, `NORTHEAST`, `NORTHWEST`, `EAST`, `WEST`, `SOUTHEAST`, `SOUTHWEST`, and `SOUTH`. The following example shows the syntax:

```
cl.setAnchor(JCChartLabel.EAST);
```

9.3.6 Adding Connecting Lines

You can add lines that connect a label to its point of attachment. This can help the end-user pinpoint what a label refers to on a chart.
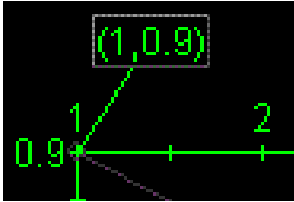
To add a connecting line to a label, set the Connected property to true, as follows:

```
cl.setConnected(true);
```

9.4 Chart Styles

Chart styles define all of the visual attributes of how data appears in the chart, including:

- Lines and points in plots and financial charts.
- Color of each bar in bar charts.
- Slice colors in pie charts.
- Color of each filled area in area charts.

Each series in a data view has its own JCChartStyle object; as new series are added, new JCChartStyle objects are created automatically by the chart. JClass Chart *automatically* defines a set of visually different styles for up to 13 series, so while you can customize any chart style, you may not need to.

Every ChartStyle has a FillStyle, a LineStyle, and a SymbolStyle. FillStyles are used for Area, Bar, Candle, Hi-Lo, Hi-Lo-Open-Close, Pie, and Stacking Bar charts. LineStyles and SymbolStyles are used for plots.
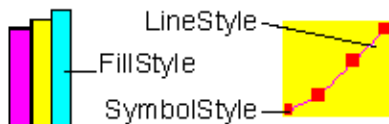

Figure 35   Types of ChartStyles available.

ChartStyle is an indexed property of ChartDataView that "owns" the JCChartStyle objects for that data view. It can be manipulated like any other indexed property, for example:

```
arr.setChartStyle(0, new JCChartStyle());
```

This adds the specified `ChartStyle` to the indexed property at the specified index. If the `ChartStyle` is null, the `JCChartStyle` at the specified point is removed. The following lists some of the other ways `ChartStyle` can be used:

■ `getChartStyle(index)` – retrieves the chart style at the specified index.

■ `setChartStyle(List)` – replaces all existing chart styles.

■ `List getChartStyle()` – retrieves a copy of the array of chart styles.

Normally, you will not need to add or remove `JCChartStyle` objects from the collection yourself. If a `JCChartStyle` object already exists when its corresponding series is created, the previously created `JCChartStyle` object is used to display the data in this series.

Customizing Existing ChartStyles

Each `JCChartStyle` object contains three smaller objects that control different aspects of the style: `JCFillStyle`, `JCLineStyle`, and `JCSymbolStyle`.

The most common chart style sub-properties are repeated in `JCChartStyle`. For example, `FillColor` is a property of `JCChartStyle` that corresponds to the `Color` property of `JCFillStyle` object. The following lists all of the repeated properties:

■ `LinePattern`, `LineWidth`, and `LineColor` repeat `JCLineStyle` properties

■ `SymbolShape`, `SymbolColor`, `SymbolSize`, and `SymbolCustomShape` repeat `JCSymbol` properties

■ `FillColor`, `FillPattern`, and `FillImage` repeat `JCFillStyle` properties.

FillStyle

`JCFillStyle` controls the fills used in bar, pie, area, and candle charts. Its properties include `Color` and `Pattern`. Use `Pattern` to set the fill drawing pattern and `Color` to set the fill color. The default pattern is solid fill.

For JDK 1.3.1 and higher, available fill patterns include none, solid, 25%, 50%, 75%, horizontal stripes, vertical stripes, 45 degree angle stripes, 135 degree angle stripes, diagonal hatched pattern, cross hatched pattern, custom fill, custom paint, or, for bar charts only, custom stack fill.

Custom fill and custom stack fill draw using the image set in the `Image` property. Custom paint draws using the `TexturePaint` object, which is set in the `CustomPaint` property.

Note that filled areas are not supported for Polar charts.

LineStyle

`JCLineStyle` controls line drawing, used in line and hi-lo charts. Its properties are `Color`, `Pattern` and `Width`. Use `Pattern` to set the line drawing pattern, `Color` to set the line color, and `Width` to set the line width.

Custom line patterns can be set with a `setPattern()` method that specifies the line pattern arrays to use.

SymbolStyle

`JCSymbolStyle` controls the symbol used to represent points in a data series, used in plot or scatter plot charts. Its properties are `Shape`, `Color` and `Size`. Use `Shape` to set the symbol type, `Size` to set its size, and `Color` to set the symbol color.
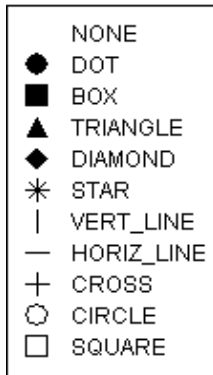
Valid symbols are shown below:

Figure 36   Symbols available in JCSymbolStyle.

You can also provide a custom shape by implementing an abstract class `JCShape` and assigning it to the `CustomShape` property.

Customizing All ChartStyles

By looping through the `JCChartStyle` indexed property, you can quickly change the appearance of all of the bars, lines, or points in a chart. For example, the following code lightens all of the bars in a chart whenever the mouse is clicked:

```
for (Iterator i = c.getDataView(1).getChartStyle().listIterator();
i.hasNext();)
        {
            JCChartStyle cs = (JCChartStyle) i.next();
            JCFillStyle fs = cs.getFillStyle();
            fs.setColor(fs.getColor().brighten);
        }
```

9.5   OutlineStyle

The `ChartDataView`'s `OutlineStyle` property controls the outlines of area, stacking area, area radar, bar, stacking bar, and pie charts. It is of type `JCLineStyle` and thus the properties of the line can be controlled by getting the `JCLineStyle` object using `getOutlineStyle()` and setting its properties.

The default outline style is a solid line of width one in the chart area's foreground color.

9.6   Borders

One way to highlight important information or improve the chart's appearance is to use a border. You can customize the border of the following chart objects:

■   Header and Footer titles

■   Legend

■   ChartArea

■   each `ChartLabel` added to the chart

■   the entire chart

Border properties are set using the standard `JComponent` border facilities, `getBorder()` and `setBorder()`.

9.7   Fonts

A chart can have more impact when you customize the fonts used for different chart elements. You may also want to change the font size to make an element better fit the overall size of the chart. Any font available when the chart is running can be used. You can set the font for the following chart elements:

■   Header and Footer titles

■   Legend

■   Axis annotation and title

■   each `ChartLabel` added to the chart

Changing a Font

Font properties are set using the standard `JComponent` font facilities, `getFont()` and `setFont()`.

Use the font properties to set the font, style, and size attributes.

9.8   Colors

Color can powerfully enhance a chart's visual impact. You can customize chart colors using Java color names or RGB values. Using an interactive tool like The Chart Customizer can make selecting custom colors quick and easy. Each of the following visual elements in the chart has a background and foreground color that you can customize:

■   the entire chart

■   Header and Footer titles

■   Legend

■   Chart Area

- Plot Area (foreground colors `JCChartArea`'s `AxisBoundingBox`)
- each `ChartLabel` added to the chart

Other chart objects have color properties too, including `ChartDataView` (bar/pie outline color) and `ChartStyles`.

Color Defaults

All chart subcomponents are transparent by default with no background color. If made opaque, the legend, chart area and plot will inherit background color from the parent chart. The same objects will always inherit the foreground color from the chart.

Headers and footers are independent objects and behave according to the rules of whatever object they are.

However, once the application sets the colors of an element, they do not change when other elements' colors change.

Specifying Foreground and Background Colors

Each chart element listed above has a `Background` and `Foreground` property that specifies the current color of the element. The easiest way to specify a color is to use the built-in colornames defined in `java.awt.Color`. The following table summarizes these colors:

Built-in Colors in java.awt.Color		
black	blue	cyan
darkGray	gray	green
lightGray	magenta	orange
pink	red	white
	yellow	

Alternately, you can specify a color by its RGB components, useful for matching another RGB color. RGB color specifications are composed of a value from 0 – 255 for each of the red, green and blue components of a color. For example, the RGB specification of Cyan is "0-255-255" (combining the maximum value for both green and blue with no red).

The following example sets the header background using a built-in color, and the footer background to an RGB color (a dark shade of Turquoise):

```
c.getHeader().setBackground(Color.cyan);

mycolor = new Color(95,158,160);
c.getFooter().setBackground(mycolor);
```

Take care not to choose a background color that is also used to display data in the chart. The default `ChartStyles` use all of the built-in colors in the following order: Red, Orange, Blue, Light Gray, Magenta, Yellow, Gray, Green, Dark Gray, Cyan, Black, Pink, and

White. Note that JClass Chart will skip colors that match background colors. For example, if the chart area background is Red, then the line, fill, and symbol colors will start at Orange.

For all JClass Chart charts, the foreground and background colors of the plot area are adjustable.

Transparency

If the JClass Chart component is meant to have a transparent background, set the `Opaque` property to `False`; then generated GIFs and PNGs will also contain a transparent background.

9.9 Positioning Chart Elements

Each of the main chart elements (`Header`, `Footer`, `Legend`, and `ChartArea`) has properties that control its position and size. While the chart can automatically control these properties, you can also customize the following:

■ positioning of any element

■ size of any element

When the chart controls positioning, it first allows space for the `Header`, `Footer`, and `Legend`, if they exist (size is determined by contents, border, and font). The `ChartArea` is sized and positioned to fit into the largest remaining rectangular area. Positioning adjusts when other chart properties change.

`ChartLabels` do not figure into the overall Chart layout. Instead, they are positioned above all other Chart elements.

Changing the Location and Size

To specify the absolute location and size of a chart element, call `setLayoutHints()` in JClass Chart with the object you wish to move and a rectangle containing its desired X- and Y-location, width, and height. If you desire any of those values to be calculated rather than set, make them equal to `Integer.MAX_VALUE`.

For example,

```
chart.setLayoutHints(legend, newRectange(0,150,200,200))
```

will set the legend to be 200 x 200 and place it at (0,150), while

```
chart.setLayoutHints(legend, Rectangle(0,150,
        Integer.MAX_VALUE,Integer.MAX_VALUE, Integer.MAX_VALUE))
```

will leave the legend's size alone but still move it to (0,150).

9.10   3D Effect

Data in bar, stacking bar and pie charts can be displayed with a three-dimensional appearance using several `JCChartArea` properties:

- **■**   `Depth` – Specifies the apparent depth as a percentage of the chart's width. No 3D effect appears unless this property is set greater than zero.

- **■**   `Elevation` – Specifies the eye's position above the horizontal axis, in degrees.

- **■**   `Rotation` – Specifies the number of degrees the eye is positioned to the right of the vertical axis. This property has no effect on pie charts.

You can set the visual depth and the "elevation angle" of the 3D effect. You can also set the "rotation angle" on bar and stacking bar charts. `Depth`, `Rotation`, and `Elevation` are all properties of the `ChartArea`.
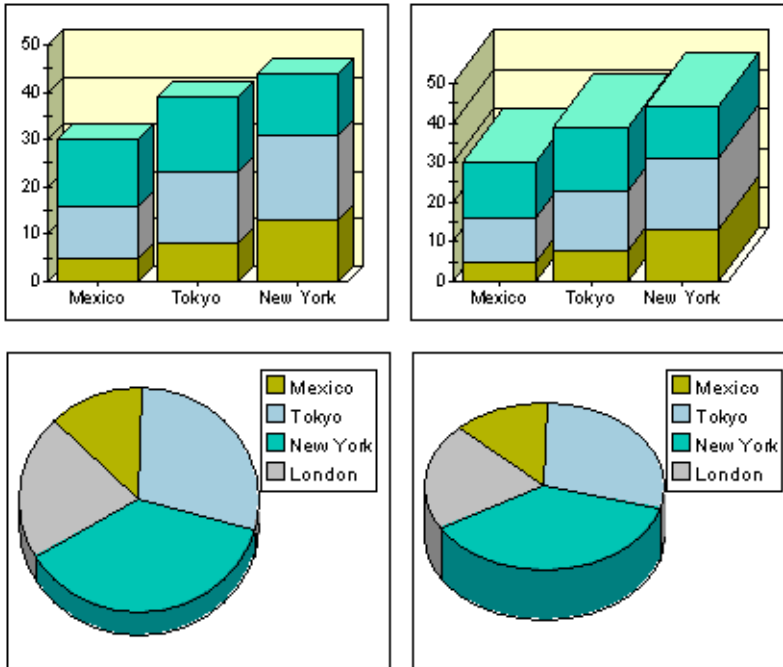

Figure 37   Four JClass Charts illustrating 3D effects.

9.11   Anti-Aliasing

Anti-aliasing is the process of smoothing out lines and curves to remove the pixelated appearance of text and graphics. The smoothing is done by padding pixels with

intermediate colors. For example, a black and white image would be smoothed out using gray.

Figure 38   The appearance of text with and without anti-aliasing.

JClass Chart is equipped with the `antiAliasing` property which can turn anti-aliasing on or off when the chart and its subcomponents are painted.

■   `JCChart.ANTI_ALIASING_ON` turns on anti-aliasing for the chart;

■   `JCChart.ANTI_ALIASING_OFF` turns off anti-aliasing for the chart;

■   `JCChart.ANTI_ALIASING_DEFAULT`, which is the default value, ensures that the graphics object will be untouched with respect to anti-aliasing when the chart is painted.

10

Advanced Chart Programming

Controlling the chart in an application program is generally straightforward once you are familiar with the programming basics and the object hierarchy. For most JClass Chart objects, all the information needed to program them can be found in the API. In addition, extensive information on how they can be used can be found in the numerous example and demonstration programs provided with JClass Chart.

This chapter covers more advanced programming concepts for JClass Chart and also looks at more complex chart programming tasks.

10.1   Outputting JClass Charts

Many applications require that the user has a way to get an image or a hard copy of a chart. JClass Chart allows you to output your chart as a GIF, PNG, or JPEG image, to either a file or an output stream.

(If you have JClass PageLayout installed, you can also encode your charts as an EPS, PS, PCL, or PDF file [in addition to GIF, PNG, or JPEG]. For more information, please see the *JClass PageLayout Programmer's Guide*. Refer to *Quest Software's web site* for information on evaluating or purchasing JClass PageLayout.)

Please note that in order to enable GIF encoding, you must obtain a license from Unisys and send a copy of this license to Quest Software. Quest will send the enabling software for GIF encoding upon receipt of a valid proof of license. There are also public sources of Java image to GIF converters.

Located in `com.klg.jclass.util.swing.encode`, the JCEncodeComponent class is used to encode components into different image file formats. When you include this class in your program, you can call one of two methods that allow you to save the chart image as a GIF, PNG, or JPEG file, sending it to either a file or an output stream.

The parameters of the two methods are the same, except for output.

10.1.1 Encode method

The method to output to a file is:

```
public static void encode(JCEncodeComponent.Encoding encoding,
        Component component, File file)
```

The method to output to an output stream is the same, except that the last parameter is `OutputStream output`, that is `...Component component, OutputStream output)`

The `component` parameter refers to the component to encode, that is, the chart; the `encoding` parameter refers to the type of encoding to use (a GIF, PNG, or JPEG; if you have JClass PageLayout installed, you can also encode your chart as an EPS, PS, PCL, or PDF file); and the `output` parameter refers either to the file to which to write the encoding or to the stream to which to write the encoding.

10.1.2 Encode example

To see this encode method in action, please look at the Encode example, found in the "Example & Demo Gallery" that was installed automatically with JClass Chart. This example appears in the Advanced folder.
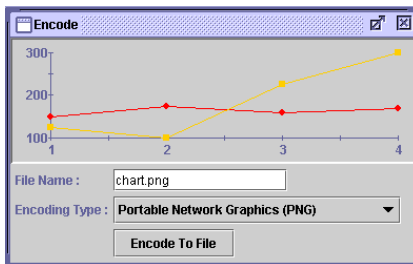
In this example, you can alter the encoding type by selecting a different encoding type from the drop-down menu. Another option provided is your choice of file name. Also, you can right-click the example to bring up the Property Editor and further manipulate the properties of the chart.

10.1.3   Code example

The following code snippet was used to create the example above.

```
public void actionPerformed(ActionEvent evt) {
    int typeIndex = encTypesCB.getSelectedIndex();
    String fileName = encFileTF.getText();
    if (evt.getSource() == encButton) {
        // if encode button pressed, get the encoding type and file name
        // and use them to encoding the chart

        if (typeIndex >= 0 && !(fileName.equals(""))) {
            // Call chart's encoding method, but make sure to catch
            // possible exception
            try {
JCEncodeComponent.encode (JCEncodeComponent.ENCODINGS
    [typeIndex], chart, new File(fileName));
            } catch (EncoderException ee) {
                ee.printStackTrace();
            }
            catch (IOException io) {
                io.printStackTrace();
            }
        }
    }
}
```

10.2   Batching Chart Updates

Normally, the chart is repainted immediately after a property is set. To make several
changes to a chart before causing a repaint, set the `Batched` property of the `JCChart`
object to `true`. Property changes do not cause a repaint until `Batched` is reset to `false`.

The `Batched` property is also defined for the `ChartDataView` object. This `Batched`
property is independent of `JCChart.Batched`. It is used to control the update requests sent
from the `DataSource` to the chart.

Note: It is **highly recommended** that you batch around the creation or updating of
multiple chart labels.

10.3   Coordinate Conversion Methods

The `ChartDataView` object provides methods that enable you to do the following:

■   Convert from data coordinates (x and Y-data values) to pixel coordinates (where these
    data coordinates appear on screen) and vice versa.

■   Determine the pixel coordinates of a given data point in a series, or the closest point
    to a given set of pixel coordinates.

The following table outlines which method or functional equivalent to use for each action.

Method	Functional equivalent	Action
`dataCoordToCoord()`	`unmap`	Converts from data coordinates to pixel coordinates
`coordToDataCoord()`	`map`	Converts from pixel coordinates to data coordinates
`dataIndexToCoord()`	`unpick`	Determines the pixel coordinates of a given data point in a series
`coordToDataIndex()`	`pick`	Determines the closest point in pixels to a given data point in a series

10.3.1 CoordToDataCoord and DataIndexToCoord

To convert from data coordinates to pixel coordinates, call the `dataCoordToCoord()` method. For example, the following code obtains the pixel coordinates corresponding to the data coordinates (5.1, 10.2):

```
Point p=c.getDataView(0).dataCoordToCoord(5.1,10.2);
```

This works in the same way as `unmap`. Note that the pixel coordinate positioning is relative to the upper left corner of the `JCChart` component display.

To convert from pixel coordinates to data coordinates, call `coordToDataCoord()`. For example, the following converts the pixel coordinates (225, 92) to their equivalent data coordinates:

```
JCDataCoord cd=c.getDataView(0).coordToDataCoord(225,92);
```

This works in the same manner as `map`. So, `coordToDataCoord()` returns a `JCDataCoord` object containing the X- and Y-values in the data space.

To determine the pixel coordinates of a given data point, call `dataIndexToCoord()`. For example, the following code obtains the pixel coordinates of the third point in the first data series:

```
JCDataIndex di=new JCDataIndex(3,c.getDataView(0).getSeries(0));
Point cdc=c.getDataView(0).dataIndexToCoord(di);
```

To determine the closest data point to a set of pixel coordinates, call `coordToDataIndex()`:

```
JCDataIndex di=c.getDataView(0).coordToDataIndex(225,92,
      ChartDataView.PICK_FOCUSXY);
```

Essentially, these last two examples demonstrate that `dataIndexToCoord()` works in much the same way as `pick` and `unpick`. The third argument passed to `coordToDataIndex()` specifies how the nearest series and point value are determined.

This argument can be one of `ChartDataView.PICK_FOCUSXY`, `ChartDataView.PICK_FOCUSX`, or `ChartDataView.PICK_FOCUSY`. For more information on the `pick` and `unpick` methods, please see Section 10.9, Using Pick and Unpick.

`JCDataIndex` contains the series and point value corresponding to the closest data point, and also returns the distance in pixels between the pixel coordinates and the point. `coordToDataIndex()` returns a `JCDataIndex` instance.

10.3.2 Map and Unmap

`map` and `unmap` are functionally equivalent to the `coordToDataCoord()` and `dataIndexToCoord()` methods. They are provided as convenience methods, and are more in keeping with typical Java terminology than `coordToDataCoord()` and `dataIndexToCoord()`.

For Polar charts, the X- and Y-values are interpreted as (*theta*, *r*) coordinates. The X-units used will depend on the current value of angle unit. The case for Radar and Area Radar charts is similar, except that X-values will be ignored.

10.4 FastAction

The `FastAction` property determines whether chart actions will use an optimized mode in which it does not bother to update display axis annotations or gridlines during a chart action. Default value is `false`.

Using `FastAction` can greatly improve the performance of a chart display, because relatively more time is needed to draw such things as axis annotations or gridlines than for simply updating the points on a chart. It is designed for use in dynamic chart displays, such as charts that enable the user to perform translation or rotation actions.

The following line of code shows how `FastAction` can be used in a program:

```
c.getChartArea().setFastAction(true);
```

10.5 FastUpdate

The `FastUpdate` property optimizes chart drawing – if possible, only new data that has been added to the datasource is drawn when the chart updates, with little recalculating and redrawing of existing points. (Please see Making Your Own Chart Data Source, in Chapter 8, for a guide on how to build an updating chart data source.) However, if the new data goes outside of the current axis boundaries, then a full redraw is done.

Using `FastUpdate` can improve the performance of a chart display, especially with dynamic chart displays.

The following line of code shows how `FastUpdate` can be used in a program:

```
c.getDataView(0).setFastUpdate(true);
```

A chart using the fast update feature will not draw correctly when the chart object is placed within an `JInternalFrame` object or when items from a `JPopupMenu` overlay the chart.

Please see the FastUpdate demo, found in *JCLASS_HOME/demos/chart/fastupd/*, for a demonstration of this feature.

Note: This feature is not supported in Area Radar or Radar charts. For Polar charts, there is no need to check the axis bounds in the X-direction. The routines for checking axis bounds can still be used for the y-direction.

10.6   Programming End-User Interaction

An end-user can interact with a chart more directly than using the Customizer. Using the mouse and keyboard, a user can examine data more closely or visually isolate part of the chart. JClass Chart provides the following interactions:

■   moving the chart

■   zooming into or out of the chart

■   rotation (only for bar or pie charts displaying a 3D effect)

■   adding depth cues to the chart

■   interactively change data points (using the pick feature)

It is also possible in most cases for the user to reset the chart to its original display parameters. The interactions described here affect the chart displayed inside the `ChartArea`; other chart elements, such as the header, are not affected.

Note: The keyboard/mouse combinations that perform the different interactions can be changed or removed by a programmer. The interactions described here may not be enabled for your chart.

A chart action is a user event that causes some interactive action to take place in the control. In JClass Chart, actions like zoom, translate and rotate can be mapped to a mouse button and a modifier. For example, it is possible to bind the translate event to the combination of mouse button 2 and the **Control** key. Whenever the user hits **Control** and mouse button 2 and drags the mouse, the chart will move.

10.6.1   Event Triggers

An event trigger is a mapping of a mouse operation and/or a key press to a chart action. In the example above, the trigger for translate is a combination of mouse button 2 and the **Control** key.

An event trigger has two parts:

■ the modifier, which specifies the combination of meta keys and mouse buttons that will trigger the action

■ the action, which specifies the combination of chart action that will occur

Valid actions include `EventTrigger.CUSTOMIZE`, `EventTrigger.DEPTH`, `EventTrigger.EDIT`, `EventTrigger.PICK`, `EventTrigger.ROTATE`, `EventTrigger.TRANSLATE`, and `EventTrigger.ZOOM`.

10.6.2 Valid Modifiers

The value of a modifier is specified using *java.AWT.event* modifiers, as shown in the following list:

■ `InputEvent.SHIFT_MASK`

■ `InputEvent.CTRL_MASK`

■ `InputEvent.ALT_MASK`

■ `InputEvent.META_MASK`

You can also specify the mouse button using one of the following modifiers:

■ `InputEvent.BUTTON1_MASK`

■ `InputEvent.BUTTON2_MASK`

■ `InputEvent.BUTTON3_MASK`

10.6.3 Programming Event Triggers

To program an event trigger, use the `setTrigger` method to add the new action mapping to the collection.

For example, the following tells JClass Chart to add a zoom operation as its first trigger (first trigger denoted by 0) when **Shift** and mouse button are pressed:

```
c.setTrigger(0,newEventTrigger(Event.SHIFT_MASK,
      EventTrigger.ZOOM);
```

10.6.4 Removing Action Mappings

To remove an existing action mapping, set the trigger to null, as in the following example:

```
c.setTrigger(0,null);
```

10.6.5 Calling an Action Directly

In JClass Chart, it is possible to force some actions by calling a method of `JCChart`. The following is a list of the methods that can be called upon to force a particular action:

■ Translation – `translateStart()`, `translate()`, `translateEnd()`

- Rotation – `rotateStart()`, `rotate()`, `rotateEnd()`
- Zoom – `zoomStart()`, `zoom()`, `zoomEnd()`
- Scale – `scale()`
- Reset – `reset()`

10.6.6 Specifying Action Axes

Actions like translation occur with respect to one or more axes. In JClass Chart, the axes can be set using the `HorizActionAxis` and `VertActionAxis` properties of `JCChartArea`, as the following code fragment illustrates:

```
ChartDataView arr = c.getDataView(0);
c.getChartArea().setHorizActionAxis(arr.getXAxis());
c.getChartArea().setVertActionAxis(arr.getYAxis());
```

Note that it is possible to have a null value for an action axis. This means that chart actions like translation do not have any effect in that direction. By default, the `HorizActionAxis` is set to the default X-axis, and the `VertActionAxis` is set to the default Y-axis.

10.7   Image-Filled Bar Charts

It is possible to use image files as chart elements within a bar chart. This is accomplished by using the `Image` in `JCFillStyle`. `Image` sets the image used to paint the fill region of bar charts. It takes `img` as a parameter, which is an AWT `Image` class representing the image to be used to paint image fills. If set to null, no image fill is done.

The following code fragment shows how `Image` can be incorporated into a program:

```
String imageStrings[] = {"cd.gif", "tape.gif"};
List seriesList = arr.getSeries();
Iterator iter = seriesList.iterator();
for (int i = 0; iter.hasNext(); i++) {
    ChartDataViewSeries thisSeries = (ChartDataViewSeries)
        iter.next();
    if (i < seriesLabels.length) {
        if (imageStrings[i] != null) {
            Class cl = getClass();
            URL url = cl.getResource("/examples/chart/intro/"+
                                                imageStrings[i]);
            if (url != null) {
                ImageIcon icon = new ImageIcon(url);
                thisSeries.getStyle().getFillStyle().
                    setImage(icon.getImage());
                thisSeries.getStyle().getFillStyle().
                    setPattern(JCFillStyle.CUSTOM_STACK);
            }
        }
    }
}
```

The effects can be seen in the ImageBar demonstration program (in the *JCLASS_HOME/examples/chart/intro/ImageBar.java* directory), which comes with JClass Chart.
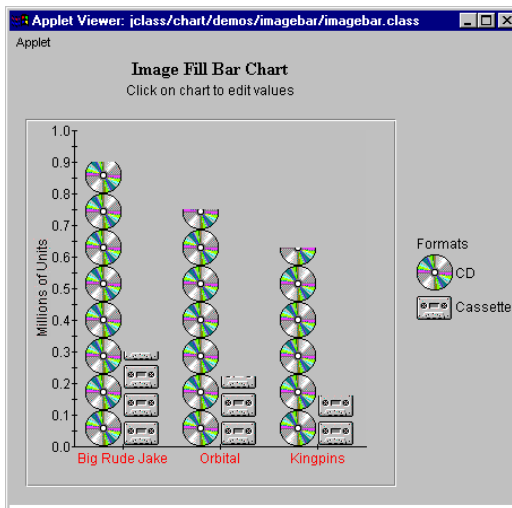
Figure 39   The ImageBar demonstration program.

The image is clipped at the point of the highest value indicated for the bar chart.

`Image` only tiles the image along a single axis. For example, if the bars were widened in the above illustration, it would still tile along the vertical Y-axis only, and would not fill in the image across the horizontal X-axis. This same principle applies (though along different axes) when the bar chart is rotated 90 degrees.

Note: `Image` can only be used with the image formats that can be used in Java.

10.8 Pick

The `pick()` method is used to translate a pixel coordinate on a chart to the data point that is closest to it. The method takes a `Point` object containing a pixel coordinate and an optional `ChartDataView` object to check against, and returns the resulting data point encapsulated in a `JCDataIndex` object.

For `pick()` to work correctly, the `JCChart` instance must first be laid out. This is automatically done whenever a chart is drawn, such as when the `snapshot()` method is called. Alternately, layout can be accomplished manually by called the `doLayout()` method of `JCChart`.

Pick Methods for Polar and Radar Charts

The `pick()` method for Polar and Radar charts is implemented in two stages. The data point closest to the pick point is identified in a primary search, thus obeying the specified pick focus rule. In some cases (for example, Radar charts with more than one series), there may be two or more data points that have the same X- or Y-value. The primary search result may be ambiguous if the pick focus rule is `PICK_FOCUS_X` or `PICK_FOCUS_Y`. To determine which of those points is the desired one, a secondary search is carried out using the `PICK_FOCUS_XY` rule.

Pick Methods for Area Radar Charts

The pick behavior for Area Radar charts differs from that of Polar or Radar charts. If the user clicks on a point within a filled polygon, the search for the closest point (again, obeying the pick focus rule) is limited to the data series represented by that polygon. Pick points within a polygon have the `JCDataIndex.distance` variable set to 0. If the pick point is not within a filled polygon (that is, the user clicked on a point outside of the largest polygon), then the smallest distance from the pick point to the polygon is taken. As with the Polar and Radar chart types, primary and secondary searches are conducted to resolve ambiguities that may arise for `PICK_FOCUS_X` or `PICK_FOCUS_Y`.

10.9 Using Pick and Unpick

The `pick` method is used to retrieve an *x,y* coordinate in a Chart from user input and then translate that into selecting the data point nearest to it. For example, if a user clicks within a single bar within a bar chart, `pick` takes the coordinates of the mouse-click and selects that bar for any action within the program. Similarly, if a user clicks in an area

immediately above a bar chart, `pick` is used to select the bar that is closest to the mouse click.

To use the pick listener, you must first set up a PICK event trigger on the chart. See Section 10.6.3, Programming Event Triggers, for more details.

Consider the following code listing (the code that comprises the DrillDown demonstration program that comes with JClass Chart, in *JCLASS_HOME/demos/chart/drilldown/*) that demonstrates how pick can be used to "drill down" to reveal more information.

Note: This example assumes that a data.class file exists that understands different data levels.

```
package demos.chart.drilldown;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Event;
import java.awt.GridLayout;
import javax.swing.JComponent;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.JEditorPane;
import javax.swing.BorderFactory;

import java.util.Iterator;
import java.util.List;

import com.klg.jclass.chart.ChartDataView;
import com.klg.jclass.chart.ChartDataViewSeries;
import com.klg.jclass.chart.EventTrigger;
import com.klg.jclass.chart.JCAxis;
import com.klg.jclass.chart.JCPickListener;
import com.klg.jclass.chart.JCPickEvent;
import com.klg.jclass.chart.JCChartStyle;
import com.klg.jclass.chart.JCLineStyle;
import com.klg.jclass.chart.JCChart;
import com.klg.jclass.chart.ChartText;
import com.klg.jclass.chart.JCDataIndex;
import com.klg.jclass.chart.JCChartArea;
import com.klg.jclass.util.swing.JCExitFrame;
import com.klg.jclass.util.legend.JCLegend;

/*
 * This demo demonstrates using pick to drill down to more
 * refined data
 */
public class DrillDown extends javax.swing.JPanel
                          implements JCPickListener {

protected Data    d = null;
protected JCChart c = null;

public DrillDown()
{
    setLayout(new BorderLayout(10,10));
    setPreferredSize(new Dimension(600,400));

    d = new Data();

    Color Turquoise = new Color(64,224,208);
    Color DarkTurquoise = new Color(0x00,0xce,0xd1);

    c = new JCChart();
    c.setTrigger(0, new EventTrigger(0, EventTrigger.PICK));
```

```
c.setBackground(DarkTurquoise);

JCChartArea area = c.getChartArea();
area.getPlotArea().setBackground(Turquoise);
area.setOpaque(true);
area.setBorder(BorderFactory.createEtchedBorder());

JComponent header = c.getHeader();
header.setBackground(Turquoise);

((JLabel)header).setText("<html><center>
                                <font color=black><b>Drill Down
                               Demo</b><P>Independent Comic Book
                                      Sales 1996</center>");

header.setBorder(BorderFactory.createRaisedBevelBorder());
header.setVisible(true);

JCLegend legend = c.getLegend();
legend.setVisible(true);
legend.setBackground(Turquoise);
legend.setForeground(Color.black);
legend.setBorder(BorderFactory.createLoweredBevelBorder());

ChartDataView dataView = c.getDataView(0);
c.setBatched(false);
dataView.setDataSource(d);
dataView.setChartType(JCChart.BAR);
dataView.setHoleValue(-1000);
dataView.getOutlineStyle().setColor(Color.darkGray);

JComponent footer = c.getFooter();
footer.setVisible(true);

((JLabel)footer).setText("<html><font color=black size=-1>
                               <CENTER><i>Drill Down -> Mouse Down
                                  on Bar or Legend<P>Drill Up ->
                                     Mouse Down on Other Area of
                                          Graph</i></CENTER>");

area.setDepth(10);
area.setElevation(20);
area.setRotation(20);

JCAxis yAxis = area.getYAxis(0);
yAxis.setGridVisible(true);
yAxis.getGridStyle().getLineStyle().setColor(new
    Color(154,154,229));

// Set colors for each data series
setSeriesColor();

// Set up pick and rotate trigger
c.setTrigger(0, new EventTrigger(0, EventTrigger.PICK));
c.setTrigger(1, new EventTrigger(Event.SHIFT_MASK,
                                        EventTrigger.ROTATE));
```

```
        c.setTrigger(2, new EventTrigger(Event.META_MASK,
                                                EventTrigger.CUSTOMIZE));
        c.setAllowUserChanges(true);

        // Add listener for pick events
        c.addPickListener(this);

        add("Center",c);
    }

    void setSeriesColor()
    {
        // Set colors for each data series
        Color colors[] = {Color.red, Color.blue, Color.white,
                                Color.magenta, Color.green, Color.cyan,
                                Color.orange, Color.yellow};
        ChartDataView dataView = c.getDataView(0);
        List seriesList = dataView.getSeries();
        Iterator iter = seriesList.iterator();
        for (int i = 0; iter.hasNext(); i++) {
            ChartDataViewSeries series =
            (ChartDataViewSeries)iter.next();
            series.getStyle().setFillColor(colors[i]);
        }
    }

    /**
     * Pick event listener.  Upon receipt of a pick event, it either
     * drills up or down to more general or refined data.
     */
    public void pick(JCPickEvent e)
    {
        boolean doLevel = false;
        boolean doUpLevel = true;
        JCDataIndex di = e.getPickResult();
        int srs = 0;

        // If clicked on bar or legend item, drill down. If clicked on
        // any other area of chart, drill up.
        if (di != null) {
            Object obj = di.getObject();
            ChartDataView vw = di.getDataView();
            srs = di.getSeriesIndex();
            int pt = di.getPoint();
            int dist = di.getDistance();

            if (vw != null && srs != -1) {
                if (srs >= 0) {
                    if ((obj instanceof JCLegend) ||
                        (obj instanceof JCChartArea && dist == 0))
                    {
                        doLevel = true;
                        doUpLevel = false;
                    }
                    else {
                        doLevel = true;
```

```
                    }
                }
            }
            else {
                doLevel = true;
            }
        }
        else {
            doLevel = true;
        }


        if (doLevel) {
            c.setBatched(true);
            if (doUpLevel) {
                d.upLevel();
            }
            else {
                    d.downLevel(srs);
            }
            setSeriesColor();
            c.setBatched(false);
        }
    }

    public static void main(String args[])
    {
        JCExitFrame f = new JCExitFrame("Basic Drilldown example");
        DrillDown tc = new DrillDown();
        f.getContentPane().add(tc);
        f.pack();
        f.setVisible(true);
    }

}
```

When compiled and run, the *DrillDown.class* program displays the following:
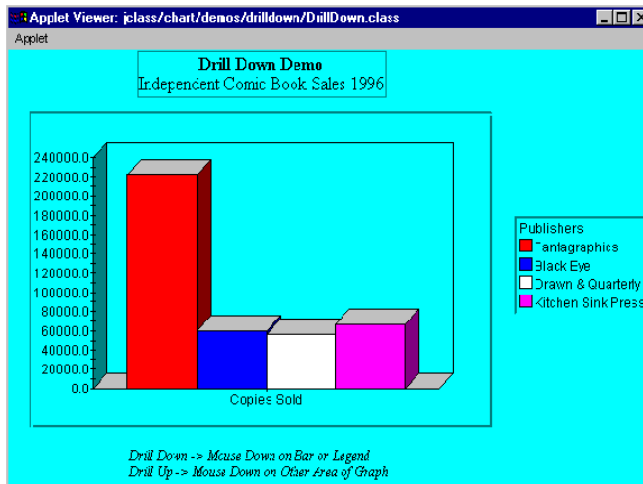
Figure 40   The DrillDown demonstration program displayed.

When a bar or legend within this chart is clicked by the user, the program "drills down" to reveal more refined data comprising that bar. If an area outside of the bars is clicked upon, then the program "drills up" to reveal more general data.

pick is key to this program, determining the way the program interacts with the user. pick requires an event trigger and listener to work, as the following code fragment shows:

```
c.setTrigger(0, new EventTrigger(0, EventTrigger.PICK));
   c.addPickListener(this);
public void pick(JCPickEvent e)
{
    JCDataIndex di = e.getPickResult();
}
```

When a user clicks in the DrillDown demonstration program, the event is triggered, and the *x,y* coordinates are passed along to the pick event listener, which in turn takes the information and performs the indicated action. The pick() method returns a JCDataIndex, which encapsulates the point index and data series of the selected point.

It is also possible to send a pick event to objects manually. When the sendPickEvent() method is called, it sends a pick event to all objects listening for it.

10.9.1   Pick Focus

pick normally takes an *x,y* coordinate value, but it can take an X- or Y-value only, which is useful for specific chart types. This can be specified using the PickFocus property of ChartDataView, which specifies how distance is determined for pick operations. When set to PICK_FOCUS_XY (default), a pick operation will use the actual distance between the point

and the drawn data. When set to values of `PICK_FOCUS_X` or `PICK_FOCUS_Y`, only the distance along the X- or Y-axis is used.

This is a particularly useful method within programs that display typical bar charts. In most cases it is more desirable to know which bar the user is *over* than which bar the user is *closest to* when the user clicks their mouse over a chart.

For example, a user may click over a relatively small bar in a bar chart, with the intention of raising the value of the bar displayed. If an adjacent bar in the chart is closer to the area of the mouse click along the Y-axis than the X-axis, then the adjacent bar could be selected instead of the intended target bar.

To overcome this, use `PickFocus` and select the axis whose values are to be reported back to the program. For example, the following line of code sets `PickFocus` to only report the X-coordinate of a pick event:

```
arr.setPickFocus(ChartDataVies.PICK_FOCUS_X);
```

10.10   Unpick

The `unpick()` method essentially functions in the opposite manner of `pick`: given a data series and a data point within that series, `unpick` returns the pixel co-ordinates of that point relative to the chart area. It takes two sets of parameters: `pt` for the point index, and `series` for the data series. For bar charts it returns the top-middle location for a given bar, and the middle of an arc for a pie chart. `unpick` can be used to display information at a given point in a chart, and can be used for attaching labels to chart regions.

For `unpick()` to work correctly, the `JCChart` instance must first be laid out. This is automatically done whenever a chart is drawn, such as when the `snapshot()` method is called. Alternately, layout can be accomplished manually by calling the `doLayout()` method of `JCChart`.

Part II

Reference Appendices

Appendix A

JClass Chart Property Listing

ChartDataView ■ *ChartDataViewSeries* ■ *ChartText* ■ *JCAreaChartFormat* ■ *JCAxis*
JCAxisFormula ■ *JCAxisTitle* ■ *JCBarChartFormat* ■ *JCCandleChartFormat* ■ *JCChart*
JCChartArea ■ *JCChartLabel* ■ *JCChartLabelManager* ■ *JCChartStyle* ■ *JCFillStyle*
JCGridLegend ■ *JCHLOCChartFormat* ■ *JCLegend* ■ *JCLineStyle* ■ *JCMultiColLegend*
JCPieChartFormat ■ *JCPolarRadarChartFormat* ■ *JCSymbolStyle*
JCValueLabel ■ *PlotArea* ■ *SimpleChart*

This appendix summarizes the JClass Chart properties for all commonly-used classes, in
alphabetical order.

A.1   ChartDataView

Name	Description
AutoLabel	The `AutoLabel` property determines if the chart automatically generates labels for each point in each series. The default is `false`. The labels are stored in the `AutoLabelList` property. They are created using the `Label` property of each series.
Batched	The `Batched` property controls whether the `ChartDataView` is notified immediately of data source changes, or if the changes are accumulated and sent at a later date.
BufferPlotData	The `BufferPlotData` property controls whether plot data is to be buffered to speed up the drawing process. This property is applicable for Plot, Scatter, Area, Hilo, HLOC, and Candle chart types only. Normally it is `true`. The property is ignored if the `FastUpdate` property is `true`. Plot data will be buffered for `FastUpdate`.
ChartFormat	The `ChartFormat` property represents an instance of `JCAreaChartFormat`, `JCBarChartFormat`, `JCCandleChartFormat`, `JCHiloChartFormat`, `JCHLOCChartFormat`, or `JCPieChartFormat`, depending on the current chart type.

Name	Description
Changed	The `Changed` property manages whether the data view requires recalculation. If set to `true`, a recalculation may be triggered. Default value is `true`.
ChartStyle	The `ChartStyle` property contains all the ChartStyles for the data series in this data view. Default value is generated.
ChartType	The `ChartType` property of the `ChartData` object specifies the type of chart used to plot the data. Valid values are `JCChart.AREA`, `JCChart.AREA_RADAR`, `JCChart.BAR`, `JCChart.CANDLE`, `JCChart.HILO`, `JCChart.HILO_OPEN_CLOSE`, `JCChart.PIE`, `JCChart.PLOT` (default), `JCChart.POLAR`, `JCChart.RADAR`, `JCChart.SCATTER_PLOT`, `JCChart.STACKING_AREA`, and `JCChart.STACKING_BAR`.
ColorHandler	The `ColorHandler` property specifies a class used to override the default color determination. The `ColorHandler` property must implement `JCDrawableColorHandler`.
DataSource	The `DataSource` property, if non-null, is used as a source for data in the `ChartDataView`. The object must implement `ChartDataModel`.
DrawFrontPlane	The `DrawFrontPlane` property determines whether a data view that has both axes on the front plane of a 3d chart will draw on the front or back plane of that chart. If `true`, it will draw on the front plane; if `false` it will draw on the back plane. If either axis associated with the data view is on the back plane, this property will be ignored and the data view will automatically be drawn on the back plane. This property is also ignored for 3d chart types such as bar and stacking bars that automatically appear on the front plane.
DrawingOrder	The `DrawingOrder` property determines the drawing order of items. When the `DrawingOrder` property is changed, the order properties of all `ChartDataView` instances managed by a single `JCChart` object are normalized.
FastUpdate	The `FastUpdate` property controls whether column appends to the data are performed quickly, only recalculating and redrawing the newly-appended data.

Name	Description
HoleValue	The `HoleValue` property is a floating point number used to represent a hole in the data. Internally, `ChartDataView` places this value in the X- and Y-arrays to represent a missing data value. Note that if the `HoleValue` is changed, values in the X- and Y-data previously set with `HoleValue`s will **not** change their values but will now draw.
Inverted	If the `Inverted` property is set to `true`, the X-axis becomes vertical, and the Y-axis becomes horizontal. Default value is `false`.
Name	The `Name` property is used as an index for referencing particular `ChartDataView` objects.
NumPointLabels	The `NumPointLabels` property determines the number of labels in the `PointLabels` property. The `PointLabels` property is an indexed property consisting of a series of Strings representing the desired label for a particular data point.
NumSeries	The `NumSeries` property determines how many data series there are in a `ChartDataView`.
OutlineColor	The `OutlineColor` property determines the color with which to draw the outline around a filled chart item (e.g. bar, pie).
PickFocus	The `PickFocus` property specifies how distance is determined for pick operations. When set to `PICK_FOCUS_XY`, a pick operation will use the actual distance between the point and the drawn data. When set to values of `PICK_FOCUS_X` or `PICK_FOCUS_Y`, the distance only along the X- or Y-axis is used.
PointLabel	Sets a particular `PointLabel` from the `PointLabels` property (see below).
PointLabels	The `PointLabels` property is an indexed property comprising a series of Strings representing the desired label for a particular data point.
Series	The `Series` property is an indexed property that contains all data series for a particular `ChartDataView`. The order of `ChartDataViewSeries` objects in the series array corresponds to the drawing order.
Visible	The `Visible` property determines whether the dataview is showing or not. Default value is `true`.

Name	Description
VisibleInLegend	The VisibleInLegend property determines whether or not the view name and its series will appear in the chart legend.
XAxis	The XAxis property determines the X-axis against which the data in ChartDataView is plotted.
YAxis	The YAxis property determines the Y-axis against which the data in ChartDataView is plotted.

A.2   ChartDataViewSeries

Name	Description
DrawingOrder	The DrawingOrder property determines the order of display of data series. When the DrawingOrder property is changed, ChartDataView will normalize the order properties of all the ChartDataViewSeries objects that it manages.
FirstPoint	The FirstPoint property controls the index of the first point displayed in the ChartDataViewSeries.
Included	The Included property determines whether a data series is included in chart calculations (like axis bounds).
Label	The Label property controls the text that appears next to the data series inside the legend.
LastPoint	The LastPoint property controls the index of the first point displayed in the ChartDataViewSeries.
LastPointIsDefault	The LastPointIsDefault property determines whether the LastPoint property should be calculated from the data.
Name	The Name property represents the name of the data series. In JClass Chart, data series are named, and can be retrieved by name.
Style	The Style property defines the rendering style for the data series.
Visible	The Visible property determines whether the data series is showing in the chart area. Note that data series that are not showing are still used in axis calculations. See the Included property for details on how to omit a data series from chart calculations.

Name	Description
VisibleInLegend	The VisibleInLegend property determines whether or not this series will appear in the chart legend.

A.3　ChartText

Name	Description
Adjust	The Adjust property determines how text is justified (positioned) in the label. Valid values include ChartText.LEFT, ChartText.CENTER, and ChartText.RIGHT. The default value is ChartText.LEFT.
Background	The Background property determines the background color used to draw inside the chart region. Note that the Background property is inherited from the parent ChartRegion.
Font	The Font property determines what font is used to render text inside the chart region. Note that the Font property is inherited from the parent ChartRegion.
Foreground	The Foreground property determines the foreground color used to draw inside the chart region. Note that the Foreground property is inherited from the parent ChartRegion.
Height	The Height property determines the height of the ChartRegion. The default value is calculated.
HeightIsDefault	The HeightIsDefault property determines whether the height of the chart region is calculated by Chart (true) or taken from the Height property (false). The default value is true.
Insets	The Insets property specifies the space that a container must leave at each of its edges. The space can be a border, a blank space, or a title.
Left	The Left property determines the location of the left of the ChartRegion. The default value is calculated.
LeftIsDefault	The LeftIsDefault property determines whether the left position of the chart region is calculated by Chart (true) or taken from the Left property (false). The default value is true.

Name	Description
Name	The Name property specifies a String identifier for the ChartRegion object.
Rotation	The Rotation property controls the rotation of the label. Valid values include ChartText.DEG_90, ChartText.DEG_180, ChartText.DEG_270, and ChartText.DEG_0. The default value is ChartText.DEG_0.
Text	The Text property is a String property that represents the text to be displayed inside the chart label. Default value is " " (empty String).
Top	The Top property determines the location of the top of the ChartRegion. The default value is calculated.
TopIsDefault	The TopIsDefault property determines whether the top position of the chart region is calculated by Chart (true) or taken from the Top property (false). The default value is true.
Visible	The Visible property determines whether the associated ChartRegion is currently visible. Default value is true.
Width	The Width property determines the width of the ChartRegion. The default value is calculated.
WidthIsDefault	The WidthIsDefault property determines whether the width of the chart region is calculated by Chart (true) or taken from the Width property (false). The default value is true.

A.4   JCAreaChartFormat

Name	Description
100Percent	The 100Percent property determines whether a stacking area will be charted versus an axis representing a percentage between 0 and 100. Default value is false.

A.5 JCAxis

Name	Description
AnnotationMethod	The `AnnotationMethod` property determines how axis annotations are generated. Valid values are `JCAxis.VALUE` (annotation is generated by Chart, with possible callbacks to a label generator), `JCAxis.VALUE_LABELS` (annotation is taken from a list of value labels provided by the user – a value label is a label that appears at a particular axis value), `JCAxis.POINT_LABELS` (annotation comes from the data source's point labels that are associated with particular data points), and `JCAxis.TIME_LABELS` (Chart generates time/date labels based on the `TimeUnit`, `TimeBase` and `TimeFormat` properties). Default value is `JCAxis.VALUE`.
AnnotationRotation	The `AnnotationRotation` property specifies the rotation of each axis label. Valid values are `JCAxis.ROTATE_90`, `JCAxis.ROTATE_180`, `JCAxis.ROTATE_270`, or `JCAxis.ROTATE_NONE`. Default value is `JCAxis.ROTATE_NONE`.
Background	The `Background` property determines the background color used to draw inside the chart region. Note that the `Background` property is inherited from the parent ChartRegion.
Editable	The `Editable` property determines whether the axis can be affected by edit/translation/zooming. Default value is `true`.
Font	The `Font` property determines what font is used to render text inside the chart region. Note that the `Font` property is inherited from the parent ChartRegion.
Foreground	The `Foreground` property determines the foreground color used to draw inside the chart region. Note that the `Foreground` property is inherited from the parent ChartRegion.
Formula	The `Formula` property determines how an axis is related to another axis object. If set, the `Formula` property overrides all other axis properties. See `JCAxisFormula` for details.
Gap	The `Gap` property determines the amount of space left between adjacent axis annotations, in pixels.

Name	Description
GeneratedValueLabels	The GeneratedValueLabels property reveals the value label at the specified index in the list of value labels generated for this axis.
GridSpacing	The GridSpacing property controls the spacing between gridlines relative to the axis. Default value is 0.0.
GridSpacingIsDefault	The GridSpacingIsDefault property determines whether Chart is responsible for calculating the grid spacing value. If true, Chart will calculate the grid spacing. If false, Chart will use the provided grid spacing. Default value is true.
GridStyle	The GridStyle property controls how grids are drawn. The default value is generated.
GridVisible	The GridVisible property determines whether a grid is drawn for the axis. Default value is false.
Height	The Height property determines the height of the ChartRegion. The default value is calculated.
HeightIsDefault	The HeightIsDefault property determines whether the height of the chart region is calculated by Chart (true) or taken from the Height property (false). Default value is true.
LabelGenerator	The LabelGenerator property holds a reference to an object that implements the JCLabelGenerator interface. This interface is used to externally generate labels if the AnnotationMethod property is set to JCAxis.VALUE. Default value is null.
Left	The Left property determines the location of the left of the ChartRegion. The default value is calculated.
LeftIsDefault	The LeftIsDefault property determines whether the left position of the chart region is calculated by Chart (true) or taken from the Left property (false). Default value is true.
Logarithmic	The Logarithmic property determines whether the axis will be logarithmic (true) or linear (false). Default value is false.

Name	Description
Max	The Max property controls the maximum value shown on the axis. The data max is determined by Chart. Default value is calculated.
MaxIsDefault	The MaxIsDefault property determines whether Chart is responsible for calculating the maximum axis value. If true, Chart calculates the axis max. If false, Chart uses the provided axis max. Default value is true.
Min	The Min property controls the minimum value shown on the axis. The data min is determined by Chart. Default value is calculated.
MinIsDefault	The MinIsDefault property determines whether Chart is responsible for calculating the minimum axis value. If true, Chart will calculate the axis minimum. If false, Chart will use the provided axis minimum. Default value is true.
Name	The Name property specifies a String identifier for the ChartRegion object. Note that the Name property is inherited from the parent ChartRegion.
NumSpacing	The NumSpacing property controls the interval between axis labels. The default value is calculated.
NumSpacingIsDefault	The NumSpacingIsDefault property determines whether Chart is responsible for calculating the numbering spacing. If true, Chart will calculate the spacing. If false, Chart will use the provided numbering spacing. Default value is true.
Origin	The Origin property controls location of the origin along the axis. The default value is calculated.
OriginIsDefault	The OriginIsDefault property determines whether Chart is responsible for positioning the axis origin. If true, Chart calculates the axis origin. If false, Chart uses the provided axis origin value. Default value is true.

Name	Description
OriginPlacement	The OriginPlacement property specifies where the origin is placed. Note that the OriginPlacement property is only active if the Origin property has not been set. Valid values include AUTOMATIC (places origin at minimum value). ZERO (places origin at zero), MIN (places origin at minimum value on axis), or MAX (places origin at maximum value on axis). Default value is AUTOMATIC.
OriginPlacementIsDefault	The OriginPlacementIsDefault property determines whether Chart is responsible for determining the location of the axis origin. If true, Chart calculates the origin positioning. If false, Chart uses the provided origin placement.
Placement	The Placement property determines the method used to place the axis. Valid values include JCAxis.AUTOMATIC (Chart chooses an appropriate location), JCAxis.ORIGIN (appears at the origin of another axis, specified via the PlacementAxis property), JCAxis.MIN (appears at the minimum axis value), JCAxis.MAX (appears at the maximum axis value), or JCAxis.VALUE_ANCHORED (appears at a particular value along another axis, specified via the PlacementAxis property). Default value is AUTOMATIC.
PlacementAxis	The PlacementAxis property determines the axis that controls the placement of this axis. In JCChart, it is possible to position an axis at a particular position on another axis (in conjunction with the PlacementLocation property or the Placement property). Default value is null.
PlacementIsDefault	The PlacementIsDefault property determines whether Chart is responsible for determining the location of the axis. If true, Chart calculates the axis positioning. If false, Chart uses the provided axis placement.
PlacementLocation	The PlacementLocation property is used with the PlacementAxis property to position the current axis object at a particular point on another axis. Default value is 0.0.
Precision	The Precision property controls the number of zeros that appear after the decimal place in chart-generated axis labels. The default value is calculated.

Name	Description
PrecisionIsDefault	The PrecisionIsDefault determines whether Chart is responsible for calculating the numbering precision. If true, Chart will calculate the precision. If false, Chart will use the provided precision. Default value is true.
Reversed	The Reversed property of JCAxis determines if the axis direction is reversed. Default value is false.
TickSpacing	The TickSpacing property controls the interval between tick lines on the axis. Note: if the AnnotationMethod property is set to POINT_LABELS, tick lines appear at point values. The default value is calculated.
TickSpacingIsDefault	The TickSpacingIsDefault property determines whether Chart is responsible for calculating the tick spacing. If true, Chart will calculate the tick spacing. If false, Chart will use the provided tick spacing. Default value is true.
TimeBase	The TimeBase property defines the start time for the axis. Default value is the current time.
TimeFormat	The TimeFormat property controls the format used to generate time labels for time labelled axes. The formats supported are the same as in Java's SimpleDateFormat class. Default value is calculated based on TimeUnit.
TimeFormatIsDefault	The TimeFormatIsDefault property determines whether a time label format is generated automatically, or the user value for TimeFormat is used. Default value is true.
TimeUnit	The TimeUnit property controls the unit of time used for labelling a time labelled axis. Valid TimeUnit values include JCAxis.SECONDS, JCAxis.MINUTES, JCAxis.HOURS, JCAxis.DAYS, JCAxis.WEEKS, JCAxis.MONTHS, and JCAxis.YEARS. Default value is JCAxis.SECONDS.
Title	The Title property controls the appearance of the axis title.
Top	The Top property determines the location of the top of the ChartRegion. The default value is calculated.

Name	Description
TopIsDefault	The TopIsDefault property determines whether the top position of the chart region is calculated by Chart (true) or taken from the Top property (false). Default value is true.
ValueLabels	The ValueLabels property is an indexed property containing a list of all annotations for an axis. Default value is null, no value labels.
Vertical	The Vertical property determines whether the associated Axis is vertical. Default value is false.
Visible	The Visible property determines whether the associated Axis is currently visible. Default value is true. Note that the Font property is inherited from the parent ChartRegion.
Width	The Width property determines the width of the ChartRegion. The default value is calculated.
WidthIsDefault	The WidthIsDefault property determines whether the width of the chart region is calculated by Chart (true) or taken from the Width property (false). Default value is true.

A.6   JCAxisFormula

Name	Description
Constant	The Constant property specifies the "c" value in the axis relationship $y2 = my + c$.
Multiplier	The Multiplier property specifies the "m" value in the relationship $y2 = my + c$.
Originator	The Originator property specifies an object representing the axis that is related to the current axis by the formula $y = mx + c$. The originator is "x".

A.7   JCAxisTitle

Name	Description
Adjust	The `Adjust` property determines how text is justified (positioned) in the label. Valid values include `ChartText.LEFT`, `ChartText.CENTER`, and `ChartText.RIGHT`. Default value is `ChartText.LEFT`.
Background	The `Background` property determines the background color used to draw inside the chart region. Note that the `Background` property is inherited from the parent `ChartText`.
Font	The `Font` property determines what font is used to render text inside the chart region. Note that the `Font` property is inherited from the parent `ChartText`.
Foreground	The `Foreground` property determines the foreground color used to draw inside the chart region. Note that the `Foreground` property is inherited from the parent `ChartText`.
Height	The `Height` property defines the height of the chart region. The default value is calculated.
HeightIsDefault	The `HeightIsDefault` property determines whether the height of the chart region is calculated by Chart (`true`) or taken from the `Height` property (`false`).
Left	The `Left` property determines the location of the left of the chart region. This property is read-only.
LeftIsDefault	The `LeftIsDefault` property determines whether the left position of the chart region is calculated by Chart (`true`) or taken from the `Left` property (`false`).
Placement	The `Placement` property controls where the `JCAxis` title is placed relative to the "opposing" axis. Valid values include `JCLegend.NORTH` or `JCLegend.SOUTH` for horizontal axes, and `JCLegend.EAST`, `JCLegend.WEST`, `JCLegend.NORTHEAST`, `JCLegend.SOUTHEAST`, `JCLegend.NORTHWEST`, or `JCLegend.SOUTHEAST` for vertical axes. The default value is calculated.
PlacementIsDefault	The `PlacementIsDefault` property determines whether Chart is responsible for calculating a reasonable default placement for the axis title. Default value is `true`.

Name	Description
Rotation	The `Rotation` property controls the rotation of the label. Valid values include `ChartText.DEG_90`, `ChartText.DEG_180`, `ChartText.DEG_270`, and `ChartText.DEG_0`. Default value is `ChartText.DEG_0`.
Text	The `Text` property is a String property that represents the text to be displayed inside the chart label. Default value is " " (nothing).
Top	The `Top` property determines the location of the top of the chart region. This property is read-only.
TopIsDefault	The `TopIsDefault` property determines whether the top position of the chart region is calculated by Chart (`true`) or taken from the Top property (`false`).
Visible	The `Visible` property determines whether the associated Axis is currently visible. Default value is `true`.
Width	The `Width` property defines the width of the chart region. The default value is calculated.
WidthIsDefault	The `WidthIsDefault` property determines whether the width of the chart region is calculated by Chart (`true`) or taken from the Width property (`false`).

A.8   JCBarChartFormat

Name	Description
100Percent	The `100Percent` property determines whether stacking bar charts will be charted versus an axis representing a percentage between 0 and 100. Default value is `false`.
ClusterOverlap	The `ClusterOverlap` property specifies the overlap between bars. Valid values are between -100 and 100. Default value is 0.
ClusterWidth	The `ClusterWidth` property determines the percentage of available space which will be occupied by the bars. Valid values are between 0 and 100. Default value is 80.

A.9   JCCandleChartFormat

Name	Description
CandleOutlineStyle	The CandleOutlineStyle determines the candle outline style of the complex candle chart.
Complex	The Complex property determines whether candle charts use the simple or the complex display style. When false, Chart only uses the style referenced by getHiLoStyle() for the candle appearance. When set to true, all four styles are used. Default value is false.
FallingCandleStyle	The FallingCandleStyle determines the candle style of the falling candle style of the complex candle chart.
HiloStyle	The HiloStyle determines the candle style of the simple candle or the HiLo line of the complex candle chart.
RisingCandleStyle	The RisingCandleStyle determines the rising candle style of the complex candle chart.

A.10   JCChart

Name	Description
About	The About property displays contact information for Quest Software in the bean box.
AllowUserChanges	The AllowUserChanges property determines whether the user viewing the graph can modify graph values. Default value is false.
Background	The Background property determines the background color used to draw inside the chart region. Note that the Background property is inherited from the parent JCComponent.
Batched	The Batched property controls whether chart updates are accumulated. Default value is false.
CancelKey	The CancelKey property specifies the key used to perform a cancel operation.
Changed	The Changed property determines whether the chart requires recalculation. Default value is false.
ChartArea	The ChartArea property controls the object that controls the display of the graph. Default value is null.

Name	Description
ChartLabelManager	The ChartLabelManager property manages all chart labels.
CustomizerName	The CustomizerName property specifies the full class name of the Chart Customizer. Default is com.klg.jclass.chart.customizer.ChartCustomizer.
DataView	The DataView property is an indexed property that contains all the data to be displayed in Chart. See ChartDataView for details on data format. By default, one ChartDataView is created.
FillColorIndex	The FillColorIndex property controls the fill color index. Default value is 0.
Font	The Font property determines what font is used to render text inside the chart region. Note that the Font property is inherited from the parent JCComponent.
Footer	The Footer property controls the object that controls the display of the footer. Default value is a JLabel instance
Foreground	The Foreground property determines the foreground color used to draw inside the chart region. Note that the Foreground property is inherited from the parent JCComponent.
Header	The Header property controls the object that controls the display of the header. Default value is a null.
LayoutHints	The LayoutHints property sets layout hints for a child component of JClass Chart.
Legend	The Legend property controls the object that controls the display of the legend. Default value is an instance of JCGridLegend.
LineColorIndex	The LineColorIndex property controls the line color index. Default value is 0.
NumData	The NumData property indicates how many ChartDataView objects are stored in JCChart. This is a read-only property. Default value is 1.
NumTriggers	The NumTriggers property indicates how many event triggers have been specified.
ResetKey	The ResetKey property specifies the key used to perform a reset operation.
SymbolColorIndex	The SymbolColorIndex property controls the symbol color index. Default value is 0.

Name	Description
SymbolShapeIndex	The SymbolShapeIndex property controls the symbol shape index. Default value is 1.
Trigger	The Trigger property is an indexed property that contains all the information necessary to map user events into Chart actions. The Trigger property is made up of a number of EventTrigger objects. Default value is empty.
WarningDialog	The WarningDialog property determines whether JClass Chart will display a warning dialogue when required.

A.11   JCChartArea

Name	Description
AngleUnit	The AngleUnit property determines the unit of all angle values. Default value is DEGREES.
AxisBoundingBox	The AxisBoundingBox property determines whether a box is drawn around the area bound by the inner axes.
Background	The Background property determines the background color used to draw inside the chart region. Note that the Background property is inherited from the parent JCChart.
Depth	The Depth property controls the apparent depth of a graph. Default value is 0.0.
Elevation	The Elevation property controls distance from the X-axis. Default value is 0.0.
FastAction	The FastAction property determines whether chart actions will use an optimized mode in which it does not bother to display axis annotations or gridlines. Default value is false.
Font	The Font property determines what font is used to render text inside the chart region. Note that the Font property is inherited from the parent JCChart.
Foreground	The Foreground property determines the foreground color used to draw inside the chart region. Note that the Foreground property is inherited from the parent JCChart.
HorizActionAxis	The HorizActionAxis property determines the axis used for actions (zooming, translating) in the horizontal direction. Default value is null.

Name	Description
PlotArea	The PlotArea property represents the region of the ChartArea that is inside the axes. This property is read-only.
Rotation	The Rotation property controls the position of the eye relative to the Y-axis. Default value is 0.0.
VertActionAxis	The VertActionAxis property determines the axis used for actions (zooming, translating) in the vertical direction. Default value is null.
Visible	If true, the ChartRegion will appear on the screen. If false, it will not appear on the screen. (Legend, header, footer and chart area are all ChartRegion instances.) Default value is true.
XAxis	The XAxis property is an indexed property that contains all the X-axes for the chart area. Default value is one X-axis.
YAxis	The YAxis property is an indexed property that contains all the y axes for the chart area. Default value is one Y-axis.

A.12   JCChartLabel

Name	Description
Anchor	Specifies how the label is to be positioned relative to the specified point. Valid values are JCChartLabel.NORTHEAST, JCChartLabel.NORTHWEST, JCChartLabel.NORTH, JCChartLabel.EAST, JCChartLabel.WEST, JCChartLabel.SOUTHEAST, JCChartLabel.SOUTHWEST, and JCChartLabel.SOUTH.
AttachMethod	Specifies how the label is attached to the chart. Valid values are JCChartLabel.ATTACH_COORD (attach label to an absolute point anywhere on the chart), JCChartLabel.ATTACH_DATACOORD (attach label to a point in the data space on the chart area), and JCChartLabel.ATTACH_DATAINDEX (attach the label to a specific point/bar/slice on the chart).
Component	The Swing component used as the chart label. By default, this is a JLabel instance.
Coord	The coordinate in the chart's space where the label is to be attached.
DataCoord	The coordinate in the chart area's data space where the label is to be attached.

Name	Description
DataIndex	A data index representing the point/bar/slice in the chart to which the label is to be attached.
DataView	For labels using ATTACH_DATACOORD, this property specifies which ChartDataView's axes should be used.
DwellLabel	When DwellLabel is set to true, the label is only displayed when the cursor is over the point/bar/slice that the label is attached to. This property is only used when the label is attached using ATTACH_DATAINDEX. When set to false (the default), the label is always displayed.
Offset	Offset specifies where the label should be positioned relative to the position the labels thinks it should be, depending on what the label's attachMethod is.
ParentManager	The ParentManager property is the ChartLabelManager instance that controls the JCChartLabel.
Text	The Text property controls the text displayed inside the label.

A.13   JCChartLabelManager

Name	Description
AutoLabelList	The AutoLabelList property is a two-dimensional array of automatically-generated JCChartLabel instances, one for every point and series. The inner array is indexed by point; the outer array by series. Default is empty.

A.14   JCChartStyle

Name	Description
FillColor	The FillColor property determines the color used to fill regions in chart. Default value is generated.
FillImage	The FillImage property determines the image used to paint the fill region of Bar and Area charts. Default value is null.
FillPattern	The FillPattern property determines the fill pattern used to fill regions in chart. This is only supported in JDK 1.3.1 and higher. Default value is JCFillStyle.SOLID.

Name	Description
FillStyle	The FillStyle property controls the appearance of filled areas in chart. See JCFillStyle for additional properties. Note that all JCChartStyle properties of the format Fill* are virtual properties that map to properties of JCFillStyle.
LineCap	The LineCap property specifies the cap style used to end a line. This is only supported in JDK 1.3.1 and higher. Valid values include BasicStroke.CAP_BUTT, BasicStroke.CAP_ROUND, and BasicStroke.CAP_SQUARE.
LineColor	The LineColor property determines the color used to draw a line. Default value is generated.
LineJoin	The LineJoin property specifies the join style used to join two lines. This is only supported in JDK 1.3.1 and higher. Valid values include BasicStroke.JOIN_MITER, BasicStroke.JOIN_BEVEL, and BasicStroke.JOIN_ROUND.
LinePattern	The LinePattern property dictates the pattern used to draw a line. Valid values include JCLineStyle.NONE, JCLineStyle.SOLID, JCLineStyle.LONG_DASH, JCLineStyle.SHORT_DASH, JCLineStyle.LSL_DASH, and JCLineStyle.DASH_DOT. This is only supported in JDK 1.3.1 and higher. Default value is JCLineStyle.SOLID.
LineStyle	The LineStyle property controls the appearance of lines in chart. See JCLineStyle for additional properties.
LineWidth	The LineWidth property controls the line width. This is only supported in JDK 1.3.1 and higher. Default value is 1.
SymbolColor	The SymbolColor property determines the color used to paint the symbol. Default value is generated.
SymbolCustomShape	The SymbolCustomShape property contains an object derived from JCShape that is used to draw points. See JCShape for details. Default value is null.
SymbolShape	The SymbolShape property determines the type of symbol that will be drawn. Valid values include JCSymbolStyle.NONE, JCSymbolStyle.DOT, JCSymbolStyle.BOX, JCSymbolStyle.TRIANGLE, JCSymbolStyle.DIAMOND, JCSymbolStyle.STAR, JCSymbolStyle.VERT_LINE, JCSymbolStyle.HORIZ_LINE, JCSymbolStyle.CROSS, JCSymbolStyle.CIRCLE, and JCSymbolStyle.SQUARE. Default value is generated.

Name	Description
SymbolSize	The `SymbolSize` property determines the size of the symbol. Default value is 6.
SymbolStyle	The `SymbolStyle` property controls the symbol that represents an individual point. See `JCSymbolStyle` for additional properties. Note that all `JCChartStyle` properties of the format Symbol* are virtual properties that map to properties of `JCSymbolStyle`.

A.15   JCFillStyle

Name	Description
Background	The `Background` property determines the background color used when painting patterned fills.
Color	The `Color` property determines the color used to fill regions in chart. The default value is generated.
CustomPaint	The `CustomPaint` property specifies the `TexturePaint` object used to paint the fill region when the pattern is set to `CUSTOM_PAINT`. This is only supported in JDK 1.3.1 and higher.
Image	The `Image` property determines the image used to paint the fill region when the pattern is set to `CUSTOM_FILL` or `CUSTOM_STACK`. The default value is `null`.
Pattern	The `Pattern` property determines the fill pattern used to fill regions in chart. This is only supported in JDK 1.3.1 and higher. The default value is `JCFillStyle.SOLID`.

Available fill patterns are: `NONE`, `SOLID`, `25_PERCENT`, `50_PERCENT`, `75_PERCENT`, `HORIZ_STRIPE`, `VERT_STRIPE`, `45_STRIPE`, `135_STRIPE`, `DIAG_HATCHED`, `CROSS_HATCHED`, `CUSTOM_FILL`, `CUSTOM_PAINT`, or, for bar charts only, `CUSTOM_STACK`. |

A.16   JCGridLegend

Name	Description
Anchor	The `Anchor` property determines the position of the legend relative to the ChartArea. Valid values include `JCLegend.NORTH`, `JCLegend.SOUTH`, `JCLegend.EAST`, `JCLegend.WEST`, `JCLegend.NORTHWEST`, `JCLegend.SOUTHWEST`, `JCLegend.NORTHEAST`, and `JCLegend.SOUTHEAST`. The default value is `JCLegend.EAST`.
Background	The `Background` property determines the background color used to draw inside the legend. Note that the `Background` property is inherited from the parent `JCChart`.
Font	The `Font` property determines what font is used to render text inside the legend. Note that the `Font` property is inherited from the parent `JCChart`.
Foreground	The `Foreground` property determines the foreground color used to draw inside the legend. Note that the `Foreground` property is inherited from the parent `JCChart`.
GroupGap	The `GroupGap` property determines the gap between groups of items in the chart legend (e.g. the columns/rows associated with a data view).
InsideItemGap	The `InsideItemGap` property determines the gap between the symbol and text portions of a legend item.
ItemGap	The `ItemGap` property determines the gap between the legend items in the same group.
MarginGap	The `MarginGap` property determines the gap between the edge of the legend and the start of the item layout.
Orientation	The `Orientation` property determines how legend information is laid out. Valid values include `JCLegend.VERTICAL` and `JCLegend.HORIZONTAL`. The default value is `JCLegend.VERTICAL`.
SymbolSize	The `SymbolSize` property determines the size of the symbol. Default value is 6.
Visible	The `Visible` property determines the gap between the legend items in the same group.

A.17   JCHLOCChartFormat

Name	Description
OpenCloseFullWidth	The `OpenCloseFullWidth` property indicates whether the open and close tick indications are drawn across the full width of the Hi-Lo bar or just on one side. The default value is `false`.
ShowingClose	The `ShowingClose` property indicates whether the close tick indication is shown or not. The tick appears to the right of the Hi-Lo line. The default value is `true`.
ShowingOpen	The `ShowingOpen` property indicates whether the open tick indication is shown or not. The tick appears to the left of the Hi-Lo line. The default value is `true`.
TickSize	The `TickSize` property specifies the tick size for open and close ticks.

A.18   JCLegend

Name	Description
Anchor	The `Anchor` property determines the position of the legend relative to the ChartArea. Valid values include `JCLegend.NORTH`, `JCLegend.SOUTH`, `JCLegend.EAST`, `JCLegend.WEST`, `JCLegend.NORTHWEST`, `JCLegend.SOUTHWEST`, `JCLegend.NORTHEAST`, and `JCLegend.SOUTHEAST`. The default value is `JCLegend.EAST`.
Background	The `Background` property determines the background color used to draw inside the legend. Note that the `Background` property is inherited from the parent `JCChart`.
Border	The `Border` property sets the border of a component. Note that the Border property is inherited from `JComponent`.
Font	The `Font` property determines what font is used to render text inside the legend. Note that the `Font` property is inherited from the parent `JCChart`.
Foreground	The `Foreground` property determines the foreground color used to draw inside the legend. Note that the `Foreground` property is inherited from the parent `JCChart`.

Name	Description
Opaque	The `Opaque` property determines the background color. If the component is completely opaque, the background will be filled with the background color. Otherwise, the background is transparent, and whatever is underneath will show through. Note, that the Opaque property is inherited from `JComponent`.
Orientation	The `Orientation` property determines how legend information is laid out. Valid values include `JCLegend.VERTICAL` and `JCLegend.HORIZONTAL`. The default value is `JCLegend.VERTICAL`.
Visible	The `Visible` property determines whether the legend is currently visible. Default value is `false`.

A.19   JCLineStyle

Name	Description
Cap	The `Cap` property specifies the cap style used to end a line. This is only supported in JDK 1.3.1 and higher. Valid values include `BasicStroke.CAP_BUTT`, `BasicStroke.CAP_ROUND`, and `BasicStroke.CAP_SQUARE`.
Color	The `Color` property determines the color used to draw a line. The default value is generated.
Join	The `Join` property specifies the join style used to join two lines. This is only supported in JDK 1.3.1 and higher. Valid values include `BasicStroke. JOIN_MITER`, `BasicStroke.JOIN_BEVEL`, and `BasicStroke. JOIN_ROUND`.
Pattern	The `Pattern` property dictates the pattern used to draw a line. Valid values include `JCLineStyle.NONE`, `JCLineStyle.SOLID`, `JCLineStyle.LONG_DASH`, `JCLineStyle.SHORT_DASH`, `JCLineStyle.LSL_DASH`, and `JCLineStyle.DASH_DOT`. This is only supported in JDK 1.3.1 and higher. The default value is `JCLineStyle.SOLID`.
Width	The `Width` property controls the line width. This is only supported in JDK 1.3.1 and higher. The default value is 1.

A.20   JCMultiColLegend

Name	Description
Anchor	The Anchor property determines the position of the legend relative to the ChartArea. Valid values include JCLegend.NORTH, JCLegend.SOUTH, JCLegend.EAST, JCLegend.WEST, JCLegend.NORTHWEST, JCLegend.SOUTHWEST, JCLegend.NORTHEAST, and JCLegend.SOUTHEAST. The default value is JCLegend.EAST.
Background	The Background property determines the background color used to draw inside the legend. Note that the Background property is inherited from the parent ChartRegion.
Border	The Border property sets the border of a component. Note that the Border property is inherited from JComponent.
Font	The Font property determines what font is used to render text inside the legend. Note that the Font property is inherited from the parent JCChart.
Foreground	The Foreground property determines the foreground color used to draw inside the legend. Note that the Foreground property is inherited from the parent JCChart.
GroupGap	The GroupGap property determines the gap between groups of items in the chart legend (e.g. the columns/rows associated with a data view).
Insets	The Insets property specifies the space that a container must leave at each of its edges. The space can be a border, a blank space, or a title.
InsideItemGap	The InsideItemGap property determines the gap between the symbol and text portions of a legend item.
ItemGap	The ItemGap property determines the gap between the legend items in the same group.
MarginGap	The MarginGap property determines the gap between the edge of the legend and the start of the item layout.
NumColumns	The NumColumns property determines the number of columns in this legend. If the number of columns is set to zero (the default), then the NumColumns will be adjusted automatically.
NumRows	The NumRows property determines the number of rows in this legend. If the number of rows is set to zero (the default), the number of rows will be adjusted automatically.

Name	Description
Orientation	The `Orientation` property determines how legend information is laid out. Valid values include `JCLegend.VERTICAL` and `JCLegend.HORIZONTAL`. The default value is `JCLegend.VERTICAL`.
SymbolSize	The `SymbolSize` property determines the size of the symbol. Default value is 6.

A.21 JCPieChartFormat

Name	Description
ExplodeList	The `ExplodeList` property specifies a list of exploded pie slices in the pie charts. Default value is an empty list.
ExplodeOffset	The `ExplodeOffset` property specifies the distance a slice is exploded from the center of a pie chart. Default value is 10.
MinSlices	The `MinSlices` property represents the minimum number of pie slices that Chart will try to display before grouping slices into the other slice. Default value is 5.
OtherLabel	The `OtherLabel` property represents the label used on the "other" pie slice. As with other point labels, the "other" label is a ChartText instance. Default value is " " (empty String).
OtherStyle	The `OtherStyle` property specifies the style used to render the "other" pie slice.
SortOrder	The `SortOrder` property determines the order in which pie slices will be displayed. Note that the other slice is always last in any ordering. Valid values include `JCPieChartFormat.ASCENDING_ORDER`, `JCPieChartFormat.DESCENDING_ORDER`, and `JCPieChartFormat.DATA_ORDER`. Default value is `JCPieChartFormat.DATA_ORDER`.

Name	Description
StartAngle	The position in the pie chart where the first pie slice is drawn. A value of zero degrees represents a horizontal line from the center of the pie to the right-hand side of the pie chart; a value of 90 degrees represents a vertical line from the center of the pie to the top-most point of the pie chart; a value of 180 degrees represents a horizontal line from the center of the pie to the left-hand side of the pie chart; and so on. Slices are drawn clockwise from the specified angle. Values must lie in the range from zero degrees to 360 degrees. The default value is 135 degrees.
ThresholdMethod	The ThresholdMethod property determines how the ThresholdValue property is used. If the method is SLICE_CUTOFF, the ThresholdValue is used as a cutoff to determine what items are lumped into the other slice. If the method is PIE_PERCENTILE, items are groups into the other slice until it represents "ThresholdValue" percent of the pie. Default value is SLICE_CUTOFF.
YAxisAngle	The YAxisAngle property determines the angle that the Y-axis makes with the axis origin base. Default value is 0 degrees.

A.22   JCPolarRadarChartFormat

Name	Description
HalfRange	The HalfRange property determines whether the X-axis for Polar charts consists of two half-ranges or one full range from 0 to 360 degrees.
OriginBase	The OriginBase property determines the angle of the theta axis origin in Polar, Radar, and Area Radar charts. Angles are based on zero degrees pointing east (the normal rectangular X-axis direction) with positive angles going counter-clockwise. The angle units are assumed to be the current value of the chart area's angleUnit property.
RadarCircularGrid	The YAxisGridCircular property determines whether gridlines are circular or "webbed" for Radar and Area Radar charts.
YAxisAngle	The YAxisAngle property determines the angle of the Y-axis in Polar, Radar, and Area Radar charts. Angles are relative to the current origin base. The angle units are assumed to be the current value of the chart area's angleUnit property.

A.23   JCSymbolStyle

Name	Description
Color	The `Color` property determines the color used to paint the symbol. The default value is generated.
CustomShape	The `CustomShape` property contains an object derived from `JCShape` that is used to draw points. See `JCShape` for details. The default value is `null`.
Shape	The `Shape` property determines the shape of symbol that will be drawn. Valid values include `JCSymbolStyle.NONE`, `JCSymbolStyle.DOT`, `JCSymbolStyle.BOX`, `JCSymbolStyle.TRIANGLE`, `JCSymbolStyle.DIAMOND`, `JCSymbolStyle.STAR`, `JCSymbolStyle.VERT_LINE`, `JCSymbolStyle.HORIZ_LINE`, `JCSymbolStyle.CROSS`, `JCSymbolStyle.CIRCLE` and `JCSymbolStyle.SQUARE`. The default value is `JCSymbolStyle.DOT`.
Size	The `Size` property determines the size of the symbol. The default value is 6.

A.24   JCValueLabel

Name	Description
ChartText	The `ChartText` property controls the ChartText associated with this Value label. The default value is a ChartText instance.
Text	The `Text` property specifies the text displayed inside the label. The default value is " " (empty String).
Value	The `Value` property controls the position of a label in data space along a particular axis. The default value is 0.0.

A.25   PlotArea

Name	Description
Background	The `Background` property determines the background color used to draw inside the chart region. Note that the `Background` is inherited from the parent ChartRegion.
Bottom	The `Bottom` property determines the location of the bottom of the PlotArea

Name	Description
BottomIsDefault	The BottomIsDefault property determines whether the Bottom of the chart region is calculated by Chart (true) or taken from the Bottom property (false).
Foreground	The Foreground property determines the color used to draw the axis bounding box controlled by JCChartArea. Note that the Foreground property is inherited from the parent ChartRegion.
Left	The Left property determines the location of the left of the PlotArea
LeftIsDefault	The LeftIsDefault property determines whether the left position of the chart region is calculated by Chart (true) or taken from the Left property (false).
Right	The Right property determines the Right of the PlotArea.
RightIsDefault	The RightIsDefault property determines whether the Right of the chart region is calculated by Chart (true) or taken from the Right property (false).
Top	The Top property determines the location of the top of the PlotArea.
TopIsDefault	The TopIsDefault property determines whether the top position of the chart region is calculated by Chart (true) or taken from the Top property (false).

A.26   SimpleChart

Name	Description
AxisOrientation	The AxisOrientation property determines if the X- and Y-axes are inverted and reversed.
Background	The Background property determines the background color used to draw inside the chart region. Note that the Background property is inherited from the parent JCComponent.
ChartType	The ChartType property determines the chart type of the first set of data in the chart.
Data	The Data property controls the file or URL used for the first set of data in chart.

Name	Description
Font	The Font property determines what font is used to render text inside the chart region. Note that the Font property is inherited from the parent JCComponent.
FooterFont	The FooterFont property determines what font is used to render text inside the footer region.
FooterText	The FooterText property holds the text that is displayed in the footer. The default value is " " (empty String).
Foreground	The Foreground property determines the foreground color used to draw inside the chart region. Note that the Foreground property is inherited from the parent JCComponent.
HeaderFont	The HeaderFont property determines what font is used to render text inside the header region.
HeaderText	The HeaderText property holds the text that is displayed in the header. The default value is " " (empty String).
LegendAnchor	The LegendAnchor property determines the position of the legend relative to the ChartArea. Valid values include NORTH, SOUTH, EAST, WEST, NORTHWEST, SOUTHWEST, NORTHEAST, and SOUTHEAST. The default value is EAST.
LegendOrientation	The LegendOrientation property determines how legend information is laid out. Valid values include VERTICAL and HORIZONTAL. The default value is VERTICAL.
LegendVisible	The LegendVisible property determines whether the legend is currently visible. Default value is false.
SwingDataModel	Sets the chart's data source to use a specified Swing TableModel object, instead of using the Data property.
View3D	The View3D property combines the values of the Depth, Elevation, and Rotation properties defined in JCChartArea. Depth controls the apparent depth of a graph. Elevation controls the distance above the X-axis for the 3D effect. Rotation controls the position of the eye relative to the Y-axis for the 3D effect. The default value is "0.0,0.0,0.0".

Name	Description
XAxisAnnotationMethod	The XAxisAnnotationMethod property determines how axis annotations are generated. Valid values include VALUE (annotation is generated by Chart, with possible callbacks to a label generator), VALUE_LABELS (annotation is taken from a list of value labels provided by the user – a value label is a label that appears at a particular axis value), POINT_LABELS (annotation comes from the data source's point labels that are associated with particular data points), and TIME_LABELS (Chart generates time/date labels based on the TimeUnit, TimeBase and TimeFormat properties). The default value is VALUE.
XAxisGridVisible	The XAxisGridVisible property determines whether a grid is drawn for the axis. The default value is false.
XAxisLogarithmic	The XAxisLogarithmic property determines whether the first X-axis will be logarithmic (true) or linear (false). The default value is false.
XAxisMinMax	The XAxisMinMax controls both the XAxisMin and XAxisMax properties. The XAxisMin property controls the minimum value shown on the axis. If a null String is used, Chart will calculate the axis minimum. The data minimum is determined by Chart. The default value is calculated. The XAxisMax property controls the maximum value shown on the axis. If a null String is used, Chart will calculate the axis maximum. The data maximum is determined by Chart. The default value is calculated.
XAxisNumSpacing	The XAxisNumSpacing property controls the interval between axis labels. If a null String is used, Chart will calculate the interval. The default value is calculated.
XAxisTitleText	The XAxisTitleText property specifies the text that will appear as the X-axis title. The default value is " " (empty String).
XAxisVisible	The XAxisVisible property determines whether the first X-axis is currently visible. Default value is true.

Name	Description
YAxisAnnotationMethod	The `YAxisAnnotationMethod` property determines how axis annotations are generated. Valid values include `VALUE` (annotation is generated by Chart, with possible callbacks to a label generator), `VALUE_LABELS` (annotation is taken from a list of value labels provided by the user – a value label is a label that appears at a particular axis value), `POINT_LABELS` (annotation comes from the data source's point labels that are associated with particular data points), and `TIME_LABELS` (Chart generates time/date labels based on the `TimeUnit`, `TimeBase` and `TimeFormat` properties). The default value is `VALUE`.
YAxisGridVisible	The `YAxisGridVisible` property determines whether a grid is drawn for the axis.
YAxisLogarithmic	The `YAxisLogarithmic` property determines whether the first Y-axis will be logarithmic (`true`) or linear (`false`). The default value is `false`.
YAxisMinMax	The `YAxisMinMax` controls both the `YAxisMin` and `YAxisMax` properties. The `YAxisMin` property controls the minimum value shown on the axis. If a null String is used, Chart will calculate the axis min. The data min is determined by Chart. The default value is calculated. The `YAxisMax` property controls the maximum value shown on the axis. If a null String is used, Chart will calculate the axis max. The data max is determined by Chart. The default value is calculated.
YAxisNumSpacing	The `YAxisNumSpacing` property controls the interval between axis labels. If a null String is used, Chart will calculate the interval. The default value is calculated.
YAxisTitleText	The `YAxisTitleText` property specifies the text that will appear as the Y-axis title. The default value is " " (empty String).
YAxisVisible	The `YAxisVisible` property determines whether the first Y-axis is currently visible. Default value is `true`.

Appendix B

Distributing Applets and Applications

Using JClass JarMaster to Customize the Deployment Archive

B.1   Using JClass JarMaster to Customize the Deployment Archive

The size of the archive and its related download time are important factors to consider when deploying your applet or application.

When you create an applet or an application using third-party classes such as JClass components, your deployment archive will contain many unused class files unless you customize your JAR. Optimally, the deployment JAR should contain only your classes and the third-party classes you actually use. For example, the *jcchart.jar*, which you used to develop your applet or application, contains classes and packages that are only useful during the development process and that are not referenced by your application. These classes include the Property Editors and BeanInfo classes. JClass JarMaster helps you create a deployment JAR that contains only the class files required to run your application.

JClass JarMaster is a robust utility that allows you to customize and reduce the size of the deployment archive quickly and easily. Using JClass JarMaster you can select the classes you know must belong in your JAR, and JarMaster will automatically search for all of the direct and indirect dependencies (supporting classes).

When you optimize the size of the deployment JAR with JClass JarMaster, you save yourself the time and trouble of building a JAR manually and determining the necessity of each class or package. Your deployment JAR will take less time to load and will use less space on your server as a direct result of excluding all of the classes that are never used by your applet or application.

For more information about using JarMaster to create and edit JARs, please consult its online documentation.

JClass JarMaster is installed automatically as part of the install process for JClass DesktopViews. For more details please refer to Quest Software's Web site.

Appendix C

HTML Property Reference

This appendix lists the syntax of JClass Chart properties when specified in an HTML file.
For example, the following HTML code sets the X-axis annotation method property:

```
<PARAM NAME="xaxis.annotationMethod" VALUE="POINT_LABELS">
```

C.1 ChartDataView Properties

Java Property	HTML Syntax	Value Type
Auto Label	*data*.autoLabel	boolean
Buffer Plot Data	*data*.bufferPlotData	boolean
Character Set	*data*.fileCharset	String
Chart Type	*data*.chartType	enum
Data	*data*	AppletDataSource
Data File	*data*File, *data*1File, or *data*2File	URLDataSource, FileDataSource
Data Name	dataName*n*	String[1]
Draw Front Plane	*data*.drawFrontPlane	boolean
Fast Update	*data*.fastUpdate	boolean
File Access	*data*.fileAccess	String
File Type	*data*.fileType	■ XML ■ Text
Hole Value	*data*.holeValue	double
Inverted	*data*.inverted	boolean
Outline Color	*data*.line.color	Color
Outline Cap	*data*.line.cap	enum
Outline Join	*data*.line.join	enum
Outline Pattern	*data*.line.pattern	enum
Outline Width	*data*.line.width	int
Point Labels	*data*.pointLabels	String
Visible	*data*.visible	boolean
Visible In Legend	*data*.visibleInLegend	boolean
X Axis	*data*.xaxis	X axis name
Y Axis	*data*.yaxis	Y axis name

[1]n is the data view number; not needed for first data view.

Note: `data` is the name of the first dataset, generated when chart properties are saved to an HTML file; additional datasets are named `data1`, `data2`, `data`*n*.

C.2   ChartDataViewSeries Properties

Java Property	HTML Syntax	Value Type
File Name	*data*.series*n*.fill.image.fileName	String
File Access Type	*data*.series*n*.fill.image.fileAccess	String
Fill Background	*data*.series*n*.fill.background	enum
Fill Color	*data*.series*n*.fill.color	Color
Fill Color Index	*data*.series*n*.fill.colorIndex	int
Fill Image	*data*.series*n*.fill.image	Image
Fill Pattern	*data*.series*n*.fill.pattern	enum
First Point	*data*.series*n*.firstPoint	int
Included	*data*.series*n*.Included	boolean
Label	*data*.series*n*.label	String
Last Point	*data*.series*n*.lastPoint	int
Line Color	*data*.series*n*.line.color	Color
Line Color Index	*data*.series*n*.line.colorIndex	int
Line Cap	*data*.series*n*.line.cap	enum
Line Join	*data*.series*n*.line.join	enum
Line Pattern	*data*.series*n*.line.pattern	enum
Line Width	*data*.series*n*.line.width	int
Symbol Color	*data*.series*n*.symbol.color	Color
Symbol Color Index	*data*.series*n*.symbol.colorIndex	int
Symbol Shape	*data*.series*n*.symbol.shape	enum
Symbol Shape Index	*data*.series*n*.symbol.symbolIndex	int
Symbol Size	*data*.series*n*.symbol.size	int
Visible	*data*.series*n*.visible	boolean
Visible In Legend	*data*.series*n*.visibleInLegend	boolean

Note: data is the name of the first dataset, generated when chart properties are saved to an HTML file; additional datasets are named data1, data2, data*n*.

C.3   JCAreaChartFormat Properties

Java Property	HTML Syntax	Value Type
100 Percent	*data*.Area.100Percent	boolean

Note: data is the name of the first dataset, generated when chart properties are saved to an HTML file; additional datasets are named data1, data2, data*n*.

C.4   JCAxis X- and Y-axes Properties

Java Property	HTML Syntax	Value Type
Annotation Method	*[xy]axis*.annotationMethod	enum
Annotation Rotation	*[xy]axis*.annotationRotation	enum
Editable	*[xy]axis*.editable	boolean
Font	*[xy]axis*.font	Font
Foreground	*[xy]axis*.foreground	Color
Formula Constant	*[xy]axis*.formula.constant	double
Formula Multiplier	*[xy]axis*.formula.multiplier	double
Formula Originator	*[xy]axis*.formula.originator	Axis Name, eg, xaxis1
Gap	*[xy]axis*.gap	int
Grid Color	*[xy]axis*.grid.color	Color
Grid Visible	*[xy]axis*.grid.visible	boolean
Grid Spacing	*[xy]axis*.grid.spacing	double
Logarithmic	*[xy]axis*.logarithmic	boolean
Max	*[xy]axis*.max	double
Min	*[xy]axis*.min	double
Num Spacing	*[xy]axis*.numSpacing	double
Origin	*[xy]axis*.origin	double
Origin Placement	*[xy]axis*.originPlacement	enum
Placement	*[xy]axis*.placement	enum
Placement Axis	*[xy]axis*.placementAxis	Axis Name, eg. xaxis1
Placement Location	*[xy]axis*.placementLocation	double
Precision	*[xy]axis*.precision	int
Reversed	*[xy]axis*.reversed	boolean

Java Property	HTML Syntax	Value Type
Tick Spacing	*[xy]axis*.tickSpacing	double
Time Base	*[xy]axis*.timeBase	Date
Time Format	*[xy]axis*.timeFormat	String
Time Unit	*[xy]axis*.timeUnit	enum
Title Adjust	*[xy]axis*.title.adjust	enum
Title Background	*[xy]axis*.title.background	Color
Title Font	*[xy]axis*.title.font	Font
Title Foreground	*[xy]axis*.title.foreground	Color
Title Placement	*[xy]axis*.title.placement	enum
Title Rotation	*[xy]axis*.title.rotation	0, 90, 180, 270
Title Text	*[xy]axis*.title.text	String
Title Visible	*[xy]axis*.title.visible	boolean
Value Labels	*[xy]axis*.valueLabels	String[] (values separated by ";")
Visible	*[xy]axis*.visible	boolean

Note: xaxis and yaxis are the names of the first axes, generated when chart properties are saved to an HTML file; additional axes are named [xy]axis1, [xy]axis2, [xy]axis*n*.

C.5   JCBarChartFormat Properties

Java Property	HTML Syntax	Value Type
100 Percent	*data*.Bar.100Percent	boolean
Cluster Overlap	*data*.Bar.clusterOverlap	int
Cluster Width	*data*.Bar.clusterWidth	int

Note: data is the name of the first dataset, generated when chart properties are saved to an HTML file; additional datasets are named data1, data2, data*n*.

C.6   JCCandleChartFormat Properties

Java Property	HTML Syntax	Value Type
Complex	*data*.Candle.Complex	boolean

Note: data is the name of the first dataset, generated when chart properties are saved to an HTML file; additional datasets are named data1, data2, data*n*.

C.7 JCChart Properties

Java Property	HTML Syntax	Value Type
Allow User Changes	allowUserChanges	boolean
Background	background	Color
Batched	batched	boolean
Border	border	String[1]
Cancel Key	cancelKey	int
Customize Trigger	customizeTrigger	enum (see Note for details)
Depth Trigger	depthTrigger	enum (see Note for details)
Edit Trigger	editTrigger	enum (see Note for details)
Font	font	Font
Foreground	foreground	Color
Label Name	label$n$	String[2]
Opaque	opaque	boolean
Parameter File	paramFile	File from which to load additional properties
Pick Trigger	pickTrigger	enum (see Note for details)
Reset Key	resetKey	int
Rotate Trigger	rotateTrigger	enum (see Note for details)
Translate Trigger	translateTrigger	enum (see Note for details)
Zoom Trigger	zoomTrigger	enum (see Note for details)

[1]String of format bordertype|param1|param2|...
[2]label$n$ is the number of Chart Labels when chart properties are saved to HTML.

Note: Valid values for any Trigger property are NONE, CTRL, SHIFT, ALT, or META (equivalent to right-mouse-click).

C.8   JCChartArea Properties

Java Property	HTML Syntax	Value Type
Angle Unit	chartArea.angleUnit	enum
Axis Bounding Box	chartArea.axisBoundingBox	boolean
Background	chartArea.background	Color
Border	chartArea.border	String[1]
Depth	chartArea.depth	int
Elevation	chartArea.elevation	int
Fast Action	chartArea.fastAction	boolean
Font	chartArea.font	Font
Foreground	chartArea.foreground	Color
Height	chartArea.height	int
Horiz Action Axis	chartArea.horizActionAxis	Axis Name, eg. xaxis1
Insets	chartArea.insets	Insets
Opaque	chartArea.opaque	boolean
Plot Area Background	chartArea.plotArea.background	Color
Plot Area Bottom	chartArea.plotArea.bottom	int
Plot Area Foreground	chartArea.plotArea.foreground	Color
Plot Area Left	chartArea.plotArea.left	int
Plot Area Right	chartArea.plotArea.right	int
Plot Area Top	chartArea.plotArea.top	int
Rotation	chartArea.rotation	int
Vert Action Axis	chartArea.vertActionAxis	Axis Name, eg. xaxis1
Visible	chartArea.visible	boolean
Width	chartArea.width	int
X	chartArea.x	int
Y	chartArea.y	int

[1]String of format `bordertype|param1|param2|...`

C.9   JCChartLabel Properties

Java Property	HTML Syntax	Value Type
Anchor	label*n*.anchor	enum
Attach Method	label*n*.attachMethod	enum
Background	label*n*.background	Color
Connected	label*n*.connected	boolean
Coord	label*n*.coord	Point
Data Attach X	label*n*.dataAttachX	int
Data Attach Y	label*n*.dataAttachY	int
Data Index	label*n*.dataIndex	DataIndex Name, eg. indexName
Data View	label*n*.dataView	ChartDataView
Dwell Label	label*n*.dwellLabel	boolean
Font	label*n*.font	Font
Foreground	label*n*.foreground	Color
Label Name	labelName*n*	String (where *n* is the label number)
Last Label Index	lastLabelIndex	int[1]
Offset	label*n*.offset	Font
Text	label*n*.text	String
Visible	label*n*.visible	boolean

[1]The index of the last label. Used as the upper boundary on labels and data indices during load. Only needs to be explicitly specified if *n* is greater than 99.

Note: label1 is the name of the first Chart Label, generated when chart properties are saved to an HTML file; additional labels are named label2, label3, label*n*.

C.10   JCDataIndex Properties

Java Property	HTML Syntax	Value Type
Data View	index*n*.dataView	ChartDataView
Distance	index*n*.distance	int
Index Name	indexName*n*	String
Point	index*n*.point	Font
Series Index	index*n*.seriesIndex	int

Note: *n* is the index number.

C.11   JCHLOCChartFormat Properties

Java Property	HTML Syntax	Value Type
Line Color	*data*.HLOC.series*n*.hilo.line.color	Color
Line Width	*data*.HLOC.series*n*.hilo.line.width	int
Open Close Full Width	*data*.HLOC.openCloseFullWidth	boolean
Showing Close	*data*.HLOC.showingClose	boolean
Showing Open	*data*.HLOC.showingOpen	boolean
Tick Size	*data*.HLOC.series*n*.tickSize	int

Note: data is the name of the first dataset, generated when chart properties are saved to an HTML file; additional datasets are named data1, data2, data*n*.

C.12   JCHiLoChartFormat Properties

Java Property	HTML Syntax	Value Type
Line Color	*data*.Hilo.series*n*.line.color	Color
Line Width	*data*.Hilo.series*n*.line.width	int

Note: data is the name of the first dataset, generated when chart properties are saved to an HTML file; additional datasets are named data1, data2, data*n*.

C.13   JCLegend Properties

Java Property	HTML Syntax	Value Type
Anchor	legend.anchor	enum
Background	legend.background	Color
Border	legend.border	String[1]
Font	legend.font	Font
Foreground	legend.foreground	Color
Height	legend.height	int
Opaque	legend.opaque	boolean
Orientation	legend.orientation	enum
Visible	legend.visible	boolean
Width	legend.width	int
X	legend.x	int
Y	legend.y	int

[1]String of format `bordertype|param1|param2|...`

C.14   JCPieChartFormat Properties

Java Property	HTML Syntax	Value Type
Explode Offset	*data*.Pie.explodeOffset	int
Min Slices	*data*.Pie.minSlices	int
Other Fill Background	*data*.Pie.other.fill.background	enum
Other Fill Color	*data*.Pie.other.fill.color	Color
Other Fill Color Index	*data*.Pie.other.fill.colorIndex	int
Other Fill Image	*data*.Pie.other.fill.image	Image
Other Fill Image File Name	*data*.Pie.other.fill.image. fileName	String
Other Fill Image File Access	*data*.Pie.other.fill.image. fileAccess	String
Other Fill Pattern	*data*.Pie.other.fill.pattern	enum
Other Label	*data*.Pie.other.label	String
Sort Order	*data*.Pie.sortOrder	ASCENDING, DESCENDING
Start Angle	*data*.Pie.startAngle	double
Threshold Method	*data*.Pie.thresholdMethod	enum
Threshold Value	*data*.Pie.thresholdValue	int

Note: data is the name of the first dataset, generated when chart properties are saved to an HTML file; additional datasets are named data1, data2, data*n*.

C.15   JCPolarRadarChartFormat Properties

Java Property	HTML Syntax	Value Type
HalfRange	*data*.PolarRadar.halfRange	boolean
OriginBase	*data*.PolarRadar.originBase	double
RadarCircularGrid	*data*.PolarRadar.radarCircularGrid	boolean
YAxisAngle	*data*.PolarRadar.yAxisAngle	double

Note: data is the name of the first dataset, generated when chart properties are saved to an HTML file; additional datasets are named data1, data2, data*n*.

C.16   Header and Footer Properties

Java Property	HTML Syntax	Value Type
Background	header.background footer.background	Color
Border	border	String[1]
Font	header.font footer.font	Font
Foreground	header.foreground footer.foreground	Color
Height	height	int
Opaque	opaque	boolean
Text	header.orientation footer.orientation	String
Visible	header.visible footer.visible	boolean
Width	width	int
X	x	int
Y	y	int

[1]String of format bordertype|param1|param2|...

C.17   Example HTML File

The following HTML file defines the chart shown below:
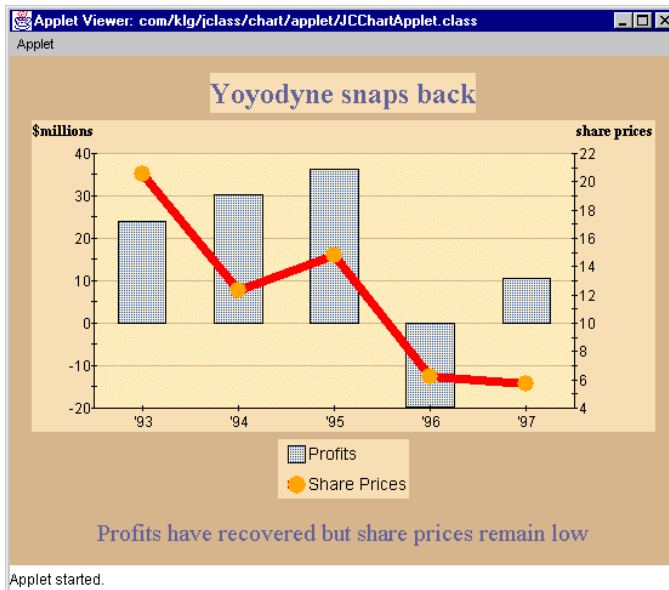

```
<HTML>
<HEAD>
<TITLE>JClass Chart</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFCC">
<FONT FACE="ARIAL,VERDANA,HELVETICA" SIZE="-1">
<CENTER><H2>Bar/Plot Combination</H2></CENTER>
<P>
<HR COLOR=CC3333>
<P>
<BLOCKQUOTE>
</BLOCKQUOTE>
<P>
<CENTER>
<APPLET CODE=com/klg/jclass/chart/applet/JCChartApplet.class ARCHIVE="lib/jcchart.jar"
CODEBASE="../../.." HEIGHT=420 WIDTH=550>
<PARAM NAME=background VALUE="210-180-140">
<PARAM NAME=foreground VALUE="black">
<PARAM NAME=font VALUE="Dialog-PLAIN-12">
<PARAM NAME=CustomizeTrigger VALUE="Meta">
<PARAM NAME=allowUserChanges VALUE="true">
<PARAM NAME=footer.y VALUE="55">
<PARAM NAME=footer.font VALUE="TimesRoman-PLAIN-20">
<PARAM NAME=footer.text VALUE="Profits have recovered but share prices remain low">
<PARAM NAME=footer.visible VALUE="true">
<PARAM NAME=header.border VALUE="bevel|raised">
<PARAM NAME=header.font VALUE="TimesRoman-BOLD-24">
```

```
<PARAM NAME=header.background VALUE="245-222-180">
<PARAM NAME=header.text VALUE="Yoyodyne snaps back">
<PARAM NAME=header.visible VALUE="true">
<PARAM NAME=legend.y VALUE="345">
<PARAM NAME=legend.border VALUE="etched|raised">
<PARAM NAME=legend.font VALUE="Dialog-PLAIN-14">
<PARAM NAME=legend.background VALUE="245-222-180">
<PARAM NAME=legend.visible VALUE="true">
<PARAM NAME=legend.anchor VALUE="South">
<PARAM NAME=legend.orientation VALUE="Horizontal">
<PARAM NAME=chartArea.y VALUE="90">
<PARAM NAME=chartArea.border VALUE="bevel|lowered">
<PARAM NAME=chartArea.background VALUE="245-222-180">
<PARAM NAME=chartArea.plotArea.background VALUE="255-232-190">
<PARAM NAME=xaxis.annotationMethod VALUE="Value_Labels">
<PARAM NAME=xaxis.placement VALUE="Min">
<PARAM NAME=xaxis.placementAxis VALUE="yaxis">
<PARAM NAME=xaxis.grid.Color VALUE="210-180-140">
<PARAM NAME=xaxis.valueLabels VALUE="1.0; '93; 2.0; '94; 3.0; '95; 4.0; '96; 5.0; '97">
<PARAM NAME=xaxis.title.visible VALUE="false">
<PARAM NAME=yaxis.placement VALUE="Min">
<PARAM NAME=yaxis.grid.visible VALUE="true">
<PARAM NAME=yaxis.grid.Color VALUE="210-180-140">
<PARAM NAME=yaxis.title.font VALUE="TimesRoman-BOLD-12">
<PARAM NAME=yaxis.title.text VALUE="$millions">
<PARAM NAME=chartArea.yaxisName1 VALUE="yaxis1">
<PARAM NAME=yaxis1.placement VALUE="Max">
<PARAM NAME=yaxis1.min VALUE="4.0">
<PARAM NAME=yaxis1.max VALUE="22.0">
<PARAM NAME=yaxis1.grid.Color VALUE="black">
<PARAM NAME=yaxis1.title.font VALUE="TimesRoman-BOLD-12">
<PARAM NAME=yaxis1.title.text VALUE="share prices ">
<PARAM NAME=data.chartType VALUE="BAR">
<PARAM NAME=data.line.color VALUE="black">
<PARAM NAME=data.series1.line.colorIndex VALUE="0">
<PARAM NAME=data.series1.line.width VALUE="8">
<PARAM NAME=data.series1.fill.colorIndex VALUE="0">
<PARAM NAME=data.series1.fill.color VALUE="0-84-255">
<PARAM NAME=data.series1.fill.pattern VALUE="Per_25">
<PARAM NAME=data.series1.symbol.colorIndex VALUE="0">
<PARAM NAME=data.series1.symbol.color VALUE="255-165-0">
<PARAM NAME=data.series1.symbol.size VALUE="7">
<PARAM NAME=data.series1.label VALUE="Profits">
<PARAM NAME=data.Bar.clusterWidth VALUE="50">
<PARAM NAME=data VALUE="
 ARRAY ' ' 1 5
 1.0 2.0 3.0 4.0 5.0
 24.0 30.2 36.4 -19.8 10.6
 ">
<PARAM NAME=dataName1 VALUE="data1">
<PARAM NAME=data1.outlineColor VALUE="black">
<PARAM NAME=data1.series1.line.colorIndex VALUE="1">
<PARAM NAME=data1.series1.line.color VALUE="red">
<PARAM NAME=data1.series1.line.width VALUE="7">
<PARAM NAME=data1.series1.fill.colorIndex VALUE="1">
<PARAM NAME=data1.series1.symbol.colorIndex VALUE="1">
```

```
<PARAM NAME=data1.series1.symbol.color VALUE="255-165-0">
<PARAM NAME=data1.series1.symbol.shape VALUE="Dot">
<PARAM NAME=data1.series1.symbol.size VALUE="14">
<PARAM NAME=data1.series1.label VALUE="Share Prices">
<PARAM NAME=data1.yaxis VALUE="yaxis1">
<PARAM NAME=data1 VALUE="
ARRAY ' ' 1 5
 1.0 2.0 3.0 4.0 5.0
 20.5 12.3 14.8 6.2 5.75
 ">
</APPLET>
<P>
<B><I><A HREF="index.html">More Applet Demos...</A></I></B>
<P>
</CENTER>
<!-- copyright information added -->
<P>
<HR COLOR=CC3333>
<P>
<P><FONT FACE="ARIAL,VERDANA,HELVETICA" SIZE=-2><A HREF="
http://www.quest.com/corporate/copyright.html">Copyright&#169;</A>
2002 Quest Software </FONT></FONT>
</BODY>
</HTML>
```

Appendix D

Porting JClass 3.6.x Applications

D.1 Overview

The major changes are listed in the table below. Each change is discussed in more detail with a recommended porting strategy.

Change	Rationale
applet subpackage	Makes it easier to find applet load/save code. Important for users who wish to remove the applet code from deployment JARs.
beans subpackage	Makes it easier to find beans. Important for users who wish to remove the beans from deployment JARs.
Chart label components	Chart labels are no longer derived from components. Instead, they contain components. This is a more flexible scheme, since any `JComponent`-derived object can be used as a chart label.
Data Binding Changes	The data binding for Chart has been rewritten, resulting in some minor API changes.
data subpackage	Makes it easier to find stock data sources. Stock data sources now include the JC prefix.
`JCChartLabelManager`	Not every user requires chart labels. To reduce download, chart label management is deferred to an object called `JCChartLabelManager`.
New data model	Old model dated back to JDK 1.0.2. New model is easier to understand.
No more `JCString`	`JCString` has been replaced by HTML in cells.
Package name change (`com.klg.jclass.chart`)	Old package name pre-dated naming standard.

Change	Rationale
Pluggable header/footer	Header and footer are now `JComponents`. This allows re-use of Swing code, and adds flexibility to the product. It also adds casts to your code.
Swing-like API	JClass 4 is Swing-based. Applies to applet `PARAM` tags as well.
Use of collection classes	Collection classes weren't available for JDK 1.0.2. Use of collection classes adds flexibility.

D.2   Swing-like API

Chart's header, footer, chart area, and legend, and the chart itself are all derived from `JComponent`. The following changes to methods apply:

Chart 3.*	Chart 4.* and higher
`get/setBorderType()`	Replaced by `JComponent.setBorder()` Note that enum-based replacements for standard Swing borders from BorderFactory may be created.
`get/setBorderWidth()`	In Swing, borders have their own width.
`get/setHeight()`	All replaced by `JComponent.setBounds()`.
`get/setHeight()` `get/setWidth()` `get/setLeft()` `get/setTop()` Related `IsDefault` methods	All replaced by `JComponent.setBounds()`, `JComponent.setLocation()` and `JComponent.setSize()`. In Chart 4.* and higher, layout options for chart area, legend, header and footer are somewhat more limited. However, `JCChart` will now accept new layout managers. Also, `JCChart` allows specification of layout hints for header, footer, chart area and legend using `JCChart.setLayoutHints()`.
`setInets()`	No direct equivalent. Use borders.
`get/setIsShowing()`	`JComponent.get/setVisible()`
`draw()`	Now using Swing's paint mechanism.

In general, any property in Chart 3.* that started with "Is" has been modified. Changes include:

Category	Chart 3.*	Chart 4.* and higher
IsShowingVisible	ChartDataViewSeries.IsShowing	ChartDataViewSeries. visible
	ChartDataViewSeries.IsShowing InLegend	ChartDataViewSeries. visibleInLegend
	ChartDataView.IsShowingInLeg end	ChartDataView. visibleInLegend
	JCAxis.GridIsShowing	JCAxis.GridVisible
	JCAxis.IsShowing	JCAxis.visible
	JCAxisTitle.IsShowing	JCAxisTitle.visible
IsIncluded	ChartDataViewSeries.IsIncluded	ChartDataViewSeries. Included
IncludedIs Editable	JCAxis.IsEditable	JCAxis.Editable

D.3   New Data Model

The data model for Chart 4 is a change to a data series-based model from a table-based model used in Chart 3.

As an example, consider charting the following data points:

```
(1,20)
(2,70)
(3,50)
```

In Chart 3.*, the data model would have looked like:

```
import jclass.chart.Chartable;
import java.util.Vector;

public class simple implements Chartable {

double xdata[] = {1, 2, 3,};
double ydata[] = {20, 70, 50,};

public  int getDataInterpretation() {
    return Chartable.ARRAY;
}

public Object getDataItem(int row, int column) {
    if (row == 0) {
        return new Double(xdata[column]);
    }
    else if (row == 1) {
        return new Double(ydata[column]);
    }
    return null;
}

public Vector getRow(int row) {
    Vector rval = new Vector();
    if (row == 0) {
        for (int i = 0; i < xdata.length; i++) {
            rval.addElement(new Double(xdata[i]));
        }
    }
    else if (row == 1) {
        for (int i = 0; i < ydata.length; i++) {
            rval.addElement(new Double(xdata[i]));
        }
    }
    return rval;
}


public int getNumRows() {
    return 2;
}


public String[] getPointLabels() {
    return pointLabels;
}
}
```

(Note that the series and point label methods are not shown.)

In Chart 4.* and higher, the corresponding code is much simpler:

```
import com.klg.jclass.chart.ChartDataModel;

public class simple implements ChartDataModel {
double xdata[] = {1, 2, 3,};
double ydata[] = {20, 70, 50,};

public double[] getXSeries(int index) {
    return xdata;
}

public double[] getYSeries(int index) {
    return ydata;
}

public int getNumSeries() {
    return 1;
}
}
```

Most important to note is the different focus. In Chart 3.*, the model viewed data as a table. Depending on the data interpretation, each row was either an X-series or a Y-series. In Chart 4.* and higher, the X- and Y-data series are returned explicitly. Also, Double objects are no longer used. (Chart simply converted them to double.)

Chart 3.* allowed data models to update chart via Observer/Observable or event/listener. Chart 4.* and higher only allows event/listener.

Listed below are Chart 3.* data model classes, and their equivalent in Chart 4.* and higher

Chart 3.*	Chart 4.* and higher
Chartable	ChartDataModel and LabelledChartDataModel
EditableChartable	EditableChartDataModel
ChartDataModel	No equivalent. Observer/Observable is no longer used for updated chart data sources.
ChartDataListener	ChartDataListener
ChartDataEvent	ChartDataEvent
ChartDataSupport	ChartDataSupport
No equivalent	ChartDataManageable Tells JCChart that an object can manage ChartDataListeners
No equivalent	ChartDataManager Manages ChartDataListeners

D.4   New Data Subpackage

All stock data sources have been moved into a data subpackage. Some of the data source names have been changed. The next table explains the changes.

Chart 3.* (jclass.chart)	Chart 4.* and higher (com.klg.jclass.chart.data)
No equivalent	`BaseDataSource` Common base class for most stock data sources.
`AppletDataSource`	`JCAppletDataSource`
`ChartSwingDataSource`	`JCChartSwingDataSource`
`VectorDataSource`	`JCDefaultDataSource` Note that `JCDefaultDataSource` provides functionality that `VectorDataSource` did not.

No equivalent	`JCEditableDataSource`
	Editable version of `JCDefaultDataSource`
	`JCFileDataSource`

`InputStreamDataSource`	`JCInputStreamDataSource`
`StringDataSource`	`JCStringDataSource`
`URLDataSource`	`JCURLDataSource`
`JDBCDataSource`	`JDBCDataSource`

D.5   New Beans Subpackage

All the beans have been moved to the beans subpackage. There has been no bean property changes.

D.6   Data Binding Changes

The data binding beans remain in the same places. However, the `dataBindingMetaData` property has been replaced by `dataBindingConfig`.

D.7   New Applet Subpackage

All code dealing with loading or saving of Chart as HTML PARAM tags has been moved to an applet subpackage. This change should be transparent to users. Deployment JARs

for users not using HTML load/save can be made smaller by removing the applet subpackage.

Some parameter changes were necessary, mostly as a result of core chart API changes.

These changes are shown below:

Chart 3.*	Chart 4.* and higher
`LeftMargin`, `TopMargin`, `BottomMargin`, `RightMargin`	No equivalent
`BorderType`	No equivalent
`BorderWidth`	No equivalent
`DoubleBuffer`	No equivalent
`Offset`	No equivalent
`IsShowing`	`IsVisible`
`IsShowingInLegend`	`VisibleInLegend`
`IsIncluded`	Included
No equivalent	`Join`, `Cap` and `Background` in `series.line`
`axis.IsVertical`	`axis.Vertical`
`axis.IsLogarithmic`	`axis.Logarithmic`
`axis.IsReversed`	`axis.Reversed`
`axis.GridIsShowing`	`axis.grid.visible`
`axis.Grid*`	`axis.grid.*` Note that `axis.grid` now supports all line style properties, including patterns
`axis.IsEditable`	`axis.editable`
`chartLabel.attachX/Y`	`chartLabel.coord` format: x,y
`chartLabel.IsConnected`	`chartLabel.connected`
`chartLabel.IsDwellLabel`	`chartLabel.dwellLabel`
`candleChartFormat.isComplex`	`candleChartFormat.complex`
`HLOCChartFormat.isShowingOpen`	`HLOCChartFormat.showingOpen`
`HLOCChartFormat.isShowingClose`	`HLOCChartFormat.showingClose`
`HLOCChartFormat.isOpenCloseFullWidth`	`HLOCChartFormat.openCloseFullWidth`

D.8   Pluggable Header/Footer

Headers and footers can now be any JComponent-derived object. By default,
`JCChart.getHeader()` and `JCChart.getFooter()` return a JLabel. However, both
methods return objects of type JComponent. This means a cast is required. Code that
used to look like this:

```
chart.getHeader().setText("Foo")
```

can be converted to look like this:

```
JLabel header = (JLabel)chart.getHeader();
header.setText("Foo");
```

The full API for headers and footers is now defined by the `JComponent`-derived object
used for header/footer. By default, this is `JLabel`. Refer to the `JLabel` API for more
details.

D.9   JCChartLabelManager

As previously mentioned, chart label management has been removed to a delegate
object. The delegate must be of type `JCChartLabelManager`. A default implementation
called `JCDefaultChartLabelManager` is provided.

Use of the delegate results in a smaller deployment JAR for users who don't use chart
labels. It also helps focus the `JCChart` API by removing the chart label-related methods.

3.* (JCChart)	4.* and higher (JCChartLabelManager)
`void addChartLabel(JCChartLabel label)`	Moved to `JCChartLabelManager`
`void removeChartLabel(JCChartLabel label)`	Moved to `JCChartLabelManager`
`int getNumChartLabels()`	Moved to `JCChartLabelManager`
`void removeAllChartLabels()`	Moved to `JCChartLabelManager`
`JCChartLabel getChartLabels(int index)`	Moved to `JCChartLabelManager`
`void setChartLabels(int index, JCChartLabel label)`	Moved to `JCChartLabelManager`
`JCChartLabel[] getChartLabels()`	`List JCChartLabelManager.getChartLabels ()`

3.* (JCChart)	4.* and higher (JCChartLabelManager)
`void setChartLabels(JCChartLabel[] s)`	`void JCChartLabelManager.setChartLabels (List s)`

D.10   Chart Label Components

`JCChartLabel` is no longer a component, but contains a component. Therefore, all of the usual component methods like `getBackground()`, `getFont()`, and so on, need to be changed and prefaced by a call to `getComponent()`

e.g. `getComponent().getBackground()`

D.11   Use of Collection Classes

JClass Chart aggregates objects like `JCAxis`, `ChartDataView` and `ChartDataViewSeries` using collections. In Chart3.*, Vectors were used. The API has been updated to take advantage of the flexibility offered by collections.

In most cases, Chart would build the object arrays manually. Collections (and their iterators) allow Chart to expose the internal collection directly.

Changes include:

Chart 3.*	Chart 4.* and higher
`String[] ChartDataView.getPointLabels()`	`List ChartDataView.getPointLabels()`
`ChartDataViewSeries[] ChartDataView.getSeries()`	`List ChartDataView.getSeries()`
`JCChartStyle[] ChartDataView.getChartStyle()`	`List ChartDataView.getChartStyle()`
`ChartDataView[] JCChart.getDataView()`	`List JCChart.getDataView()`
`JCChartLabel[] JCChart.getChartLabels()`	`List JCChartLabelManager. getChartLabels()` `List JCDefaultChartLabelManager. getChartLabels()`
`JCAxis[] JCChartArea.getXAxis()`	`List JCChartArea.getXAxes()`

Chart 3.*	Chart 4.* and higher
`JCAxis[] JCChartArea.getYAxis()`	`List JCChartArea.getYAxes()`

For convenience, many of the index-based accessors remain. For example, you can still grab axes based on an index:

```
JCAxis xaxis = chart.getChartArea().getXAxis(1);
```

Collections allow users to take advantage of iterators. In Chart 3.*, iterating over all the X-axes required the following code:

```
JCAxis[] xaxes = chart.getChartArea().getXAxis();
for (int i = 0; i < xaxes; i++) {
    JCAxis xaxis = xaxes[i];
    // Do something interesting
}
```

In Chart 4.* and higher, iterators can be used:

```
for (ListIterator li = chart.getChartArea().getXAxes().listIterator();
    i.hasNext();) {
    JCAxis xaxis = (JCAxis)li.next();
}
```

D.12   No More JCString

`JCString`s have been replaced by HTML in cells. This is supported by Swing, and has been added to Chart where appropriate.

You can now put raw HTML into headers and footers, as long as the text starts with "<html>". HTML is also valid in axis annotations, axis titles and legend elements.

Index

foreground color 178
formatted file 134
full-range X-axis
  polar charts 31

G

gap 80
general
  data format 141
  data layout 13
  layout 143
get object properties 13
  Java code 14
getOutlineColor() 165
GIF 179
GIF file format 183, 184
gridlines 61, 81, 129

H

half-range X-axis
  polar charts 31
header 105
  HTML property syntax 251
  positioning 179
  titles 159
headers and footers 159
HeaderText 89
hi-lo chart 12, 46
  ChartStyle properties used 47
  customizing chart styles 47
  logical series 46
hi-lo-open-close chart 12, 46
hole value 142
  specifying 155
HoleValueChartDataModel 155
HTML property syntax
  ChartDataView 238
  ChartDataViewSeries 240
  Footer 251
  Header 251
  JCAreaChartFormat 241
  JCAxis X- and Y-axes 242
  JCBarChartFormat 243
  JCCandleChartFormat 244
  JCChart 245
  JCChartArea 246
  JCChartLabel 247
  JCDataIndex 248
  JCHiLoChartFormat 248
  JCHLOCChartFormat 248
  JCLegend 249
  JCPieChartFormat 250

JCPolarRadarChartFormat 251
Hypertext Markup Language (HTML) 14

I

IDE
  setting properties 20
image formats 183
images
  transparent 179
inheritance hierarchy 21
initialization 142
intelligent defaults 78
interacting with the chart 188
interactive
  chart 172
  labels 172
interactively
  setting properties 20
internationalization 26
introduction to JClass Chart 1
inverting a chart 124
inverting X- and Y-axis 109
IsOpenCloseFullWidth
  using for error bar charts 46

J

JavaBeans 49
  overview 49
JBdbChart
  data binding 146
JBdbChart Bean 70
JBuilder 67
JCAppletDataSource
  pre-built DataSource 134
JCAreaChartFormat
  HTML property syntax 241
  property summary 208
JCAxis
  AnnotationRotation property 128
  containment hierarchy 24
  IsLogarithmic property 127
  IsReversed property 125
  Min and Max properties 125
  property summary 209
  second Y-axis 131
JCAxis X- and Y-axes
  HTML property syntax 242
JCAxis.POINT_LABELS 114
JCAxis.TIME_LABELS 114
JCAxis.VALUE 114
JCAxis.VALUE_LABELS 114
JCAxisFormula

use with SimpleChart 69, 93, 95

T

TableModel 136
   use with SimpleChart 69
TableModel, use with SimpleChart 93, 95
technical support 5, 6
   contacting 5
   FAQs 6
terminology 9
TexturePaint 223
Title property (axis) 128
titles 159
transparency 179
transparent images 179
transposed data 144
trigger property 25
tutorial 101
   SimpleChart Bean 49
typographical conventions 2

U

unmap 186, 187
unpick 186, 199
updating chart data source 155

V

valid modifier 189
value annotation 60
Value_Labels notation 60
ValueLabels 117
ValueLabels axis annotation 117
values annotation 115
   numbering precision 115
valueToDate() method 121

X

X-axis
   full or half-range 31
   when chart inverted 124
   when logarithmic 127
XML 136
   examples in JClass 137
   further information 137
   hole parameter 140
   interpreter 137
   labels 140
   name parameter 140
   other parameters 140

primer 136
special characters 137
specifying data by point 139
specifying data by series 138
using in JClass 137

Y

Y-axis
   second Y-axis 130
   when chart inverted 124