JClass Elements™

Programmer's Guide

Version 6.3
for Java 2 (JDK 1.3.1 and higher)

Essential Enhancements and Extensions to the Basic Swing Components

8001 Irvine Center Drive
Irvine, CA 92618
949-754-8000
www.quest.com

April 2004

RefNo: PRGDE-JCSWS/630-04/2004

Warranty

The information contained in this document is subject to change without notice. Quest Software makes no warranty of any kind with respect to this information. QUEST SOFTWARE SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTY OF THE MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Quest Software shall not be liable for any direct, indirect, incidental, consequential, or other damage alleged in connection with the furnishing or use of this information.

Trademarks

JClass, JClass Chart, JClass Chart 3D, JClass DataSource, JClass Elements, JClass Field, JClass HiGrid, JClass JarMaster, JClass LiveTable, JClass PageLayout, JClass ServerChart, JClass ServerReport, JClass DesktopViews, and JClass ServerViews are trademarks of Quest Software, Inc. Other trademarks and registered trademarks used in this guide are property of their respective owners.

World Headquarters
8001 Irvine Center Drive
Irvine, CA 92618
www.quest.com
e-mail: info@quest.com
U.S. and Canada: 949.754.8000

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

This product includes software developed by the Apache Software Foundation http://www.apache.org/.

The JPEG Encoder and its associated classes are Copyright © 1998, James R. Weeks and BioElectroMech. This product is based in part on the work of the Independent JPEG Group.

> Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
>
> 1. Redistributions of source code must retain the above copyright notice, this list of conditions, all files included with the source code, and the following disclaimer.
> 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
>
> THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,

EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This product includes software developed by the JDOM Project (http://www.jdom.org/). Copyright © 2000-2002 Brett McLaughlin & Jason Hunter, all rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution.
3. The name "JDOM" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact license@jdom.org.
4. Products derived from this software may not be called "JDOM", nor may "JDOM" appear in their name, without prior written permission from the JDOM Project Management (pm@jdom.org).

THIS SOFTWARE IS PROVIDED “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE JDOM AUTHORS OR THE PROJECT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Table of Contents

Preface

Introducing JClass Elements

The Swing components are the most significant part of the Java Foundation Classes (JFC). Swing components cover basic needs, but some commonly useful items are missing. For instance, a Color Chooser component is included, but a Font Chooser is not.

JClass Elements is a broad collection of GUI components and utility classes designed to augment Swing's basic offerings. With JClass Elements, you have an extended set of off-the-shelf user interface components at your disposal. Moreover, because of their open design, it's easy to adapt them to your own custom needs.

Feature Overview

The classes of JClass Elements are distributed over three packages. `com.klg.jclass.util` contains a collection of utilities and `com.klg.jclass.swing` contains the more elaborate GUI components. There are additional classes in `com.klg.jclass.util.swing`. This package contains both utilities and some basic GUI components that add functionality to their Swing ancestors.

JClass Elements is a collection of utilities and GUI components which:

- Extends basic Swing functionality by rounding out the list of much-needed components.
- Simplifies your work by providing built-in functionality.
- Implements commonly-required utility functions.

JClass Elements may be used in conjunction with all of Quest Software's other JClass products, as well as with ordinary Swing components.

JClass Elements 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.

You can freely distribute Java applets and applications containing JClass components according to the terms of the *License Agreement* that appears at install time.

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 in this Manual

`Typewriter Font`	■ Java language source code and examples of file contents. ■ JClass Elements 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 the Manual

Part I –Components and Layout Managers – contains information about JClass Elements's GUI components. Each chapter explains what the component is, and describes how to use it in your development project. There is also a chapter describing the behavior of the layout managers that JClass Elements provides. These functional yet simple-to-use layout managers can ease your layout tasks.

Chapter 1, Introducing JClass Elements, provides an overview of the components in JClass Elements.

Chapter 2, CheckBox-List Component, describes the use of a component that associates check boxes with list items.

Chapter 3, Circular and Linear Gauges, describes the `JCCircularGauge` structure, a component for displaying and setting values on a circular dial or gauge.

Chapter 4, Date Chooser, describes the use of a graphical date chooser component.

Chapter 5, JCPopupCalendar Component, introduces a component that allows you to edit the date and time using a drop-down calendar.

Chapter 6, Exit Frame, outlines this subclass of Swing's `JFrame`, which is used to detect and react to window-closing events.

Chapter 7, Font Choosers, details the `JCFontChooser` class, which gives you an easy way of letting your end users change fonts.

Chapter 8, HTML/Help Panes, covers the use of this subclass of Swing's `JEditorPane`, which provides added HTML, hyperlink, and cursor changing functionality.

Chapter 9, Sortable Table, offers information about this index map sorting class.

Chapter 10, Multiple Document Frame, outlines this multiple document interface component, which allows you to put multiple windows in the same pane.

Chapter 11, Multi-Select List, covers the use of this dual-list component, which handles tasks like specifying file inclusion and exclusion by providing a GUI containing two list areas. Items can be moved from one list area to the other. The names in the *selected* list are marked for the action you designate, while those in the *deselected* list are excluded.

Chapter 12, Spin Boxes, presents an overview of this incrementing and decrementing component, which is used with `java.lang.Number` type objects.

Chapter 13, Splash Screen, shows you how to include a splash screen with your application.

Chapter 14, Tree/Table Components, provides information about the table component, which presents data as a hierarchical/tree listing or a non-hierarchical grid listing.

Chapter 15, Wizard Creator, covers the use of a component that manages pages with wizard-like behavior. Typically, these pages are dialogs that assist the end user in setting up custom configurations by organizing the setup procedure.

Chapter 16, Layout Managers, covers the behavior and use of the JClass Elements layout managers.

Part II– Utilities – describes how to use the utility classes in JClass Elements. Each chapter explains what the class is and describes how to use it in your development project.

Chapter 17, Introduction to the Utility Classes, describes JClass Elements's utility classes.

Chapter 18, Debugging Tools, covers this tool that provides three different types of debug printout control.

Chapter 19, JCFileFilter, provides a convenient way of passing Windows-style filename extensions to a Swing `JFileChooser` so that only files of the named types appear in the file chooser dialog.

Chapter 20, Icon Creator, outlines how you can use `String` arrays to create icon images. This eliminates the need to supply separate image files for the icons in your class.

Chapter 21, Image Encoder, describes how to use this class to provide a picture of your component.

Chapter 22, Listener List, describes how to use this class for keeping track of event listeners.

Chapter 23, Progress Helper, offers information about this index map sorting class.

Chapter 24, String Tokenizer, outlines the capabilities of this class, which lets you specify a delimiter and split a String into tokens.

Chapter 25, Thread Safety Utilities, describes the classes that help with thread safety.

Chapter 26, Tree Set, outlines the features of this class, which allows you to represent a set's elements as a sort tree.

Chapter 27, Type Converters, outlines how to use these classes to convert between data types.

Chapter 28, Word Wrap, shows how to add word wrapping functionality to a String.

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

Appendix A, Bean Properties Reference, gives important property details of all JClass Elements's components.

Appendix B, Distributing Applets and Applications, describes how to package your application for distribution using JClass JarMaster.

Appendix C, Colors and Fonts, provides you with a useful table of `Color` values.

API Reference

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

Licensing

In order to use JClass Elements, 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 Elements.

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://www.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 are at *http://java.sun.com/beans/resources.html*

These documents are not required to develop applications using JClass Elements, 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 Elements, 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

Components and Layout Managers

1

Introducing JClass Elements

1.1 How the Manual is Organized

For the most part, each chapter is devoted to a single component. This makes it easy to find a component or utility, and makes it easy to review its structure and usage. In very few cases, utilities that are very closely related are covered in the same chapter.

You're reading Part I right now. You'll find JClass Elements's Graphical User Interface (GUI) components and layout managers in Part I and JClass Elements's utility classes in Part II.

This chapter contains a general description of each GUI component and layout manager in the product.

1.2 Components and Layout Managers

It's as easy to use a JClass Elements component in your program as it is to use a Swing component. Where you would reference a `JComponent` if it were part of Swing, you instead reference a `JCComponent`, after making sure that the JClass Elements packages are on your CLASSPATH.

Some of JClass Elements's components, like `JCMDIFrame`, augment the standard Swing components. Others, like `JCFontChooser`, provide a new component, one that is not part of the standard Swing package. Either way, they add functionality.

Here's a brief note on each component or layout manager:

Component Name	Description
`JCCheckBoxList` 	This component lets you show a columnar list of items. A check box appears at the left of each item. A mode switch lets you set the selection policy between (a) only one box at a time, (b) a contiguous range, or (c) any group whatsoever. You can implement the `javax.swing.event.ListSelectionListener` interface to respond immediately when a user checks a box.
`JCCircularGauge` `JCLinearGauge` 	A graphical component for setting and displaying one or more quantifiable values. GUI designers may create innovative linear sliders and progress meters having a distinctive and unique flavor. `JCCircularGauge` makes it easy to create switches and dials for setting discrete values, as well as circular gauges for monitoring and setting a continuous range of values. `JCLinearGauge` vastly expands your design choices for linear interactive displays that have the same flexible functionality as the circular gauge.
`JCDateChooser` 	This component provides a convenient way of viewing and setting calendar information. The days of the month appear as a standard monthly calendar, the year is in a spin box, and you have a choice of a spin box or a pop-down list for the month.
`JCExitFrame` 	A frame that responds to window-closing events either by exiting or by becoming invisible.
`JCFontChooserPane` 	`JCFontChooser` is the abstract base class for `JCFontChooserBar` and `JCFontChooserPane`. It provides common data and methods for both components. Place one of these in any application where you want to let the end user choose fonts.
`JCHTMLPane` and `JCHelpPane` 	Ease of use is the key feature for these two components. Simply pass an HTML-encoded file to the pane and you have a mini browser. `JCHelpPane` supports up to three `JCHTMLPane`**s** and provides basic navigation buttons, allowing you to implement a simple HTML-based help system. The left pane functions as a table of contents, and the right pane shows the help pages.

Component Name	Description
`JCMDIFrame` and `JCMDIPane` 	Multiple document windows are great where multiple views of some multi-faceted object may be required, or multiple forms need to be simultaneously present in a window. The JClass versions optimize space by managing the active window's main menu bar, while providing the standard window-management options.
`JCMultiSelectList` 	`JCMultiSelectList` matches the API for `JList` except that two lists instead of one appear in the component's GUI. There are four buttons in between the list areas that move items between the lists. The left-hand list contains non-selected items and the right-hand list contains the selected items.
`JCSpinBox` 	Swing provides checkboxes and radio buttons, but no spin boxes. The JClass spin boxes fill the need for components that let the user select a number or a `String` by clicking on up- or down-arrows.
`JCSpinNumberBox` 	Use `JCSpinNumberBox` for incrementing and decrementing objects of type `java.lang.Number`. You can select numbers of type `Byte`, `Short`, `Integer`, `Long`, or `Float`, and you can set maximum and minimum values for the spin operation.
`JCSortableTable` 	A subclass of `JTable` that internally wraps any `TableModel` it is given with a `JCRowSortTableModel` and provides a `Comparator` that has a adjustable list of the column indexes that it uses for sorting. Clicking on a column header invokes the sorting behavior tied to that column, clicking again reverses the sort. It can be used to sort `Date`s, `Object`s that implement `Comparable`, and wrapped primitive types. For more information, see Features of JCSortableTable, in Chapter 9, for a description and examples.
`JCSplashScreen` 	A splash screen is an image that appears while an application is loading. It serves both as an indication that the program is being loaded from disk and as a place to put notices, such as copyrights, version or release numbers, and the like.
`JCTreeExplorer` 	A subclass of `JTable` that handles listeners, rendering, editing, and painting of a component that combines tree-like and table-like properties.

Component Name	Description
JCPopupCalendar 	JCPopupCalendar is a component that allows you to edit the date and time using a dropdown calendar.
JCTreeTable 	Swing's JTree and JTable are the two components that do more than merely display data; they attempt to manage the data as well. This becomes important when you need to organize large amounts of data and provide a view that displays a portion of it along with an indication of its relationship to the rest. Information that has a hierarchical structure, like a file system, can be displayed as tree data, while other types of data nicely fit a tabular format. There are a large number of data structures that combine tree-like and a table-like properties. A file system has a hierarchical organization that begs to be represented as a tree, yet the individual directories and files have properties, such as name, size, type, and date modified, that fit nicely in a row-column organization. Obviously there is a need for a component that lets you combine the look and functionality of both a tree and a table.
JCWizard and JCSplitWizard 	JCWizard and JCSplitWizard let you create and manage a Wizard-style group of dialogs by supplying informative events and special page components with standard buttons. You add a JCWizardListener to your JCWizardPages to invoke the actions that each page needs to perform.
Layout Managers	The layout managers are JCAlignLayout, JCColumnLayout, JCElasticLayout, JCGridLayout, and JCRowLayout. JCBorder, JCBox, JCBrace, and JCSpring are the associated components. Use them as enhancements to the AWT layout managers. Use JCAlignLayout to vertically arrange components with their associated labels, and JCRowLayout to arrange components in a single row. JCGridLayout improves AWT's GridLayout by sizing cells more intelligently. JCBorder lets you place your borders anywhere, not just around components.

The following table lists some JClass Elements objects, with their nearest Swing relatives. The accompanying description informs you about the advantages you gain by using the JClass Elements component.

Swing	JClass Elements	Description
No Swing equivalent	`JCCircularGauge` `JCLinearGauge`	Highly configurable circular and linear dials and gauges.
`JEditorPane`	`JCHTMLPane`	Its constructor takes either an HTML String or a URL, making it easy to add HTML pages to a pane. Follows hyperlinks without having to add listeners explicitly.
`JPane`	`JCMDIPane`	A pane that can hold multiple document interface (MDI) frames.
`JInternalFrame`	`JCMDIFrame`	Supports the multiple document interface paradigm with the automatic addition of a "Windows" menu to the parent menubar.
`JFrame`	`JCExitFrame`	Automatically responds to window closing events.
`JTree` `JTable`	`JCTreeTable` `JCTreeExplorer`	Components that combine tree and table views of hierarchically ordered data.

No Swing equivalent.	`JCFontChooser`
	`JCFontChooserBar`
	`JCFontChooserPane`
		Choose fonts from a menu or a dialog.

`(AWT) GridLayout`	`JCAlignLayout`	An easy way to lay out a two-column grid.
No Swing equivalent.	`JCElasticLayout`	For laying out components in a single row or a single column. Any leftover space in the component is divided among the components in the way you specify.
`JSpinner`	`JCSpinBox` `JCSpinNumberBox`	These spin boxes are the top half of a combo box. They are useful when you don't need a drop-down list, and they don't subclass from JComboBox. Instead, they inherit from Swing's `AbstractSpinBox`.

Swing	JClass Elements	Description
No Swing equivalent.	`JCDateChooser`	`JCDateChooser` is a component that displays a calendar in one of four variant forms. Each one displays the days of the month in the familiar form of a calendar, but varies the ways that the month and year are displayed.
No Swing equivalent.	`JCWordWrap`	Wraps lines, given a length and a newline delimiter.
`JProgressBar`	`JCProgressHelper`	A thread-safe class that reports via a dialog just how far along some time-consuming operation is.

1.3 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 will be 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*.)

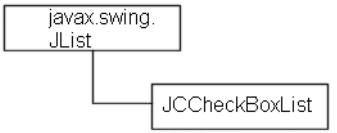
Essentially, developers of the localized software create their own resource bundles for their own locale. Developers should check every package for a */resources/* directory; if one is found, then the *.java* files in it will need to be localized.

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

2

CheckBox-List Component

2.1 Features of JCCheckBoxList



A `JCCheckBoxList` functions just like a `JList`, except that a check box appears to the left of the list items. (See Figure 2.)

- It is a subclass of the `JList` component that implements a `JCheckBox` as the cell renderer.
- To use a `com.klg.jclass.util.swing.JCCheckBoxList` in your application, simply ensure that the JClass Elements JAR is part of your CLASSPATH.
- Constructors are of the no argument type, or a single argument consisting of an instance of a `ListModel`, an array of `Objects` (usually `Strings`), or a `Vector` of list items.

- For comparison purposes, a `JCCheckBoxList` is shown beside a `JList` in the following figure.

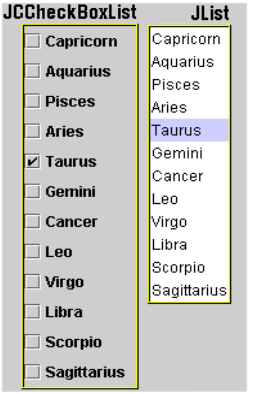


Figure 1 Comparison of a JCCheckBoxList with Swing's default JList.

2.2 Classes

`com.klg.jclass.util.swing.JCCheckBoxList` – The component itself.

`javax.swing.event.ListSelectionListener` – for listening to changes in the list.

2.3 Properties

`JCCheckBoxList` does not need any extra properties. You are free to use all the properties it inherits from `JList`.

For a full listing of the properties, see Appendix A, Bean Properties Reference.

2.4 Methods

`JCCheckBoxList` does not need any extra methods. You are free to use all the methods it inherits from `JList`. Selection is handled just like a `JList`; you can choose one of three selection modes: single, block, or multiple block.

2.5 Examples

This example shows the use of a `JCCheckBoxList`, including the action taken when an item is checked. If you look at the full listing in `examples.elements.CheckBoxList`, you'll

observe that the only modification needed to replace a `JList` with a `JCCheckBoxList` is the change of the component's name and this import line:

```
import com.klg.jclass.util.swing.JCCheckBoxList;
```

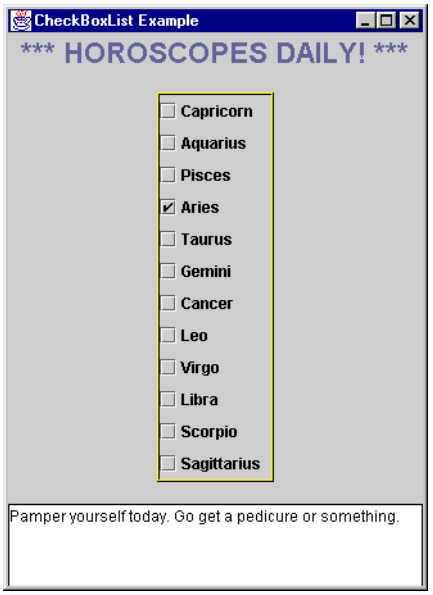


Figure 2 A JCCheckBoxList.

The code that instantiates a `JCCheckBoxList` is:

```
public JCCheckBoxList list;

        list = new JCCheckBoxList(data);
        list.setBorder(new EtchedBorder(Color.black, Color.yellow));
        list.setForeground(Color.white);
        list.setSelectionMode(0);
        list.addListSelectionListener(this);
        add(list);
```

Since `JCCheckBoxList` inherits all the properties of `JList`, it also uses `ListSelectionEvent` and `ListSelectionListener` to let you track the list items that have been selected. The code fragment shown below, also taken from `examples.elements.CheckBoxList`, shows how to use the `ListSelectionEvent` to react when items in a `JCCheckBoxList` are selected. In this example, a horoscope based on the selected item is placed in a text area.

```
//================== ListSelectionListener interface methods ========

public void valueChanged(ListSelectionEvent e) {
```

```
        int index = list.getSelectedIndex();

            switch(index) {
            case 0:
                horoscope.setText(
                "A homeless puppy will follow you home. Be good to it.");
                            break;
              ...
              More messages for the JTextArea
              ...
            case 19:
                horoscope.setText("You're running low on supplies. Run out &
                                  stock up.");
                break;

            }

        }

    }
```

3

Circular and Linear Gauges

3.1 Circular and Linear Gauges

The `JCGauge` components in JClass Elements provide you with realistic-looking instruments for your application's GUI design. End users can interact with colorful, easy-to-read meters on which they may view or set values. There are two basic types: circular and linear. Circular gauges may be used to give your application the flavor of an automobile instrument cluster, or alternatively an airline cockpit, control room, old time radio, or numerous other designs. Similarly, linear gauges may be made to look like thermometers (for hot and not-so-hot stock opportunities as well as temperature measurement), jazzed-up progress meters, and a variety of level and volume indicators. Your GUI design possibilities have been expanded, and you have the potential to make your GUIs even more visually appealing and user friendly.

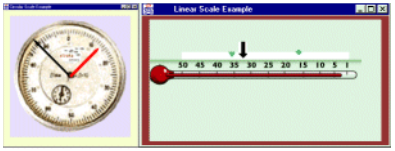


Figure 3 Sample circular (left) and linear (right) gauges.

3.1.1 Parts of a Gauge

Gauges, whether circular or linear, consist of visible objects and objects whose purpose is functional. The visible objects are:

- **Header.** A header provides a title for the gauge. Headers are `JComponent`s, and by default they are `JLabel`s.

- **Footer.** A footer provides another option for titling a gauge. It too is a `JComponent`, and by default a `JLabel`.
- **Placeable Labels.** Any component may be placed on a gauge at a specified position by employing an `add()` method that takes a `LinearConstraint` or `RadialConstraint` as its second parameter.
- **Scale.** A scale on which values may be defined. The scale can have associated `JCTick`, `JCNeedle`, `JCIndicator`, and `JCRange` objects.
- **Tick** and **Tick Label.** A tick object is used to show the scale values. It is a collection of uniformly spaced marks and labels.
- **Indicators** and **Needles.** Whereas a tick object is actually a collection of tick marks, an indicator is a single marker placed at a particular scale value. A needle is a subclass of an indicator that may be made interactive, allowing end users to set a new value for a parameter by clicking and dragging the needle to a new position. Like indicators, needles may be non interactive, but made to point to values under program control.

The diagram shows the various visible components of a gauge. Note that the gauge area (`JCGaugeArea`) contains the scale and its objects, but not the header, footer, or legend.

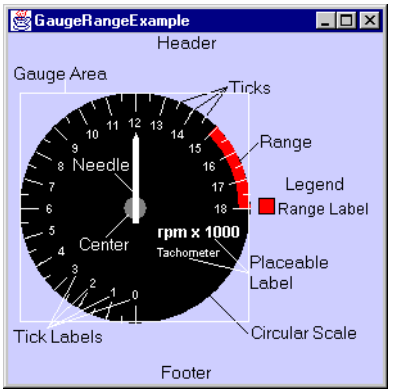


Figure 4 Objects within a gauge.

The non visible parts of a gauge are:

- Its **Pick Mechanism** and its **Mouse Interaction Mechanism**. These are described in Events and Listeners in JCGauge, in Chapter 3.
- **Gauge Constraint** and **Gauge Layout.** The gauge's layout and constraint classes assist in placing the gauge's various subcomponents.

Gauge appearance and interactivity

Because the gauge and many of its constituents are subclassed from `JComponent`, you have considerable flexibility in designing its appearance. Interfaces for indicators, needles, ranges, scales, and ticks let you replace the built-in objects with those you design yourself.

As well, it is easy to add additional items like numerical counters, images, and such to the gauge.

Gauges behave like analog devices whose readings are only approximate. If a more precise digital readout is desired, the needle's `getValue()` method, or the component's pick listener is employed to extract numeric values from the gauge. Figure 5 shows a circular gauge functioning as a speedometer. The needle points to the car's current speed. A numeric counter has been added to function as an odometer, and another counter displays a digital readout of the speed. This reading is more precise than the indication of the speed given by the needle.

Interactions between mouse and needle may be turned on or off. When an interaction is enabled, an end user can control the placement of the needle. If you wish, interactions can be turned off and the component can be used simply as a display device.

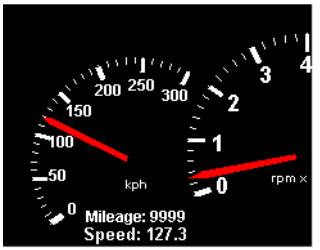


Figure 5 The odometer is a label positioned at the bottom of the speedometer.

You control the order in which objects are rendered to achieve your desired layering effect. See Rendering order for an explanation.

Caution: When setting a scale on a gauge, a center on a circular gauge, or adding indicators, needles, ranges, and ticks to a gauge, use the special set() and add() methods provided in JCGauge. Using standard `java.awt.Container`'s `add()` methods does not take into account the gauge's special requirements. When using the gauge's special `add()` methods you are still able to include an optional index that specifies the rendering order.

A gauge may have multiple instances of indicators, needles, ranges, and tick marks. In this case, the gauge maintains a collection for each of the object groups. The gauge has indicator and needle lists for keeping track of scale pointers, a range list for keeping track of the bands, called *ranges*, that can be used to mark various regions around the scale, and a tick list to keep track of the different tick objects.

Labels, which may be any `JComponent`, not just a `JLabel`, are created and manipulated individually rather than being stored in a list. There are special-purpose methods called `addLabel(label, radialConstraint)` and `addLabel(label, linearConstraint)` for placing labels at a specified location on the gauge.

Gauges may be added to other gauges. They may or may not share the same origin. See Section 3.21, Adding Other Components to a Gauge for a discussion on how to place components on a gauge and for an example of placing a smaller circular gauge on top of a larger one.

A gauge's size is determined by the size of its container. The container's layout manager, along with its preferred size setting, determine the gauge's initial size. Gauges may be resized as long as the layout manager permits it.

Rendering order

A render list, which in Java is often called the *z*-order of the components, is effectively created by the order in which components are added at execution time. This list determines in what order child objects are to be drawn. Components added last are drawn first. In a circular scale for example, by adding a needle and then a center, the center object is drawn first, then a needle, so that the needle is fully visible from center to tip instead of being partially covered by the center object. There are ways of manipulating the list so that a different drawing order can be specified. By drawing a needle first, it can appear to be attached to the edge of the center object rather than beginning at the center of the circle.

Inner and outer extents

The size of indicators, needles, ticks, and ranges is specified in part by two parameters called their inner and outer extents. These extents are defined as ratios based on the underlying scale. In the circular case, inner and outer extents refer to locations in a radial direction, with the center defined as 0.0. Thus, an object with an inner extent of 0.0 means that it is drawn from the center outwards to the position defined by the outer extent. If this outer extent is 0.8, the object extends out from the center a distance equal to 80% of the radius of the circular scale. Inner and outer extents on a circular scale are diagrammed in Figure 18.

In the linear case, how extents are measured depends on whether the orientation of the scale is horizontal or vertical. Since extents are measured in the direction transverse to the direction in which scale values increase, they are measured from the top edge of a horizontal scale, and from the left edge of a vertical scale. For example, an indicator on a vertical linear scale whose inner extent is 0.15 and whose outer extent is 0.75 is drawn beginning at 15% of the gauge's width from the left edge to 75% of the gauge's width.

3.1.2 Switches

A *switch* is the software equivalent to its electrical counterpart, a device that has a set of discrete, selectable positions. A simple switch may have only two states, on and off, or it may possess a number of selectable states. Figure 6 shows a switch shaped like a pointer that has six discrete states.

Switches may be implemented by having your code control the way that the gauge reacts to user input, but the easiest way to implement a switch is to choose a scale with integer

values that correspond to switch positions, then set the gauge's `snapToValue()` property to `true`:

```
// make it discrete
gauge.setSnapToValue(true);
```

By default, `snapToValue` is `false`.

See *GaugeSwitchExample.java* in the *examples/elements/switchdemo* directory for an example of a switch with six positions.

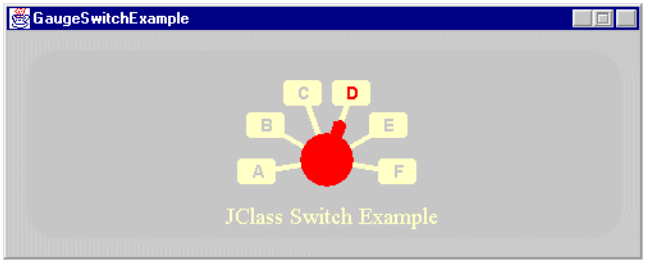


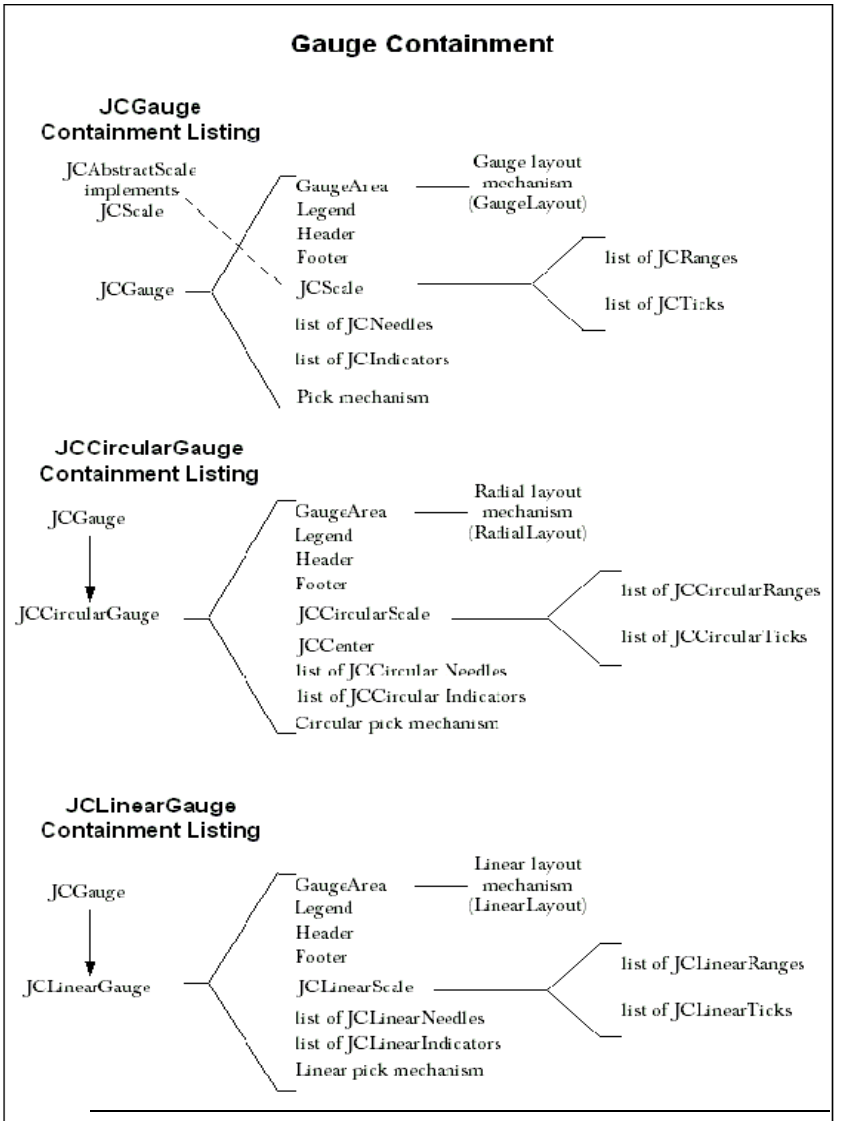
Figure 6 A switch with six discrete states.

If you require a switch with irregularly spaced stops you will need to add a pick listener and place the needle yourself in response to an end user's actions.

3.1.3 Organization of the Gauge Classes

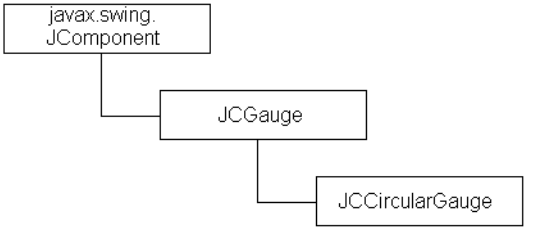
In keeping with the goal of making the gauges as configurable as possible, all gauge classes inherit from an abstract class `JCGauge`, which is itself a `JComponent`. `JCGauge` implements `MouseListener` and `MouseMotionListener` so that it may respond to mouse clicks and drags. This allows an end user to set a value by dragging a needle to a desired location on a scale, or by clicking on the scale and having the needle jump to the value determined by the position of the mouse pointer.

The diagram shows where you'll find the components and sub-objects that make up the gauge. Objects are designated by their class names.



3.2 Features of JCCircularGauge

`JCCircularGauge` is a subclass of `JComponent` whose on-screen representation looks like an analog circular measuring instrument. `JCCircularGaugeBean` in the *com/klg/jclass/swing/gaugebeans* directory is a JavaBean. It wraps properties found in `JCLinearGauge`'s contained objects so that they are easily accessible in an IDE.



`JCCircularGauge`s are containers for interactive circular meters that use a needle to point to a value and allow it to be measured on a scale. Circular gauges are constructed using an assortment of objects that provide structured functionality to the component as a whole.

Figure 7 shows the components that may be used in a gauge. The central element is a *circular scale*, which resides in a `JCGaugeArea`. The only way that a scale indicates its presence is if its foreground color is different from that of the gauge. The visible objects are a *center*, and collections of *indicators*, *needles*, *ticks,* and *ranges.* A center object, as its name implies, marks the center of a circular gauge. It may be a disk or an image. Needles perform measurement functions by pointing at scale values. Ticks provide visible scale markings with optional value labels so that scale values may be read. Ranges are bands that mark part or all of the scale to distinguish one part from another, like the red-line area of a tachometer. The range shown in Figure 7 runs along the circumference of the circular scale, thus marking its location. Outside the gauge area there is room for a *header*, *footer*, and *legend.* Header and footer elements are `JComponent`s, typically `JLabel`s, but the choice is yours. The legend is a `JCLegend`, a special type of `JComponent` that makes it easy to provide a legend that itemizes each component of a chosen type that the scale contains. Ranges are the default items for a legend, and may be itemized simply by naming the ranges and calling `gauge.getLegend().setVisible(true)`.

Alternatively, needles or other objects may be the items referred to in the legend list. If you do decide to place items other than ranges in the legend you will have to write your own legend populator. See Custom legends for details.

Also, through its inner class `GaugeType`, `JCCircularGauge` defines a set of configurations for circular gauges, such as `LOWER_RIGHT_QUARTER_CIRCLE` or `TOP_HALF_CIRCLE`, allowing

you to confine the gauge to a quadrant or a semicircle. There are nine choices in all. See Section 3.5.3, Properties, for the list of constants for `GaugeType`.

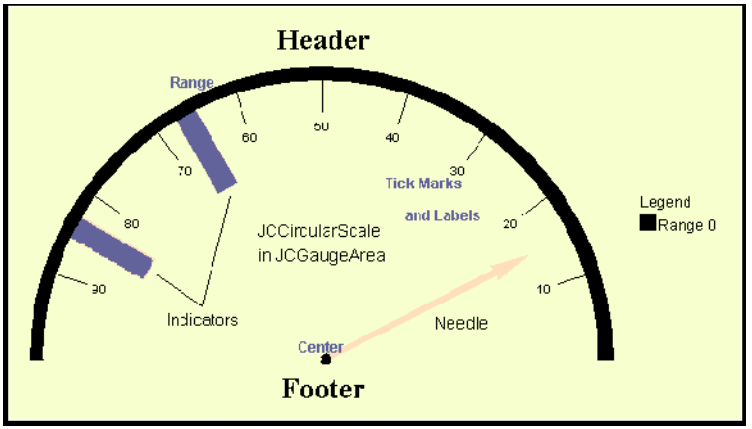
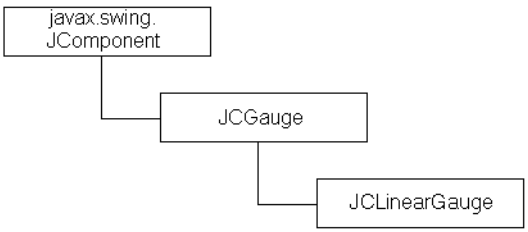


Figure 7 The major components in a circular gauge.

3.3 Features of JCLinearGauge

`JCLinearGauge` is a subclass of `JComponent` whose on-screen representation looks like an analog linear measuring instrument. `JCLinearGaugeBean` is a JavaBean. It wraps properties found in `JCLinearGauge`'s contained objects so that they are easily accessible in an IDE.



`JCLinearGauge`s are containers for interactive linear meters that use needles to point to values and allow these values to be measured on a scale. Linear gauges are constructed using an assortment of objects that provide structured functionality to the component as a whole.

Figure 8 shows the components that may be used in a linear gauge. The central element is a *linear scale*, which resides in a `JCGaugeArea`. The only way that a scale indicates its presence is if its foreground color is different from that of the gauge. The visible objects

are collections of *indicators*, *needles*, *ticks*, and *ranges*. There is no center object in a linear gauge. Needles perform measurement functions by pointing at scale values. Ticks provide visible scale markings with optional value labels so that scale values may be read. Ranges are bands that mark part or all of the scale to distinguish one part from another, like range shown in Figure 8, which is the small rectangle that runs between 85 and 100. Outside the gauge area there is room for a *header*, *footer*, and *legend*. Header and footer elements may be any `JComponent`, typically a `JLabel`. The legend is a `JCLegend`, a special type of `JComponent` that makes it easy to provide a legend that itemizes each component of a chosen type that the scale contains. Ranges are the default items for a legend, and may be itemized simply by naming the ranges and calling `gauge.getLegend().setVisible(true)`.

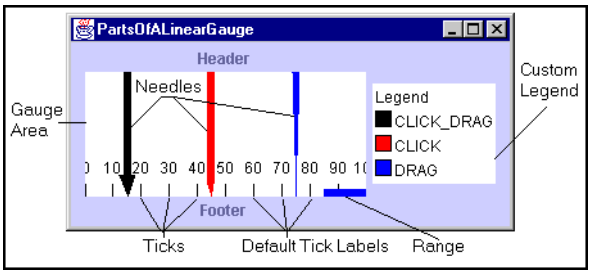


Figure 8 The major components of a linear gauge.

Note: Place a border around a linear gauge to ensure tick labels at the scale's extremities are fully visible.

3.4 JCGauge

`JCGauge` is the abstract superclass for `JCCircularGauge` and `JCLinearGauge`. `JCGauge` creates a header, footer, and legend whenever a circular or linear gauge is instantiated, and it has methods for adding or removing indicators, needles, ranges, and ticks.

Use the gauge's `add()` methods rather than the `add()` methods in `JComponent`. The `addLabel()` methods in `JCCircularGauge` and `JCLinearGauge` are there because these objects depend on `LinearConstraint` and `RadialConstraint` classes to position labels based on a linear extent and a pixel value, or a specified angle and radial distance. (See Section 3.16.1, RadialConstraint and RadialLayout, for more information.)

At any given time, each gauge contains one scale. A scale holds information about the minimum and maximum values for the scale, and the direction, forward or backward, in which scale values increase. The circular scale has start and stop angles that determine the portion of a full circle occupied by the scale. A needle object provides the visual indication of a particular value on the scale. Its length is specified by setting its *inner extent* and its *outer extent*. If the scale changes its size, the needle adjusts itself accordingly,

maintaining a proper position and proportional length relative to the size of the scale. See Section 3.14, The Indicator and Needle Objects, for details.

3.4.1 Constructor

`JCGauge` has an abstract no-argument constructor that is called when a circular or linear gauge is instantiated. It creates a default header and footer as `JLabel`s, and a default legend as a `JCGridLegend`.

3.4.2 Methods and Properties for JCGauge

`JCGauge` has these methods and properties:

Methods

JCGauge Method	Description
`addIndicator` `removeIndicator`	Adds or removes an indicator to the indicator collection for this gauge. Both take an indicator as a parameter, and an optional second parameter, an index, specifying its position in the `Vector` of needles.
`addNeedle()` `removeNeedle()`	Adds or removes a needle to the needle collection for this gauge. Both take a needle as a parameter, and an optional second parameter, an index, specifying its position in the `Vector` of needles.
`addPickListener()` `removePickListener()`	Adds or removes a pick listener for this needle. See Section 3.18, Events and Listeners in JCGauge.
`addRange()` `removeRange()`	A gauge maintains a list of ranges associated with it. These methods add or remove a specified range from the collection. They take the range as a parameter and an optional second parameter, an index, for its position in the Vector of ranges.
`addTick()` `removeTick()`	A gauge maintains a list of ticks associated with it. These methods add or remove a specified tick from the collection. They take the tick as a parameter and an optional second parameter, an index, for its position in the `Vector` of ticks.
`getComponentArea()`	Returns the JClass component's subcomponent on which the gauge will be drawn.
`getDrawingAreaHeight()`	Gets the height of the drawing area represented by this gauge.
`getDrawingAreaWidth()`	Gets the width of the drawing area represented by this gauge.
`getIndicators()`	Returns the list of indicators associated with this gauge.
`getNeedles()`	Returns the list of needles associated with this gauge.
`getRanges()`	Returns the list of ranges associated with this gauge.
`getScale()`	Returns the scale associated with this gauge.
`getTicks()`	Returns the tick objects for this gauge.

JCGauge Method	Description

`mouseClicked()`
`mouseDragged()`
`mouseEntered()`
`mouseExited()`
`mouseMoved()`
`mousePressed()`
`mouseReleased()`
	`mouseClicked` and `mouseDragged` send pick events to listeners. The other methods are empty, and must be overridden in a subclass if you wish to use them for your own purposes.

`pick()`	Given a screen location in pixels, returns the closest scale value as a `JCGaugePickEvent`.
`sendPickEvent()`	Broadcasts the pick event to interested listeners.

Properties

JCGauge Property	Description
`getFooter()` `setFooter()`	Returns or sets the footer for this gauge, a `JComponent`.
`getGaugeArea()` `setGaugeArea()`	Returns or sets the `JCGaugeArea` for this gauge, a `JCGaugeArea`. These methods, although declared public, are internal to JClass. There should be no need to obtain a reference to a `JCGaugeArea`.
`getHeader()` `setHeader()`	Returns or sets the header for this gauge, a `JComponent`.
`getLegend()` `setLegend()`	Returns or sets the legend for this gauge, a `JCLegend`.
`getRepaintEnabled()` `setRepaintEnabled()`	Disables or enables repaints of the gauge and its components.
`getSnapToValue()` `setSnapToValue()`	Returns or sets the `snapToValue` property that controls whether the needle should snap to the closest discrete integral scale value (`true`) or to any scale value (`false`).

3.5 JCCircularGauge

3.5.1 Constructors

Of the three constructors for `JCCircularGauge`, the default no-argument one supplies its own scale, then populates it with a center (a disk or an image), a needle, and a set of tick marks. The constructor that takes a Boolean value creates a gauge with an associated

circular scale if the Boolean parameter is `true`, or an empty gauge otherwise. In either case, the center, indicators, needles, and ticks must be added separately. The third constructor takes a gauge type as a parameter. It too creates an empty gauge of the specified type. The scale, center, indicators, needles, and ticks must be added separately.

3.5.2 Methods for JCCircularGauge

`JCCircularGauge` subclasses from `JComponent`, making it a `JComponent` with special capabilities for laying out subcomponents radially.

JCCircularGauge Method	Description
`addLabel()` `removeLabel()`	Adds or removes a label on the gauge. This is a general-purpose method, suitable for laying out any `JComponent` it is given.
`getClosestNeedle()`	Returns the closest needle to the clicked/dragged point. Returns closest needle of type CLICK/DRAG/CLICK_DRAG, or if there are none of these, the closest needle of type NONE.
`getScale()` `setScale()`	Gets or sets the scale used by the gauge. A scale contains the measurement parameters associated with a circular gauge. The `getScale()` method is inherited from `JCGauge`, but the `setScale()` method is overridden so that its argument must be a `JCCircularScale`. An optional second parameter to `setScale()` allows you to suppress the addition of the scale to the gauge area, but it is not expected that you will need to use this option.
`mouseClicked()`	Sends pick events to listeners and moves the closest needle with a `CLICK` interaction enabled to the value indicated by the mouse click.
`mouseDragged()`	Called during mouse drag events in the gauge to move the needle closest to the mouse.

3.5.3 Properties

A `JCCircularGauge` has the same properties as a `JComponent` and these additional ones:

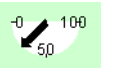
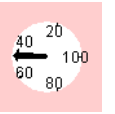
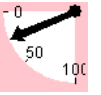
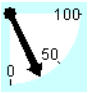
JCCircularGauge Property	Description
`getCenter()` `setCenter()`	Returns or sets the center for this gauge.

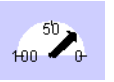
`getGaugeType()` `setGaugeType()`	Returns or sets the gauge type, one of the `JCCircularGauge.GaugeType` enums.

For a full listing of the properties, see the API for *com.klg.jclass.swing.gauge.JCCircularGauge*.

Type constants for JCCircularGauge

The inner class `JCCircularGauge.GaugeType` has these constants and methods:

Circular GaugeType Constant	Angular Span of the Circle, Semicircle, or Quadrant
`BOTTOM_HALF_CIRCLE`	180-0 degrees. See Section 3.10.3, Angles In a Circular Scale for a discussion of angular measurement in `JCCircularGauge`. 
`FULL_CIRCLE`	0-360 degrees 
`LEFT_HALF_CIRCLE`	90-180 degrees 
`LOWER_LEFT_QUARTER_CIRCLE`	180-270 degrees 
`LOWER_RIGHT_QUARTER_CIRCLE`	270-0 degrees 
`RIGHT_HALF_CIRCLE`	270-90 degrees 

Circular GaugeType Constant	Angular Span of the Circle, Semicircle, or Quadrant
TOP_HALF_CIRCLE	0-180 degrees 
UPPER_LEFT_QUARTER_CIRCLE	90-180 degrees 
UPPER_RIGHT_QUARTER_CIRCLE	0-90 degrees 
getStartAngle()	A method that returns the start angle as an integer.
getSweepAngle()	A method that returns the sweep angle as an integer.

3.6 JCLinearGauge

3.6.1 Constructors

JCLinearGauge has a no-argument constructor that supplies its own scale, then populates it with a needle and a set of tick marks. The constructor that takes a Boolean value creates a gauge with an associated linear scale if the Boolean parameter is true, or an empty gauge otherwise. In either case, indicators, needles, and ticks must be added separately.

3.6.2 Methods for JCLinearGauge

A linear gauge has methods for adding and removing labels, getting or setting its associated linear scale, and finding the needle closest to the point where a mouse click or drag occurred.

JCLinearGauge Method	Description
addLabel() removeLabel()	Adds or removes a label on the gauge. This is a general-purpose method, suitable for laying out any JComponent it is given.
getClosestNeedle()	Returns the closest needle to the clicked/dragged point.

`getScale()` `setScale()`	Gets or sets the scale used by the gauge. A scale contains the measurement parameters associated with a linear gauge. The `getScale()` method is inherited from `JCGauge`, but the `setScale()` method is overridden so that its argument must be a `JCLinearScale`. An optional second parameter to `setScale()` allows you to suppress the addition of the scale to the gauge area, but it is not expected that you will need to use this option.
`mouseClicked()`	Sends pick events to listeners and moves the closest needle with a `CLICK` interaction enabled to the value indicated by the mouse click.
`mouseDragged()`	Called during mouse drag events in the gauge to move the needle closest to the mouse.

3.6.3 Properties

A `JCLinearGauge` has the same properties as a `JCGauge`.

3.7 Headers, Footers, and Legends

Both `JCCircularGauge` and `JCLinearGauge` create a header and footer, which by default is a `JLabel`, but may be any `JComponent`. If you wish to choose some other header or footer than the default, methods `setHeader()` and `setFooter()` in `JCGauge` specify which `JComponent`s, typically `JLabel`s, to use. By default, the header, footer, and legend do not show. To display a header, use:

```
// gauge is a reference to either type of JCGauge
gauge.getHeader().setVisible(true);
```

A gauge's legend is an instance of `com.klg.jclass.util.JCLegend`, an abstract class that requires a subclass to provide a specific layout. By default, a gauge uses `JCGridLegend` to provide a default implementation of `JCLegend` and delegates `DefaultLegendPopulatorRenderer` to populate and render the legend. In the default case, range names are the items in the legend. If you wish to itemize ranges in a legend, simply show the gauge's default legend and `DefaultLegendPopulatorRenderer` does the rest.

```
// gauge is a reference to either type of JCGauge
gauge.getLegend().setVisible(true);
```

A legend's built-in behavior is to use range names for its items. Ranges have default names like *range0*, *range1*, and so on. Give a range a name of your own choosing as follows:

```
// range is a reference to either type of range
range.setRangeName("Danger zone");
```

Custom legends

If you need a legend that itemizes other things like needles or ticks then you can subclass `DefaultLegendPopulatorRenderer` and override `getLegendItems()` to provide your chosen item list with instances of `JCLegendItem`. The method in `DefaultLegendPopulatorRenderer` called `createLegendItem()` is used both for items and for the legend's title, the difference being that a null `Color` specification (the constructor's last parameter) indicates a title. Alternatively, you can use `setLegendTitle()` to set a title on an existing `JCLegend`. Note that `getLegendItems()` returns a list of items. See *GaugeInteractionExample.java*, which creates a legend that lists the needles used in the gauge.

3.7.1 JCLegend

Interfaces

There are two interfaces associated with `JCLegend`. `JCLegendPopulator` is an interface implemented by classes that wish to populate a legend with data, and `JCLegendRenderer` is an interface implemented by a class that wishes to help render the legend.

`DefaultLegendPopulatorRenderer` implements both interfaces and provides a built-in mechanism for itemizing range objects in a legend.

Methods in JCLegend

Method	Description
`getLegendPopulator()` `setLegendPopulator()`	Returns or sets the JCLegendPopulator instance used to populate this legend.
`getLegendRenderer()` `setLegendRenderer()`	Returns or sets the legend renderer class that is used to help draw the legend.
`getOrientation()` `setOrientation()`	Returns or sets the Orientation property that determines how the legend information is laid out. Possible values are `JCLegend.HORIZONTAL` or `JCLegend.VERTICAL`.

3.8 JCScale

`JCScale` is the interface that represents a graduated scale. A scale has a minimum value, a maximum value, and a direction. Lists of other objects like `JCTick` and `JCRange` objects are associated with the scale. These associated objects use the scale to get information that they need to render themselves.

Both circular and linear scale objects implement the `JCScale` interface, whose methods are as follows.

Interface JCScale Method	Description
`addRange()` `removeRange()`	Adds or removes a scale's range object. By supplying an optional index you can control where this range is placed in the list of range objects.
`addTick()` `removeTick()`	Adds or removes a scale's tick object. By supplying an optional index you can control where this group of ticks is placed in the list of tick objects.
`getDirection()`	Returns the `JCAbstractScale.Direction` for this scale, `FORWARD` or `BACKWARD`, giving the direction in which scale values increase.
`getExtent()`	Returns the `zoomFactor` for this scale. (This method is retained for backwards compatibility. Its use is deprecated.)
`getGauge()`	Returns the gauge associated with this scale.
`getMax()`	Returns the maximum value for this scale.
`getMin()`	Returns the minimum value for this scale.
`getRanges()`	Returns the Vector of range objects for this scale.
`getTicks()`	Returns the Vector of tick objects for this scale.
`getZoomFactor()`	Returns the `zoomFactor` for this scale.
`inBounds()`	Returns true if the value is within the scale's minimum and maximum.
`pick()`	Given a screen position in pixels, returns the closest scale value.
`setBorder()`	Sets a `Border` on the scale.
`setDirection()`	Sets the `JCAbstractScale.Direction` for this scale, `FORWARD` or `BACKWARD`, giving the direction in which scale values increase.
`setExtent()`	Sets the `double` that represents the `zoomFactor` for this scale. (This method is retained for backwards compatibility. Its use is deprecated.)
`setMax()`	The `double` that represents the maximum value for this scale.

Interface JCScale Method	Description
`setMin()`	The `double` that represents the minimum value for this scale.
`setZoomFactor()`	The `double` that represents the `zoomFactor` value for this scale.

Those wishing to use their own type of scale should implement this interface.

3.9 JCAbstractScale

`JCAbstractScale` implements `JCScale`. It is the superclass of both `JCCircularScale` and `JCLinearScale`, and it encapsulates the common properties of both these concrete classes. It does not add any methods beyond those in the interface it implements.

3.9.1 JCAbstractScale Properties

A `JCAbstractScale` holds the following information: the minimum and maximum values for the quantity being measured, a direction setting, and a zoom factor. Its `pick()` method is used for processing a scale value corresponding to the point at which a mouse click occurred. Here, it is declared abstract because circular and linear scales have differing implementations.

Min, max

These `JCScale` properties specify the beginning and ending values for the scale. Note that multi-turn functionality (multiple turns required to move from *min* to *max*) is not supported. Example:

```
scale.setMax(25.0); // Maximum value for the scale
scale.setMin(5.0);   // Minimum value for the scale
```

Specifying the direction of travel

By default, a circular scale increases in a counterclockwise direction. A linear scale increases from left to right if its orientation is horizontal or from bottom to top if its orientation is vertical. In these cases this is called the *forward* direction. To set this direction explicitly, call `setDirection(JCAbstractScale.Direction direction)` on the scale. The two field values are:

`JCAbstractScale.Direction.FORWARD`	Values increase clockwise, or from left to right.
`JCAbstractScale.Direction.BACKWARD`	Values increase counterclockwise, or from right to left.

The default value for `JCScale.Direction` is `FORWARD`.

See Deprecated way of specifying the direction of travel for a circular gauge for the pre-JClass 5 constants for setting a direction. These have been retained for backwards compatibility, but their use is deprecated.

Zoom factor

By default, the scale is drawn so that it fills the gauge area. If labels, ticks, or other components need to be placed outside the scale they may be cropped. To prevent this, use the `zoomFactor` property.

A circular scale has a `zoomFactor` property (called an *extent* before JClass 5) whose purpose is to avoid scaling problems when you want to have objects extend past the circumference of the scale. Objects exterior to the circumference may not resize properly because the border remains a fixed number of pixels no matter how greatly the window is resized, therefore there is a chance that the exterior objects may be clipped. By setting a scale's zoom factor to be less than 1.0 you can place objects outside the scale yet keep them on the interior of the gauge area, thus avoiding clipping problems if the scale is magnified.

For example, the code that places tick marks and their labels seemingly well outside a circular scale's boundary is:

```
// Sets the scale factor
scale.setZoomFactor(0.4);
// Places the tick marks and labels
tick.setInnerExtent(1.85);
tick.setOuterExtent(2.0);
tick.setLabelExtent(1.75);
```

These settings produce the scale shown in Figure 9 on the left.

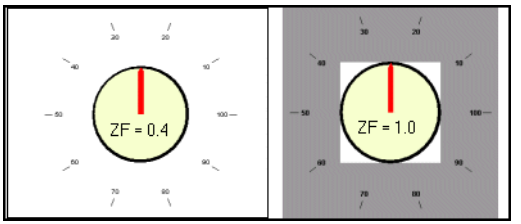


Figure 9 Placing ticks and their labels well outside the scale.

The left-hand figure is labeled "ZF = 0.4," which indicates the zoom factor setting. No border was used to pad the gauge's exterior. The effect has been accomplished just by setting the zoom factor.

The right-hand gauge in Figure 9 has a zoom factor of 1.0. If nothing else were done, the tick marks would lie outside the gauge area (the white rectangle), but by adding a border

(the gray area enclosing the gauge), the tick marks are visible within it. If you are allowing end users to resize the gauge, use a zoom factor rather than a border.

Borders are required for linear gauges. Setting a zoom factor on a linear gauge does not remove the necessity of setting a border to avoid the clipping of tick labels at the extremities of the scale. Figure 10 shows that setting a zoom factor on a linear scale compresses it in the direction perpendicular to the scale, but leaves the length of the scale unaffected.

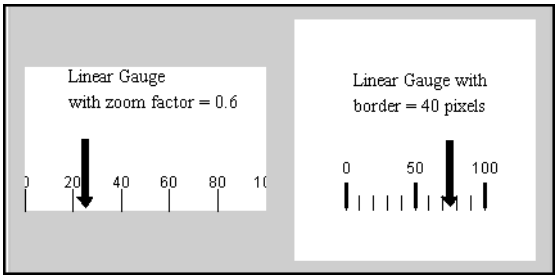


Figure 10 Comparison of a zoom factor (left) and a border (right) for a linear gauge.

On the other hand, setting a border around the scale allows room for the tick labels to be properly drawn. You can adjust the individual border thicknesses to suit your application. For more on setting a zoom factor for a linear gauge, see Setting a zoom factor, in Chapter 3.

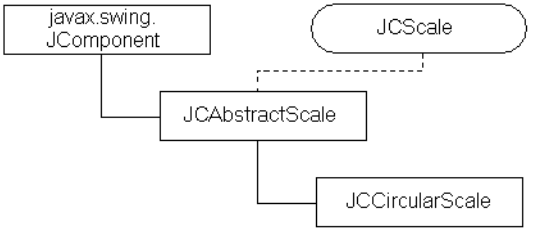
3.10 The Circular Scale Object

The `JCCircularScale` object and its superclass `JCAbstractScale` form the basis of all the quantifiable elements of the circular gauge[1]. Since a circular scale belongs to a circular gauge, the constructor for a `JCCircularScale` requires an argument of type `JCCircularGauge`. One circular scale object at a time may belong to a `JCCircularGauge` component. Also, the gauge keeps a reference to its circular scale in `JCGauge`.

1. JCCircularGauge uses a start angle and a stop angle to define the angular range, rather than employing Java's notion of a start angle and a sweep angle. For example, a scale that occupies a lower half circle has a start angle of 180° and a stop angle of 360°. Avoid the temptation to specify these two angles as 180° and 180°! See Section 3.10.3, Angles In a Circular Scale, for a discussion on how angles are measured in a `JCCircularScale`.

3.10.1 Circular Scales

A `JCCircularScale` inherits its direction, minimum and maximum values, lists of ranges and ticks, and zoom factor from `JCAbstractScale`. In addition it defines a start angle and stop angle, and a radius.



Other than being able to specify its foreground color, the circular scale contains no other visual information. Tick objects use the scale to determine what values to assign to tick labels, and needles have a value based on their location on a scale.

3.10.2 Notes on Circular Scale's Properties

The circular scale object's job is to hold the required numerical information for the construction of a circular gauge, but not any visual information except for the scale's foreground color. Displaying the information is the responsibility of indicators, needles, tick marks, ranges, and so on. Some of these are shown in Figure 11, along with the circular scale's property values that define the direction, set the numerical range values, determine the part of the circle that is to be displayed, and set the scale's start and stop angles.

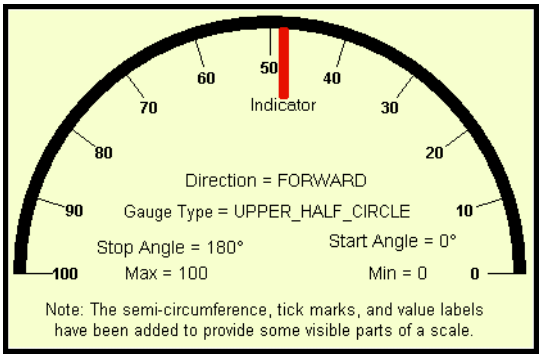


Figure 11 A circular scale showing representative values for its properties.

It is also possible to provide a foreground color such as the one shown in Figure 11.

3.10.3 Angles In a Circular Scale

The convention for angular measurement in a circular scale defines due east as the zero degree line. Angles increase in a counterclockwise direction, so that 90° is due north, 180° is due west, and 270° is due south. This is the direction specified by the constant

```
JCAbstractScale.Direction.FORWARD
```

Figure 12 illustrates the way that angular measurement is done in the circular scale, showing the X-axis, Y-axis, and location of the zero degree line.

The start angle is usually less than the stop angle but it is not a requirement. In the default situation, the direction is counterclockwise and the scale's *min* value is attached to the scale's *start* angle.

Once the start and stop angles of a circular scale have been chosen along with the maximum and minimum numerical values to be associated with them, one additional parameter, *direction*, sets whether the start angle corresponds to the minimum value or the maximum value. A maximum value may be attached to the scale's *start* angle by setting the *direction* parameter to *backward*.

For example, if the start and stop angles are chosen as 90° and 270°, and the values being measured begin at 100 and end at 200, setting the direction to `JCAbstractScale.BACKWARD` causes the value 200 to be located at 90°. The values decrease around the scale in a clockwise direction, ending at 270° where a value of 100 is located. Thus, while the coordinate system for angular measurement never changes, the direction in which the scale's value parameter increases can be reversed.

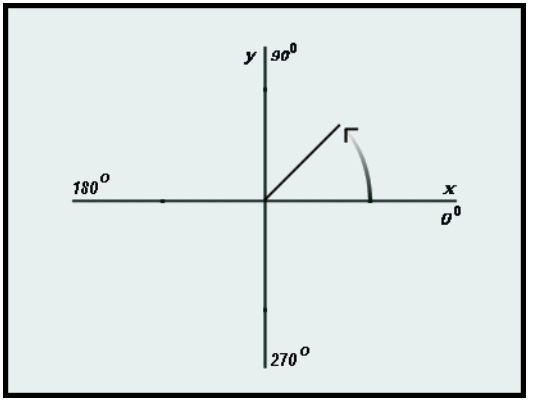


Figure 12 Diagram of a circular scale's reference plane with direction set to 'forward'.

Deprecated way of specifying the direction of travel for a circular gauge

Previously, before the release of JClass 5, the direction of travel for circular scales was defined as:

`JCAbstractScale.Direction.CLOCKWISE`	Values increase clockwise.
`JCAbstractScale.Direction.COUNTERCLOCKWISE`	Values increase counterclockwise.

The default value for `JCScale.Direction` is `COUNTERCLOCKWISE`.

These constants have been retained. However, if you use the new constants defined in Specifying the direction of travel you will be able to switch between circular and linear gauge types without causing confusion over inappropriate direction names.

Start and stop angles

The start and stop angles specify where measurements begin and end on the circular scale. These two angles, defined in `JCCircularScale`, specify the compass positions at which the *min* and *max* values are located. A circular scale positions zero degrees at due east. Angles increase in the counterclockwise direction no matter what the value of `JCAbstractScale.Direction` is. A start angle may be greater than a stop angle, which would be the case if a scale's *min* value begins at the twelve o'clock position and its *max* value is at the three o'clock position. In this case, the start angle is 90° and the stop angle is zero degrees.

These angles may be set in `JCCircularScale`'s constructor. Alternatively, start and stop angles may be set using `setStartAngle()` and `setStopAngle()`. For instance:

```
JCCircularScale scale = new JCCircularScale();
scale.setStartAngle(15);
```

sets the scale's start angle at 15°.

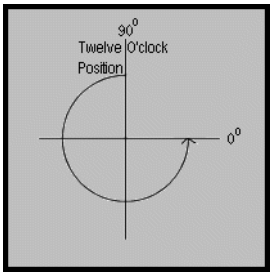


Figure 13 A case where the start angle is greater than the stop angle.

Radius

`JCCircularScale` has a `getRadius()` method that returns the size of the circular scale. Initially, the size of the gauge may be set using `JComponent`'s `setPreferredSize()`

method. Also, the size of the scale relative to its container is controlled by the `zoomFactor` property.

Assigning a color

A scale's color may be assigned in a `JCCircularScale`'s constructor, or you may use `JComponent`'s `setForeground()` method. By default, only the portion of the scale between its *start* and *stop* angles is colored. Any remaining portion retains the color of the gauge. If you wish to assign the color to the full scale, set the scale's `paintCompleteBackground` property to `true`:

```
(JCCircularScale)
 gauge.getScale().setPaintCompleteBackground(true);
```

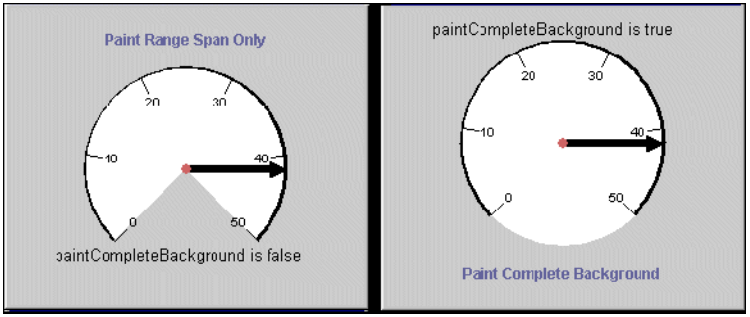


Figure 14 Using paintCompleteBackground to determine how much of the scale is colored.

If you want to color only the portion of the scale between the start and stop angles, use `setPaintCompleteBackground(false)`.

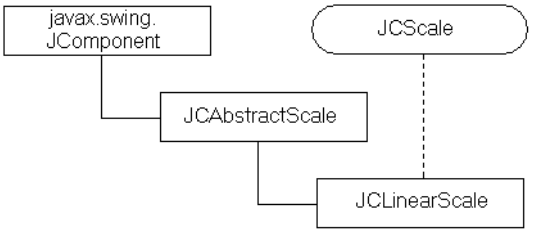
Labels in a circular gauge

Labels in a circular gauge work similarly to the way they do in a linear gauge except that they use `CircularConstraint`s. See Labels in a linear gauge for information.

3.11 The Linear Scale Object

`JCLinearScale` provides a graduated scale drawn in a linear fashion. Figure 15 shows a simple linear scale. In addition to the scale, the gauge in the figure has two triangularly-shaped indicators, a needle, two ranges, a set of labeled tick marks, and a

collection of labels. There is a border around the scale. It has the same color as the scale's background, but its presence is important to scale's layout. See the section on Borders.



Setting a linear scale's direction and orientation

By default, scale values increase from left to right, as shown in Figure 15. The direction may be controlled by setting either one of:

```
scale.setDirection(JCAbstractScale.Direction.BACKWARD);
scale.setDirection(JCAbstractScale.Direction.FORWARD);
```

A linear scale is oriented either horizontally or vertically. The default orientation is horizontal, and the setting is controlled by `JCLinearScale.Orientation`, whose values are:

```
JCLinearScale.Orientation.HORIZONTAL
JCLinearScale.Orientation.VERTICAL
```

A typical call is:

```
JCLinearScale scale = new JCLinearScale();
scale.setOrientation(JCLinearScale.Orientation.VERTICAL);
```

Borders

You will likely want to place a border around each of your linear gauges. If you don't, the tick label numbers may appear to be clipped by the sides of the container. The linear gauge in Figure 15 has a 20-pixel-wide border that was constructed as follows:

```
Border border =
    BorderFactory.createLineBorder(new Color(247, 255, 206), 20);
    scale.setBorder(border);
```

In this case, the scale's border has the same color as the scale's background color, so the viewer is not aware of its presence. The visible dark border is part of the gauge, not the scale. Note that it's possible to write into the border area. Parts of the *Direction*, *Orientation*, *Min*, *Max*, and *Ranges* labels are actually in the scale's border area.

You can use `createEmptyBorder()` to assign different widths to all four sides.

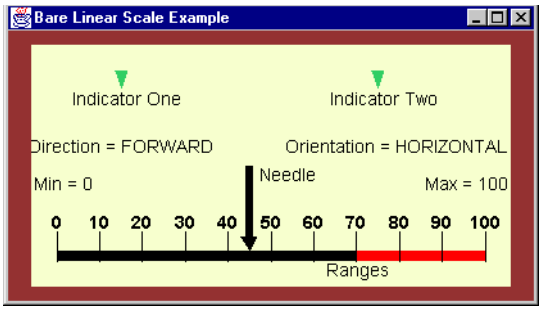


Figure 15 All elements shown, including the dark border, are parts of a linear gauge.

Labels in a linear gauge

Place labels using a `LinearConstraint`. Its constructor takes a reference to the parent gauge, an `extent`, and an `int` representing a `position` along the scale. The `extent` parameter is the distance from the top-left corner of the scale[1]. This distance is vertical for horizontal scales and horizontal for vertical scales, and is specified as a ratio of the required distance to the height of the scale when its orientation is horizontal, or to the width of the scale when its orientation is vertical. The position parameter measures the distance in pixels from the left-hand side when the gauge's orientation is horizontal, or from the top of the scale when its orientation is vertical.

```
JLabel l1     = new JLabel("<html><font color=black>Indicator One");
l1.setToolTipText("Indicator One");
gauge.getGaugeArea().add(l1, new LinearConstraint(gauge, 0.15, 75));
```

Setting a zoom factor

Your design may require that indicators, labels, and the like should appear on the outside of the scale. You may be able to use a border to accomplish this task; however, a linear scale provides a more flexible mechanism called a `zoomFactor`. In a linear scale, a zoom factor less than one compresses the height of the scale (if the orientation is horizontal) while leaving the width (the scale direction) unchanged. In general, the zoom factor compresses the dimension at 90° to the scale no matter what the orientation is. Thus, setting

```
scale.setZoomFactor(0.6);
```

compresses the scale to 60% of its size and leaves room totalling 40% of the scale height evenly above and below the scale. Needles or ticks with inner extents less than one or with outer extents greater than one will display nicely even when the component is resized.

1. The extent is measured from the scale boundary, not from any border that may enclose the scale.

For even more control, two Boolean properties, `useZoomFactorForMin` and `useZoomFactorForMax` are available. By default, both of these are `true`, but if one of these is set to `false`, the zoom factor will not be applied to the appropriate *min* and *max* extent portions of the scale. If both `useZoomFactorForMin` and `useZoomFactorForMax` are set to false, the zoom factor is ignored.

Pick: getting values from the scale by user interaction
The `pick()` method returns the scale value corresponding to an end user's mouse click.

3.11.1 Methods in JCLinearScale

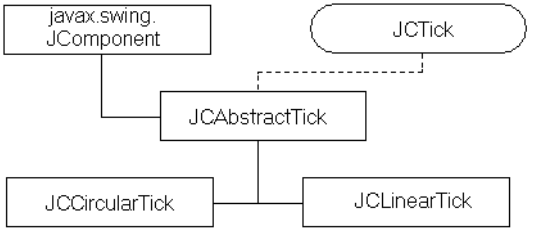
The table lists the commonly used methods for a `JCLinearScale`:

JCLinearScale Method	Description
`add()`	Overrides add to pass in a linear constraint. The first parameter is the component to be added and the optional second component is an index denoting the order among the other components that have been added to the parent.
`getLinearGauge()`	Returns the linear gauge associated with this scale.
`getOrientation()` `setOrientation()`	Returns or sets the `JCLinearScale.Orientation` for this linear scale. Values are: `JCLinearScale.Orientation.HORIZONTAL` or `JCLinearScale.Orientation.VERTICAL`, specifying the orientation in which scale values are rendered.
`getScaleSize()`	Calculates the rectangle that defines the position and dimensions of the linear scale.
`getUseZoomFactorForMax` `setUseZoomFactorForMax`	Determines whether to apply the zoom factor to the max extents portion of the scale. Default: `true`.
`getUseZoomFactorForMin` `setUseZoomFactorForMin`	Determines whether to apply the zoom factor to the min extents portion of the scale. Default: `true`.

3.12 Tick Objects

Tick objects provide annotation marks on the circular or linear scale. They provide the usual graduations that are often found on measuring devices. It is possible to have many different tick objects associated with a scale; for instance, two tick objects can provide major and minor graduations. Typically, major tick marks have associated labels

displaying numerical values, or you may supply your own definition for tick labels using the `JCLabelGenerator` interface.



Both tick marks and tick labels may be turned on and off independently for each tick object. A property called `drawTicks` controls whether tick labels are drawn, and another called `drawLabels` controls whether the associated labels are drawn. Examples are given later in this section.

You can set the start value, stop value, the tick increment value, and the inner and outer extents for the tick marks. These *extent* parameters control the length of the tick line in the radial direction in the case of a circular gauge, or the height/width in the case of a linear gauge. They are specified as decimal fraction of the scale's radius in the circular case, or the scale's height/width in the linear case. If you wish, you may specify a color and a width for the tick object. If you do not set a particular property, the object chooses a default value.

You can set the precision for values displayed on tick mark labels, but be aware that reducing the precision may introduce rounding issues that may make the scale markings confusing. See A note on precision for an explanation.

The following diagrams illustrate both circular and linear ticks.

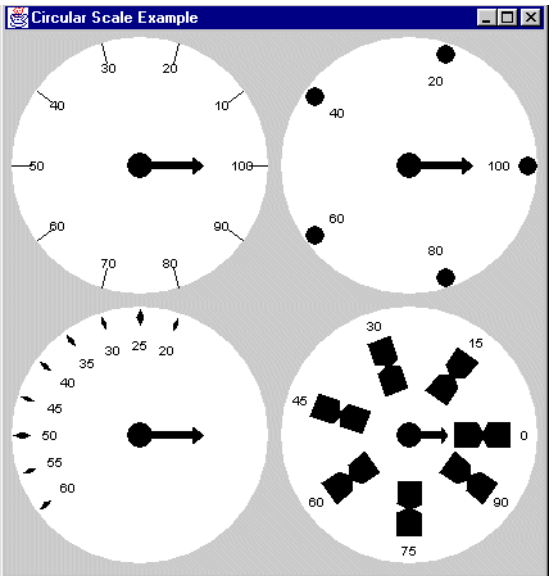


Figure 16 Some circular gauge tick objects and their associated labels.

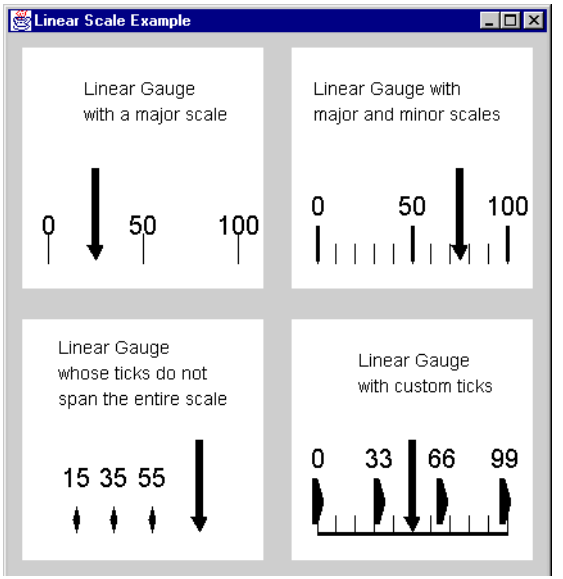


Figure 17 Linear gauge tick objects and their associated labels.

You specify where the tick marks are to begin and where they are to end. This allows you to place tick marks over a portion of the scale as shown in the bottom left scale in Figure 16 and Figure 17. Note the different tick styles in the figures. There are custom tick style set for the scales in the lower right of Figure 16 and Figure 17. See Defining your own tick style for details.

Since you can control the visibility of tick marks and tick labels independently, you may choose to have only the labels showing. This may be useful if you wish to have a single style of tick mark but label every second one.

3.12.1 Notes on the Tick Object

Associating a tick object with a scale

Tick objects need to know the scale for which they are to provide graduations. One of the parameters in the tick object's constructor is the scale associated with the tick object. Thus, a tick object cannot be instantiated without specifying its associated scale. Once you have a tick object, you still need to add it:

```
JCLinearTick tickMark;
// Instantiate the tick...
gauge.getScale().addTick(tickMark);
```

Setting the tick type

There are six built-in types: circle, diamond, line, rectangle, reverse triangle, and triangle. The types are held in the `JCTickStyle` class. Example:

```
JCTick tick = new JCCircularTick(...);
tick.setTickStyle(JCTickStyle.TRIANGLE);
```

Defining your own tick style

You can define your own tick style by defining its shape using two arrays, one for the X-coordinates and one for the Y-coordinates. Pass these arrays and an `int` specifying the number of points to `JCTickStyle`'s constructor. Alternatively you could subclass `JCTickStyle` and define your new styles as constants. In either case, define a shape as an array of X- and Y-coordinate points as you would for any `Rectangle`:

```
import com.klg.jclass.swing.gauge.JCTickStyle;
public class MyTickStyle extends JCTickStyle {
    public static final JCTickStyle NOTCHED_RECTANGLE = new JCTickStyle(
        new int[] {-10, -2, 0,  2,  10,  10,  2,  0, -2, -10},
        new int[] { 3,   3, 1,  3,   3,  -3, -3, -1, -2,  -2},
        10);
}
```

Use your newly defined tick style by calling

```
tick.setTickStyle(MyTickStyle.NOTCHED_RECTANGLE);
```

Alternatively, you can use the `JCTickStyle` constructor that allows you to define your own (*x*, *y*) coordinate pairs. This example can be seen in the lower right of Figure 17:

```
int xpoints[] = {-100, 0,   100,   100,      0, -100};
int ypoints[] = {   0, 100, 100,  -100,   -100,    0};
int numpoints = 6;
tick.setTickStyle(new JCTickStyle(ypoints, xpoints, numpoints));
```

Note: The layout algorithm assumes that the center of the tick mark's bounding rectangle is at (0, 0).

Setting the tick object's placement

Tick marks are normally required at constant increments along a scale. The tick object accomplishes this objective by using the associated scale's *min* and *max* values, and the *precision* value, to determine appropriate values for the tick object's *start*, *stop*, and *increment* values. Alternatively, you can control the spacing by setting `automatic` to `False`. You still control where the tick marks are to begin with `startValue` and where they are to end with `stopValue`, but you set how many tick marks there are with `incrementValue`.

Setting a tick object's dimensions

The placement of a tick mark is controlled with `innerExtent` and `outerExtent`. The values for both of these properties are numbers representing a proportion relative to the size of the associated circular or linear scale. For instance, if the value set on `innerExtent` is 1.0, tick marks begin right at the circumference of a circular scale, or at the edge[1] of a linear scale. For circular scales, tick marks are drawn radially outward to the value set in `outerExtent`, which is also given as a ratio based on the radius of the associated circular scale. For linear scales, the outer extent of 1.1 causes the tick mark to terminate 10% of the scale's vertical dimension below the bottom edge of the scale when the orientation is horizontal, or 10% of the scale's horizontal dimension to the right of the right edge of the scale when the orientation is vertical.

Note: If your tick marks extend outside the scale, that is, at extents less than 0 or greater than 1.0, and you allow the scale to be resized, you may have to increase the dimensions of the scale's borders to ensure that there is enough space to hold the tick marks, otherwise their outer extents may be clipped. Alternatively, set the scale's *zoomFactor* property to a value less than 1.0 and the tick's outer extent to 1.0 to give the appearance of tick marks lying outside the scale. In fact, they are within the scale's boundary, but they appear to lie outside the colored portion of the scale. See *GaugeOutsideExample.java* in the examples directory for an illustration of this technique.

If the inner extent is equal to the outer extent, no ticks are drawn, but the preferred way of hiding tick marks is to set `drawTicks` to `false`.

1. The lower edge of a linear scale whose orientation is horizontal, or the right edge of a linear scale whose orientation is vertical. An inner extent of 1.0 is definitely a case for surrounding the scale with a border.

The width in pixels of the tick marks is specified in the `tickWidth` property. This property must be set for any tick style other than `JCTickStyle.LINE`.

Labeling tick increments

Labeling may be on or off, depending on the value of `drawLabels`. In automatic mode, one of the ways the object manages the span between labeled tick marks is by using the `precision` property's value. If you are controlling the placement of labeled tick marks (`tick.setAutomatic(false)`), make sure you have set `precision` properly, as explained in A note on precision.

Note: If you do place labels outside the scale and you allow the scale to be resized, you may have to increase the dimensions of the scale's borders to ensure that there is enough space to hold the labels. Rather than using borders, the recommended method is to adjust the scale's `zoomFactor` property. See the previous note in Setting a tick object's dimensions.

Custom tick labels

Tick labels are drawn with the help of the `JCLabelGenerator` interface. It contains a single method, `makeLabel()`, which takes three parameters: a `JCTick`, a scale value, and a `GaugeConstraint`[1]. Use the method in `JCAbstractTick` called `setLabelGenerator()` to tell the gauge to use your custom labeling mechanism.

The following code snippet uses an anonymous inner class to add an implementation of `JCLabelGenerator` to a tick object. The `makeLabel()` method is passed a reference to the tick object in question, the scale value for the tick mark in question, and a reference to the tick's `RadialConstraint`. Only the `value` parameter is used in this example. Since the tick marks are supposed to display temperature values, the custom labels are coded to produce temperature values along with their units of measurement, for example, 20° C.

```
// create a label generator to mark
// the temperature values with their units
    tick.setLabelGenerator(new JCLabelGenerator() {
        public JComponent makeLabel(JCTick tick, double value,
                                    RadialConstraint constraint) {
            String s = (value != 0) ? String.valueOf((int)value)
                                    : "zero";
         JLabel label = new JLabel(s + "\u00B0 C");
         label.setToolTipText(s + "\u00B0 C");
            return label;
        }
});
```

The code adds "° C" to each value except 0°, where it supplies the word "zero" instead.

For an example of a custom label in a user-defined class, see *GaugeSwitchExample.java*.

1. The two subclasses of `GaugeConstraint` are `LinearConstraint` and `RadialConstraint`.

Another reason for using custom tick labels is to provide an offset between a label and its associated tick mark. Normally a label lines up with its tick mark so that a line joining the center of the scale and a tick mark also goes through the center of the tick mark's label. You can use `JCLabelGenerator` to offset the label, since a `GaugeConstraint` is passed as one of the parameters of `makeLabel()`.

A note on precision

If `precisionUseDefault()` is `true`, a default value for precision is determined. This value should be sufficient for most situations. However, you can specify your own precision using `setPrecision()`. This has the side effect of setting *precisionUseDefault* to `false`. The precision setting affects the width of the label and therefore the number of labels that may be used without overlapping. The effect is quite noticeable if `automatic` is `true`. The number of labels changes as the scale is zoomed, becoming fewer during contraction so that the label on one number does not overlap adjacent labels, or becoming more numerous during expansion so that labeled marks do not become too widely separated. Wider labels will be fewer in number compared to the same scale with shorter tick labels. All this happens because the tick object automatically calculates the number of tick labels. Note that `drawLabels` must be `true` or the labels won't show.

The value of the `precision` property controls how tick labels are interpreted. There are three cases to consider:

- Positive values of `precision` indicate the number of places after the decimal place to include. For example, setting the `precision` property to 3 causes the labels to be multiples of 0.001.
- Negative values indicate the positive number of zeros to use before the decimal place. For example, if the value of the `precision` property is –3, numbering will be in multiples of 1000.
- A value of zero causes the labels to be integers.

Failure to set the precision property may be the source of a misleading scale. As an example, setting the start value of a tick object at -25, the stop value at 160, and allowing the default precision sets the precision to -1. In this case the first tick mark is where the scale starts, that is, at -20, which is probably not what was desired. Setting the precision to 0 rectifies the problem because this specifies that digits in the units column must not be rounded.

3.12.2 Methods

The tick object has the following get/set methods:

JCAbstractTick Method	Description
`getAutomatic()` `setAutomatic()`	Controls whether the tick object is functioning in automatic mode or in manual mode. If it is in automatic mode, the associated scale's min and max values and the value of precision are used to determine "pleasing" tick start, stop, and increment values. Tick spacing is calculated by the gauge and any settings to these properties will be ignored, that is, the values reset to the automatic values. If the tick object is in manual mode, whatever values you set in `startValue`, `stopValue`, and `incrementValue` are honored provided they are legal.
`getDrawLabels()` `setDrawLabels()`	Returns the Boolean controlling the drawing of labels. If `true`, use the value of `precision` to place a numeric label on each tick value.
`getDrawTicks()` `setDrawTicks()`	Returns the Boolean controlling the drawing of tick marks. If `true`, use the value of `precision` or `incrementValue` to determine how many tick marks to draw.
`getFont()`	Returns the `Font` used for tick mark labels (`java.awt.Component`).
`getFontColor()` `setFontColor()`	Returns or sets the `Color` used for tick mark labels.
`getIncrementValue()` `setIncrementValue()`	The tick increment value. In automatic mode, this value is read-only. In non automatic mode, this value can be changed.
`getInnerExtent()` `setInnerExtent()`	The inner (towards the center of a circular scale) extent of each tick drawn. This value is related to the radius of an associated circular or linear scale. For example, a value of 0.9 means each tick's inner extent is 0.9 times the radius value of the scale away from the center, or a 0.9 times the scale's height/width away from the top left of the scale.

JCAbstractTick Method	Description
`getLabelExtent()` `setLabelExtent()`	The location of the center of each label. This value is related to the radius of the associated circular or the height/width of the associated linear scale; a value of 1.1 means each label's extent is 1.1 times the radius value of a circular scale away from the origin, or 1.1 times the height of a linear scale away from its top edge, or 1.1 times the width away from the left edge, depending on its direction. **Note:** extending the label outside the scale may require adjustment of the borders.
`getLabelGenerator()` `setLabelGenerator()`	Returns the label generator associated with this tick. Consult the API for interface `JCLabelGenerator` and its `makeLabel()` method for information on defining your own labels. A minimal example is given in the section on Labeling tick increments in this chapter.
`getOuterExtent()` `setOuterExtent()`	The outer extent of each tick drawn. Outer extent values increase towards the component's boundary. This value is related to the radius of the associated circular scale, or the width/height of the associated linear scale. For example, a value of 1.1 means each tick's outer extent is 1.1 times the radius value of the scale in a direction away from the center, or 1.1 times the height of a linear scale away from its top edge or 1.1 times the width away from its left edge, depending on its orientation. **Note:** extending the tick mark outside the scale may require adjustment of the borders.
`getPrecision()` `setPrecision()`	Affects the format of the tick label. This property defines the number of digits of precision after the decimal point to be used for labels. If its value is zero or less, the labels will all be integers. Positive values denote the number of places after the decimal point (for example, 3 means multiples of 0.001); negative values indicate the number of zeros to be used before the decimal place (for example, -3 means numbering will be in multiples of 1000). A side effect of setting this property is to set the corresponding `precisionUseDefault` property to `false`.

JCAbstractTick Method	Description
`getPrecision UseDefault()` `setPrecision UseDefault()`	If `true`, use the gauge-determined precision for tick labels, otherwise use `startValue`, `stopValue` and `incrementValue` to place ticks along the scale. If you set this property to false you should set a value for precision as well (unless a value has been calculated previously).
`getScale()`	Returns the scale associated with this tick. Tick marks must be associated with a scale (this is enforced by the Tick object's constructor).
`getStartValue()` `setStartValue()`	In non automatic mode use this value as the start value. The value must be between the associated scale's min and max values, including end points.
`getStopValue()` `setStopValue()`	In non-automatic mode use this value as the stop value. The value must be between the associated scale's min and max values, including end points.
`getTickColor()` `setTickColor()`	The `Color` of the tick mark.
`getTickStyle()` `setTickStyle()`	Predefined tick styles are `JCTickStyle.CIRCLE`, `JCTickStyle.DIAMOND`, `JCTickStyle.LINE`, `JCTickStyle.RECTANGLE`, `JCTickStyle.REVERSE_TRIANGLE`, and `JCTickStyle.TRIANGLE`. Extend `JCTickStyle` to define your own shape for tick marks, or define one by passing X- and Y-coordinate points to `JCTickStyle`'s constructor.
`getTickWidth()` `setTickWidth()`	The width in pixels of the widest part of the object. Ticks whose style is circle or line ignore this property.
`isReversed()` `setReversed()`	The reversed property is used to choose a mirror image of a tick mark. If the default mark is <, the reversed tick mark is >.

3.12.3 Sample Code

The following example illustrates using the tick object's constructor. If you create a gauge using one of the convenience constructors you already have a tick object associated with the scale that you can access using code; for example:

```
JCCircularTick tick = (JCCircularTick)
                        gauge.getScale().getTicks().elementAt(0);
```

Then simply set the properties you wish to change from their default values. Nonetheless, it is often useful to create tick objects from scratch and set properties in the constructor.

This way you inspect each property and determine its appropriateness for your application. Here's an example:

```
// create a Tick object and set its properties
JCTick tick = new JCCircularTick(
    scale,          // the associated circular scale
    false,          // automatic tick generation
    0,              // start
    300,            // stop
    25,             // increment
    false,          // precisionUseDefault
    0,              // precision
    2,              // width
    true,           // draw labels
    true,           // draw ticks
    0.75,           // label extent
    0.85,           // inner extent
    1.0,            // outer extent
    Color.white,
    JCTickStyle.LINE,
    new Font("Helvetica", Font.BOLD, 18),
    Color.white);
```

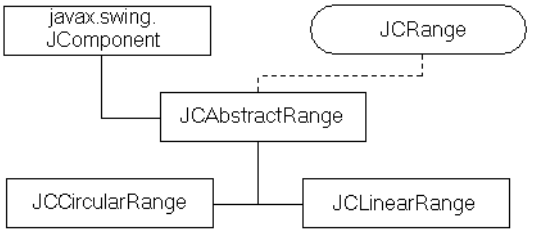
3.13 The Range Object

On circular scales, ranges are rings or partial rings used to emphasize certain sections of the scale by coloring them. By using range objects, an implementor can draw arcs (which may be full circles) and provide differently colored subdivisions on the scale.

On linear scales, ranges are rectangular bars running along the direction of the scale.

Ranges may be either colored bands or images.

If the range is narrow enough it has the appearance of a tick mark. This effect may be of use if you wish to provide special tick marks, such as those marking the dynamically changing maximum and minimum values on a temperature scale. See the demo *TemperatureFluctuationExample.java*.



Ranges are associated with a scale. There may be multiple ranges for the same scale.

You can control the inner and outer radii (and thus the thickness) of the range. As well, you set start-stop values (and thus the breadth), and the color. Figure 18 shows two range objects on a circular scale. The thinner one spans the entire scale and appears as the circumference of the scale. The thicker one spans the region between tick marks 20 and 80. The diagram shows how a start value, stop value, inner extent, and outer extent determine the shape of a range.

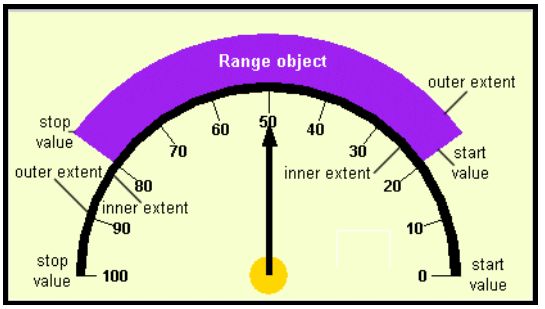


Figure 18 Two range objects for a Circular Scale object.

Figure 19 shows a Circular Gauge component in which one range object covers another, partially obscuring it. The drawing order in Figure 19 is first the semicircular range, followed in succession by the 20-80 range, the tick marks, the needle, and finally the center. If the objective is to show the semicircular range, draw the 20-80 range first. This topic is discussed in more detail in the next section.

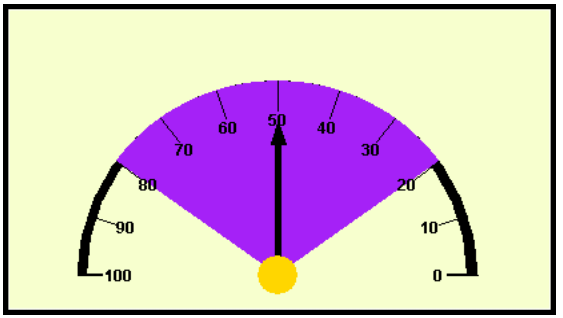


Figure 19 A range object that is the same size as its associated scale.

3.13.1 Notes on the Range Object's Properties

Coloring a range

A range object's color is set by the first parameter in its constructor. Use the `setForeground()` method on a `JCCircularRange` or `JCLinearRange` object to change the color of a range once it has been instantiated.

Associating a range with a circular or linear scale

A range object has to be associated with a parent scale by providing a reference to it in the constructor, or by using the `setScale()` method once a range has been instantiated.

A range spans a region of its associated scale, but to be visible it must have a thickness as well as a span. Properties `innerExtent` and `outerExtent` control the inner and outer limits of its thickness. Values for these properties are given as ratios based on the size of its associated scale. For example, in a circular gauge an inner extent of 0.75 and an outer extent of 1.25 mean that the thickness is half the radius of the associated circular scale and is placed symmetrically over the circumference.

Extending past the scale

If you wish to use an offset to shift a range beyond the end of the tick marks, create a circular scale spanning all the values you need to display but supply tick marks for one portion of the scale and a range for another portion.

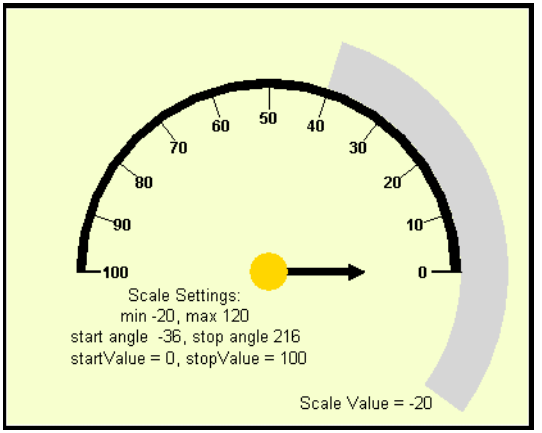


Figure 20 An offset is applied to the outer range of Figure 18.

As shown in Figure 20, an offset appears to begin before the start of the scale. The scale's settings are *max* = 120, *min* = -20, *startAngle* = -36°, *stopAngle* = 216°. These were calculated to be consistent with the offset given to the range, and to maintain a semicircular appearance for the labeled part of the scale.

3.13.2 Constructors for JCCircularRange and JCLinearRange

The constructors are:

```
JCCircularRange(java.awt.Color        foreground,
                JCCircularScale       scale,
                double                innerExtent,
                double                outerExtent,
                double                startValue,
```

```
                      double                stopValue)

public JCLinearRange(java.awt.Color         foreground,
                     JCLinearScale          scale,
                     double                 innerExtent,
                     double                 outerExtent,
                     double                 startValue,
                     double                 stopValue)
```

All range properties are set in the constructor, along with associating a range with a circular scale. Range properties are listed next.

3.13.3 Methods and Properties for JCRange and JCAbstractRange

JCAbstractRange Method	Description
`getInnerExtent()` `setInnerExtent()`	Returns or sets the ratio of the radius' length at which to start drawing this range from the center outwards for circular scales, or the distance away from the inner edge for linear scales. For example, a value of 0.5 means the inner extent should begin one-half the distance between the scale's origin and the scale's circumference or outer edge.
`getOuterExtent()` `setOuterExtent()`	Returns or sets the value at which the outer boundary of the range is drawn. The value is expressed as a ratio of the radius' length. For example, a value of 1.5 means the outer extent lies 1.5 times the scale's radius away from the center, or 1.5 times the linear scale's height away from the inner edge.
`getScale()`	Returns the scale associated with this range.
`getScaleImage()` `setScaleImage()`	Returns or sets the flag that controls whether to scale the image within the dimensions of the range.
`getStartValue()` `setStartValue()`	Returns or sets the scale value (not the angle) where the range is to begin. This value should be between the associated scale's min and max values.
`getStopValue()` `setStopValue()`	Returns or sets the scale value (not the angle) where the range ends. This value should be between the associated scale's min and max values.

Additional Methods	Description
`setForeground()`	Method inherited from class `javax.swing.JComponent`. Used to dynamically color the range after it has been instantiated.

Additional Methods	Description
`setVisible()`	Method inherited from class `javax.swing.JComponent`. Used to dynamically show or hide a range.

3.13.4 Sample Code

Set the property values for the range object in its constructor using the sample below as a guide.

```
// create a range that marks the circumference of the scale
circumference = new JCCircularRange(
            Color.black,          // range color
            gauge.getScale(),     // one way of referencing a scale
            0.95,                 // range inner extent
            1.00,                 // outer extent is at the radius
            0,                    // start value for the range
            100);                  // stop value for the range
```

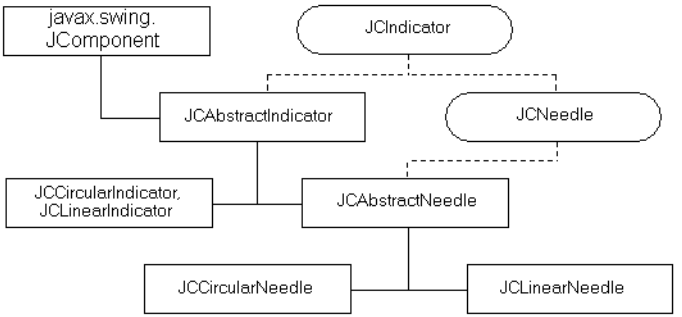
3.14 The Indicator and Needle Objects

Indicators are a general mechanism for placing static[1] markers on a scale. Needles are meant for user interaction, so the `JCAbstractNeedle` subclass of `JCAbstractIndicator` has built-in capabilities for control by end users. If you permit it, an end user can reposition a needle by clicking, dragging, or both.

Indicator and needle objects are the visible pointers on a scale object in a `JCGauge` component. Any number of indicators may be displayed on the same scale. Each

1. As far as direct end user interaction is concerned. Indicators may be repositioned under program control if desired.

indicator has its own value and this value determines where on the scale it is positioned. The gauge manages its list of indicators by keeping them in a `Vector`.



You can control the indicator's color, length, width, and shape. The shape scales itself by using its *length* and *width* parameters.

The needle is drawn using its *shape* attribute. Basic constants exist for drawing triangles, pointers, and arrows, or you may provide your own shape.

The position of an indicator, and therefore its associated value on a scale may be controlled programmatically, and the needle subclass may be controlled via an end user action. A needle can be positioned by either clicking or dragging on the scale so long as a needle interaction is enabled. You can add a `ChangeListener` to a needle to respond to mouse actions. Additionally, the gauge has a `JCGaugePickListener` interface that permits retrieval of the scale value corresponding to the spot where a mouse click occurs. You can enable click and drag interactions with a needle via the inner class called `JCNeedle.InteractionType`, which contains constants that specify the possible types of interactions between mouse actions and needles.

These are:

Interaction Type	Description
`CLICK`	The needle snaps to a mouse click.
`CLICK_DRAG`	The needle snaps to the mouse click, or follows a mouse drag.
`DRAG`	The needle snaps to the position of the mouse when the drag operation begins, then follows it until the mouse button is released.
`NONE`	Default case: neither clicking nor dragging affects the needle.

When a needle's value changes, a value changed callback is called to allow the program to disable or limit the change. Use `sendEvents()` to control whether an event is generated as a result of an action taken in the callback.

3.14.1 Notes on the Indicator's Properties

Indicator shapes

An indicator has seven possible built-shapes: arrow, tailed arrow, circle, pointer, rectangle, tailed pointer, or triangle, as shown in Figure 21. These shapes are controlled by the constants in `JCIndicatorStyle`. They are `ARROW`, `CIRCLE`, `POINTER`, `RECTANGLE`, `TAILED_ARROW`, `TAILED_POINTER`, and `TRIANGLE`.

Setting an indicator's length

The indicator's length is based on the associated scale and is set as a decimal fraction of the scale's dimensions using its `innerExtent` and `outerExtent` properties. In the circular case, an indicator begins at the center of the circular scale and extends outwards. For example, if its `outerExtent` property is set to 1.0, the indicator's tip lies on the circumference of the associated circular scale. In the linear case, an indicator's extents are measured from the top of the gauge area when the orientation is horizontal, or from the left-hand edge when the scale is vertical.

Setting a needle's length

As a subclass of `JCAbstractIndicator`, a needle has inner and outer extents, and because it is a subclass of `JCAbstractNeedle`, it has a length property as well. Setting the needle's length is equivalent to setting its outer extent. If you want to have the needle begin away from the center of a circular gauge, set its inner extent to some positive value. The value is expressed as a ratio based on the radius. Likewise, a linear needle may be offset from the top of a linear horizontal scale by setting its inner extent. In this case, the inner extent may be positive or negative, but you will have to set a border on the scale to prevent the needle from being clipped. Example:

```
((JCLinearNeedle)
gauge.getNeedles().firstElement()).setInnerExtent(0.3);
```

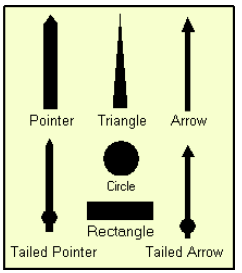


Figure 21 The seven built-in indicator shapes.

Setting an indicator's width

An indicator's width is set using the `indicatorWidth` property, or the `needleWidth` property in the case of a needle.

Defining a custom indicator style

It is possible to provide your own indicator style if you require a custom shape. The method is the same as in Defining your own tick style. The simplest way is to use the `JCIndicatorStyle` constructor that allows you to define a new shape. If you wish to keep your new indicator styles for general reuse as class constants, extend `JCIndicatorStyle` and define a shape as arrays of coordinate points using the same format as you would for `java.awt.Rectangle`. Here is an example:

```
import com.klg.jclass.swing.gauge.JCIndicatorStyle;

public class MyIndicatorStyle extends JCIndicatorStyle {
/**
 * A needle in the form of a diamond using these array values
 */

    public static final JCIndicatorStyle DIAMOND
            = new JCIndicatorStyle(
        new int[] { -100,   0, 100,    0},
        new int[] {    0, 100,   0, -100},
        4);
```

The `JCIndicatorStyle` constructor that allows you to define a new shape has been used here to define the class constant.

Often an indicator style is defined so that the shape starts at the center of a circular scale. A shape defined this way also starts at the inner edge of a linear scale. Since the `DIAMOND` defined above starts at -100, it does not start at the scale's center or edge when it is drawn. Instead, this `DIAMOND` straddles the center. You'll need to test your design to see that it scales properly if you begin your needle away from the center and you intend to allow scaling.

Indicators may be sized however you wish, but if the inner or outer extent specification causes the indicator to be drawn outside the component's boundary the indicator will be clipped. You may increase the border size to compensate, but borders are not scaled when components are resized. Since the indicator's length is defined as a fraction of its associated scale's radius, it may still elongate past the border if the component is expanded too much. An alternative approach is to use the scale's `zoomFactor` property, set to some value less than one. This has the effect of shrinking the scale so that its boundary is less than its true radius. An indicator whose length is greater than 1.0 appears to extend beyond the scale. Because it is really inside the scale's actual boundary it can be resized without clipping.

Coloring an indicator object

An indicator's color is set in its constructor or by using `javax.swing.JComponent.setForeground()`.

Controlling a indicator's visibility

You can show or hide an indicator by setting its `java.awt.Component.setVisible()` method to `true` or `false` respectively. Test the visibility with `java.awt.Component.isShowing()`.

Positioning an indicator with the mouse

Use the `JCAbstractNeedle.InteractionType` constants to determine how the needle responds to mouse clicks within the Circular Gauge component. To stop the needle from responding to the mouse, use `InteractionType.NONE`. Other possible values are `CLICK`, `DRAG`, or `CLICK_DRAG`.

3.14.2 Constructors

Both `JCAbstractIndicator` and `JCAbstractNeedle` have two constructors. The first takes a `JCScale` object and creates an indicator or needle with default settings. The following ones allow you to set properties, associate it with its parent scale, and, in the case of a needle, specify an interaction type:

```
public JCAbstractIndicator(
    Color foreground,         // foreground color
    double indicatorWidth,    // indicatorWidth
    JCScale scale,            // associated scale
    boolean visible,          // true: indicator is visible
    double inner_extent,      // inner extent
    double outer_extent,      // outer extent
    JCIndicatorStyle indicatorStyle, // indicator style
    double value              // where to place this indicator
    )

public JCAbstractNeedle(
    java.awt.Color,           // foreground color
    double width,             // needleWidth
    JCScale scale,            // associated scale
    JCAbstractNeedle.InteractionType,    // needle interactionType
    boolean visible,          // true: needle is visible
    double,                   // length
    JCIndicatorStyle,         // needleStyle
    double value,             // where to place this needle
    )
```

3.14.3 Methods and Properties

`JCAbstractIndicator` is the super class for `JCCircularIndicator`, `JCLinearIndicator`, and `JCAbstractNeedle`. Indicators are used to point to zero or more selected values on a scale.

Indicator methods and properties

Indicator Properties	Description
`getForeground()` `setForeground()`	In `java.awt.Component`, returns or sets the needle's color, a `java.awt.Color`.

`getIndicatorStyle()`
`setIndicatorStyle()`
	Returns or sets the shape of the indicator. Possible values are:
	`JCIndicatorStyle.ARROW`
	`JCIndicatorStyle.POINTER`
	`JCIndicatorStyle.TAILED_ARROW`
	`JCIndicatorStyle.TAILED_POINTER`
	`JCIndicatorStyle.TRIANGLE`
	
	It is possible to define your own indicator style by extending `JCIndicatorStyle` or by using its constructor.

`getIndicatorWidth()` `setIndicatorWidth()`	Returns or sets the width of the indicator in pixels.
`getInnerExtent()` `setInnerExtent()`	Returns or sets the inner extent of this indicator as a ratio of the scale's width, height, or radius (depending on the scale's type and orientation).
`getOuterExtent()` `setOuterExtent()`	Returns the outer extent of this indicator as a ratio of the scale's width, height, or radius (depending on the scale's type and orientation).
`getPreferredSize()`	Overridden so that the indicator scales with the gauge.
`getScale()`	Returns the `JCScale` associated with this indicator.
`getValue()` `setValue()`	Returns or sets the scale value (not the scale angle) to which the indicator points. When setting this value care should be taken to ensure it is between the scale's min and max values. An indicator may not be dragged outside the scale's range, that is, before its `startValue` or after its `stopValue`.
`isReversed()` `setReversed()`	An indicator that lacks longitudinal symmetry may be reversed. For example, the default `JCIndicatorStyle.ARROW` points outwards on a circular scale. By setting the `reversed` property to `true` the arrow points toward the center rather than at the circumference.

Needle methods and properties

Needle properties allow you to control most aspects of the needle's appearance save for its drawing order (*z* order) with respect to the other gauge components. By managing the order in which `gauge.addNeedle()` is called relative to the other *gauge.addComponent*

methods you achieve your intended layering effect. For instance, by adding a center and then a needle, the part of the needle under the center is obscured.

`JCAbstract Needle` inherits all the methods of:

Needle Methods	Description
`addChangeListener()` `removeChangeListener()`	Adds or removes a listener interested in needle movements.

Needle Properties	Description
`getForeground()` `setForeground()`	In `java.awt.Component`, returns or sets the needle's color, a `java.awt.Color`.
`getGauge()` `setGauge()`	Returns or sets the circular gauge associated with the needle.

`getInteractionType()`
`setInteractionType()`
	The type of allowed mouse interaction. Possible values are:
	`JCAbstractNeedle.InteractionType.NONE`
	`JCAbstractNeedle.InteractionType.CLICK`
	`JCAbstractNeedle.InteractionType.DRAG`
	`JCAbstractNeedle.InteractionType.CLICK_DRAG`

`getLength()` `setLength()`	Returns or sets the needle's length, expressed as a decimal fraction based on the circular scale's radius. Setting a needle's length is the same as setting its outer extent.

`getNeedleStyle()`
`setNeedleStyle()`
	Returns or sets the shape of the needle. Possible values are:
	`JCIndicatorStyle.ARROW`
	`JCIndicatorStyle.POINTER`
	`JCIndicatorStyle.TAILED_ARROW`
	`JCIndicatorStyle.TAILED_POINTER`
	`JCIndicatorStyle.TRIANGLE`
	
	It is possible to define your own needle style by extending `JCIndicatorStyle` or by using its constructor.

`getNeedleWidth()` `setNeedleWidth()`	Returns or sets the width of the needle in pixels.
`getPreferredSize()`	Overridden so that the needle scales with the gauge.

Needle Properties	Description
`getSendEvents()` `setSendEvents()`	Returns or sets a `sendEvents` flag; `true` means events will be sent when the needle's value changes, `false` means don't send the events. Use this method if your code might otherwise trigger a non-terminating sequence of events, such as setting a needle to its value plus one.
`getValue()` `setValue()`	Returns or sets the scale value (not the scale angle) to which the needle points. When setting this value care should be taken to ensure it is between the scale's min and max values. A needle may not be dragged outside the scale's range, that is, before its `startValue` or after its `stopValue`.

3.14.4 Code Sample

If you have used the default gauge constructor, a needle is supplied. You can obtain a handle to it as follows:

```
JCNeedle needle = ((JCNeedle)gauge.getNeedles().firstElement());
```

You can completely configure a needle at the time it is created. The following code sample shows how:

```
JCNeedle needleOne =
    new JCCircularNeedle(
        Color.red,              // needle's color
        10,                     // needle width
        cscale,                 // associated circular scale
        true,                   // visible?
        0.0,                    // inner extent
        1.0,                    // outer extent
        JCNeedleStyle.POINTER,  // needle style
        25.0                    // needle value
        );
```

The remaining step is to add the needle to the gauge:

```
gauge.addNeedle(needleOne, 0);      // Ensure the needle is on top
```

Adding a change listener for needle movements

If you wish to take some action when a needle is moved you can use the fact that a needle movement generates a change event. You may add a change listener to the needle, for example:

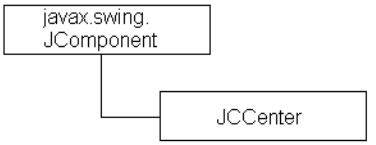
```
needleOne.addChangeListener(this); // For needle movements
```

The class (referenced by the `this` pointer in our example here) that is to respond to needle movements defines a `stateChanged(ChangeEvent e)` method to handle whatever action needs to be taken.

3.15 The Center Object

Center object

A center object is associated with a gauge. You can set the center's radius and its color, then add it using the gauge's `setCenter()` method. Alternatively, you can specify an image for the center object.



A center object is used to mark the position of the center of a circular scale in a `JCCircularGauge` component. It can be a colored circle or a user-supplied image. If an image is specified, the circular disk will not be drawn.

A center object must be associated with a circular scale. Each of its three constructors demands a scale as one of its parameters.

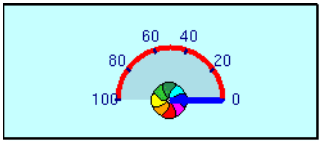


Figure 22 An image used as a center object.

3.15.1 Notes on the Center Object

Setting the center's color

Set the color of the center object using the `setForeground()` method, or in the center object's constructor. The result is a circular disk of this color at the center of the scale. The size of the disk is controlled by the center's `radius` property.

Associating a center with a circular scale

Each circular gauge may have zero or one center objects. A center is associated with the gauge through its circular scale object. Specify the circular scale using the `scale` parameter in the center object's constructor. The center object may be replaced with an image if desired. The image does not rotate.

Sizing the disk

Set the size of the disk using the `radius` property. The value you pass it is a fraction based on the size of the radius of its associated circular scale. For instance, if you set a value of

0.2, the circular disk marking the center has a radius 20% as long as the radius of the circular scale.

Using an image as the visual

If you require anything other than a circular disk to mark the center of your scale, you can supply an `Image`. The image is scaled to the radius of the center object if `setScaleImage()` is `true`. In this case resizing the gauge causes a proportionate change in the image. See Section 3.15.3, Sample Code, for an example.

The center image can be sized to fill the entire scale. In this case it can be made to appear as a background image on which needles, ranges, and extra tick marks may be placed as desired.

Figure 23 Using a center image as a background.

In Figure 23 the parameters for the circular scale have been chosen so that they match the scale markings on the image, which fills the entire circular scale area. The result is an interactive dial with a distinctive look.

Controlling visibility

Use the `visible` parameter in the constructor to control whether the Center object is visible initially. After the object is created, control its visibility using the `setVisible()` method.

3.15.2 Constructors, Properties, and Methods

Constructors

`JCCenter` has three constructors. The one-parameter version requires a reference to the parent `JCCircularGauge`. The other two require a color parameter and either a size parameter in the case of a circular disk, or an image parameter if an image is to be used.

```
JCCenter(JCCircularScale scale)
// Creates a drawn, black center disk using a default radius
// that is 10% of the scale's radius.

JCCenter(JCCircularScale scale,
         java.awt.Color foreground, // color of the disk
         double radius)             // Center is a disk

JCCenter(JCCircularScale scale,
         java.awt.Color foreground,
         java.awt.Image image)       // Center is an image
```

Properties

Property	Description
`getImage()` `setImage()`	Determines the `java.awt.Image` that is to be drawn at the center of the scale.
`getRadius()` `setRadius()`	Gets or sets the size parameter for the center object when it is a circular disk.
`getScaleImage()` `setScaleImage()`	Boolean that controls whether or not to scale the image when the gauge is resized.

Methods

Protected methods `drawDisc()` and `drawImage()` are used to draw the Center, and are used by `paintComponent()` to actually draw the center object on the screen. `getPreferredSize()` is overridden so that the center object's proportional size relative to the circular scale can be maintained. These methods are only of concern to those who wish to subclass the center for custom purposes.

3.15.3 Sample Code

Follow these examples if you want to provide a center in your circular scale.

To use a colored disk as the center of circular scale, use this code:

```
// create a center
JCCenter center = new JCCenter(circularScale, Color.white, 0.1);
gauge.setCenter(center);
```

To provide an image, use this code:

```
// use an image
Image image;
image = Toolkit.getDefaultToolkit().getImage("arrow.gif");
JCCenter center = new JCCenter(circularScale, Color.white, image);
gauge.setCenter(center);
```

3.16 The Constraint Mechanism in JCGauge

3.16.1 RadialConstraint and RadialLayout

The `RadialLayout` class uses an instance of `RadialConstraint` to position a component at a given angle and at a specified proportional distance from the center of the associated circular gauge. Thus, the gauge employs `RadialConstraint` classes to facilitate laying out gauge objects in such a way that the objects' angular positions are maintained as the gauge is resized, as well as maintaining their proper radial proportions.

It supports the placement of any component on the gauge area, not just indicators, needles, ranges, and ticks. Usually these are labels used to annotate a Circular Gauge, but they may be any `JComponent`, even another gauge.

Constructors

`RadialConstraint` has a single constructor which is passed a *gauge*, an *extent*, and an *angle*. The `extent` parameter specifies the radial distance for the placement of the component. The `angle` parameter specifies the angle. The center of component's bounding rectangle is placed on the gauge at the point defined by the two parameters. Typically an instance of `RadialConstraint` is passed via the `addLabel()` method in `JCCircularGauge`, which passes it to an `add()` method that knows how to use `RadialLayout` to position the component.

Here's an example:

```
JCCircularGauge gauge = new JCCircularGauge();
JLabel label = new JLabel("<html>Pressure (lbs/in&sup2;)");
gauge.addLabel(label, new RadialConstraint(gauge, 0.35, 90));
```

3.16.2 Linear Constraint and Linear Layout

The `LinearLayout` class uses an instance of `LinearConstraint` to position a component at a given extent and at a specified pixel distance from the origin of the associated circular gauge. Thus, the gauge employs `LinearConstraint` classes to facilitate laying out gauge objects in such a way that the objects' relative positions are maintained as the gauge is resized.

It supports the placement of any component on the gauge area, not just indicators, needles, ranges, and ticks. Usually these are labels used to annotate a linear gauge, but they may be any `JComponent`, even another gauge.

Constructors

`LinearConstraint` has a single constructor which is passed a *gauge*, an *extent*, and a *position*. The `extent` parameter specifies the proportional distance from the top left of the rectangle enclosing the gauge. The distance is vertical for horizontal scales and horizontal for vertical scales, and is specified as a ratio of this distance to the height or width of the scale. The `position` parameter specifies the distance as an integer representing a percentage of the height or width from the top or left of the scale. The center of the positioned component's bounding rectangle is placed on the gauge at the point defined by these two parameters. Typically an instance of `LinearConstraint` is passed via the `addLabel()` method in `JCLinearGauge`, which passes it on an `add()` method that knows how to use `LinearLayout` to position the component.

Here's an example:

```
JCLinearGauge gauge = new JCLinearGauge();
JLabel label = new JLabel("Pressure Point");
gauge.addLabel(label, new LinearConstraint(gauge, 0.35, 90));
```

3.17 Labels

Labels are used to annotate a gauge. Any number of them may be placed anywhere within the boundaries of the gauge area using a gauge's `addLabel()` method, which in turn uses a `RadialConstraint` or a `LinearConstraint`. Because it is a `JLabel`, it has user-controllable text, position, background and foreground color, images, and borders.

The `RadialConstraint` class, whose constructor is `RadialConstraint(JCGauge gauge, double extent, double angle)`, lets you specify a label's position by giving a distance from the center of the scale and an angle. The constructor for a linear constraint is `LinearConstraint(JCGauge gauge, double extent, int position)`, which lets you specify a label's position by giving the extent in the transverse direction to the scale and a position along the scale in pixels.

Note that there is an automatic mechanism for providing numeric labels on tick objects or for specifying labeled ticks in user-specified formats. See Section 3.12, Tick Objects, for a discussion of tick labels.

3.17.1 Notes on the Using Labels

You can choose a location within the gauge area by specifying the location of the center of the rectangle containing the `RadialConstraint` or `LinearConstraint` class.

You can choose a border type using `setBorder` and adjust its appearance using the various border factory methods.

One way of setting the font is by using a `JLabel`'s ability to process HTML tags. Set a font using the `<font>` tag and the color using `color = HTMLcolorValue` within the tag. If you are adding text to a circular gauge, you can do the following:

```
JLabel l1 = new JLabel("<html><font color=black>
                        Start Angle = 90\u00B0");
l1.setToolTipText("Start Angle = 90\u00B0");
gauge.addLabel(l1, new RadialConstraint(gauge, 0.60, 15), 0);
```

You can control whether a text object is drawn using `setVisible()` and determine its visibility with `isVisible`.

Aligning text

The lines of text within a label may be centered, or right justified. If text is not centered it may appear that `RadialConstraint` is not positioning the text at the correct angle. In Figure 24 both text areas are aligned vertically, but without the borders on the components it appears that the lower label is not vertically aligned. Here is the code that produces this layout:

```
JLabel l1     = new JLabel(
            "<html><font color=black>
                        <P ALIGN=CENTER>Start Angle <br>= 0\u00B0");
l1.setToolTipText("Start Angle = 0\u00B0");
l1.setBorder(new BevelBorder(BevelBorder.RAISED));
gauge.getGaugeArea().add(l1,
                        new RadialConstraint(gauge, 0.50, 90), 0);

JLabel l2     = new JLabel(
        "<html><font color=black><P>Stop Angle <br>= 360\u00B0");
l2.setToolTipText("Stop Angle = 360\u00B0");
l2.setBorder(new BevelBorder(BevelBorder.RAISED));
gauge.getGaugeArea().add(l2,
                        new RadialConstraint(gauge, 0.50, 270), 0);
```

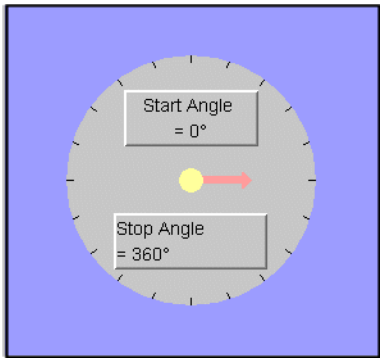


Figure 24 If text is not centered it may appear to be placed at the wrong angle.

Adding a component

A label (in reality, any `JComponent`) is an independent rectangular region that can be placed anywhere within the gauge area by specifying the center of the component using either the `LinearConstraint` or the `RadialConstraint` class.

3.17.2 Sample Code

The first code snippet shows the addition of a label positioned half way from the center of a gauge to its circumference at an angle of 45°.

```
RadialConstraint rConstraint = new RadialConstraint(gauge, 0.50, 45)
JLabel label     = new JLabel("Start Angle = 0");
label.setToolTipText("Start Angle = 0\u00B0");
gauge.getGaugeArea().add(label, rContstraint, 0);
```

Usage is the same for a linear constraint. This one puts a label at (0, 0) on a linear scale:

```
JLabel l0 = new JLabel("<html><font color=black>0");
l0.setToolTipText("0 marks the spot!");
gauge.getGaugeArea().add(l0, new LinearConstraint(gauge, 0.0, 0));
```

This snippet shows how to label tick marks with a String that specifies the units of the measurement. It uses the `JCLabelGenerator` interface and its method, `makeLabel()`.

```
// create a label generator to mark ticks with their units
          tick.setLabelGenerator(new JCLabelGenerator() {
                  public JComponent makeLabel(JCTick tick, double value,
                      RadialConstraint constraint) {
                      String s = String.valueOf((int) value) + " units";
                  JLabel label = new JLabel(s);
                  label.setToolTipText(s);
                      return label;
                  }
      });
```

3.18 Events and Listeners in JCGauge

`JCGaugePickEvent` represents a pick event in `JCGauge`. A pick event occurs when a `JCGaugePickListener` is installed on a gauge and the mouse button is pressed over a `JCGauge` object.

Method	Description
`getComponent()`	The component associated with this event.
`getGauge()`	Returns the gauge associated with this event.
`getPoint()`	Returns the (*x*, *y*) point of the click.
`getValue()`	The value associated with this event. This is the scale value corresponding to the place where the mouse click occurred.

Method	Description
`toString()`	Returns the point where the mouse was clicked and the associated scale value.

Interface `JCGaugePickListener` has one method, `pick()`. It is called on the object that has installed itself as a listener by invoking `gauge.addPickListener()`. See *GaugePickExample.java* for an example of the use of a pick listener.

3.19 Utility Functions for JCGauge

Static utility functions are found in `com.klg.jclass.swing.gauge.GaugeUtil`. They perform conversions or other common functions needed within `JCGauge`.

3.19.1 GaugeUtil

The following table lists some methods that you might find useful. Consult the API for full details.

GaugeUtil Method Name	Description
`clamp()`	If the passed-in value in the first parameter is between the second (*min)* and third (*max)* parameters, leave it unchanged. Otherwise return the *min* or *max* value, whichever is closer.
`drawCircleFor Circular Scale()`	In a circular scale, draws a filled circle based on the supplied scale value, inner extent, and outer extent. Since the method takes a `Graphics` context as a parameter, you will have to override one of the `paint()` methods to use it.
`drawCircleFor Linear Scale()`	In a linear scale, draws a filled circle based on the supplied scale value, inner extent, and outer extent. Since the method takes a `Graphics` context as a parameter, you will have to override one of the `paint()` methods to use it.
`drawLinear Polygon()`	Draws a polygon given a linear scale, inner and outer extent, width, a value on the scale at which to draw, and various other attributes. The type of polygon drawn is a `JCPolygon`, described in the next section.
`normalize Angle()`	Transforms its (double) argument to an angle between 0° and 360°.
`rotate()`	Rotates a `Polygon` by an amount given in degrees by its second argument.

GaugeUtil Method Name	Description
`scale()`	Scales a `Polygon` specified in its first argument by independent amounts in the horizontal and vertical directions. The second and third arguments are the X- and Y-scaling ratios.

`fromRadians toRadians()`
 	
`normalize Angle()`
	Transforms its (double) argument, which should be an angle, to an angle in degrees or in radians.
	
	Converts an angle of any size to one in the range 0° - 360°.

`translate()`	Translate a `Polygon` by the amount given by X- and Y-parameters, given as integers representing pixels.

`valueToAngle()`
 	
`valueTo Position()`
	Converts a double to an angle in degrees. Parameters are `value`, `start_value`, `stop_value`, `start_angle`, and `stop_angle`. The returned value is the angle that corresponds to the input value, which is interpreted as a scale value and thus should be between the *min* and *max* values for the scale.
	
	Example: A circular scale's `start_angle` is 45° and its `start_value` is 10. Its `stop_angle` is 135° and its `stop_value` is 55°. Given a value of 20, which is between the *min* and *max* values for the scale, the function returns 65°, the angular position on the given scale for that value. Note that the input value must be within the range set by the scale.
	
	`valueToPosition()` converts a value on a linear scale to its position in pixels measured from the edge of the component.

3.19.2 JCPolygon

`JCPolygon` is the abstract super class for `JCIndicatorStyle` and `JCTickStyle`. It is a `Polygon` that retains the dimensions of its bounding box.

GaugeUtil Method Name	Description
`getExtrema()`	Given a `Polygon`, returns its bounding `Rectangle`.

3.20 JCCircularGaugeBean and JCLinearGaugeBean

The two JavaBeans in the gauge package are `JCCircularGaugeBean` and `JCLinearGaugeBean`. Their purpose is to make it easy to set gauge properties at design time in an integrated development environment (IDE) tool.

The `JCGauge` components are designed to be highly configurable. Interfaces support the possibility of replacing indicators, needles, ranges, scales, and ticks with custom-designed components. In the same vein, the concrete objects in JClass Elements based on these interfaces may be subclassed to provide extra functionality. All this flexibility comes at the price of having a gauge's properties distributed throughout the sub objects, making them hard to get at via an IDE's property sheet. To solve this problem, the gauge's JavaBeans provide accessors for the most-needed properties, placing them all within the JavaBean so that they are presented in one table by the IDE.

The property names as they appear in an IDE are listed in Appendix A.

The following figure shows the property sheets for `JCCircularGauge` and `JCCLinearGauge` in JBuilder. You see that you can set many needle, scale, and tick properties.

Circular Gauge Bean in JBuilder

name	jCCircularGaugeBean1
constraints	null
autoTickGeneration	True
centerColor	Black
centerRadius	0.1
direction	DIRECTION_FORWARD
drawTickLabels	True
drawTickMarks	True
needleColor	Black
needleInnerExtent	0.0
needleInteractionType	INTERACTION_NONE
needleLength	1.0
needleOuterExtent	1.0
needleStyle	NEEDLE_ARROW
needleValue	0.0
needleWidth	15.0
paintCompleteBackground	False
precision	-1
scaleColor	White
scaleExtent	1.0
scaleMax	100.0
scaleMin	0.0
snapToValue	False
startAngle	0.0
stopAngle	360.0
tickColor	Black
tickFont	"Dialog", 0, 12
tickFontColor	Black
tickIncrement	10.0
tickInnerExtent	0.85
tickLabelExtent	0.8
tickOuterExtent	1.0
tickStartValue	0.0
tickStopValue	100.0
tickStyle	TICK_LINE
tickWidth	2.0
type	TYPE_FULL_CIRCLE
useDefaultPrecision	True
zoomFactor	1.0

Linear Gauge Bean in JBuilder

name	jCLinearGaugeBean1
constraints	null
autoTickGeneration	True
direction	DIRECTION_FORWARD
drawTickLabels	True
drawTickMarks	True
needleColor	Black
needleInnerExtent	0.0
needleInteractionType	INTERACTION_NONE
needleLength	1.0
needleOuterExtent	1.0
needleStyle	NEEDLE_ARROW
needleValue	0.0
needleWidth	15.0
precision	-1
scaleColor	White
scaleExtent	1.0
scaleMax	100.0
scaleMin	0.0
snapToValue	False
tickColor	Black
tickFont	"Dialog", 0, 12
tickFontColor	Black
tickIncrement	20.0
tickInnerExtent	0.85
tickLabelExtent	0.8
tickOuterExtent	1.0
tickStartValue	0.0
tickStopValue	100.0
tickStyle	TICK_LINE
tickWidth	2.0
useDefaultPrecision	True
zoomFactor	1.0

Figure 25 The properties tables for JCGaugeBeans in JBuilder.

3.21 Adding Other Components to a Gauge

As an example of adding any `JComponent` to a gauge, consider a case where the need is to provide a smaller circular gauge within a larger one, such as a stopwatch whose larger scale counts off seconds and whose smaller scale indicates the number of minutes that have elapsed. In the code snippet that follows, `bigGauge` is the one containing the second hand and `smallGauge` is the one containing the minute hand. It does not show the details

of setting up the properties of the two scales, but once they are configured it is easy to place the smaller gauge within the larger.

```
JCCircularGauge bigGauge;
JCCircularGauge smallGauge;
bigGauge.addLabel(smallGauge,
                  new RadialConstraint(smallGauge, 0.35, 90), -1);
```

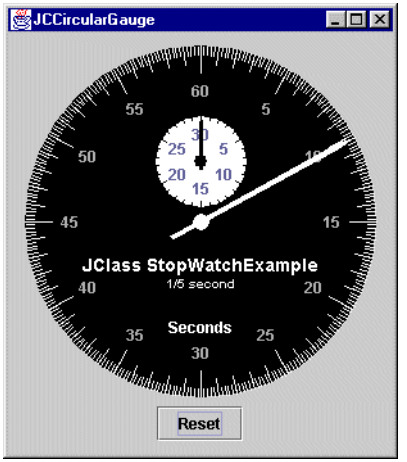


Figure 26 One circular gauge within another.

3.22 JClass 4 to JClass 5: A Mini-porting Guide

The gauge classes have been extensively reorganized for the JClass 5 release, but you should find that there is little or no impact on your existing applications that use circular gauges. You may find that your code compiles against the new classes without the need to change anything; however, if you discover that your application does not run with JClass 5 or newer releases, you can refer to the table below. It should encompass almost all the issues that need attention.

In JClass 4 was:	In JClass 5 becomes:
`gauge.getNeedles().firstElement()`	`(JCCircularNeedle) gauge.getNeedles().firstElement()`
`gauge.getRanges().firstElement()`	`(JCCircularRange) gauge.getRanges().firstElement()`
`gauge.getScale().firstElement()`	`(JCCircularScale) gauge.getScale().firstElement()`

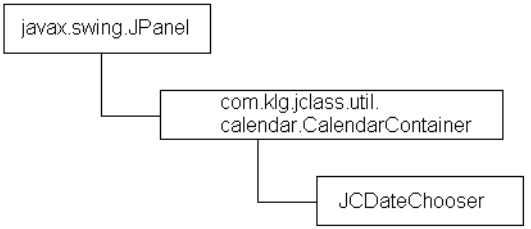
In JClass 4 was:	In JClass 5 becomes:
`gauge.getTicks().firstElement()`	`(JCCircularTick) gauge.getTicks().firstElement()`
`JCCenter(gauge)`	`JCCenter(circularScale)`
`JCCircularNeedle(gauge)`	`JCCircularNeedle(scale)`
`JCNeedle.InteractionType`	`JCAbstractNeedle.InteractionType`
`JCNeedle.setForeground()`	`JCAbstractNeedle.setForeground()`
`JCNeedleStyle`	`JCIndicatorStyle`
`JCScale.Direction`	`JCAbstractScale.Direction`

4

Date Chooser

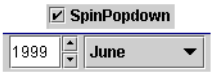
4.1 Features of JCDateChooser

`JCDateChooser` is a component that displays a calendar in one of four variant forms. Each one displays the days of the month in the familiar form of a calendar, but varies the ways that the month and year are displayed.

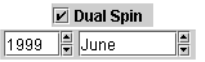


The different styles are:

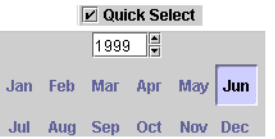
- **Spin Popdown**
  The year is shown in a spin box; the month is shown in a popdown.



- **Dual Spin**
  Spin boxes are used to display both the year and the month.



- **Quick Select**
  The year is shown in a spin box; a table is used to display all twelve months. One of the months may be highlighted to indicate that it has been selected.



- **Read Only**
  The year and month are shown in non-editable fields. The table showing the days of the month is read-only as well. Selected "special" dates still appear highlighted.

- Like the standard Swing components, `JCDateChooser` provides for the optional use of a *Tool Tip*.

The accompanying figure shows the full component in its *Quick Select* form, so the year is shown in a spin box, while tables are used to show months and days of the month. Note that special days, in this case Saturdays and Sundays, are distinguishable from the others.

You can observe the other calendar styles by running the example called *examples.elements.DateChooser.java*.

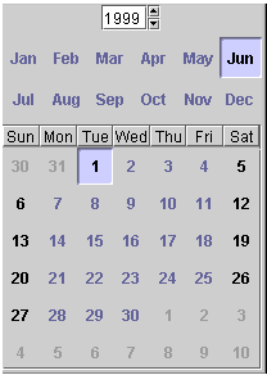


Figure 27 A JCDateChooser.

4.2 Classes and Interfaces

Classes in the com.klg.jclass.util.calendar Package

`AbstractLabel`	An abstract class for setting dates in a locale-dependent fashion. It is used for `MonthLabel`s and `YearLabel`s.
`DayTable`	The days of the month.
`JCCalendarContainer`	A container that manages `CalendarComponent` children. That is, any calls to the calendar component interface methods are automatically passed to any calendar component children.
`JCCalendar`	A calendaring utility class that can define special dates and custom date classes, such as "Tuesdays" or "April Fool's Day" (April 1). `JCCalendar` augments `java.util.Calendar` by providing extra date-specific capabilities.
`JCDateChooser`	A GUI component with four styles of calendar. Special dates display differently from other dates.
`MonthLabel`	Displays a locale-specific table of month labels.
`MonthPopdown`	Encapsulates the months of the year in a popdown.
`MonthSpin`	Encapsulates the months of the year in a spin box.
`MonthTable`	Encapsulates the months of the year in a table.
`YearLabel`	Presents the designated year in a label.
`YearSpin`	Presents the designated year in a spin box.

4.2.1 The CalendarComponent Interface

The calendar component uses a single model for the day, month, and year.

The methods declared in `public interface CalendarComponent` are:

- `public void setCalendarModel(JCValueModel model)`
  Sets the model which provides the current date being used by the calendar component.
- `public void setSpecialDates(JCCalendar special_dates`
  Sets the special dates being used by the calendar component.
- `public void setLocale(Locale locale);`
  Sets the locale being used by the calendar component.

- `public void addActionListener(ActionListener l)`
  Adds an action listener to detect specific actions on this component.
- `public void removeActionListener(ActionListener l)`
  Removes action listener to detect specific actions on this component.

4.2.2 The SpecialDate Interface

This interface has only one method:

```
boolean isSpecialDate(int year, int month, int date, int week)
```

You'll note that the numeric value for the week (1 - 52) is a redundant parameter in `isSpecialDate`. This is done for efficiency's sake. If you implement the `SpecialDate` interface, you will have to supply a numeric value for the week even though it is possible to compute it from the first three parameters in `isSpecialDate`. Note that you use `JCCalendar`'s `dayofweek()` method to calculate this value.

4.3 Properties

Properties of JCDateChooser

`chooserType`	For specifying the date chooser type, use one of `JCDateChooser.DUAL_SPIN`, `JCDateChooser.QUICK_SELECT`, `JCDateChooser.READ_ONLY`, or `JCDateChooser.SPIN_POPDOWN`.
`days`	The days array* used by the date chooser
`minimumDate`, `maximumDate`	Bounds between which dates are valid. Any date outside these bounds will be rejected by the validator.
`months`	The months array* used by the date chooser
`shortMonths`	The months' short form array* used by the date chooser
`toolTipText`	The text that appears in the tool tip box when the mouse pointer rests over the component.
`value`	The currently selected date.

*This array must be at least as long as what the JCDateChooser's locale expects. By default, the array is initialized to the locale's default list.

Properties of JCCalendar

`addSpecialDate`, `removeSpecialDate`	Mark a special date on the calendar, or remove one that has already been designated as special.
`isSpecialDate`	A Boolean indicating whether the given date is special.

For a full listing of the properties, see Appendix A.

4.4 Methods

JCDateChooser

There are four visual aspects to the date chooser: *Quick Select*, *Dual Spin*, *Spin Popdown*, and *Read Only*. Use `setChooserType(int type)` to select the type you want to display. `type` is one of `JCDateChooser.DUAL_SPIN`, `JCDateChooser.QUICK_SELECT`, `JCDateChooser.READ_ONLY`, or `JCDateChooser.SPIN_POPDOWN`.

The `CalendarComponent` interface provides the mechanism for extracting parts of a `JCDateChooser` date. The methods are `getDayComponent()`, `getMonthComponent()`, and `getYearComponent()`.

As noted in the section on properties, you set minimum and maximum dates by providing `setMinimumDate()` and `setMaximumDate()` with a `java.util.Calendar` object.

Set the currently selected date programmatically with `setValue()`, or determine what its value is with `getValue()`. The parameter is once again a `java.util.Calendar` object.

JCCalendar

While not subclassed from `java.util.Calendar`, `JCCalendar` is used in conjunction with it to provide for a classification of some dates as "special." Special days are managed through these methods: `addSpecialDate()` and `removeSpecialDate()`, which take a `SpecialDate` object as a parameter. and `isSpecialDate()`. There is no restriction on how many dates may be deemed special.

The class contains a number of utility methods. One, called `isLeapYear()`, can be used to determine if any given year is a leap year. With `dayOfWeek()`, you can determine the day of the week given a year, month, and day. You can clone a `Calendar` object using `copyCalendar()`.

Certain applications involving calendars require that certain days are treated specially. For example, some businesses that are open on the weekend close on Mondays. In such a case, it is useful to be able to lump all Mondays together and classify them as days when the store is closed. Perhaps the store's founder always holds a sale on his birthday, the 29th of February. In this and similar cases, it's useful to be able to denote anniversary days that occur on the same date every year. There are other days, such as Labor day, which is defined as the first Monday in September. `JCCalendar` contains inner classes `DayOfWeek`, `MonthDayOfMonth`, `MonthWeekDayOfWeek` to help you deal with these situations. These classes allow you to store various calendar objects of the types just mentioned. The first of these allows you to store a day, like Sunday, by declaring an instance variable

```
DayOfWeek sunday = new DayOfWeek(0);
```

From the example, you see that the seven days of the week are mapped using a zero-based index.

To store a date like July 4, use:

```
MonthDayOfMonth july4 = new MonthDayOfMonth(7, 4)
```

To store a date like Labor Day, use:

```
MonthWeekDayOfWeek laborDay = new MonthWeekDayOfWeek(9, 1, 1)
```

4.5 Examples

The illustrative code snippets shown here demonstrate how you can create special days and how you can set bounds on the permissible dates. Refer to the Date Chooser

example, automatically installed into *com/klg/jclass/examples/elements/* when you install JClass Elements, for the complete example.

```
//The location of JCDateChooser
import com.klg.jclass.util.calendar.*;

//Create an instance of JCDateChooser within your class.
dateChooser = new JCDateChooser();
...
//Create a "special day"
JCCalendar special_dates = new JCCalendar();
// Make Sundays special days
special_dates.addSpecialDate(new JCCalendar.DayOfWeek(0));
...
dateChooser.setSpecialDates(special_dates);
...
//Set bounds for the calendar
Calendar max = Calendar.getInstance();
   max.set(max.YEAR, 2050);
dateChooser.setMaximumDate(max);
```

5

JCPopupCalendar Component

5.1 Features of JCPopupCalendar

`JCPopupCalendar` is a component that allows you to edit the date and time using a drop-down calendar. In its editable form, the popup calendar displays a text field with a button next to it. Pushing on the button pops down a calendar from which a date and time can be selected. By default, the calendar has spinboxes for the year, month, and time along with a table which displays the days of the month. The day table updates each time the year and month are changed with the mouse clicks. The time spinbox allows editing of the hour, minute, second, and meridiem.

`JCPopupCalendar` is an extension of `JComboBox`. Instead of selecting an item from a drop-down list, the user selects a date/time value using a popup calendar editor. `JCPopupCalendar` uses a `JFormattedTextField` that is configured to edit dates as its text editor. The text editor's value is kept in sync with the popup calendar editor's value, so changing one will automatically update the other. As with `JComboBox`, `JCPopupCalendar` is non-editable by default. In this case, the text field is replaced with a button which when selected activates the popup calendar editor. The popup calendar editor can still change the value in the non-editable case.

Note: This component can only be used with JDK 1.4 and above. Those using JDKs prior to JDK 1.4 can use `JCPopupField` which is a part of JClass Field.

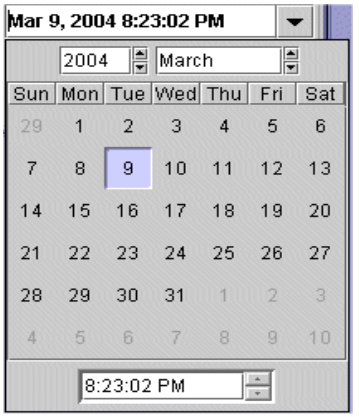


Figure 28 A sample popup calendar.

The default value for a `JCPopupCalendar` component is the current date and time.

5.2 Classes

The pertinent classes to `JCPopupCalendar` are:

`JCPopupCalendar`	Creates the `JCPopupCalendar` component.
`JCPopupCalendarEditor`	Interface that the popup editor must implement.
`JCPopupCalendarBeanInfo`	Contains the `JCPopupCalendar` bean information.
`DateTimeChooser`	Creates the component that the popup calendar editor displays to the user. This contains the `JCDateChooser` and `TimeSpin` components that are manipulated by the user to select a new date and/or time.
`DateTimePopup`	Creates the actual calendar popup editor. This class is responsible for implementing the `JCPopupCalendarEditor` interface, containing the `DateTimeChooser`, and communicating with the actual popup object.
`JCDateChooser`	Allows the date to be edited. This component can be configured to use various formats. For more information, see Features of JCDateChooser, in Chapter 4.

`TimeSpin`	Allows the time to be edited with a spinner.
`JCPopupListener`	Responds to events that happen in the popup.
`JCPopupEvent`	Gets passed to `JCPopupListeners` when the value is committed from the popup calendar to `JCPopupCalendar`.

5.3 Properties

Method	Type	Description
`calendarType`	int	Calendar type. This must be one of the following: ■ JCPopupCalendar.DATE_TIME (default): indicates that both the date and time can be edited. ■ JCPopupCalendar.DATE: indicates that only the date can be edited.
`hidePopupOnDayTableClick`	boolean	If the value is TRUE, the popup calendar editor will pop down when a day is selected. If the value is FALSE (default), the day can be changed without the editor popping down, and the calendar will pop down when it is double-clicked.
`maximumDate`	java.util.Date	Maximum date value. If a date is provided, the popup calendar cannot be set later than that value. Default is `null`, meaning that there is no maximum.
`minimumDate`	java.util.Date	Minimum date value. If a date is provided, the popup calendar cannot be set earlier than that value. Default is `null`, meaning that there is no minimum.
`popupEditor`	JCPopupCalendarEditor	Current calendar popup editor. Default is `DateTimePopup`. The `DateTimeChooser` is retrieved through this property, allowing for other properties to be set (for example, `chooserType`). **Note:** It is not recommended that you build your own `popupEditor`, but that you customize the one that is provided.

showApplyButton	boolean	If the value is TRUE, an Apply button is available on the calendar which, when selected, will commit the current value and will pop down the editor. This may be useful if the hidePopupOnDayTableClick property is FALSE, but it is still desirable to provide another way to dismiss the popup, other than double clicking on the day table. If the value is FALSE (default), the Apply button is not available.
showPopupOnUpDownArrow	boolean	If the value is TRUE (default), the calendar popup editor will pop up when the up or down arrow is selected in the text editor. If the value is FALSE, the calendar popup editor will not pop up when the up or down arrow is selected in the text editor.
value	java.util.Date	Current value of the component. Defaults to the current date and time.

5.4 Constructors and Methods

JCPopupCalendar Constructors

`JCPopupCalendar`'s constructor constructs a popup calendar, where the default date and time can be configured, as well as the locale and calendar type.

Constructor	Description
`JCPopupCalendar()`	Constructs a `JCPopupCalendar.DATE_TIME` calendar type, with the current date, the current time, and the default locale selected.
`public JCPopupCalendar (int calendarType)`	Constructs a `JCPopupCalendar` of the given calendar type with the current date, the current time, and the default locale selected.
`public JCPopupCalendar (Date d)`	Constructs a `JCPopupCalendar.DATE_TIME` calendar type, with the default locale and provided date and time.
`public JCPopupCalendar (int calendarType, Date d)`	Construct a `JCPopupCalendar` of the given type with the default locale and provided date and time.

`public JCPopupCalendar (Date d, Locale l)`	Constructs a `JCPopupCalendar.DATE_TIME` calendar type, with a specified locale and provided date and time.
`public JCPopupCalendar (int calendarType, Date d, Locale l)`	Constructs a `JCPopupCalendar` of the calendar type, with a specified locale and provided date and time.

5.5 Listeners and Events

The `JCPopupListener` listens for `JCPopup` events, which are generated when the calendar popup editor's value is committed to `JCPopupCalendar` and the popup is popped down. `JCPopupEvent` has the following methods:

Method	Description
`getSource()`	The source of the event which is the `DateTimePopup` object.
`getNewValue()`	The new value to be committed.

5.6 Examples

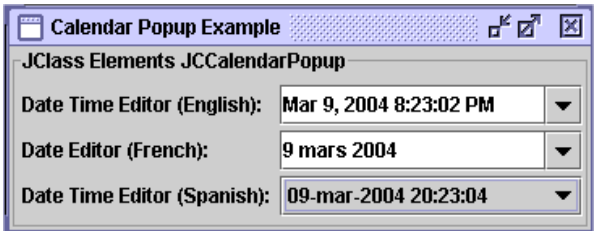
Please refer to `examples.elements.CalendarPopup.java` to see a working popup calendar, or refer to `examples.elements.CalendarDialog.java` to see a how to use the `DateTimeChooser` component in a dialog editor.

The following code produces a screen with three possible popup calendars: one in English, one in French, and one in Spanish.

```
import com.klg.jclass.swing.JCPopupCalendar;
import com.klg.jclass.util.swing.JCExitFrame;
import com.klg.jclass.util.swing.JCAlignLayout;
import com.klg.jclass.util.JCEnvironment;
import javax.swing.*;
import javax.swing.border.TitledBorder;
import java.awt.*;
import java.util.Locale;
import java.util.Date;

public class CalendarPopup extends JPanel {

protected JCPopupCalendar popup1, popup2, popup3;

public CalendarPopup()
{
// Set the layout
setLayout(new BorderLayout());
```

```
// Place all the popup fields in a panel
JPanel p = new JPanel();
add(p, BorderLayout.CENTER);
JCAlignLayout mgr = new JCAlignLayout(2, 3, 3);
p.setLayout(mgr);
p.setBorder(new TitledBorder("JClass Elements JCCalendarPopup"));

//
// Example of a Date/Time JCPopupCalendar in English
//
Locale locale = new Locale("en", "US");
popup1 = new JCPopupCalendar(JCPopupCalendar.DATE_TIME, new Date(),
       locale);
popup1.setEditable(true);
Component c = popup1.getEditor().getEditorComponent();
if (c instanceof JTextField) {
((JTextField)c).setColumns(15);
}
p.add(new JLabel("Date Time Editor (English): "));
p.add(popup1);
mgr.setResizeWidth(popup1, true);

//
// Example of a Date JCPopupCalendar in French
//
locale = new Locale("fr", "FR");
popup2 = new JCPopupCalendar(JCPopupCalendar.DATE, new Date(), locale);
popup2.setEditable(true);
p.add(new JLabel("Date Editor (French): "));
p.add(popup2);
mgr.setResizeWidth(popup2, true);

//
// Example of a non-editable Date/Time JCPopupCalendar in Spanish.
//
locale = new Locale("es", "ES");
popup3 = new JCPopupCalendar(JCPopupCalendar.DATE_TIME, new Date(),
       locale);
popup3.setEditable(false);
p.add(new JLabel("Date Time Editor (Spanish): "));
p.add(popup3);
mgr.setResizeWidth(popup3, true);
}

public static void main(String[] args)
{
   if (JCEnvironment.getJavaVersion() < 140) {
       System.err.println("\nThis example is incompatible " +
                          "with JDKs prior to 1.4.0
       System.exit(1);
}
```

```
JCExitFrame frame = new JCExitFrame("JCPopupCalendar Examples");
CalendarPopup t = new CalendarPopup();
frame.getContentPane().add(t);
frame.pack();
frame.show();
}

}
```

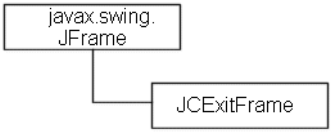


6

Exit Frame

6.1 Features of JCExitFrame

A subclass of `JFrame` that listens for window close events and exits the application when the event is received, or hides the window so that it can be made visible later on. There is a `JFrame` constant in Java™2 v1.3 called `EXIT_ON_CLOSE` that performs the same function.



It is useful for applications containing a single frame. If you used the utility frames available in JClass 3.x versions of *jclass.contrib*, it is useful to know that this replaces `DemoFrame` and `ContribFrame`.

6.2 Properties

A `JCExitFrame` has the same properties as a `JFrame`, and one additional one:

`exitOnClose` — A Boolean property that determines whether the application should exit when the user closes the frame or when `close()` is called (default: `true`). If set to `false`, the frame is hidden; it can be made visible later.
Note: Compare this to using `JFrame.EXIT_ON_CLOSE` in JDK 1.3, which performs the same function.

For a full listing of the properties, please see Appendix A, Bean Properties Reference.

6.3 Methods and Constructors

Methods

JCExitFrame subclasses from JFrame, making it a JFrame with a built-in mechanism for catching window-closing events. The following methods report or control which action is taken when a window-closing event is received.

getExitOnClose()	Returns false if the window will be hidden rather than exiting when a window-closing event is received.
setExitOnClose()	A Boolean method that determines whether the application should exit when the user closes the frame or when close() is called (default: true). If set to false, the frame is hidden; it can be made visible later.

Constructors

There are two constructors. The default constructor provides an untitled frame while the other accepts a parameter which is used to set the frame's title.

JCExitFrame()	Default constructor.
JCExitFrame(String title)	The parameter provides a title for the frame.

6.4 Examples

Use a JCExitFrame as you would a JFrame, and manage window closing events using the exitOnClose property.

```
import java.awt.Font;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import com.klg.jclass.util.swing.*;

public class ExitFrameExample {

        static String message0 = "Many JClass examples and demos
                                                use a JCExitFrame.";
        static String message1 = "\n\nKeep in mind that you can
          hide a JCExitFrame \nrather than disposing of it entirely.";
        static String message = message0 + message1;

        static JTextArea messageArea = new JTextArea(message);
```

```
    public static void main(String[] args){
        String title = "A Basic Frame That Responds to Window-Closing
                        Events";
        JCExitFrame frame;

        frame = new JCExitFrame(title);
        frame.setSize(450, 100);
        frame.setVisible(true);
        frame.setExitOnClose(false); // Hide the window
                                     // instead of closing it.



        messageArea.setFont(new Font("Times-Roman", Font.BOLD, 14));
        frame.getContentPane().add(new JScrollPane(messageArea),
                                                          "Center");
        messageArea.setVisible(true);

        }
}
```

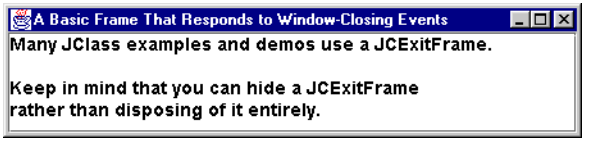


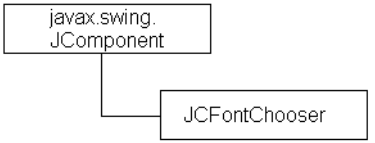
Figure 29 A JCExitFrame containing a JTextArea.

7

Font Choosers

7.1 Features of JCFontChooser and its Subclasses

`JCFontChooser` is the abstract base class for `JCFontChooserBar` and `JCFontChooserPane`. It provides common data and methods for both components.



- Constructors let you specify what the default fonts and sizes are, as well as letting you set whether underlining is on.
- `JCFontChooserPane` – provides a pane of controls designed to allow a user to manipulate and select a font. It is suitable for use in a tab pane or a dialog window. `JCFontChooserPane` includes a preview area with sample text.
- `JCFontChooserBar` provides a pane of controls designed to allow a user to manipulate and select a font. It is suitable for use in a `JToolbar`.
- Like the standard Swing components, `JCFontChooserBar` provides for the optional use of a *Tool Tip*.

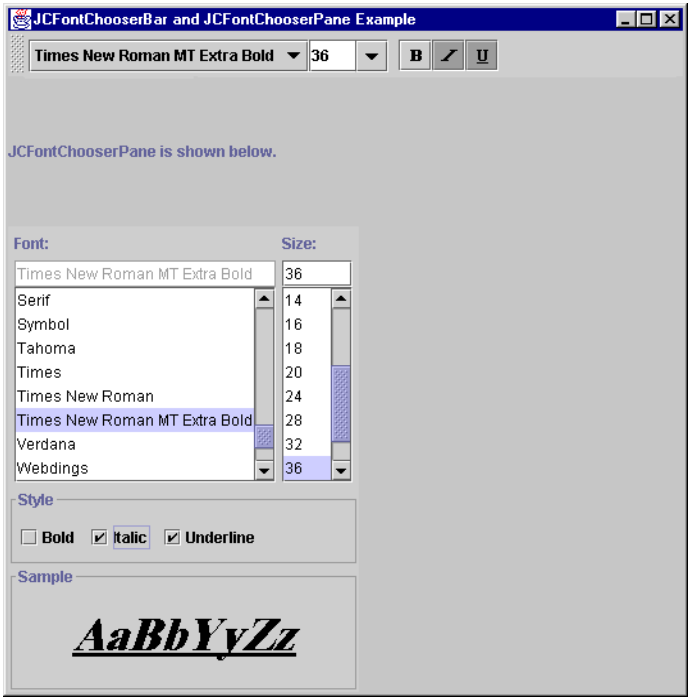


Figure 30 A JCFontChooserBar and a JCFontChooserPane.

7.2 Classes

JCFontEvent	Used to inform listeners that the font has been changed. Its constants are: JCFontEvent.FONT_NAME_CHANGE, JCFontEvent.FONT_SIZE_CHANGE JCFontEvent.BOLD_STYLE_CHANGE, JCFontEvent.ITALIC_STYLE_CHANGE, JCFontEvent.UNDERLINE_STYLE_CHANGE.
JCFontChooserBar	A GUI component suitable for a menu bar.
JCFontChooserPane	A GUI component suitable for a dialog or a tabbed pane.
JCFontListener	The listener interface. Methods are fontChanging and fontChanged.
JCFontAdapter	A convenience class that provides empty implementations of the listener interface's methods.

7.3 Properties

Properties of JCFontChooserBar and JCFontChooserPane

toolTipEnabled	A Boolean property that indicates whether Tool Tips are being used. The get method is called isToolTipEnabled.
selectedFont	The set method of this property has three different signatures: a single parameter Font font, a two parameter version, Font font, boolean underline, and a version for setting every font-related parameter, String name, int style, int size, boolean underline.

For a full listing of the properties, please see Properties of JCFontChooserBar and Properties of JCFontChooserPane in Appendix A.

7.4 Methods

Because the initial choice of font parameters is made in the constructor, and subsequent changes are made by interacting with the GUI, there are no public methods of interest in JCFontChooserBar or JCFontChooserPane. Only the listener methods need concern you.

You listen for font changes by implementing the JCFontListener interface. Its two methods are fontChanging(), and fontChanged(). Both methods take a JCFontEvent

parameter. Use the first method to inspect and possibly veto the change in font, or in the underline state. Use the second to notify of these changes.

A `JCFontEvent` contains information about its source, the type of change that was made, old and new Font values, old and new underline values, and a Boolean `fontChanging` parameter that indicates whether this is a vetoable change or not.

Note: The "old" font and underline values are read-only.

7.5 Examples

In this example we'll add both a `JCFontChooserBar` and a `JCFontChooserPane` to the same panel. Normally, you place a `JCFontChooserPane` in its own dialog, but adding it to a `JPanel` as is done here doesn't change the way `JCFontChooserPane`'s properties are set. The code snippet shows how to instantiate both components and how to add a `JCFontListener` so you can respond to font-changed events. Since the listening object is `JCFontExample`, it needs to provide an implementation of `fontChanged`, the method that is declared in interface `JCFontListener`. The `JCFontChooserBar` is added to a `JToolbar`, as is shown first.

```
public class JCFontExample extends JPanel implements JCFontListener {

    ...
    bar = new JToolBar();
    Font font3 = new Font("Serif", Font.PLAIN, 12);
    font3 = JCFontChooser.setUnderline(font3, true);
    ...

    fontBar = new JCFontChooserBar(font3);
    bar.add(fontBar);
    fontBar.addJCFontListener(this);
    ...
    fontPane = new JCFontChooserPane(font3);
    fontPane.addJCFontListener(this);
    ...
}
```

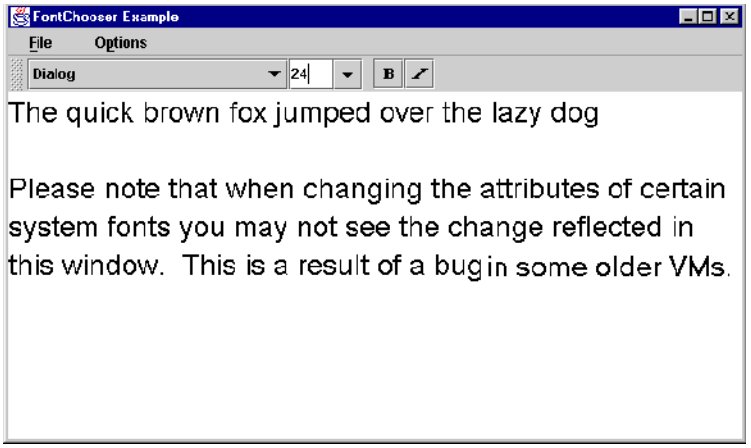


Figure 31 A bug notice in a JCFontChooserPane.

Please note that there is a problem with early Java 1.2 VMs that may require an extra block in your code. You may not see the changes in `JCFontChooserPane`'s preview area unless you add the following block of code:

```
//================== JCFontListener interface methods

/** Font is changing. Listeners can change
 */ the font and/or underline indication. */
public void fontChanging(JCFontEvent e) {
}

/** Font has been changed. */
public void fontChanged(JCFontEvent e) {
   System.out.println("Font changed to: "+ e.getFont());
    Object source = e.getSource();
     if (source instanceof JCFontChooserBar || source instanceof
            JCFontChooserPane) {
        Font font = e.getFont();
        Container parent = sampleText.getParent();
     sampleText.setFont(font);
        sampleText.repaint();
     }
}
```

8

HTML/Help Panes

8.1 Features of JCHTMLPane

`JCHTMLPane` is a subclass of Swing's `JEditorPane` which has been hard-coded to use the HTML Editor kit. HTML display can be as simple as passing the HTML code to `JCHTMLPane`'s `setText()` method. Alternatively, you can pass the text as a parameter to the constructor. This class also implements a Hyperlink listener to implement link traversal and different cursor images (hand cursor and wait cursor).

`JCHTMLPane` is an extension of `JEditorPane` that lets you:

- Construct an HTML pane, given a URL.
- Construct an HTML pane, given a pointer to HTML text.
- Change the icon for the cursor when it is over a link.
- Follow the reference in a link.
- Use an `MDIMenuBar` and `MDIToolBar` in addition to a `JMenuBar`.

Note that the HTML functionality in Swing's `JEditorPane` is based on `javax.swing.text.html.HTMLEditorKit`, which supports most, but not all, HTML 3.2 tags. The `APPLET` tag is not supported (March, 2000), and care should be taken when using `OBJECT`, `SCRIPT`, `FRAME`, and dynamic HTML.

8.2 Features of JCHelpPane

`JCHelpPane` is an extension of `JCHTMLPane` in that it contains two `JCHTMLPane`s under a header pane. A typical use places an HTML page containing a title in the header pane, a table of contents page on the left, and a contents page on the right. You can use it to provide your users with a lightweight browser for a HTML-based help facility.

- The lightweight browser becomes part of your application.
- Once links have been followed, forward and back buttons allow users to retrace their steps.

- JCHelpPane checks to see if a URL for the title pane was specified. If it wasn't, the title pane is not shown.

8.3 Classes

JCHTMLPane provides all the functionality necessary for an HTML-based pane, while JCHelpPane implements a lightweight two- or three-paned help system. Both of these are JavaBeans.

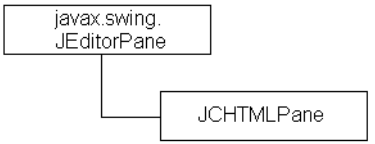


Figure 32 JCHTMLPane inherits from JEditorPane.

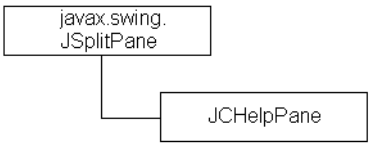


Figure 33 JCHelpPane inherits from JSplitPane.

8.4 Properties

JCHTMLPane's properties are the same as JEditorPane's. The class behaves like a JEditorPane with extra HTML awareness.

For a full listing of JCHTMLPane's properties, see Appendix A, Bean Properties Reference.

8.5 Constructors and Methods

8.5.1 Constructors

Constructors for JCHTMLPane

Along with the parameterless constructor for creating a blank pane, two others provide a handy way of instantiating a pane and providing it with HTML content in one operation.

Constructor	Description
`JCHTMLPane()`	Constructs a blank HTML pane.
`JCHTMLPane(URL url)`	Constructs an HTML pane with the specified URL.
`JCHTMLPane(String text)`	Constructs an HTML pane with the specified HTML text.

Constructors for JCHelpPane

`JCHelpPane`'s constructors let you specify source pages using URLs or `String`s. The latter may be advantageous if you generate some HTML-formatted text dynamically.

Constructor	Description
`JCHelpPane()`	Constructs a single blank "contents" pane.
`JCHelpPane(URL contents, URL view)`	Constructs, from the specified URLs, a help screen with a contents pane on the left and a view pane on the right. Note that `String`s may be used in place of URLs.
`JCHelpPane(URL contents, URL view, URL title)`	Constructs, from the URLs, a help screen with three frames: a header frame that spans the top of the window, and two side-by-side frames underneath. Note that `String`s may be used in place of URLs.

8.5.2 Methods

JCHTMLPane

The method of note is `setText()`, which is inherited from `JEditorPane`. Use it to pass text with embedded HTML tabs to the `JCHTMLPane`. An alternative way to pass the text is to via the pane's constructor, described above.

JCHelpPane

Although it is possible to construct a help browser using just the constructors for `JCHelpPane`, it has a number of methods are provided that may help your construction:

`getContentsPage()` `setContentsPage()`	Gets or sets the contents page. That is, get or set the HTML pane on the left hand side.
`getContentsPane()`	Returns the HTML pane on the left hand side.
`getTitlePage()` `setTitlePage()`	Gets or sets the title page. That is, gets or sets the HTML pane at the top.
`getViewPage()` `setViewPage()`	Gets or sets the view page. That is, gets or sets the HTML pane on the right hand side.
`getViewPane()`	Returns the HTML pane on the right hand side.
`isUseToolBar()` `setUseToolBar()`	Gets or sets the value of `useToolBar`. If the set method returns `true`, the component traverses up the tree to find its root pane container and adds a tool bar to it if one does not exist. If one exists, it adds the HTML navigation buttons to the existing toolbar. If two buttons exist in the tool bar named **Back** and **Forward**, then it will not add the buttons, but rather add listeners to those buttons.

8.6 Examples

JCHTMLPane

The following incomplete code fragment shows how you can compose your HTML text dynamically, then pass it to an instance of `JCHTMLPane`. The result is shown in the accompanying figure.

```
String myHTMLText = "<HTML><HEAD><TITLE>JCHTMLPane Demo</TITLE></HEAD>";
      myHTMLText += "<BODY><B>HTML (Bold) <P> <H1>JCHTMLPane
          understands basic HTML tags,</H1>";
      myHTMLText += "such as headings:";
      myHTMLText += "<H2 COLOR=red>A second level heading.</H2>";
      myHTMLText += "<H3 COLOR=blue><EM>And lists:</EM></H3><BR>";
      myHTMLText += "<OL><LI>Life is like a box of choco-lates";
      myHTMLText += "<LI>Judy, Judy, Judy";
      myHTMLText += "<LI>Play it again, Sam</OL>";
      myHTMLText += "<A HREF=\"http://www.quest.com\">
         And links to other Web pages</A>";
      myHTMLText += "<P>Tables too!<TABLE BORDER=10
          BORDERCOLOR=BLACK BGCOLOR=WHITE>";
      myHTMLText += "<tr><td>ROW ONE, First COLUMN cell</TD>
          <TD>ROW ONE, Second COLUMN Cell</TD>
          <TD>ROW ONE, Third COLUMN cell</TD></TR>";
```

```
        myHTMLText += "<tr><td>ROW TWO, First COLUMN cell</TD>
            <TD>ROW TWO, Second COLUMN Cell</TD>
            <TD>ROW TWO, Third COLUMN cell</TD></TR>";
        myHTMLText += "<tr><td>ROW THREE, First COLUMN cell</TD>
            <TD>ROW THREE, Second COLUMN Cell</TD>
            <TD>ROW THREE, Third COLUMN cell</TD></TR>";
        myHTMLText += "</TABLE>";
        myHTMLText += "</BODY></HTML>";

        JCHTMLPane pane = new JCHTMLPane(myHTMLText);
pane.setEditable(false);
        pane.setVisible(true);
        frame.getContentPane().add(pane, BorderLayout.SOUTH);

          frame.pack();
          frame.setVisible(true);
```

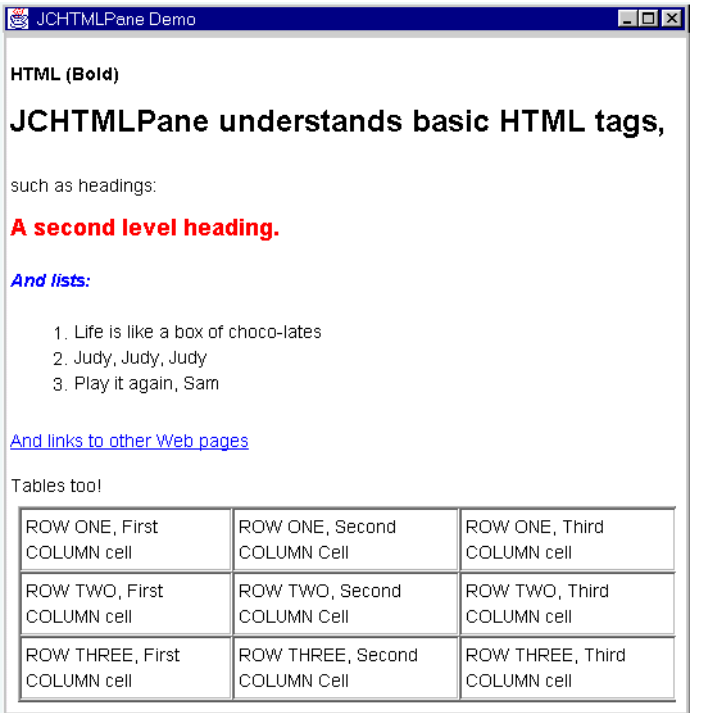


Figure 34 A JCHTMLPane whose contents are derived from HTML Strings in the class.

JCHelpPane

This example demonstrates instantiating `JCHelpPane`s with both Strings and URLs. If any of the URLs can't be found, the version of `JCHelpPane` that uses Strings is displayed.

```
import com.klg.jclass.util.swing.*;
import com.klg.jclass.util.value.*;
import javax.swing.*;
import java.awt.*;

/**
 * This example demonstrates the use of a JCHelpPaneExample
 */
public class HelpPaneExample {

// All the work is done in main()

public static void main(String args[]) {
    String contents = new String(
          "The contents pane of the JCHelpPane if URL isn't found.");
    String view = new String(
          "The view pane of the JCHelpPane if URL isn't found.");
    String title = new String("Header for the Help Pane.");
    JFrame frame = new JCExitFrame("Help Pane Example");
    JCHelpPane app = new JCHelpPane(contents, view, title);
    try {
        java.net.URL contentsFromURL = new
                          java.net.URL("http://....../toc_page.html");
        java.net.URL viewFromURL = new
                            java.net.URL("http://....../readme.html");
        java.net.URL titleFromURL = new
                         java.net.URL("http://....../jclasslogo.html");
        app = new JCHelpPane(contentsFromURL, viewFromURL,
                                                titleFromURL);
    }
    catch (java.net.MalformedURLException e) {
        System.out.println("Malformed URL");
    }

    app.setPreferredSize(new Dimension(640, 400));
    frame.getContentPane().add(app);
    frame.pack();
    frame.setSize(700, 450);
    frame.show();
}

}
```

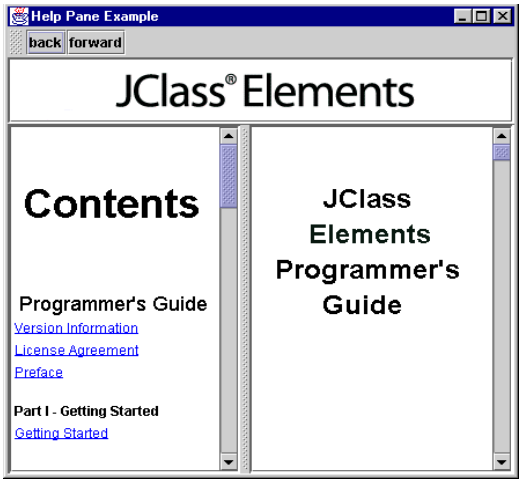


Figure 35 A JCHelpPane showing the HTML version of this manual.

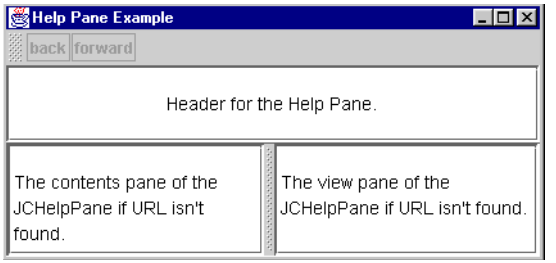


Figure 36 In this example, the alternate JCHelpPane when the URL can't be found.

9

Sortable Table

9.1 Features of JCMappingSort

Sorting can be accomplished by indexing the list of objects that are going to be ordered according to some comparison policy. It can be much more efficient to sort these indices instead of sorting the objects themselves. The idea is to form an array of indices. Initially, a[1] = 1, a[2] = 2, and so on, up to *n*, the length of the list. After sorting, the result might be a[1] = 9, a[2] = 3, ... a[n-1] = 1, ... a[n] = 7, where now the index in a[1] corresponds to the object that is the smallest element in the list according to the supplied comparison rule. The index in a[2] corresponds to the next smallest object, and so on. The list hasn't changed, but the array supplies a mechanism for traversing the list according to some ordering principle.

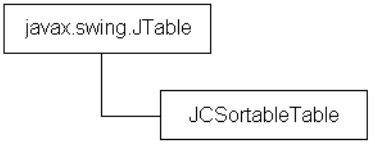
The foregoing paragraph shows you that if you want to use this type of mapping sort in your application, you'll need to supply an array of indices and a comparator to use with your list. In some cases, a comparator is already available. A number of common objects implement the `Comparable` interface in Java 2. You can compare any of these types without needing to supply an explicit comparator.

`JCMappingSort` provides a `sort()` method, which takes an implementation of `JCIntComparator` and an array of indices as parameters, and modifies the passed-in array based on the `compare()` method defined by your implementation of `JCIntComparator`.

9.2 Features of JCSortableTable

`JCSortableTable` uses a comparator and a configureable list of column indices, making this class useful for establishing a sort policy that specifies what should be done when two elements in the primary column have the same value. Elements in the primary column

that compare the same are arranged among themselves by sorting the secondary column. The process can be continued as necessary by including more columns in the list.



- Each column in a table may have an associated list of columns that are to be used as sort keys. Normally, the column itself is specified as the primary sort key.
- You can set whether a table is re-sorted automatically when its data changes.
- You can set or toggle the sorting order, permitting sorting from less to greater, or from greater to less.
- Sort `Dates`, `Objects` that implement `Comparable`, and wrapped primitive types.
- You can provide your own implementation of `JCRowComparator` to perform row comparisons in the sort algorithm. See the example at the end of this chapter for details.

9.3 Classes and Interfaces

JCMappingSort

`CollectionIntComparator`	Implements `JCIntComparator` to compare two lists.
`JCIntComparator`	An interface that declares a `compare` method taking two indices as parameters. The `compare` method must be able to compare the `Objects` corresponding to the indices.
`JCMappingSort`	Contains a static `sort()` method that is passed a `JCIntComparator` and an array of indices. The array containing the indices is sorted rather than sorting the list objects to which they refer.

You'll find these classes and interfaces in `com.klg.jclass.util`.

JCSortableTable

JCSortableTable	JCSortableTable is a subclass of JTable that internally wraps any TableModel it is given with a JCRowSortTableModel and provides a Comparator that has a configureable list of the column indexes that it uses for sorting. Clicking on a column header invokes the sorting behavior tied to that column, clicking again reverses the direction of the sort.
JCRowComparator	This interface is to be used with JCRowSortModel. It sorts rows using a specified ordered list of columns as the sort keys. By default, it sorts on the first column.
JCRowSortModel	An interface that defines methods for sorting rows by specifying which columns are to be used as keys.

Using your own comparator with JCSortableTable

If you wish to provide your own comparator for a JCSortableTable, follow these steps:

1. Create a javax.swing.table.TableModel.
2. Create a com.klg.jclass.util.swing.DefaultRowSortTableModel, giving it the TableModel.
3. Set your comparator to this instance of a DefaultRowSortTableModel.
4. Set the DefaultRowSortTableModel on your JCSortableTable.

Note that the data model you set in step 2 should be a JCRowSortTableModel. If it is not, JCSortableTable will wrap the data model you provide with a JCRowSortTableModel.

9.4 Constructors and Methods

Constructors for JCSortableTable

`JCSortableTable()`	`JCSortableTable` is a subclass of `JTable` that internally wraps any `TableModel` it is given with a `JCRowSortTableModel` and provides a `Comparator` that has a configureable list of the column indexes that it uses for sorting.

`JCSortableTable(`
`  int numRows,`
`  int numColumns)`
	Constructor that specifies the number of rows and columns in the table.

`JCSortableTable(`
`  Object[][] rowData,`
`  Object[] columnNames)`
	The constructor for a data source composed of an array of `Object`s.

`JCSortableTable(` `  TableModel dm)`	Constructor that accepts a `TableModel`.

`JCSortableTable(`
`  TableModel dm,`
`  ColumnModel cm)`
	Constructor that accepts both a `ColumnModel` and a `TableModel`.

`JCSortableTable(`
`  TableModel dm,`
`  ColumnModel cm,`
`  ListSelectionModel sm)`
	Constructor that accepts a `ColumnModel`, a `TableModel`, and a `ListSelectionModel`.

`JCSortableTable(`
`  Vector rowData,`
`  Vector columnNames)`
	The constructor for a `Vector` data source.

The core of the sorting mechanism is based on providing the `sort()` method with a list of indices specifying an ordered list of columns on which the sort is to be based:

```
public static void sort(JCIntComparator comparator, int indices[])

public static void sort(JCIntComparator comparator, int indices[],
                        int start, int end)
```

Both methods require a `JCIntComparator` and an array of indices. The second method includes two additional parameters that are useful in many sorting algorithms.

Methods

In addition to the host of methods it inherits from `JTable`, `JCSortableTable` adds many of its own:

`createDefaultColumnsFromModel()`	Overridden from the superclass to allow auto-creation of our own column model.
`getAutoSort()`	Returns whether the data is automatically sorted when it changes according the current comparator.
`getCellEditor()`	Takes parameter `int row, int column` to get the cell editor for that row and column.
`getCellRenderer()`	Takes parameter `int row, int column` to get the cell editor for that row and column.
`getKeyColumns()`	Takes parameter `int column` to return the key columns used to sort the table model when clicking on the specified column.
`getUnsortedRow()`	Takes parameter `int sortedRow` to return the unsorted row index of specified sorted row.
`setAutoSort()`	Takes parameter `boolean autoSort` to specify whether the data should be automatically sorted when it changes.
`setKeyColumns()`	Takes parameters `int column, int[] keyColumns` to set the key columns used to sort the table model when clicking on specified column.
`setModel()`	Takes parameter `javax.swing.table.TableModel newModel` to set the data model for this table to `newModel` and registers with for listener notifications from the new data model.
`setTableHeader()`	Takes parameter `javax.swing.table.JTableHeader newHeader` to overwrite the default implementation and add a `MouseListener` to the new table header.
`sort()`	Takes parameter `int column` to sort rows using the quicksort algorithm.
`tableChanged(e)`	Uses parameter `avax.swing.event.TableModelEvent e` to pass information about the event. Overrides super class method to check for a change in sorting.
`unsort()`	Restores the unsorted order.

9.5 Cell Renderers for JCSortableTable

Normally, you do not need to be concerned with the details of how table cells are rendered because renderers for most common cases have already been supplied. On the other hand, you may wish to use a custom renderer of your own design. While it is possible to use `setDefaultRenderer()` to set a cell renderer for a `JTable`, the method is not available for use with `JCSortableTable`. Instead, JClass uses its own powerful cell editor/renderer mechanism. This allows all JClass products to manage collections of `JCCellRenderer` types uniformly instead of having to manage the renderer types separately. To set your own cell renderer, use JClass Cell's `EditorRendererRegistry`, and implement one of the renderer interfaces. Please see the *com.klg.jclass.cell API* for details.

9.6 Examples

JCMappingSort example

`JCMappingSort` cannot be instantiated by calling its constructor. Instead, it has two static methods of the form:

- `public static void sort(JCIntComparator compartor, int indices[]);`
- `public static void sort(JCIntComparator compartor, int indices[], int start, int end);`

The purpose of these two methods is to sort a mapping of indices instead of an array of objects. This is particularly useful when dealing with a `Collection`, or some form of data model where you reference a data element with an index. Your implementation of the `JCIntComparator` interface provides the implementation details for the objects you are sorting.

`JCIntComparator` should look like this:

```
public interface JCIntComparator {
    public int compare(int index1, int index2);
}
```

The `CollectionIntComparator` is a specific implementation of `JCIntComparator` that can compare `Collections`. Sample code looks like this:

```
public class CollectionComparator implements JCIntComparator {

protected Collection collection;
protected Comparator comparator;

public CollectionComparator(Collection collection, Comparator
                                comparator) {
     this.collection = collection;
     this.comparator = comparator;
 }

public CollectionComparator(Collection collection) {
     this(collection, null);
 }

public int compare(int i1, int i2) {
     Object a1 = collection.get(i1);
     Object a2 = collection.get(i2);

     if (comparator != null) {
         // use comparator if provided
         return comparator.compare(a1, a2);
     }
     else if (a1 instanceof Comparable) {
         // items are comparable so get them to compare themselves
         return ((Comparable) a1).compare(a2);
     }
     else {
        // We have no comparator and the objects are not
                comparable
        throw new IllegalArgumentException("Objects are not
Comparable; please provide a Comparator with the constructor:
CollectionComparator(Collection collection, Comparator
                     comparator)");
     }
 }

 }
```

JCSortableTable examples

One use of `JCSortableTable` is given in *examples/elements/SortTable*.

A full example based on `SortTable.java` follows. It demonstrates sorting on columnar data containing `String`s, and two types of primitives: Boolean values and integers. The example provides its own implementation of `JCRowComparator` to perform a comparison between two rows in the table.

```
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

```
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Comparator;
import java.text.*;

import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.UIManager;
import javax.swing.table.AbstractTableModel;
import javax.swing.BoxLayout;
import javax.swing.table.DefaultTableCellRenderer;

import com.klg.jclass.util.swing.JCExitFrame;
import com.klg.jclass.util.swing.JCSortableTable;
import com.klg.jclass.util.swing.DefaultRowSortTableModel;
import com.klg.jclass.util.swing.DefaultRowComparator;
import com.klg.jclass.util.swing.JCRowSortModel;
import com.klg.jclass.util.swing.JCComparableRow;

/**
 * Sorting is allowed on these columns:
 * "First Name", "Last Name", "Position", "Favorite Number", and
 * "Vegetarian"
 */
public class SortDateJCSortableTable extends JPanel implements
    ActionListener {

    protected final static String[] names =
     {"First Name", "Last Name", "Position",
      "Favorite Number", "Vegetarian",
      "Calendar","GregorianCalendar"};

protected final static Object[][] data = {

        {"Diana", "Zukerman", "Research Officer",
        new Integer(1), new Boolean(false),"",""},
         {"Adam", "Petersen", "Consultant",
        new Integer(2), new Boolean(false),"",""},
        {"Mary", "Binfield", "Research Associate",
        new Integer(5), new Boolean(false),"",""},
        {"Michael", "Rizzo", "Research Fellow",
        new Integer(2), new Boolean(true),"",""},
        {"Ahmad", "Baldi", "Consultant",
        new Integer(3), new Boolean(false),"",""},
        {"Ian", "Clemente", "Research Fellow",
        new Integer(7), new Boolean(false),"",""},
         {"David", "Rubinstein", "Consultant",
        new Integer(4), new Boolean(false),"",""},
    };

    protected JButton buttonUnsort = null;
    protected JCSortableTable sortableTable = null;

    /** Indicates that the first object is less than the second object.
     */
```

```
   public static final int LESS_THAN = -1;
   /** Indicates that the first object is equal to the second object. */
   public static final int EQUAL = 0;
   /** Indicates that the first object is greater than the second object. */
   public static final int GREATER_THAN = 1;

public SortDateJCSortableTable() {
    // Set a simple BoxLayout manager
    setLayout(new BoxLayout(this,BoxLayout.X_AXIS));

    //set up the calender values to be tested

    for (int r =0 ; r < data[0].length ; r++){
       Calendar c = Calendar.getInstance();
       c.set(1998+r, r, 1);

       GregorianCalendar gc = (GregorianCalendar)c;

       data[r][5] = c;
       data[r][6] = gc;
       }
    // Create and add the table
    sortableTable = createTable();
    add(new JScrollPane(sortableTable));

    // Create and add an Unsort button for the table
    buttonUnsort = new JButton("Unsort");
    buttonUnsort.addActionListener(this);
    add(buttonUnsort);
}

public static JCSortableTable createTable() {
   EditableTableModel model = new EditableTableModel();
    JCSortableTable table = new JCSortableTable();

// JCSortTable will do this anyway,
    // but this way we have a member handle to it.
    DefaultRowSortTableModel mRSmodel =
     new DefaultRowSortTableModel(model);
    mRSmodel.setComparator(new MyComparator());

    //set model and cast it down to the "DefaultRowSortTableModel"
    table.setModel(mRSmodel);

   // We use the last name if the first name is the same.
    int sort0[] = {0, 1};
    table.setKeyColumns(0, sort0);

   // We use the first name if the last name is the same.
    int sort1[] = {1, 0};
    table.setKeyColumns(1, sort1);

   // We use person's name if the department is the same.
    int sort2[] = {2, 0, 1};
```

```
        table.setKeyColumns(2, sort2);

        //set the non primitive renderers, no editor defined for this
              example
        table.getColumn("GregorianCalendar").setCellRenderer(
           new CGFCalendarCellRenderer());
        table.getColumn("Calendar").setCellRenderer(
           new CGFCalendarCellRenderer());

        return table;
    }

    public static void main(String args[]) {
       JCExitFrame frame = new JCExitFrame(
           "SortDateJCSortableTable Example");
       SortDateJCSortableTable app =
           new SortDateJCSortableTable();

       if (args.length > 0) {
           if (args[0].equals("windows")) {
               try {
                UIManager.setLookAndFeel(
                   "com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
            } catch (Exception e) {}
           }
       }

       frame.getContentPane().add(app);
       frame.setBounds(50, 50, 650, 350);
       frame.show();
    }


    //===== ActionListener interface method =========

    public void actionPerformed(ActionEvent e) {
       if (e.getSource() instanceof JButton) {
        sortableTable.unsort();
        }
    }

    public static class CGFCalendarCellRenderer
    extends DefaultTableCellRenderer
    {
    protected java.text.DateFormat date_formatter =
        java.text.DateFormat.getDateInstance();

    public void setValue(Object o)
    {
       String str = null;
       if (o instanceof String) {
           str = (String)o;
       }
       else if (o instanceof Calendar) {
           str = date_formatter.format(((Calendar)o).getTime());
       }
```

```
    else {
        str = o.toString();
    }
  super.setValue(o == null ? "" : str);
}
}
private static class EditableTableModel extends AbstractTableModel {

    EditableTableModel() {
    super();
    }

    public int getColumnCount() {
        return names.length;
    }

    public int getRowCount() {
        return data.length;
    }

    public Object getValueAt(int row, int col) {
        return data[row][col];
    }

    public String getColumnName(int column) {
        return names[column];
    }

    public Class getColumnClass(int col) {
        return getValueAt(0,col).getClass();
    }

   // Disallow edits on dates, and on favorite number
    public boolean isCellEditable(int row, int col) {
        return col != 3 && col != 5 && col != 6;
    }

    public void setValueAt(Object aValue, int row, int column) {
        data[row][column] = aValue;
    }
} // EditableTableModel

static class MyComparator extends DefaultRowComparator {

   public MyComparator(){
      super(JCRowSortModel.FORWARD);
   }
   public int compare(JCComparableRow row1, JCComparableRow row2) {
      int[] kc = super.getKeyColumns();

   for (int i = 0; i < kc.length; i++) {
       Object o1 = row1.getValueAt(kc[i]);
       Object o2 = row2.getValueAt(kc[i]);

      if(o1 instanceof Calendar && o2 instanceof Calendar) {
           Calendar c1 = (Calendar)o1;
```

```
            Calendar c2 = (Calendar)o2;

              if(c1.equals(c2)) {
                return EQUAL;
            }
            else if(c1.before(c2)) {
                return LESS_THAN;
            }
            else {
                return GREATER_THAN;
            }
          }
        }
        return super.compare(row1, row2);

      }

}//MyComparator

} // SortDateJCSortableTable
```

10

Multiple Document Frame

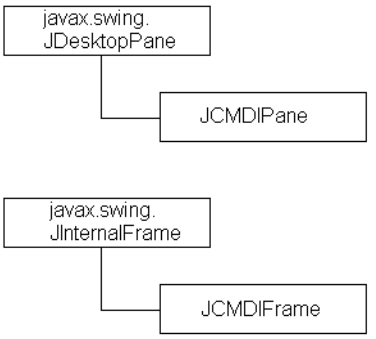
10.1 Features of JCMDIPane and JCMDIFrame

The *Multiple Document Interface* (MDI) model is a common way of organizing and presenting information in windows-style environments. The MDI model makes it possible to arrange multiple child windows inside a parent window, such as multiple files in a text editor, or multiple charts in one frame. In effect, the parent window becomes the desktop within which the children operate. Before Swing, there was no way of building MDI applications using the Abstract Windowing Toolkit (AWT).

If you were limited to using raw Swing components, you would likely build your primary GUI application within a `JFrame`. The container used to hold a multiple-document interface is a `JDesktopPane`, which you would put into the content pane of your `JFrame`. Finally, you would add `JInternalFrame`s as needed for your document windows.

The JClass Elements components `JCMDIPane` and `JCMDIFrame` augment the functionality of `JDesktopPane` and `JInternalFrame` respectively. Simply replace `JDesktopPane` with `JCMDIPane`, and `JInternalFrame` with `JCMDIFrame`, and your job is almost complete! The only other thing you need to do is to use the `setMDIMenuBar()` method to set individual

menu bars on each of your internal frames. These menu bars will replace the default menu bar that you set on `JCMDIPane`.



JClass Elements's multiple document `JCMDIPane` interface component extends Swing's `JDesktopPane` view to provide the following standard MDI features:

- True maximization. Instead of keeping two menu bars when an internal pane is maximized, `JCMDIPane` optimizes screen real estate by placing menus on the desktop's menu bar. All necessary functionality is preserved.
- Automatically adds a localized **Windows** menu containing two sections.
- The upper section of the **Windows** menu allows you to select from one of three window tiling algorithms: **Cascade**, **Tile Horizontally**, or **Tile Vertically**.
- The upper section of the **Windows** menu also allows you to *Minimize/Maximize* the selected frame, or to (re)*Arrange Icons* of the minimized frames.
- The lower section of the **Windows** menu provides a list of the titles of the internal frames, giving the user the ability to switch focus to any internal frame by selecting its name from the menu.
- Adds unmaximize/close icons to the far right of the menu bar when one of the frames is maximized.

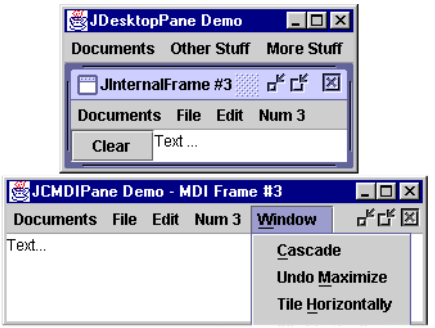


Figure 37 The differences between a JCMDIPane (lower image) and a JInternalFrame (upper image).

Public classes:

`JCMDIPane` – subclass of `JDesktopPane`
`JCMDIFrame` – subclass of `JInternalFrame`

`JCMDIPane` is API compatible with `JDesktopPane`, but the behavior differs in that it automatically generates a **Windows** menu on the first toolbar it finds in its ancestral hierarchy. This **Windows** menu has arrangement options **Cascade**, **Tile Horizontally**, **Tile Vertically**, and **Arrange Icons**, and a selectable list of all the existing internal frames. When frames are maximized the first child of an internal frame's content pane is reparented to a panel that is mapped on top of all the frames so that the maximized frame makes maximal use of the existing window real estate.

Default dragging and resizing behavior is done speedily by drawing wire frames.

JCMDIFrame, when calling getContentPane, returns an additional child that is the single child of the true content pane. This is done for easy reparenting purposes as well as for support routines that aid in the manipulation of this child.

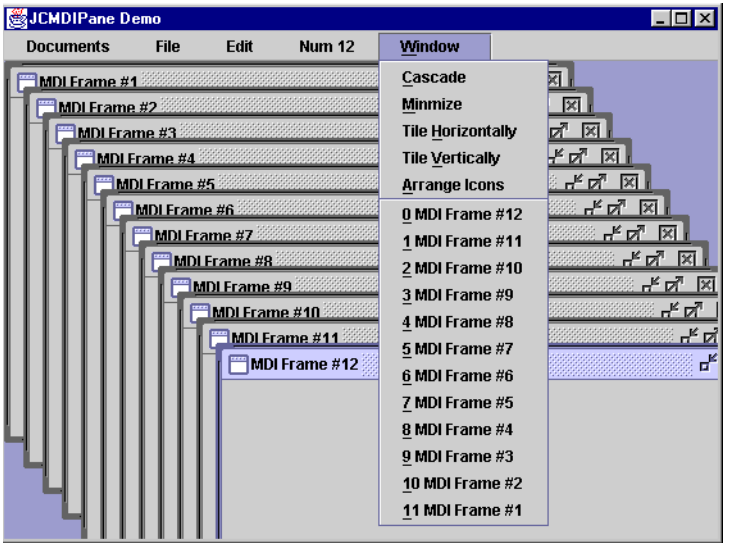


Figure 38 A JFrame containing a JCMDIPane and multiple JCMDIFrames.

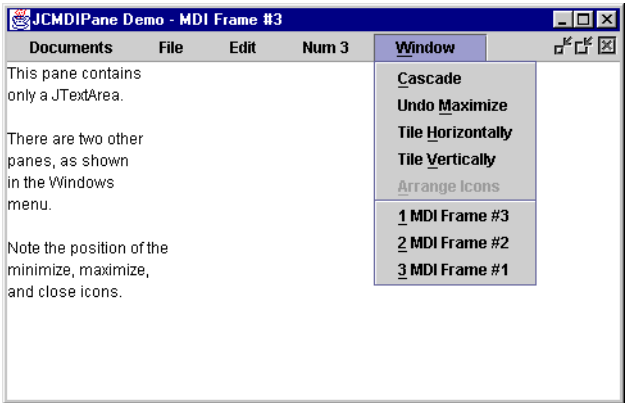


Figure 39 A maximized internal frame—the iconify and close buttons have moved to the menu bar.

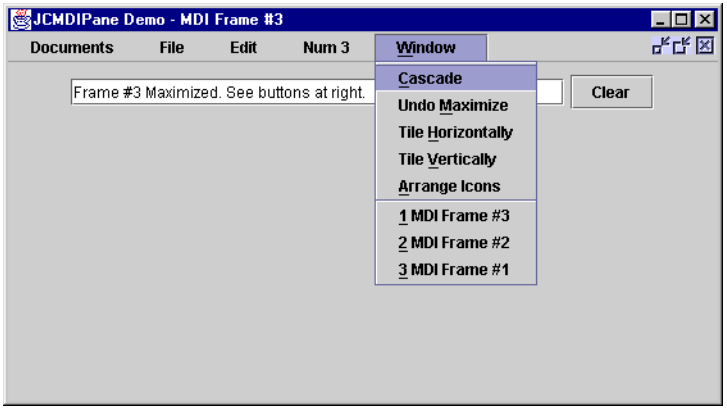


Figure 40 One frame is maximized. The Iconify, Maximize, and Close buttons appear on the menu bar.

10.2 Properties

For a full listing of these components' properties, see Properties of JCMDIFrame and Properties of JCMDIPane, in Appendix A.

`JCMDIFrame` has the same properties as `JInternalFrame`.

Along with the properties inherited from `JDesktopPane`, there are two additional properties in `JCMDIPane`: `frameManipulationStyle` and `considerIconsWhenTiling`. Setting `frameManipulationStyle` allows you to control how a frame is painted when it is dragged within the desktop. The default style, `JCMDIPane.DEFAULT`, causes `JCMDIPane` to repaint the entire frame when dragging. The second style, called wireframe, causes `JCMDIPane` to repaint a rectangle that matches the size of the frame. The wireframe dragging style is used when `frameManipulationStyle` is set to `JCMDIPane.WIREFRAME`. The `considerIconsWhenTiling` property controls the way that windows are tiled (`false` means that windows will be tiled using the entire desktop area; `true` means that windows will not be tiled over any icons that appear on the desktop).

10.3 Methods

Methods of JCMDIFrame

`getMDIMenuBar()` `setMDIMenuBar()`	Gets or sets the menu bar associated with this frame. If the parent of this frame is a `JCMDIPane`, then this menu bar will become the containing frame's menu bar when this frame becomes active. `setMDIMenuBar()` takes a `JMenuBar` as a parameter.
`getMDIToolBar()` `setMDIToolBar()`	Gets or sets the toolbar associated with this frame. If the parent of this frame is a `JCMDIPane`, then this toolbar will become the containing frame's toolbar when this frame becomes active. `setMDIToolBar()` takes a `JToolBar` as a parameter.
`getContentPane()`	Overrides `getContentPane()` to provide a container one level removed so that the frame can be maximized by reparenting its children to a different parent.

Methods of JCMDIPane

`getAllNonIconifiedFrames()` `getAllIconifiedFrames()`	Returns all non-iconified/iconified `JCMDIFrame`s currently displayed in the desktop.

`getDragMode()`
`setDragMode()`
	Sets the dragging style of the frames on the desktop. Because `JCMDIPane` uses its own Desktop Manager, it does not use the dragging implementation of `JDesktopPane`; instead, it uses the dragging implementation of `frameManipulationStyle`. However, setting this property actually sets the `frameManipulationStyle` to the equivalent style. Valid styles are:
	`OUTLINE_DRAG_MODE` – corresponds to `WIREFRAME`
	`LIVE_DRAG_MODE` – corresponds to `DEFAULT`

`getFrameManipulationStyle()` `setFrameManipulationStyle()`	Sets the frame manipulation style. Valid styles are: `WIREFRAME` – drags and resizes as a wire frame, `DEFAULT` – default style specified by the PLAF you are using. The default style causes `JCMDIPane` to paint the entire frame when dragging it.
`setInitialLayout()`	Allows the layout of the `MDIFrame` windows to be set before the `MDIPane` window has been displayed. This has no effect after the `MDIPane` has been displayed for the first time.
`isMaximized()` `setMaximized()`	Methods to manage the maximized pane.

`getMDIMenuBar()` `setMDIMenuBar()`	The parameterless version of `getMDIMenuBar()` returns the menu bar used if we have no internal frames, whereas a `JInternalFrame` parameter is used to return the menu bar used for the specified frame. If the frame is a `JCMDIFrame` with a non-null `MDIMenuBar`, then this is returned. Otherwise, the `MDIMenuBar` for the this pane is returned. `setMDIMenuBar()` with a `JMenuBar` parameter sets the toolbar to use if there are no internal frames.
`getMDIToolBar()` `setMDIToolBar()`	Returns the toolbar used for the specified frame. If the frame is a `JCMDIFrame` with a non-null `MDIToolBar`, then this is returned; otherwise, the `MDIToolBar` for the this pane is returned. `getMDIToolBar()` takes a `JInternalFrame` as a parameter. Sets the toolbar to use if there are no internal frames. `setMDIToolBar()` takes a `JToolBar` as a parameter.
`getNonSelectedIcon()` `setNonSelectedIcon()`	Gets or sets the non-selected icon, which appears before the non-selected items in the menu. The default is an empty icon that acts as a placeholder so menu items will be aligned properly. Setting both these icons to null restores the previous behavior.
`getSelectedIcon()` `setSelectedIcon()`	Gets or sets the icon which is to appear beside the selected window item in the Windows menu. The default icon is a check mark.
`getPreferredSize()`	If this pane has an ancestor that's a scroll pane or it has no children, then it returns the default preferred size. If it has children and no scroll pane for an ancestor, then it returns a size big enough to show all its children in their current locations.
`getTopFrame()`	Returns the topmost frame.
`setInitialLayout()`	Allows the layout of the MDIFrame windows to be set before the MDIPane window has been displayed. This has no effect after the MDIPane has been displayed for the first time. If not called, the initial layout is unpredictable. Pass one of these constants to the method: `JCMDIPane.TILE_HORIZONTAL`, `JCMDIPane.TILE_VERTICAL`, `JCMDIPane.CASCADE`.

`activateFrame()`	Makes this frame the active frame.
`arrangeIcons()`	Arranges "iconified" panes along the bottom.
`cascadeWindows()`	Arranges non-iconified panes in cascade form.
`closeFrame()`	Closes or deactivates a frame.
`deactivateFrame()`	
`maximize()`	Maximizes a pane, filling the host frame.
`tileWindowsHorizontally()`	Tiles the frames in the specified direction
`tileWindowsVertically()`	
`unmaximize()`	Returns the frame to its former size.
	Note: These methods were protected in version 4.0 and have been made public in version 4.0.1.

There are a number of protected methods available to application programmers who wish to subclass a `JCMDIPane`. Consult the API for a list of these methods.

10.4 Examples

This code snippet highlights the few things that need to be done to convert your MDI application based on `JInternalFrame` into one based on `JCMDIFrame`.

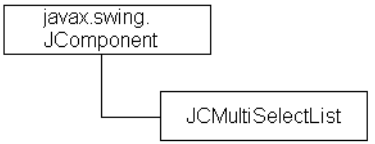
```
import com.klg.jclass.swing.JCMDIPane;
import com.klg.jclass.swing.JCMDIFrame;

/**
 * The class extends JClass Elements' JCExitFrame so you don't have to
 * write repetitive window closing code.
 */
public class MDIInternalFrameDemo extends JCExitFrame implements
                                            ActionListener {
/**
 * The internal frames reside inside a JCMDIPane
 */
public MDIInternalFrameDemo() {
       super("JCMDIPane Demo");
    ...
    desktop = new JCMDIPane(); // a custom layered pane
    createFrame(); // Create first window
    ...
}
/**
 * Each frame can have its own menu bar, whose elements are
 * defined by you. A "Window" menu is added automatically.
 */
protected void createFrame() {
         JCMDIFrame iframe = new JCMDIFrame(
                                       "MDI Frame #" + (++MDIFrameCount),
                                       true, //resizable
                                       true, //closable
                                       true, //maximizable
                                       true);//iconifiable
...
/**
 * Use this method to set the menu bar on the frame, even though
 * it appears on the desktop rather than on the individual frame.
 */
iframe.setMDIMenuBar(mbar);
...
```

11

Multi-Select List

11.1 Features of JCMultiSelectList

`JCMultiSelectList` matches the API for `JList` except that two lists instead of one appear in the component's GUI. There are four buttons between the two lists that move items back and forth. The left-hand list contains non-selected items and the right-hand list contains the selected items. In the context of a `JCMultiSelectList`, if an item is marked as selected, it means more than simply being highlighted. Besides providing a visual division of list items into the two columns, selected and non-selected, there are numerous methods for dealing with the values and indices of a set of selected values.



- `JCMultiSelectList` provides a visual component that clearly distinguishes items chosen from a given list by removing them from the original list and placing them in another container. See the next figure for details.
- You can create a `JCMultiSelectList` using one of its five constructors, four of which correspond to the constructors of a `JList`. The remaining constructor has an empty `ListModel`, but has a parameter for setting the horizontal gap between the two lists.
- As with a `JList`, you can specify content using the `ListModel` interface, or you can supply content using `Object`s or `Vector`s.
- You are able to modify content in various ways depending on which objects you used to populate the main list.
- End users may perform single or multiple, contiguous or non-contiguous selections of list items.
- The `ListSelectionModel` generates a `ListSelectionEvent` to allow you to process user interactions.

Four buttons control the movement of items in one list to the other. They are shown in the next figure. The top button moves selected items from the left-hand list to the right-hand list. The second from the top moves all items out of the left-hand list to the right-hand list. The bottom two buttons perform the analogous operation, but in the other direction.

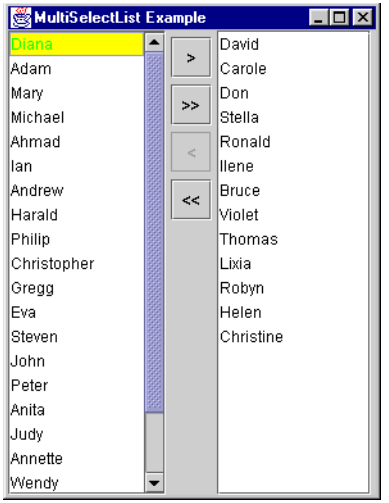


Figure 41 GUI for JCMultiSelectList.

You set a `ListModel` on the component, or you can use the default model that is provided. In the latter case, you simply add `Objects` to the existing component. The component uses `getSelected()` to determine which items should appear in the right-hand list. Only the non-selected items show on the left.

11.2 Properties

A selection of `JCMultiSelectList`'s properties are shown in the following table. Please see Properties of JCMultiSelectList and Bean Properties Reference in Appendix A for a complete list.

Property	Description
`model`	Gets or sets the model associated with the list.
`prototypeCellValue`	Sets the prototypical cell value, a cell used for the calculation of cell widths, rather than forcing the calculation to inspect every item in the list.

`toolTipText`	Gets or sets the text that appears in the tool tip.

11.3 Constructors and Methods

Constructors

There are constructors for `ListModel`, array, and `Vector` types of data models.

Methods

`JCMultiSelectList` subclasses from `JComponent`, giving it a host of inherited methods. It overrides some, like `addListSelectionListener`, to provide specific functionality. The table shows a few frequently used methods. Please refer to the API for a full list.

Method Name	Description
`addListSelectionListener()`	Adds a listener to the `ListSelectionListener`'s list. Its parameters are a `javax.swing.event` and a `ListSelectionListener`.
`addSelectionInterval()`	Adds the specified interval to the current selection. It takes two `int` parameters that specify the beginning and ending positions of the interval.
`clearSelection()`	Clears the selection.
`deselectAll()`	Moves all items from the right list to the left list.
`deselectItem()`	Moves the items selected in the right list to the left list.
`fireSelectionValueChanged()`	Forwards the given notification event to all registered listeners. It takes two `int` parameters that specify the begin and end positions of the interval, and a third `boolean` parameter that specifies whether this is one of a rapidly occurring series of events. Parameters are `int firstIndex, int lastIndex, boolean isAdjusting` See `javax.swing.event.ListSelectionEvent`.
`getAnchorSelectionIndex()`	Returns the first index argument from the most recent interval selection.
`getSelectedIndex()`	Returns the index of the first selected cell.
`getSelectedIndices()`	Returns the array of indices of selected items.
`getSelectedValues()`	Returns an array of the selected cell values.

`setModel()`	Sets the list's data model. Its single parameter is a `javax.swing.ListModel`.

11.4 Examples

See `examples.elements.MultiSelectList` for a full listing of this example.

One of `JCMultiSelectList`'s constructors takes an array of list items as its parameter. Call this array `data` and define it as follows:

```
static String[] data = {"Tom", "Dick", "Harry"};
```

Create a `JCMultiSelectList` and give it the data:

```
JCMultiSelectList list = new JCMultiSelectList(data);
```

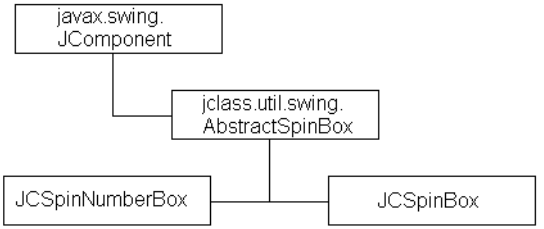
You process events generated by the list by implementing the `valueChanged()` method of the `ListSelectionListener` interface.

12

Spin Boxes

12.1 Features of JCSpinBox and JCSpinNumberBox

Swing provides checkboxes and radio buttons, but no spin boxes. The JClass spin boxes fill the need for components that let the user select a number or an `Object` by clicking on up or down arrows. You can use a `JCSpinBox` to spin through a list of `Objects` (so long as the required editors and renderers exist, or have provided them), or use a `JCSpinNumberBox`, which can display any numeric object.



- `JCSpinBox` looks very much like a `JComboBox` except that it has no dropdown. It takes a list of objects and presents these values in a spin box. You use the up and down arrows to cycle through the list.
- Use `JCSpinNumberBox` for incrementing and decrementing objects of type `java.lang.Number`. You can select numbers of type `Byte`, `Short`, `Integer`, `Long`, or `Float`, and you can set maximum and minimum values for the spin operation.
- Both components follow Swing's MVC paradigm. A `JCSpinBoxModel` interface is used to manage the spin box's data.
- `JCSpinBox` has four constructors for the various ways in which you can supply data; that is, as `Objects`, `Vectors`, or via a `JCSpinBoxModel`. A fourth parameterless constructor is available. It uses an empty `DefaultSpinBoxModel` as a placeholder for data that will be provided later.
- Contents of a spin box may be modified via a `JCSpinBoxEditor` interface.

- A KeySelectionManager interface defines a method for associating a keystroke to an item in the spin box.
- JCSpinBoxModel methods are inherited from javax.swing.ListModel and javax.swing.ComboBoxModel. These are addListDataListener, getElementAt, getSize, removeListDataListener, getSelectedItem, and setSelectedItem.
- The listener is the addValueListener, and the event is JCValueEvent.

12.2 Classes and Interfaces

Interfaces

JCSpinBoxEditor	The editor component used for JCSpinBox components.
JCSpinBoxModel	A data model for JCSpinBox modeled after javax.swing.ComboBoxModel. JCSpinBoxModel is a ListDataModel with a selected item. This selected item is in the model since it is not always in the item list. It inherits its methods from javax.swing.ComboBoxModel and javax.swing.ListModel.
JCSpinBox MutableModel	Extends JCSpinBoxModel to define models that are changeable. It declares methods for adding, inserting, and removing elements.

Helper Classes

AbstractSpinBox	The super class for JCSpinBox and JCSpinNumberBox. The class is abstract because it does not define spinUp(), spinDown(), and checkArrowButtons(), but it does provide the common functionality for JCSpinBox and JCSpinNumberBox.
JCValueEvent	The event object has methods getSource, getOldValue, and getNewValue, allowing you to find out which spin box posted the event, and its old and new values.

12.3 Properties

JCSpinBox properties

These properties contain all of the functionality of JCSpinBox. In keeping with Swing's MVC design paradigm, the JCSpinBoxModel interface contains a data model for JCSpinBox modeled after javax.swing.ComboBoxModel. JCSpinBoxModel is a

ListDataModel with a selected item. This selected item is in the model since it is not always in the item list.

Property Name	Description
actionCommand	Sets or returns the action command that is included in the event sent to action listeners.
continuousScroll	Determines how selection is handled when the mouse button is held down on a spin arrow button. If continuousScroll is true, the component scrolls continuously through the items in the scroll box until the mouse button is released. If continuousScroll is false, a separate mouse click is required to select the next item in the scroll box.
getItemAt	Returns the list item at the specified index.
getItemCount	Returns the number of items in the list.
model	Sets or returns the data model currently used by the JCSpinBox.
renderer	Sets or returns the renderer used to display the selected item in the JCSpinBox field.
selectedIndex	Returns the index of the currently selected item in the list, or selects the item at the position marked by the index.
selectedItem	Returns the currently selected item, or sets the selected item in the JCSpinBox by specifying the object in the list.
isEditable	Returns true if the JCSpinBox is editable.

JCSpinNumberBox properties

These properties let you specify the operation, that is, whether the numbers in the spin box are whole numbers or floating point numbers. Additionally, you can set the spin increment and bounds.

For a complete list of properties, please see Properties of JCSpinBox and Properties of JCSpinNumberBox in Appendix A.

Property Name	Description
continuousScroll	Determines how selection is handled when the mouse button is held down on a spin arrow button. If continuousScroll is true, the component scrolls continuously through the items in the scroll box until the mouse button is released. If continuousScroll is false, a separate mouse click is required to select the next item in the scroll box.

Property Name	Description
maximumValue	Returns or sets the maximum value. The default is Long.MAX_VALUE
minimumValue	Returns or sets the minimum value. The default is Long.MIN_VALUE.
numberFormat	The NumberFormat object used by the spinner to parse and format numbers
operation	Takes a JCSpinNumberBox.INTEGER and sets the operation.
spinStep	The spin increment. The default is 1.
value	The current value of the spinner.
valueRange	Convenience method to set maximum and minimum values together. Defaults are Long.MIN_VALUE, Long.MAX_VALUE.

12.4 Constructors and Methods

Constructors

JCSpinNumberBox()	Use this component when you want to let your users increment or decrement a object of type java.lang.Number. Long.MIN_VALUE, Long.MAX_VALUE, and floating point numbers outside the range Double.MIN_VALUE cause an exception. Use setOperation(JCSpinNumberBox.FLOATING_POINT) when you want to use floating point numbers. The use of setOperation(JCSpinNumberBox.INTEGER) is optional, since this is the default case.
JCSpinBox()	Use this component when you want a spin box containing an Object. For some non-standard objects, you may need to create your own editor and renderer.

JCSpinBox methods

These methods manage a list of items by providing methods for adding and removing items from the list of objects, and for adding listeners for these changes. See the API for the complete list of JCSpinBox methods.

Method Name	Description
addActionListener()	Adds an ActionListener.

Method Name	Description
removeActionListener()	Removes an ActionListener.
removeAllItems()	Removes all items from the item list.
addItem()	Adds an item to the item list.
removeItem()	Removes an item from the item list.
removeItemAt()	Removes the item at anIndex. To use this method, the JCSpinBox data model must implement JCSpinBoxMutableModel.
addItemListener()	Adds a java.awt.event.ItemListener. Its parameter is the class that will receive the event when the selected item changes.
removeItemListener()	Removes a java.awt.event.ItemListener.

12.5 Examples

To use a JCSpinNumberBox, simply instantiate it and set its parameters according to your needs, for example:

```
JCSpinNumberBox float_spin = new JCSpinNumberBox();
float_spin.setName("FloatingPointSpinBox");
float_spin.setValue(new Integer(0));
float_spin.setValueRange(new JCSpinNumberBox.Range(new Integer(0),
                                                   new Integer(12)));
float_spin.setSpinStep(new Double(1.5));
float_spin.setOperation(float_spin.FLOATING_POINT);
```

You don't need to use the setOperation method when you create an INTEGER version of a JCSpinNumberBox since that is the default type.

Similarly, you can create a JCSpinBox:

```
JCSpinBox string_spin = new JCSpinBox(titles);
string_spin.setName("StringSpinBox");
string_spin.setSelectedIndex(0);
string_spin.addValueListener(listener);
```

The figure shows that each time a mouse click changes a spin box's value, the generated event can report on both the old and the new value. The output in Figure 42 results from clicking each spin box in succession twice.

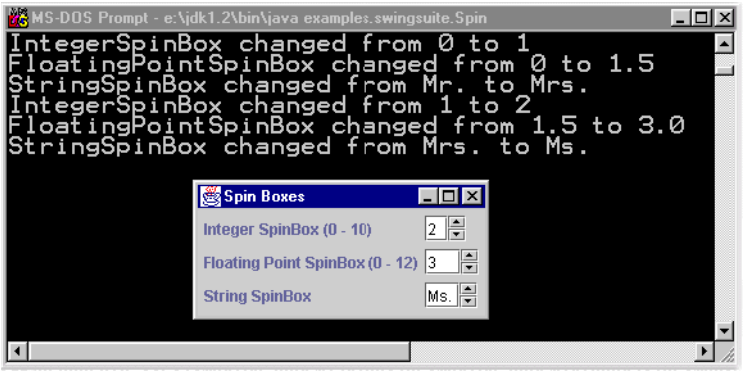


Figure 42 Capturing spin box events.

Listening for Spin Box Events

`JCNumberSpinBox` uses the `JCValueModel` interface to set and get its values, and to define its listeners. To respond to spin events, do something like this:

1. Create a listener as part of the setup for the component, for example:

```
JCValueListener listener = new ValueListener();
```

2. Then add a listener to the spin box:

```
float_spin.addValueListener(listener);
```

3. Implement the listener class and define a `valueChanged` method:

```
class ValueListener implements JCValueListener {
      public void valueChanging(JCValueEvent e) {
      }

      public void valueChanged(JCValueEvent e) {
      System.out.println(((Component) e.getSource()).getName() +
                                                    " changed from " +
                                                    e. getOldValue() +
                                                    " to " +
                                                    e.getNewValue());
}
} // end of ValueListener
```

13

Splash Screen

13.1 Features of JCSplashScreen

A splash screen is an image that appears while an application is loading. It serves both as an indication that the program is being loaded from disk and as a place to put notices, such as copyrights, version or release numbers, and the like.

`JCSplashScreen` does the following:

- Creates a splash screen given an `Icon` or the location of the image. The image location is the package path of the image and must be in the classpath. Any `Icon`, such as a GIF, JPEG, or other supported image may be used, so long as the time it takes to load is acceptable. An example of an image is in */demos/elements/gauge/gauge.gif*.
- Once instantiated, a `JCSplashScreen` appears only once. Hiding it causes it to be disposed.

13.2 Classes and Interfaces

The stand-alone class `com.klg.jclass.swing.JCSplashScreen` subclasses from `java.lang.Object`, providing an independent mechanism for displaying an image in a window in the middle of the screen.

No interfaces are used in `JCSplashScreen`.

13.3 Methods and Constructors

Constructors

`JCSplashScreen()`	`JCSplashScreen` has two constructors for instantiating a splash screen, one taking a `String` that specifies the location of the image, and the other taking an `Icon`. It throws an `illegalArgumentException` if the image String or image icon is invalid.

Methods

`setVisible()`	A Boolean method that shows or hides the splash screen. Once the splash screen is hidden, the splash screen window will be disposed. This means the splash screen cannot become visible again.

13.4 Examples

If you compile and run the code for this example, which is given below, you'll see a message printed on the console informing you that the application has started. The image for the splash screen is loaded and is made visible, then the program enters a wait state until a sleep command times out. An application that takes a long time to load would exhibit similar behavior. The user knows that loading is in progress because the splash screen is visible. It contains whatever graphic information you think is appropriate.

The example uses a `JCExitFrame` to hold a button that controls the disposal of the splash screen. All that is required for its disposal is the command `setVisible(false)`, but once it is given, the splash screen is gone for good. You would issue this command after receiving notification that the application is ready to run.

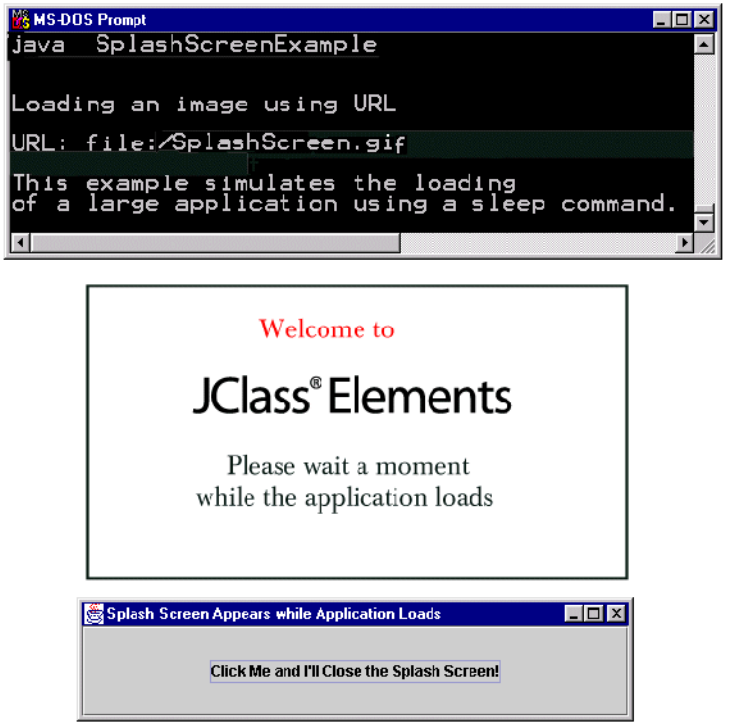


Figure 43 The visible elements in SplashScreenExample.

```
import com.klg.jclass.swing.*;
import com.klg.jclass.util.swing.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.net.*;

public class SplashScreenExample extends JFrame implements
                                                ActionListener {
    static JCSplashScreen ss;
    static Icon image;
            SplashScreenExample() {
              URL url =
                     getClass().getResource("/images/SplashScreen.gif");
              // convert URL to Image icon
              image = new ImageIcon(url);
              System.out.println(
                   "\n\nLoading an image using URL\n\nURL: " + url);
              System.out.println("This example simulates the loading");
```

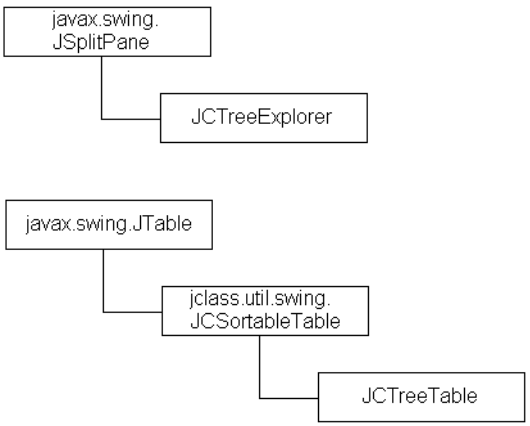
```
                System.out.println(
                      "of a large application using a sleep command.");
                  // initialize(image);
                  ss = new JCSplashScreen(image);
            }

            public void actionPerformed(ActionEvent e) {
                    if (e.getSource() instanceof JButton) {
                        ss.setVisible(false);
                        }
            }

        public static void main(String[] args){
            SplashScreenExample sse = new SplashScreenExample();
            String title = "Splash Screen Appears while Application Loads";
            JCExitFrame frame;
            frame = new JCExitFrame(title);
            ss = new JCSplashScreen(image);
            ss.setVisible(true);
            try {
                Thread.currentThread().sleep(10000);
                } catch (Exception e) {}
            Container cp = frame.getContentPane();
            JButton btn = new JButton(
                           "Click Me and I'll Close the Splash Screen!");
            btn.addActionListener(sse);
            cp.add(btn);
            frame.setSize(450, 100);
            frame.setVisible(true);

            frame.setExitOnClose(true); // Close the window so the
                                              application can exit.

            }


    }
```

14

Tree/Table Components

14.1 Features of JCTreeExplorer and JCTreeTable

Swing's `JTree` and `JTable` are the two components that do more than merely display data; they attempt to manage the data as well. This becomes important when you need to organize large amounts of data and provide a view that displays a portion of it along with an indication of its relationship to the rest. Information that has a hierarchical structure, like a file system, can be displayed as tree data, while other types of data nicely fit a tabular format. There are a large number of data structures that combine tree-like and table-like properties. A file system has a hierarchical organization that begs to be represented as a tree, yet the individual directories and files have properties, such as name, size, type, and date modified, that fit nicely in a row-column organization. Obviously there is a need for a component that lets you combine the look and functionality of both a tree and a table.



`JCTreeExplorer` and `JCTreeTable` fill the need for components that have the dual characteristics of a tree and a table, and provides these functions:

- Allows you to view the object as a tree, with an accompanying table. Any tree node may have tabular data associated with it.

- Contains a flexible "painter" object that accommodates a Swing cell renderer or a JClass cell renderer.
- Permits the construction of arbitrary data sources as treetables through its `JCTreeTableModel` interface. Any class can be used to supply data to the treetable, provided that it implements the `JCTreeTableModel` interface.
- The two components have advanced column sorting functionality. Each column can have a different ordered set of columns that are to be used as secondary sort keys. When a user clicks on a column header to sort that column, any identical cells are arranged based on the sort order of the secondary key, or keys.
- Cells may be edited by implementing `JCCellEditor`.
- Folder icons can be customized by replacing the editor/renderer, or by setting a `JCIconRenderer`.

Since `JCTreeExplorer` and `JCTreeTable` are enhancements of Swing's `JTable` and `JTree`, it's a good idea to be familiar with those components to ease the learning curve. Need a primer on Swing's table and tree components? See the tutorial on How to Use Tables and How to Use Trees at Sun's *javasoft* Web site.

`JCTreeExplorer` presents a table for the currently selected node, while `JCTreeTable` shows table rows for all visible nodes. See the following figure for the different visual characteristics between the two components.

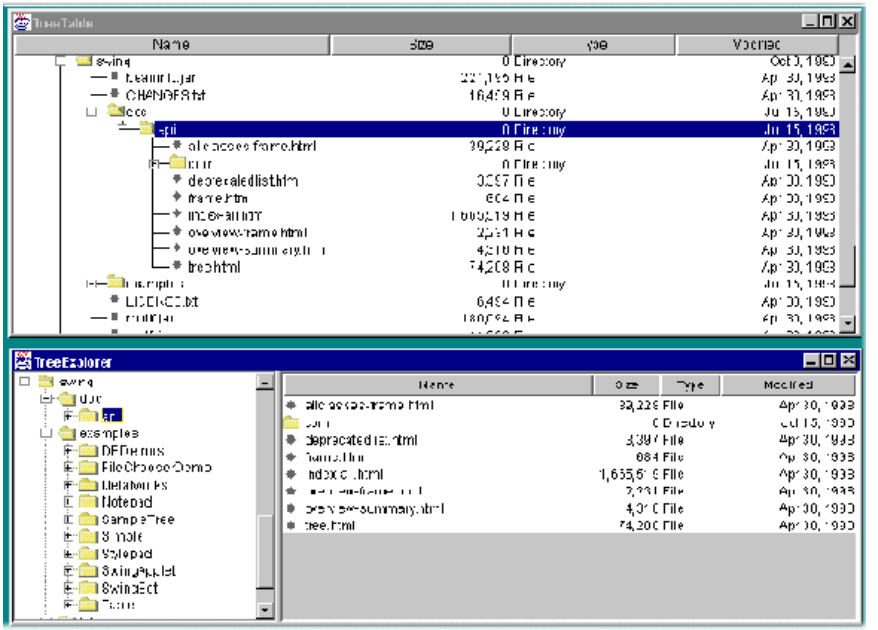


Figure 44 Views of a JCTreeTable and a JCTreeExplorer.

`JCTreeTable` and `JCTreeExplorer` support Swing's pluggable look-and-feel. The previous figure shows the Windows look and feel, while the following two figures show the default Metal look and feel.

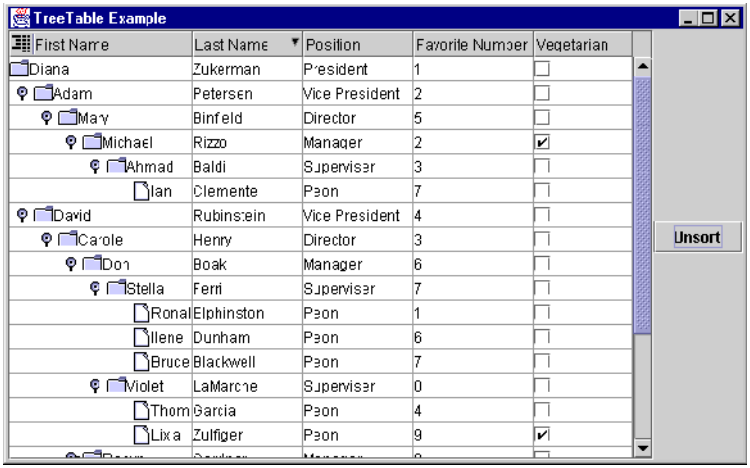


Figure 45 JCTreeTable component, metal look and feel.

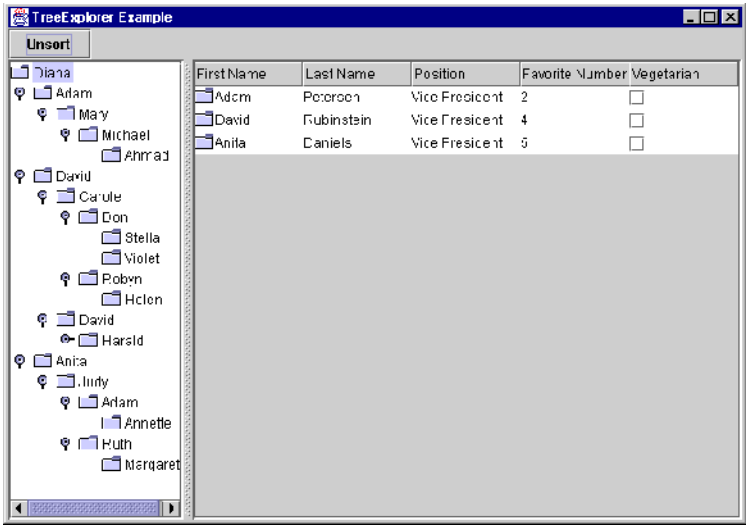


Figure 46 JCTreeExplorer component, metal look and feel.

14.2 Classes and Interfaces

Interfaces for JCTreeExplorer and JCTreeTable

Interface	Description
JCTreeIconRenderer	In com.klg.jclass.util.treetable, this interface represents a class that renders a tree icon. Not a renderer in the strict definition of the word, it provides the icon to be rendered. It is not necessary to make this interface do the rendering since Icons know how to render themselves. A default implementation of this class simply returns the *plaf* icon it is passed. The purpose of this mechanism is to allow a user to override the icons being drawn in a tree without a) having to figure out the default *plaf* for the icons that they do not wish to override, and b) overriding simple data-type editors. Its single method is getNodeIcon.
JCTreeTableModel	Model to use that combines the TreeModel and TableModel interfaces. This model allows data to be viewed as a multicolumn tree in a left-hand pane, and a table in a right-hand pane. Note that specific implementations need to implement both JCTreeTableModel and TableModel for them to work properly.

Here is the definition of the JCTreeTableModel interface:

```
public interface JCTreeTableModel {
    // Returns the value of the specific node and column
    public Object getValueAt(Object node, int column);
    // Returns whether a particular cell is editable,
    //                                    given the node and column
    public boolean isCellEditable(Object node, int column);
    // Sets the value at a particular node and column
    public void setValueAt(Object value, Object node, int column);
    // The following methods map exactly onto
    //                                  javax.swing.table.TableModel
    public void addTableModelListener(TableModelListener l);
    public Class getColumnClass(int column);
    public int getColumnCount();
    public String getColumnName(int column);
    public int getRowCount();
```

```
    public Object getValueAt(int row, int column);
    public boolean isCellEditable(int row, int column);
    public void removeTableModelListener(TableModelListener l);
    public void setValueAt(Object value, int row, int column);
    // The following methods map exactly onto javax.swing.tree.TreeModel
    public void addTreeModelListener(TreeModelListener l);
    public Object getChild(Object parent, int index);
    public int getChildCount(Object parent);
    public int getIndexOfChild(Object parent, Object child);
    public Object getRoot();
    public boolean isLeaf(Object node);
    public void removeTreeModelListener(TreeModelListener l);
    public void valueForPathChanged(TreePath path, Object newValue);
}
```

Classes

Class	Description
DefaultTreeTable SelectionModel	Extends javax.swing.tree.DefaultTreeSelectionModel and implements JCTreeTableSelectionModel. Like JCMultiSelectList, a treetable offers different selection modes. A treetable has an associated DefaultTreeTableSelectionModel when it is created, but you can define your own selection model, so long as it is a subclass of DefaultTreeSelectionModel and implements JCTreeTableSelectionModel.
TreeTableSupport	Abstract class in com.klg.jclass.util.treetable that provides an implementation that handles a TreeModel/TableModel combination for use in a Table component. Its functionality includes tracking expanded node counts, mapping and posting expansions, and selection events. It also provides a node "painter" object that can be wrapped into a Swing CellRenderer or a JClass CellRenderer.
JCSortableTable	A subclass of JTable that internally wraps any TableModel it is given with a JCRowSortTableModel and provides a Comparator that has an adjustable list of the column indexes that it uses for sorting. Clicking on a column header invokes the sorting behavior tied to that column; clicking again reverses the sort.
TreeWithSortable Children	This class implements JCTreeTableModel and JCRowSortModel. It constructs a JCTreeTableModel that wraps a given instance of a JCTreeTableModel and provides a sorted mapping of the children for any given leaf node. The sort order is defined by the configurable Comparator property.
com.klg.jclass. util.treetable BranchTree	An implementation of TreeModel that wraps a tree model so that it only exposes branches; that is, non-leaf nodes. This is useful for explorer-type views where the "tree view" portion only displays the branch nodes.
DefaultTreeIcon Renderer	An implementation of JCTreeIconRenderer, its getNodeIcon method returns the icon to render at the right of the specified value. This simple implementation returns the plaf icon passed to it.

Class	Description
JCTreeExplorer	A subclass of JSplitPane that provides a tree view on the left-hand side of the split pane, and a table view on the right. Constructor: JCTreeExplorer(JCTreeTableModel)
JCTreeTable	A subclass of JTable that handles listeners, rendering, editing, and painting of a component that combines tree-like and table-like properties. Constructor: JCTreeTable(JCTreeTableModel)
com.klg.jclass. util.treetable NodeChildrenTable	Maps the children of a particular node in a JCTreeTableModel into a standard Swing TableModel.

Providing your own sorting mechanism

If you need to provide your own sorting algorithm, one way is to subclass JCRowComparator and pass a comparator of the new type to TreeWithSortableChildren.

14.3 Properties

For a complete list of properties, please see Properties of JCTreeExplorer and Properties of JCTreeTable in Appendix A.

Properties of JCTreeExplorer

`getTree()`	Returns the `JTree` component used.
`getTable()`	Returns the `JTable` component used.
`setKeyColumns()`	Sets which columns are to be used as primary and secondary sort keys. It takes a column number (0, 1, 2 ...) as its first parameter, and an array of column numbers as its second parameter. **Example:** `setKeyColumns(0, {1, 0})` specifies that when the user clicks on the header in the first column, sorting takes place based on the second column, and identical entries in the second column are sorted based on the ordering implied by the first column. This is a useful sort key for directories, where the first column is the file or directory name, and the second column contains the object's size. Because directories have size zero, they are sorted at the top and then arranged alphabetically.

Properties of JCTreeTable

`treeTableModel`	The `treeTableModel` is the interface for the data.
`treeIconRenderer`	Gets or sets the icon renderer. If this property is set to null, no icon will be shown.
`rootVisible`	You can show the root node or not, depending on the setting of this Boolean property. The accessor method is called `isRootVisible`.
`showsRootHandles`	Determines whether the node handles are to be displayed.

14.4 Methods

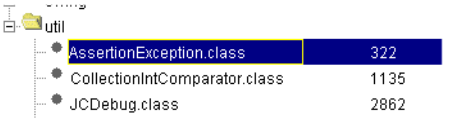
JCTreeExplorer Methods

`getSelectionPath()`	Returns the `javax.swing.tree.TreePath` of the first selected row inside the table view.
`getSelectionPaths()`	Returns the `javax.swing.tree.TreePath` of the first selected row inside the table view.
`getTable()`	Returns the `JTable` component used.
`getTree()`	Returns the `JTree` component used

`getTreeIconRenderer()`	Returns the icon renderer being used.
`setTreeIconRenderer()`	Sets the icon renderer.
`setUI()`	Sets the `javax.swing.plaf.TableUI` UI.

JCTreeTable Methods

`addSelectionPath()`	Adds the node identified by the specified `TreePath` to the current selection.
`addSelectionPaths()`	Adds each path in the array of paths to the current selection.
`addTableHeader MouseListener()`	Adds a `MouseListener` to the table header.
`addTreeExpansion Listener()`	Adds a listener for `TreeExpansion` events.
`addTreeWillExpand Listener()`	Adds a listener for `TreeWillExpand` events.
`collapsePath()`	Ensures that the node identified by the specified path is collapsed and viewable.
`collapseRow()`	Ensures that the node in the specified row is collapsed.
`createSortable TableColumn()`	Creates a `TableColumn`.
`expandPath()`	Ensures that the node identified by the specified path is expanded and viewable.
`expandRow()`	Ensures that the node in the specified row is expanded and viewable.
`getCellEditor()`	Overridden to return the appropriate data render for the first column if the treetable is in tree display mode.
`getCellRenderer()`	Overridden to return the appropriate data render for the first column if the treetable is in tree display mode.
`getClosestPath ForLocation()`	Returns the row to the node that is closest to X,Y.
`getEditingPath()`	Returns the path to the element that is currently being edited.
`getExpandedDescendants()`	Returns an `Enumeration` of the descendants of path that are currently expanded.
`getPathForLocation()`	Returns the path for the node at the specified location.
`getPathForRow()`	Returns the path that is displayed at the table row specified.
`getRowForLocation()`	Returns the row for the specified location.

`getRowForPath()`	Returns the row that displays the node identified by the specified path.
`getRowsForPaths()`	Returns the rows for the visible specified paths.
`getScrollsOnExpand()`	Returns `true` if the tree scrolls to show previously hidden children.
`getSelectedPath()`	Returns the `TreePath` of the first selected row.
`getSelectionPath()`	Returns the path to the first selected node.
`getSelectionPaths()`	Returns the paths of all selected values.
`getShowNodeLines()`	Returns the state of `ShowNodeLines`. If true, the connecting lines that are drawn between nodes in an explorer view are shown. 
`getShowsRootHandles()`	Returns `true` if handles for the root nodes are displayed.
`getTreeIconRenderer()`	Returns the icon renderer being used.
`getTreeSelectionModel()`	Returns the model for selections.
`getTreeTableModel()`	Returns the `JCTreeTableModel` that is providing the data.
`getView()`	Returns the current view.
`isPathSelected()`	Returns true if the item identified by the path is currently selected.
`isRootVisible()`	Returns true if the root node of the tree is displayed.
`isSortable()`	Is the treetable sortable? Returns the value of the `sortable` property.
`makeVisible()`	Ensures that the node identified by a path is currently viewable.
`removeTreeExpansion Listener()`	Removes a listener for `TreeExpansion` events.
`removeTreeWillExpand Listener()`	Removes a listener for `TreeWillExpand` events.
`setRootVisible()`	Determines whether or not the root node from the `TreeModel` is visible.

`setScrollsOnExpand()`	Determines the behavior of a node when it is expanded. If `true`, the viewport will scroll to show as many descendants as possible when the node is expanded; if `false`, the viewport will not scroll.
`setSelectionPath()`	Selects the node identified by the specified path.
`setSelectionPaths()`	Selects the specified paths.
`setShowNodeLines()`	Allows you override the *plaf*-specified behavior for drawing lines.
`setShowsRootHandles()`	Determines whether the node handles are to be displayed.
`setSortable()`	Sets whether the treetable is sortable.
`setSwitchPolicy()`	Sets the `switchPolicy` variable that determines whether or not to allow view switching Options are: `JCTreeTable.SWITCH_BUTTON_DONT_SHOW`, `JCTreeTable.SWITCH_VIEW_NEVER`, `JCTreeTable.SWITCH_VIEW_TO_TABLE_ON_SORT`, `JCTreeTable.SWITCH_VIEW_ON_ICON_ONLY`.
`setTreeIconRenderer()`	Sets the icon renderer.
`setTreeTableModel()`	Sets the `TreeModel` that will provide the data.
`setTreeTable SelectionModel()`	Sets the tree's selection model.
`setUI()`	Sets the `javax.swing.plaf.TableUI` UI.
`setView()`	Sets whether the view is for a tree or a table.
`updateUI()`	Updates the UI.

14.5 Examples

Implementing a custom node icon for a JCTreeTable

Your application may require that you supply your own custom node icon for the tree view. Create your own implementation of `JCTreeIconRenderer`, and write a method similar to the one whose signature is shown here:

```
public Icon getNodeIcon(TreeModel treemodel,
                        Object node,
                        Object value,
                        Class object_class,
                        boolean is_leaf,
                        boolean is_expanded,
                        Icon plaf_icon)
```

You may want to have two icons, one for nodes with children and one for leaf elements. In that case, use the `boolean` parameter `is_leaf` to choose which icon will be used.

The method should return the `Icon` you want to use. Pass your implementation of `JCTreeIconRenderer` to your instance of a `JCTreeTable` using `setTreeIconRenderer(JCTreeIconRenderer renderer)`.

Please see *TreeExplorer.java* and *TreeTable.java* in the *JCLASS_HOME\examples\elements* directory for some examples of using both `JCTreeExplorer` and `JCTreeTable`.

15

Wizard Creator

15.1 Features of JCWizard and JCSplitWizard

`JCWizard` lets you create and manage a Wizard-style group of dialogs by supplying informative events and special page components with standard buttons. You add a `JCWizardListener` to your `JCWizardPages` to invoke the actions that each page needs to perform.

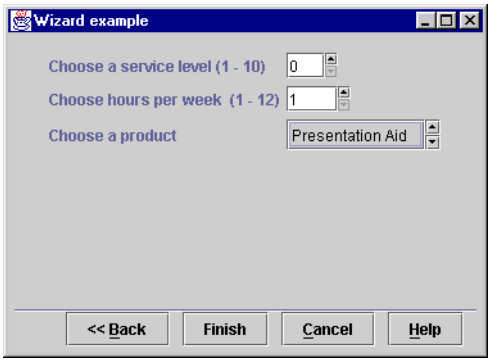
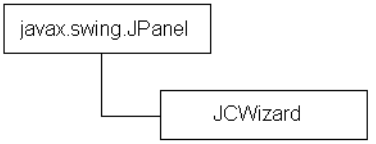


Figure 47 A sample Wizard page.

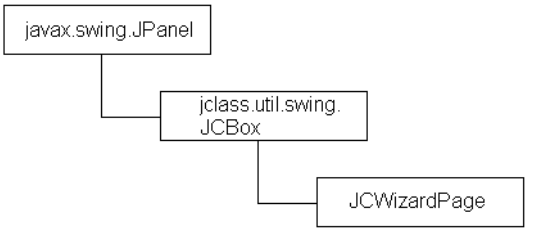
`JCWizard` supplies these features:

- Standard **Next**, **Back**, **Finish**, **Cancel** and **Help** buttons that are characteristic of Wizard dialogs.
- You provide instructions for the end-user on each page, and define the actions corresponding to the choices made by the end-user.
- The Wizard's pages are instances of `JCWizardPage`. As is usual with Swing components, you do not add children to `JCWizardPage`. Instead, you call its `getContentPane()` method and add items to it.

The `JCWizard` component is a container that manages `JCWizardPage`s. The pages are added to it in a way that only one of them shows at a time, but navigation buttons let the end user move back and forth through the deck. The component posts a `JCWizardEvent` as changes occur.



The `JCWizardPage` provides a `getContentPane()` method to return the panel to which you add content. It automatically builds content to manage the **Next**, **Previous**, **Finish**, **Cancel**, **Help** buttons at the bottom right of the page.



`JCSplitWizard`, on the other hand, creates a split-Wizard layout, which allows for one page to be created with multiple panels, rather than multiple pages with one panel.

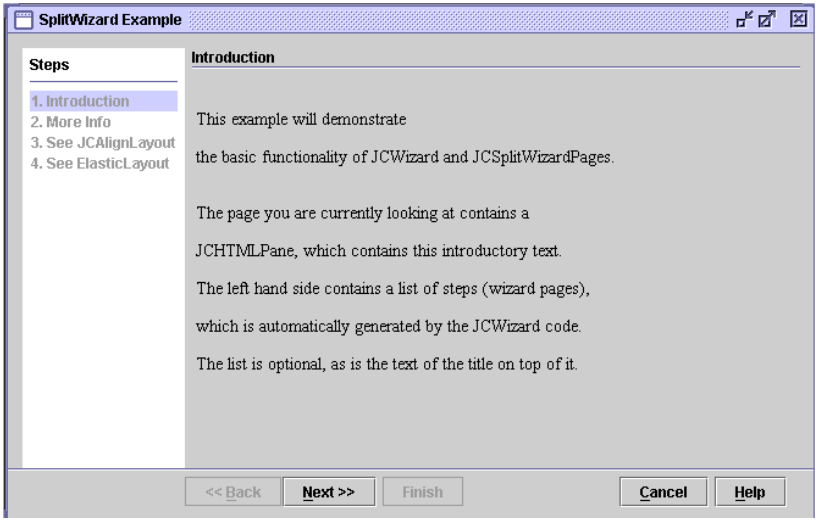


Figure 48 A sample split-Wizard page.

The button features provided for a standard Wizard are also available for the split-Wizard, though there is only one set that will apply to the entire Wizard. (The **Back** button is automatically unavailable on the first page, and the **Next** button is automatically unavailable on the last page.) Pages that are added are displayed in the right pane, while the left pane is used for the progress list, if one has been created.

15.2 Classes

The classes in the `JCWizard` group are:

`JCWizard`	Creates and manages a Wizard-style set of dialogs. To create your own Wizard, design `JCWizardPage`s and add Wizard listeners.
`JCWizardPage`	A `JCBox` that knows about Wizard-style actions.
`JCWizardEvent`	The event object that carries information about changes to `JCWizard` pages.
`JCWizardListener`	The listener for `JCWizard` events.

The classes in the `JCSplitWizard` group are:

`JCSplitWizard`	Creates a wizard with two panes, and a bottom button panel. The panes can contain any component, or left pane can optionally contain an automatically generated list of Steps.

15.3 Constructors and Methods

JCWizard Constructors

`JCWizardPage`'s constructor lets you specify the buttons you want on a page by combining `JCWizardPage` constants, as follows:

```
page = new JCWizardPage(JCWizardPage.NEXT     |
                        JCWizardPage.PREVIOUS |
                        JCWizardPage.FINISH   |
                        JCWizardPage.CANCEL   |
                        JCWizardPage.HELP);
```

JCSplitWizard Constructors

`JCSplitWizard` has two constructors. There is a constructor that can be used to create a standard two-pane Wizard, along with buttons and an empty string as a title. The second constructor is equipped with two arguments:

- The `int` argument that specifies which buttons should be included;

- The `String` title that specifies the title of the progress list.

JCWizard Methods

`JCWizard` inherits both properties and methods from `JPanel`. Listed here are the methods that `JCWizard` itself defines to provide the needed functionality for managing Wizard pages.

Method	Description
`add()`	Adds a page to the Wizard.
`addWizardListener()`	Adds a new `JCWizardListener` to the list.
`cancel()`	Invokes the registered "cancel" action.
`finish()`	Invokes the registered "finish" action.
`first()`	Moves to the first page in the Wizard.
`help()`	Invokes the registered "help" action.
`last()`	Moves to the last page in the Wizard.
`next()`	Advances to the next page in the Wizard.
`previous()`	Moves to the previous page in the Wizard.
`show()`	The method takes a parameter (`String name`), and moves to the Wizard page with the specified name.

JCSplitWizard Methods

Listed here are the methods that `JCSplitWizard` itself defines to provide the needed functionality for managing split-Wizard pages.

Method	Description
`addPage()`	Adds a page to the right panel of the Wizard.
`getLeftPanel()`	Accesses the left panel, which can contain an automatically generated list of steps based on titles, an image, help text, or any other desired components.
`cancel()`	Invokes the registered "cancel" action.
`finish()`	Invokes the registered "finish" action.
`first()`	Moves to the first page in the Wizard.
`help()`	Invokes the registered "help" action.
`last()`	Moves to the last page in the Wizard.

`next()`	Advances to the next page in the Wizard.
`previous()`	Moves to the previous page in the Wizard.

15.4 Events

A `JCWizard` or `JCSplitWizard` listens for `JCWizardEvent`s. A `JCWizardEvent` contains information on the `Object` that triggered the event, the `Component`'s current page and new page, two `Boolean`s, whether the event occurred on the last page, and whether the event should be allowed to finish processing.

Interface `JCWizardListener` methods are:

`nextBegin`	Invoked *before* advancing to the next page. Calling `e.setAllowChange(false)` will prevent the advance to the next page. Check `e.isLastPage()` to see if you are on the last page.
`nextComplete`	Invoked after advancing to the next page.
`previousBegin`	Invoked *before* returning to the previous page. Calling `e.setAllowChange(false)` will prevent the return to the previous page.
`previousComplete`	Invoked after advancing to the previous page.
`finished`	Invoked if a "finish" action is triggered.
`canceled`	Invoked if the "cancel" action is triggered.
`help`	Invoked if the "help" action is triggered

15.5 Examples

Please refer to `examples.elements.Wizard.java` to see how to construct regular Wizard pages. Briefly, these are the steps:

1. Add an instance of a `JCWizard` to a `JPanel` or similar component.
2. Create a `JCWizard` page.
3. Specify the buttons that should appear on each page.
4. Name the page.
5. The content for each page will likely be a `JPanel`. Add it to the Wizard page's content pane.
6. Add the page to the `JCWizard`.
7. Continue adding pages as necessary.

Please refer to `examples.elements.SplitWizard.java` to see how to construct split-Wizard pages. Briefly, these are the steps:

1. Create an instance of a `JCSplitWizard`, adding the desired buttons and determining whether or not a progress list should be generated.
   To create a progress list, pass the title of the list as the second argument in the constructor.
   If no progress list is desired, simply leave the title null, or use the no-argument constructor.
2. Create content for the right-hand wizard pages. These can be any instance of `JComponent`.
3. Add the pages to the wizard using `wizard.addPage(JComponent, page title)`, where page title is the title of the page.
4. Add the wizard to a container; add listeners, if desired.
5. If a progress list has not been specified, call `getLeftPage()` to add content to the left pane.

16

Layout Managers

16.1 Features of the Layout Managers in JClass Elements

This chapter describes JClass Elements' layout managers and the components that are closely associated with them. The layout managers are `JCAlignLayout`, `JCColumnLayout`, `JCElasticLayout`, `JCGridLayout`, and `JCRowLayout`. `JCBorder`, `JCBox`, `JCBrace`, and `JCSpring` are the associated components.

16.1.1 Layout Manager Classes

JCAlignLayout

`JCAlignLayout` is a layout manager that provides a simple way to lay out a vertically arranged group of control components, each with an associated label (or other component) placed to its left.

JCColumnLayout

`JCColumnLayout` is a simple subclass of `JCElasticLayout` that allows layout in a single column.

JCElasticLayout

`JCElasticLayout` is a layout manager that supports `JCElastic` components either horizontally or vertically. A component is considered *elastic* if it either implements the `JCElastic` interface or it has a constraint object that implements the `JCElastic` interface. Layout is performed in either a single row or column (depending on its orientation when created). The preferred size is calculated in the direction of orientation. If the container is bigger than the preferred size of all the components then the extra space is divided up between the components that are "elastic" in the direction of the orientation. The extra space is allocated to each of the components with respect to their "elasticity". If all the elastic components have the same elasticity (in the direction of the orientation) then they are equally stretched. If there is an uneven number of pixels to apportion, then the first n units of elasticity are allocated the extra pixels, where n is the remainder when the total elasticity is divided by the number representing the extra pixels ($n = total\_elasticity \bmod extra\_pixels$).

JCGridLayout

`JCGridLayout` is an improved subclass of `GridLayout`. It lays out a grid of rows and columns based on the attributes of the individual rows and columns. Whereas `GridLayout` uses the widest and tallest child to size each cell, `JCGridLayout` uses the widest element in a column to set the width of that column, and the tallest element in a row to set the height of that row.

JCRowLayout

`JCRowLayout` is a simple subclass of `JCElasticLayout` that allows layout in a single row.

16.1.2 Associated Component Classes

JCBorder

`JCBorder` can be used with any layout manager. With it you can place a border anywhere, not just around a component. You draw a border by overriding the component's paint method and calling `JCBorder.draw()`. Its parameters allow you to specify the `Graphics` object it will be passed, along with its border style, border size in pixels, placement of the top left corner relative to its parent, its width and height, and the shadow colors for its sides. Please refer to the API for a full description of the two variations of the parameter list for this method.

Border styles may be any one of the following:

`JCBorder.ETCHED_IN`	Double line, border appears inset.
`JCBorder.ETCHED_OUT`	Double line, border appears raised.
`JCBorder.FRAME_IN`	1-pixel shadow-in at edge, border appears framed.
`JCBorder.FRAME_OUT`	1-pixel shadow-out at edge, border appears framed.
`JCBorder.IN`	Border appears inset.
`JCBorder.OUT`	Border appears raised.
`JCBorder.CONTROL_IN`	MS-Windows control shadows.
`JCBorder.CONTROL_OUT`	MS-Windows control shadows.
`JCBorder.PLAIN`	Border drawn in foreground color.
`JCBorder.NONE`	No border drawn.

JCBox

`JCBox` is a Swing container that uses the `JCElasticLayout` to lay out components in a single row or column. Use the `orientation` property within an IDE to control the orientation of the box. The `JCSpring` and `JCBrace` components are useful Beans to use in conjunction with this container.

JCBrace

An implementation of a component that participates in a layout even though it has no view. It is called a *brace* because its main function is to reserve space as a way of controlling the layout of the visible components. A brace usually has equal minimum and preferred sizes, and an unlimited maximum size.

JCSpring

This is a stretchable concrete implementation of the `JCElasticLayout` interface, which specifies components as stretchable for the `JCElasticLayout` manager and its subclasses. A `JCSpring` has independently settable elasticity parameters for both the horizontal and vertical directions.

16.2 Interfaces

`JCElasticLayout` – The interface that informs enabled layout managers that a particular component should be stretched to its maximum before stretching any non-elastic components.

16.3 Properties

JCBox

`alignment`	One of `SwingConstants.LEFT`, `SwingConstants.CENTER`, or `SwingConstants.RIGHT`, specifying the alignment of the layout.
`orientation`	One of `JCElasticLayout.HORIZONTAL` or `JCElasticLayout.VERTICAL`, specifying how the container is to lay out its components.

JCBrace

`orientation`	One of `JCElasticLayout.HORIZONTAL` or `JCElasticLayout.VERTICAL`, specifying whether it is a horizontal or a vertical brace.
`length`	This is the value of both the minimum size and the preferred size. Whether the length refers to a horizontal or a vertical dimension depends on the `orientation`.

JCSpring

`horizontalElasticity` `verticalElasticity`	These are properties with integer values specifying the relative elasticities of the components to which they refer.

16.4 Constructors and Methods

16.4.1 Layout Managers

JCAlignLayout

`getLabelVertical Alignment()`	Returns the vertical position of a label relative to its control.
`setResizeHeight()`	Sets whether the control should be resized vertically to the height of the largest component in its row (default: `false`). This value is ignored for labels (the components in odd columns).
`setResizeWidth()`	Sets whether the control should be resized horizontally to its parent's right edge if it is in the last column (default: `false`).

`setLabelVertical Alignment()`	Sets the vertical position of a label relative to its control. Choices are `TOP`, `MIDDLE` (default), or `BOTTOM`.

JCColumnLayout

A simple subclass of `JCElasticLayout` that arranges layout in a single column.

`JCColumnLayout()`	Creates a column layout that aligns components on the left.
`JCColumnLayout( int alignment)`	Creates a column layout that aligns components to the specified alignment: `SwingConstants.LEFT`, `SwingConstants.CENTER`, or `SwingConstants.RIGHT`

JCElasticLayout

Use its constructors to provide the layout you want.

`JCElasticLayout (int orientation)`	Creates a row layout that by default aligns components to the left of the row or column.
`JCElasticLayout (int orientation, int alignment)`	Creates a column layout that aligns components as specified by the second parameter, which can be one of the `SwingConstants.LEFT`, `SwingConstants.RIGHT`, `SwingConstants.TOP`, `SwingConstants.BOTTOM`, or `SwingConstants.CENTER`.

When adding an elastic constraint to an object, you can use one of these constants:

- `JCElasticLayout.HORIZONTALLY_ELASTIC_CONSTRAINT`,
- `JCElasticLayout.VERTICALLY_ELASTIC_CONSTRAINT`,
- `JCElasticLayout.COMPLETELY_ELASTIC_CONSTRAINT`

For example:

```
add(c, JCElasticLayout.HORIZONTALLY_ELASTIC_CONSTRAINT);
```

JCGridLayout

Like `GridLayout` in the AWT, `JCGridLayout` has a two-parameter constructor in which you specify the number of rows and columns for your grid, and a four-parameter version in which you specify horizontal and vertical gaps as well. Use this constructor just as you would a `GridLayout`. Unlike the AWT's `GridLayout`, `JCGridLayout`'s rows may have different heights and its columns may have different widths. See the example later on in this chapter for a visual comparison between the two layout managers.

16.4.2 Associated Components

JCBox

`JCBox()`	Creates a horizontal `JCBox` container. The constructor may have an optional parameter, `int orientation`. In this case, valid values are `JCBox.HORIZONTAL` or `JCBox.VERTICAL`.
`createHorizontalBox()` `createVerticalBox()`	Convenience methods for creating `JCBox`es.
`getAlignment()` `setAlignment(int alignment)`	Describes how the component is aligned. The alignment parameter is one of `SwingConstants.LEFT`, `SwingConstants.CENTER`, or `SwingConstants.RIGHT`.
`getOrientation()` `setOrientation(int orientation)`	The box uses the orientation to determine whether its components are arranged horizontally or vertically.

JCBrace

`getOrientation()` `setOrientation(int orientation)`	The parameter in the `set` method can be one of `JCElasticLayout.HORIZONTAL` or `JCElasticLayout.VERTICAL`, specifying how the container is to lay out its components.
`getLength()` `setLength(int length)`	This is the value of both the minimum size and the preferred size. Whether the length refers to a horizontal or a vertical dimension depends on the `orientation`.

JCSpring

`get/setHorizontal Elasticity()` `get/setVertical Elasticity()`	The set methods take an integer parameter specifying the relative elasticity of the `JCSpring`. When two or more springs are used, the elasticities are used as weighted values for the springiness.

`get/setMaximumSize()`
`get/setMinimumSize()`
`get/setPreferredSize()`
	The set methods require a `Dimension` parameter, which the get methods return.

16.5 Examples

JCGridLayout

The example shown here illustrates the difference between AWT's `GridLayout` and JClass Elements' `JCGridLayout`, which conserves space by permitting rows to have different heights and columns to have different widths. The height of each row is determined by the height of the tallest component in that row, and the width of a column is determined by widest component in the column, independent of the width of other columns. With `JCGridLayout`, rows have varying heights and columns have varying widths.

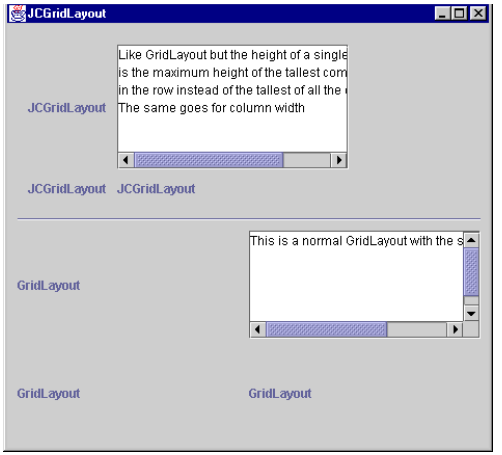


Figure 49 A comparison of JCGridLayout and GridLayout.

JCAlignLayout

This layout manager makes it easy to provide a vertical arrangement of data input fields and their associated labels. You can provide for more than two columns and, as the example shows, you aren't restricted to text fields.

Although its intended use is one with labels in the first column, `JCAlignLayout` lets you place any component in any column.

Use it as you would any layout manager for a frame or panel:

```
JCAlignLayout layout = new JCAlignLayout(2, 5, 5);
setLayout(layout);
```

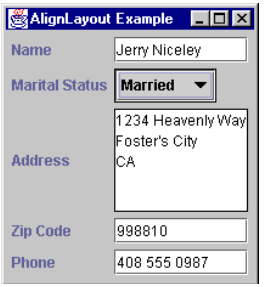


Figure 50 Using JCAlignLayout.

Part II

Utility Classes

17

Introduction to the Utility Classes

Utilities

17.1 Utilities

JClass Elements's utilities live for the most part in two packages: `com.klg.jclass.util`, and `com.klg.jclass.util.swing`. Some components, like `JCTreeTable`, rely on support classes found in a `util` subpackage, in this case in `com.klg.jclass.util.treetable`, and `JCDateChooser` has a package of its own, `com.klg.jclass.util.calendar`.

Here is a brief description of the utility classes:

Name of Utility	Description
`JCDebug`	Place debug statements in your Java code that contain an optional level indication. Use a print level setting to control the detail in your debugging printout. A PERL script lets you remove all debug statements when your code is ready for release.
`JCIconCreator`	There are times when you would like to have a custom image as part of your toolbars, labels, buttons, and so on, yet you don't want to go to the trouble of using a paint package. `JCIconCreator` lets you use `String` arrays to create image icons. The format is similar to that used in *XPixMap* (XPM) format files.

Name of Utility	Description
JCEncode Component	You can encode the image information for any component in your application with this utility. Since all the children of the chosen component are also encoded, you can capture a picture of your entire user interface for any well-behaved component hierarchy, or any single one of its child components. The utility encodes images in the public-domain Portable Network Graphics (PNG) format. **Note: If you wish to export your images in GIF format, you'll need a license from the copyright holder, Unisys Corp. Quest Software will send you the GIF encoder class upon receipt of a copy of your license from Unisys.** **You need JClass PageLayout to encode components in EPS, PS, PDF, or PCL. JClass PageLayout is available as a part of the JClass DesktopViews suite.**
JCListenerList	JCListenerList is a class that assists with keeping track of event listeners in a thread-safe manner.
JCMappingSort	Sorting can be accomplished by indexing the list of objects that are going to be ordered according to some comparison policy. It can be much more efficient to sort these indices instead of sorting the objects themselves. The idea is to form an array of indices. The utility is documented with JCSortableTable.
JCProgress Helper	JCProgressHelper is a class that lets you create and manage a thread-safe progress dialog. With it, you can monitor a potentially time-consuming operation and present a visual record of its progress. If it the operation will take more than a specified time, a progress dialog will be popped up.
JCString Tokenizer	JCStringTokenizer provides simple linear tokenization of a String. The set of delimiters, which defaults to common whitespace characters, can be specified either during creation or on a per-token basis. It is similar to java.util.StringTokenizer, but delimiters can be included as literals by preceding them with a backslash character (the default). It also fixes a known problem: if one delimiter immediately follows another, a null String is returned as the token instead of being skipped over.
JCSwing Utilities	This class currently contains a single method: setEnabled(), which takes a Component and a Boolean as parameters. It uses the Boolean parameter to enable or disable all the children of the component, which are themselves Components.

Name of Utility	Description
`Thread Safety`	The `JCSwingRunnable` class helps you manage your threads by providing a class that you can subclass easily. It provides methods that simplify handling execution in the proper threads.
`JCTreeSet`	This class adds to Swing's functionality by providing you with a way of representing the elements of a set as a binary-sorted tree. If the elements of the set have an defined ordering principle, it is used by default to construct the B-tree, but other ordering mechanisms are possible.
`JCTypeConverter` and `JCSwingType Converter`	You frequently need to convert objects to Strings, and Strings to objects, when you are coding a user interface. For example, a user types a String as input that you would like to convert to an object. The input String might consist of a sequence of integers, delimited by commas, that you would like to convert to an array. There are also times when you need to convert an object to a String so that you can place the text on a label or a button. The JClass type converters are a collection of the most useful conversions from String to object, and from object to String. The static methods of `JCTypeConverter` let you retrieve parameters from an application or applet, and convert these parameters to particular data types.
`JCWordWrap`	`JCWordWrap` provides a static method called `wrapText` that performs basic word-wrap logic on a String, given a line width and new line delimiter.

18

Debugging Tools

18.1 Features of JCDebug

`JCDebug` is a utility class that allows you to add debugging statements to your source code, in order to facilitate development tasks. Once the statements are in the code, the debugging mechanism can easily be turned on or off. To activate normal debug output, simply call `setEnabled(true)`; to turn off debugging without removing it from the source, call `setEnabled(false)` to globally disable `JCDebug` printing. To permanently remove the debugging code from the source, just the *jcdebug.pl* perl script, located in the *JCLASS_HOME/bin* directory.

It is also possible to group debugging statements by providing a tag parameter along with the text to be printed. For example:

```
JCDebug.println("tag1", "This is printed when setEnabled(true) and
   setTag(\"tag1\") are in effect.").
```

Thus, if you wish to enable print statements marked by parameter `tag1`, call `setTag("tag1")`. If you wish to turn on debugging print statements with various tags, call `setTags(String new_tags[])`, passing in an array of tag names. Initially, the array of tag names is null, which means that no tagged statements will be output. Thus, once some tags have been set, they can all be turned off by calling `setTag(null)`.

Another useful construct is the concept of "levels." The advantage of this is that by calling, for example, `setLevel(2)`, you will avoid seeing any debug messages marked 3 or below. This is useful for controlling the amount of detail you are viewing without removing the debug information.

Note that JClass distribution bytecode does *not* contain any references to `JCDebug`.

`JCDebug` helps you accomplish the following:

- Place multi-level print statements in your code for debugging purposes and remove them after testing is complete.
- Force a stack trace to occur at any point in your code.

- Optionally use a Perl script to place `/*DEBUG_START*/ ... /*DEBUG_END*/` blocks in your code. The debug statements are removed or commented out at ship-time using the PERL script *JCLASS_HOME/bin/jcdebug.pl*.

18.2 Classes and Scripts

`com.klg.jclass.util.JCDebug`	Class containing static methods for assertions and debug statements.
`jcdebug.pl`	The Perl script that removes or comments out `JCDebug` statements from source code.

18.3 Methods

Printing debug information

You can define a numerical order-of-importance to your print statements. Print statements labeled with lower numbers are deemed to be more important than those with higher-numbered labels. By setting a global print level variable at 3, for example, all print statements labeled with a number higher than three will be ignored. Those labeled with a print level variable of 1, 2, or 3 will all be printed.

Also, it's possible to supply a list of tags. All print statements with print level variables matching a `String` in the list will be processed, all other print statements will be ignored.

`setPrintStream()`	Sets the output stream to use. The default is `System.out`.
`getPrintStream()`	Returns the `PrintStream` currently in use.
`println()`	Prints debug information on the current output stream.
`println (String text)`	Always prints a message, unless debugging is turned off.
`println(int plevel, String text)`	The `int` parameter is for level-controlled diagnostic printing. Any print statement with a `plevel` greater than the currently set print level is not printed.
`println(String ptag, String text)`	The `ptag` parameter for tag-controlled diagnostic printing. Any print statement with a `ptag` matching than the currently set tag list is printed.

`println(int plevel, String ptag, String text)`	Prints the text according to level and tag filter options. This method is a combination of `println(int plevel, String text)` and `println(Sting ptag, String text)`.
`setEnabled(true)`	Activates debug output.
`isEnabled()`	Determines if debugging is on or off.
`setLevel(int new_level)`	Output occurs for statements marked with the specified level or lower. To see all debug statements, set `new_level` to a very large number.
`getLevel()`	Returns the level that determines what gets printed. All levels less than or equal to the returned integer are printed.
`setTag("myString")`	Sets one tag. A second call to `setTag()` causes the first tag to be forgotten.
`setTags(arrayOfStrings)`	Sets the array of tags to use for the debugging session.

Notes on tags and level numbers:

- If you do not supply a tag as one of the parameters in your print statement, the tag is deemed to be null. Statements with null tags are always printed as long as debugging is enabled.
- As long as you get a match from the level number or the tags you'll have some diagnostic printout.
- You don't have to use either tags or level numbers. You can simply use unadorned `JCDebug.println` statements. In this case, you don't have any selectivity other than being able to turn debugging on and off.

Forcing a stack trace

The following methods help force a stack trace:

`printStackTrace()`	Forces a stack trace at the current location.
`printStackTrace(String s)`	Forces a stack trace at the current location, and prints an identifying message as a header.
`printStackTrace(String ptag, Throwable t)`	Prints a stack trace for the specified exception if the specified tag is currently enabled.

18.4 Removing JCDebug Statements from Your Code

To remove the JCDebug statements from your code, you will need a Perl interpreter. If you don't have one, it is available at the central Web site for the Perl community (*http://www.perl.com*).

Running the Perl script

The script you will need to call is *jcdebug.pl*. It has three options: `-e`, `-d`, and `-r`. The `-e` option enables your debugging statements. It places markers in the form of comments before and after each debug statement, leaving the statement itself exposed to the Java compiler. These markers have the form `/* JCDEBUG_START* /` and `/* JCDEBUG_END */`, with the `JCDebug` statement in between.

The `-d` option removes the inner pair of matching `* /  /*` brackets to form one long comment which includes the `JCDebug` statement.

The `-r` option completely removes all `JCDebug` statements. You might want to use this option just before shipping the final product. Just realize that once the debug statements are removed using this option, they can't be recovered.

Here's an example of a command to delete `JCDebug` statements by turning them into comments:

```
perl jcdebug.pl -d MyCode.java
```

After executing this command, all `JCDebug` statements begin with `/* JCDEBUG_START` and end with `JCDEBUG_END */`. Since this is the syntax for the start and end of a comment, anything between these tags is not compiled.

Execute the following command:

```
perl jcdebug.pl -e MyCode.java
```

This will modify the file by replacing the inner pair of matching `* /  /*` brackets so that all `JCDebug` statements begin with `/* JCDEBUG_START */` and end with `/* JCDEBUG_END */`. The `JCDebug` statement itself is no longer part of the comment.

As was mentioned above, the script removes all lines with `JCDebug.print` in them, and any `import com.klg.jclass.util.JCDebug` statements that may exist. If you just want to turn off the debugging code without removing it from your source, call `setEnabled(false)` to globally disable `JCDebug` assertions and printing.

18.5 Examples

This example illustrates the following points about the use of `JCDebug`:

- `JCDebug.setEnabled(true)` must be in effect for the debug mechanism to be turned on.

- If tags are used, `JCDebug.setTag()` controls which tagged print statements are active. The example uses `tag2` and `tag3` as arbitrary labels. Print statements involving `tag2` will be active, but those involving `tag3` will not.

```
import com.klg.jclass.util.JCDebug;

public class TestJCDebug {

public static void main(String args[])
{
   JCDebug.setEnabled(true);
   System.out.println("Starting the test:");

   //Set a tag so that all JCDebug statements with this tag will print
   JCDebug.setTag("tag2");

   //These should print
   JCDebug.println("Debugging is on so this should be printed!");
   JCDebug.println("tag2", "Label tag2 is enabled, " + "so this should be
                     printed.");

   //This should not print because it does not match the tag
   JCDebug.println("tag3", "The tag for this print statement is tag3 " +
                     "so this should not be printed.");

   //Now turn off debugging
   JCDebug.setEnabled(false);

   //The following two lines will not be printed.
   JCDebug.println("Debugging is off. This should not be printed!");
   JCDebug.println("tag2", "This label is enabled, but debugging is off, "
                     + "so this should not be printed");
}

}
```

The output from this test program is reproduced here.

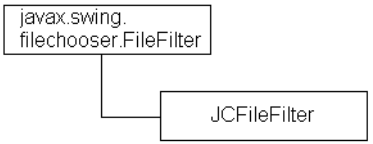
```
Starting the test:
Debugging is on so this should be printed!
Label tag2 is enabled, so this should be printed.
```

19

JCFileFilter

19.1 Features of JCFileFilter

`JCFileFilter` enhances Swing's `JFileChooser` by allowing you to pass it file extensions. These are the only ones that appear in the file chooser dialog. Extensions are of the type ".txt," ".java," and so on, which are typically found on Windows and Unix platforms, but not on the Macintosh. Case is ignored, so ".txt" is equivalent to ".TXT" as far as the filter is concerned.



The class has versatile constructors and convenience methods for setting both the extensions that are to be filtered and an optional descriptive phrase.

19.2 Constructors

`JCFileFilter` has four constructors:

`JCFileFilter()`	The default constructor. This form of the constructor is used to list all file types. To filter files, use the `addExtension()` method. To provide a human-readable description for the extension, use the `setDescription()` method.
`JCFileFilter(String extension)`	Creates a file filter that, initially at least, shows only files of the type whose extension is specified by the String argument. The period (dot) before the extension is optional. More filename extensions may be added using the `addExtension()` method.

`JCFileFilter(String[] filters)`	Creates a file filter for a list of file extensions given as a String array. More may be added using the `addExtension()` method.
`JCFileFilter(String extension, String description)`	Creates a file filter that, initially at least, shows only files of the type whose extension is specified by the String argument. The period (dot) before the extension is optional. More filename extensions may be added using the `addExtension()` method. The human-readable description is retrieved using the `getDescription()` method.

19.3 Methods

These methods let you set or examine the elements of the filtering process:

`addExtension(String extension)`	Places an additional filename extension in the list of those to filter against. Any filename matching this extension is listed in the file dialog, as well as those that have been added in the constructor. The method may be used multiple times to form a list of extensions to filter against.
`accept(File f)`	This Boolean method is used to determine if the given filename has an extension that is in the list of those to be filtered, that is, to be displayed in the file dialog. Files that begin with "." are ignored, but directories are always shown.
`getExtension(Strings)`	Returns the extension portion of a String.
`set/getDescription()`	Sets or gets the human readable description of this filter. This String is used to preface the list of file extensions that shows up in the file dialog's "Files of type:" combo box. For example, if the filter is set to: `String[] filterTypes = {"gif", "jpg"};` and the description is specified as `"JPEG & GIF Images"`, then the combo box displays "**JPEG & GIF Images (*.gif, *.jpg)**"
`setExtensionListIn Description( boolean b)` `isExtensionListIn Description()`	Determines whether the extension list, for example (*.jpg, *.gif), should show up in the human-readable description. The corresponding Boolean method returns the policy currently in effect.

19.4 Example

The following code snippet sets up a filter for GIF and Java source files. The description, which appears in the file chooser's "Files of type:" combo box along with the extensions themselves, is provided through a parameter passed to the constructor.

Note that you can control whether either part of the description actually appears through the use of `setDescription()` and `setExtensionListInDescription()`.

```
JFileChooser chooser = new JFileChooser();
String[] filterTypes = {"gif", "java"};
JCFileFilter filter =
    new JCFileFilter(filterTypes, "GIF Images and Java source files");
chooser.addChoosableFileFilter(filter);
```

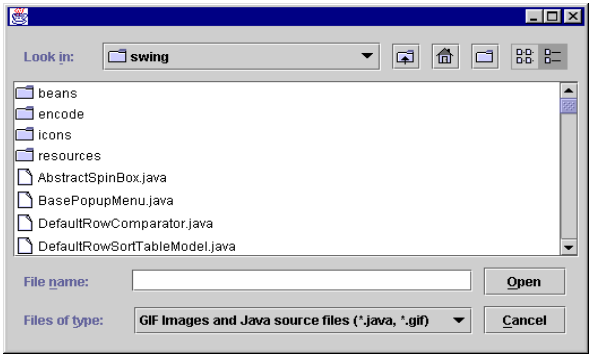


Figure 51 A JCFileFilter for Java source files and GIF images.

20

Icon Creator

20.1 Features of JCIconCreator

There are times when you would like to have a custom image as part of your toolbars, labels, buttons, and so on, yet you don't want to go to the trouble of using a paint package. `JCIconCreator` lets you use `String` arrays to create image icons. The advantages of using `JCIconCreator` include:

- A simple and convenient way of defining an image from a String of characters.
- Keeping the image information in the class that uses it, rather than having to manage the location of associated image files.
- Designing small-sized custom images or diagrams without the need of a paint program.
- Having a simple way of associating the image with the standard `javax.swing.ImageIcon` class.

20.2 Classes

This utility consists of a single class, `com.klg.jclass.util.swing.JCIconCreator`, subclassed from `java.lang.Object`.

20.3 Constructors and Methods

The `JCIconCreator` has two constructors are `JCIconCreator()`, which creates an uninitialized image icon, and `JCIconCreator(int w, int h)`, where the parameters measure the size, in pixels, of the two dimensional array used to hold the characters representing the image.

Methods in JCIconCreator

The following is a list of the methods available for `JCIconCreator`:

`clear()`	Clears the icon so that no image is associated with it.
`getIcon()`	Gets the icon created by this instance of `JCIconCreator`. An overloaded version of this method takes a passed-in byte array (as might be obtained from a database's image field) and attempts to convert it into an `Image`. Use `getIcon()` method to return an `ImageIcon`. For example: `ImageIcon myLabelIcon = ic.getIcon();`
`setColor()`	Sets the color corresponding to a character passed as its first parameter. Its second parameter is a `Color` object or an RGB `int`. Use the `setColor()` method to associate a character in the array with a color. For example, if `ic` is an instance of a `JCIconCreator`, `ic.setColor('G', Color.green);` associates a "G" in the pixel map with a green pixel.
`setPixels()`	Sets the pixel data. If its parameter is an array of Strings, this represents the data for all rows. If the parameters are a row index and a String, this represents the pixel data for one row. If `pixelMap` is an array of characters representing pixels, inform the instance about them with `ic.setPixels(pixelMap);`
`setSize()`	Width and height `int` parameters are used to set the width and height for the image.

20.4 Examples

The following code section shows how to declare a `String` array, use it as the source for defining the pixels in an icon, and how to convert the `JCIconCreator` object to an `ImageIcon` for use as the graphic part of a label.

```
...
private static final String testIcon[] = {
"  BBBBBBBBB  ",
"  B  000  B  ",
"  B 00000 B  ",
"  B 00000 B  ",
"  B 00000 B  ",
"  B  000  B  ",
"  B       B  ",
"  B       B  ",
```

```
"  B       B  ",
"  B       B  ",
"  B       B  ",
"  B       B  ",
"  BBBBBBBB   " };

JButton b1;

public ToolbarIcons() {
    JToolBar bar;
    JLabel label;

    setBackground(Color.lightGray);
    setLayout(new BorderLayout());

    JCIconCreator ic = new JCIconCreator(13, 13);
    ic.setColor('B', Color.black);
    ic.setColor('O', Color.orange);
    ic.setPixels(testIcon);
    ImageIcon icon = ic.getIcon();
    ...
    bar = new JToolBar();
    b1 = new JButton("Caution", icon);
    bar.add(b1);
    ...
```

Figure 52 Three labels with custom icons created using JCIconCreator.

21

Image Encoder

21.1 Features of JCEncodeComponent

You can encode the image information for any component in your application with this utility. Since all the children of the chosen component are also encoded, you can capture a picture of your entire user interface for any well-behaved component hierarchy, or any single one of its child components. The utility encodes images in the public-domain Portable Network Graphics (PNG) format. Other common formats are available if you also have JClass PageLayout installed. JClass PageLayout is available as part of the JClass DesktopViews suite.

The advantages of using `JCEncodeComponent` include:

- Saving an image of a component in PNG format.
- A simple way to encode a component: just call `JCEncodeComponent.encode()`.

Please note that the JPEG format is not supported because it loses information as a result of the compression.

Note: If you wish to export your images in GIF format, you'll need a license from the copyright holder, Unisys Corp. Quest Software will send you the GIF encoder class upon receipt of a copy of your license from Unisys.

21.2 Classes and Interfaces

The `com.klg.jclass.util.swing.encode` package contains an interface, a main class called `JCEncodeComponent`, and various helper classes that output the various supported image formats.

The interface that defines an image encoder contains a single method: `encode()`. Its parameters are the component whose image is to be encoded, and the stream on which to place the encoded information.

There is also an exception class, `EncoderException`. It is raised by `JCEncodeComponent` or one of its subclasses. The exception may be subclassed for exceptions thrown by

subclasses of JCEncodeComponent. It represents any problem encountered while encoding an image. The message is used to state the type of error.

JCEncodeComponent has a public static inner class named Encoding that is used to provide instances of the various valid encodings or to supply an error message if an attempt fails.

21.3 Constructors and Methods

JCEncodeComponent

The Encode inner class is used to instantiate a particular encoding type, such as PDF. It defines methods that provide information about the encoder, including a failure message if the encoder fails to load.

getEncoder()	Returns an encoder for this encoding type.
getFailureMessage()	Message stating possible reasons for encoder load failure.
getLongName()	Returns the fully qualified name of the supported encoding type.
getShortName()	Returns the descriptive name of the supported encoding type.
toString()	Returns both the short name and the long name in a single String.

The array called ENCODINGS contains, as instances of Encode, the supported encoding types. You pass an element of this array to the encode method, along with your component and a Stream specifier, to produce an encoding of the component which is sent to the stream. The method is overloaded so that you can write the information to a file if you wish.

encode(Encoding encoding, Component component, OutputStream output)	Invoke this method on a Java component to encode its image in the specified format, and send the encoded information to the specified stream.
encode(Encoding encoding, Component component, File file)	Encodes the component's image in the specified format. Sends the encoded information to the specified file.

21.4 Examples

Below is an example that encodes an entire frame in PNG format, then stores the result in a file. Most of the code simply serves to create a frame containing a few components. Since the process of encoding can result in an exception being thrown, the single-line command that does all the work is enclosed in a `try` block.

```
import com.klg.jclass.util.swing.encode.JCEncodeComponent;
import com.klg.jclass.util.swing.JCExitFrame;
import java.io.File;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.JButton;
import javax.swing.JTextField;

public class EncoderExample {

    public static void main(String[] args)   {
        JCExitFrame eFrame;
        eFrame = new JCExitFrame("Encoder Example");
        JPanel jp = new JPanel();
        JLabel jl = new JLabel("PNG Encoding");
        JButton jb = new JButton("Just a button");
        JTextField jt = new JTextField(
                                    "The entire frame will be encoded");
        jp.add(jl);
        jp.add(jb);
        jp.add(jt);
        jp.setVisible(true);
        eFrame.getContentPane().add(jp);
        eFrame.setSize(350, 100);
        eFrame.setVisible(true);
        File efps = new File("efps.png");
        try {
            JCEncodeComponent.encode(JCEncodeComponent.PNG,
                                                        eFrame, efps);
            } catch (Exception e) {
            System.out.println("Exception caught: " + e);
            }
    }
}
```

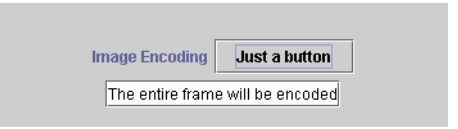


Figure 53 The result of encoding the entire JCExitFrame.

You can find another example, *Encode.java*, in the *examples/elements* directory. In that example, a single component, a button, is encoded. A combo box lets you choose the encoding format, a text field displays the current choice, and a button-press initiates encoding to a file. This example is more realistic in that the encoding process is initiated

by the end user through some action, such as a menu choice, or, as in this case, by pressing a button.

The result is shown in the next figure.

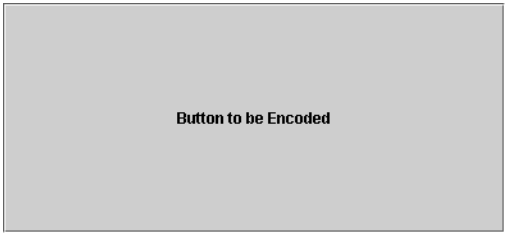


Figure 54 Encoding a single component using examples.elements.Encode.

If you attempt to encode a component using a GIF format, you will see the following error dialog:

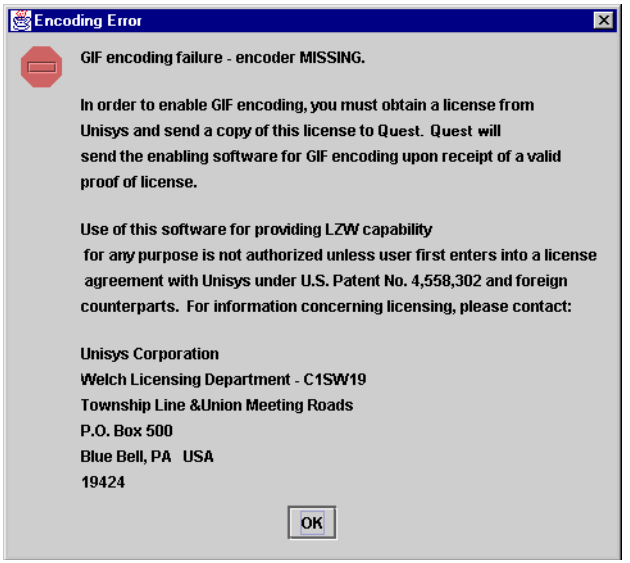


Figure 55 The error dialog that appears if you do not have GIF encoding installed.

Note: You need JClass PageLayout to encode components in EPS, PS, PDF, or PCL. JClass PageLayout is available as a part of the JClass DesktopViews suite.

22

Listener List

22.1 Features of JCListenerList

`JCListenerList` is a class that assists with keeping track of event listeners in a thread-safe manner. The use of static methods on the `JCListenerList` class prevents any problems from occurring if the list being modified is null. To send events to the listener in the list, simply get the `JCListenerListEnumeration` of the list and walk through the elements. There is no ordering guarantee.

22.2 Classes

The following is a list of the `JCListenerList` classes:

`JCListenerList`	The thread-safe listener list class.
`JClistenerListEnumeration`	Implements `java.util.Enumeration` and takes a `JCListenerList` as its parameter. It defines methods `hasMoreElements()` and `nextElement()`.

22.3 Methods

The following is a list of the `JCListenerList` methods:

`add()`	Adds an element to the list of listeners.
`remove()`	Removes an element to the list of listeners.
`elements()`	Returns an `Enumeration` object.

22.4 Examples

To add a listener using a `JCListenerList`:

```
JCListenerList someList = null;
...
public synchronized void addSomeListener(SomeListener l) {
    someList = JCListenerList.add(someList, l);
 }
```

To remove a listener:

```
  public synchronized void removeSomeListener(SomeListener l) {
      someList = JCListenerList.remove(someList, l);
  }
```

The use of static methods on the `JCListenerList` class prevents any problems from occurring if the list being modified is null.

To send events to the listener in the list, simply get the `Enumeration` of the list and walk through the elements. There is no ordering guarantee.

23

Progress Helper

23.1 Features of JCProgressHelper

`JCProgressHelper` is a class that lets you create and manage a thread-safe progress dialog. With it, you can monitor some potentially time-consuming operation and present a visual indication of its progress. If it looks like the operation will take some time, a progress dialog appears. Before the operation is started the `JCProgressHelper` should be given a numeric range and a descriptive String. Initially, there is no `JProgressBar`. As the operation progresses, call the `updateProgress()` method to indicate how far along the [*min* .. *max*] range the operation is. After the first `timeToDecideToPopup` milliseconds (default 500) the progress monitor will predict how long the operation will take. If it is longer than `timeToPopup` (default 2 seconds) a `JProgressBar` is popped up.

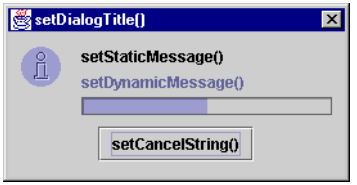


Figure 56 A JCProgressHelper showing the methods used for labelling.

The advantages of `JCProgressHelper` include:

- You are able to quantify the process that the `JCProgressHelper` is monitoring by setting two integers representing a minimum and a maximum of some value that proportionately measures the progress of some time-consuming operation. As the process continues, you call `updateProgress()` with a parameter indicating how far along things are.
- The progress helper transforms all your calls to it into Thread-safe calls to the parent Swing component to encourage frequent updating.
- A descriptive, dynamically updatable, message informs users about the progress of the operation.

- The progress dialog contains a **Cancel** button, permitting the end-user to terminate long-running processes.
- The progress dialog waits for a time that you set in `setTimeToDecideToPopup()` before checking whether to pop up, and does not pop up at all unless the operation is projected to take at least a minimum time, which you may set also, in `setPopupTime()`.

23.2 Constructors and Associated Classes

23.2.1 Constructors

- `JCProgressHelper(Component parent)`
- ```
  JCProgressHelper(Component parent,
                   String static_message,
                   int min,
                   int max)
  ```
- ```
  JCProgressHelper(Component parent,
                   String static_message,
                   int min,
                   int max,
                   boolean show_dynamic_message,
                   boolean is_modal,
                   boolean is_dismissable)
  ```

The parameters in the constructors are:

`parent`	The object whose computations are to be monitored.
`min, max`	Integer parameters that represent a time scale for the operation. The progress bar is scaled between these values. Thus, if `min` = 40 and `max` = 90, and the current value is 50, then (50 - 40)/(90 - 40) = 1/5, and 1/5, that is 20%, of the length of the progress bar is shaded.
`static_message`	The first message line. This message cannot be changed once the dialog has popped up.
`show_dynamic _message`	The changeable part of the message in the line below the static message of the progress dialog can be turned on and off using this Boolean parameter.
`is_modal`	If `true`, the progress dialog must remain the active window.
`is_dismissable`	If `false`, the user cannot cancel the computation that this progress meter is monitoring.

23.2.2 Associated Classes

Following is a list of the JCProgressHelper associated classes:

JCProgressEvent	JCProgressEvent is used to monitor the status of a process. The event contains information about the process name, the current unit being processed, and the total unit count for the process. The setAbort() method allows the listener to abort the process, and the isAborted() method checks to see if the process is aborted.
JCProgress Listener	The listener interface that may be used for processes which are to be monitored by a progress bar. Methods are: processingBegin(), invoked when a process has begun, processingEnd(), invoked when the process has been compeleted, processingError(), invoked when a process encounters an error, and processingUnit(), invoked when a process unit has been completed.
JCProgress CancelledEvent	JCProgressCancelledEvent is used to notify interested listeners when the user has cancelled progress via the **Cancel** button on the JCProgressHelper.
JCProgress CancelledListener	The listener interface which is used to detect a user-cancellation action in the JCProgressHelper. JCProgressCancelledEvent is fired when the user has cancelled progress via the **Cancel** button on the JCProgressHelper. Implementations of this interface may detect the cancellation in their application and react to it.
JCProgressHelper	The progress monitor itself.
JCProgress AbortedException	Used to create an exception that tells the process that it should exit.
JCProgressAdapter	An abstract class that acts as an adapter. It provides null implementations of all the methods defined in JCProgressListener.

23.2.3 Using the Event and Listener Classes

Depending on your needs, there are four ways to use the progress mechanism:

- Use the JCProgressHelper GUI and let it handle all updates without any need to invoke events and listeners in your code. In this case, you call startProgress() and updateProgress() to control the progress bar. See examples.elements.BasicProgressHelperExample.java in the *JCLASS_HOME/examples/elements* directory.

- Use `createProgressListener()` to have `JCProgressHelper` manage events internally. In this case, you call listener methods like `processingBegin()` and `processingUnit()`, which are implemented by the progress meter itself. You don't have to supply the code.
- Create your own `addProgressListener()` method and have it register `JCProgressListener`s. This option gives you the most control. You'll need to have one of your classes implement `JCProgressListener`, or you can extend `JCProgressAdapter`. You override the listener methods you need, and you will have to provide implementations of `addProgressListener(JCProgressListener)`, `removeProgressListener(JCProgressListener)`, and `fireProgressEvent(JCProgressEvent)` in the class that wishes to control the progress meter. See *ProgressListenerExample.java* in the *JCLASS_HOME/examples/elements* directory.
- Use an implementation of `JCProgressCancelledListener` to handle what must be done if the end user clicks the `JCProgressHelper`'s **Cancel** button. A `JCProgressCancelledEvent` is sent to all `JCProgressCancelledListener`s when this button is clicked. If you use this listener your application is notified immediately when the **Cancel** button has been clicked, rather than having to wait until `processingUnit()` is called from `JCProgressListener`.

23.3 JCProgressHelper Methods

`completeProgress()`	Call this method when your computation has finished to allow for cleanup.
`isOkayToContinue()`	Is `false` if the user has pressed the **Cancel** button. You can use it in your code to force a cancellation of the computation.
`setDynamicMessage()`	The changeable part of the progress meter's message.
`setCancelString`	Sets the String in the **Cancel** button. Default is "Cancel".
`setDialogTitle`	Sets the String in the dialog's title bar. Default is "Progress..."
`setMaximum()`	You model the duration of a process by inventing an integer range *min* .. *max*. Choose the range so that it is easy to calculate the integer that represents your computation's degree of completion. Set the maximum-time integer using this method.
`setMinimum()`	Use this method to set the minimum-time integer.
`setRange()`	A convenience method that combines `setMinimum` and `setMaximum`.
`setStaticMessage()`	Use this method to define the unchanging part of the progress meter's message.
`setTimeToDecideTo Popup()`	The progress meter waits until this time before attempting to predict how long the process it is monitoring will take. It then uses the time set in `setTimeToPopup()` to decide whether to pop up at all.
`setTimeToPopup()`	The progress meter won't pop up if the progress meter's calculation estimates that the process will take less time than the time you set here. The default is 500 ms.
`startProgress()`	Call this method within your object to inform the progress meter that timing has begun.
`updateProgress()`	Call this method with a integer parameter that represents your computation's degree of completion.

23.4 Examples

Add a `JCProgressHelper` to your component as follows:

```
JCProgressHelper jpr = new JCProgressHelper(eFrame,
                                "Here is a progress message", 0, 10,
                                                    true, true, true);
```

Set a time which it is unnecessary to display a progress meter and call `setPopupTime()` with this value. Decide when you want the progress helper to calculate its estimation of the monitored process' completion time and call `setTimeToDecideToPopup()` with this time. This time should be long enough that the tracking variable is no longer at its minimum value, so that the progress meter has a way of estimating how long the operation will take to complete.

Call the `updateProgress()` method periodically to update the tracking variable:

```
jpr.updateProgress(5); // Progress indicator is half-way along
```

When the process has completed, call `completeProgress()` to remove the dialog.

A full example

The following code causes a progress dialog to appear. It sets both a static message and a dynamic message that is changed every time the progress bar is updated. Since the dialog is managed in an AWT thread, you may have to make sure that it is given a chance to

run, especially if you wish to perform some initialization of the progress dialog in the mainline thread and you wish it to appear in the dialog.

```
import javax.swing.*;
import com.klg.jclass.util.swing.JCProgressHelper;

public class ProgressHelper extends JFrame {

    // Use the constructor that permits setting a static message
    public ProgressHelper() {
        JCProgressHelper jpr = new JCProgressHelper(this,
                               "Here is a static progress message", 0, 100,
                                                    true, true, false);
    // Pop up the progress dialog.
    // There will be a delay, determined in part
    //   by the two popup time parameters.
    jpr.startProgress();
    for (int j = 1; j < 11; j++){
    // Simulate an ongoing process ...
    try {
        Thread.sleep(1000);
    } catch (Exception e) {
    }
    // ... and update the progress meter periodically,
    jpr.updateProgress(j*10);
    // changing the dynamic message as the meter updates.
    jpr.setDynamicMessage("Dynamic "+ j*10);
    }
    // Dispose of the progress meter
    jpr.completeProgress();
    // The mainline program can continue as required...
    // ... until its tasks are completed.
    System.exit(0);
    }

    public static void main(String[] args) {
        ProgressHelper ph = new ProgressHelper();
    }
}
```

24

String Tokenizer

24.1 Features of JCStringTokenizer

`JCStringTokenizer` provides simple linear tokenization of a `String`. The set of delimiters, which defaults to common whitespace characters, can be specified either during creation or on a per-token basis. It is similar to `java.util.StringTokenizer`, but delimiters can be included as literals by preceding them with a backslash character (the default). It exhibits this useful behavior: if one delimiter immediately follows another, a null String is returned as the token instead of being skipped over.

`JCStringTokenizer` has these capabilities:

- Parses a String using a delimiter you specify.
- Parses a String using the specified delimiter and escape character.
- Counts the number of tokens in the `String` using the specified delimiter.

24.2 Classes

This utility consists of a single class called `JCStringTokenizer`.

Pass the String to be tokenized to the constructor:

```
String s = "Hello my friend";
JCStringTokenizer st = new JCStringTokenizer(s);
```

Process the tokens in the String tokenizer with methods `hasMoreTokens()` and `nextToken()`.

24.3 Methods

These are the methods of `JCStringTokenizer`:

`countTokens()`	Returns the next number of tokens in the `String` using the delimiter you specify.
`getEscapeChar()`	Gets the escape character (default: \).
`getPosition()`	Returns the current scan position within the String.
`hasMoreTokens()`	Used with `nextToken()`. Returns `true` if more tokens exist in the String tokenizer.
`nextToken()`	Gets the next token from the delimited String. If required, the delimiter can be "escaped" by a backslash character. To include a backslash character, precede it by another backslash character.
`nextToken()`	Gets the next whitespace-delimited token.
`parse()`	Given a String a delimiter, and an optional escape character, this method parses the String using the specified delimiter and returns the values in an array of `String`s. Use the second form of the command if you wish to set an escape character different from the default, which is the backslash character.
`setEscapeChar()`	Sets the escape character (default: \). If 0, no escape character is used.

24.4 Examples

At one point, there are two side-by-side commas in the String that is to be split into tokens. The delimiter for tokenization is a comma, so a null is returned as the token in this case. Upon encountering it, `println()` outputs the word "null" as part of the print stream. Note that leading spaces are not stripped from the tokenized word.

```
String token, s = "this, is, a,, test";
JCStringTokenizer st = new JCStringTokenizer(s);
while (st.hasMoreTokens()) {
    token = st.nextToken(',');
    System.out.println(token);  }
```

This prints the following to the console:

```
this
 is
 a
null
 test
```

You can remove the leading spaces by passing each token in turn to another String tokenizer whose delimiter is a space.

In the next example, a slightly longer String is parsed based on the delimiter being the space character. As in the previous example, side-by-side spaces are interpreted as having a null token between them.

```
import com.klg.jclass.util.JCStringTokenizer;

public class StringTokenizerExample {

    public static void main(String args[]){

        String token, s = "this   is a  test of the string " +
                    + "tokenizer called JCStringTokenizer. " +
        "\nThe whitespace between the repeated words is a tab     tab. ";
        System.out.println("First, the string: " + s);
         JCStringTokenizer st = new JCStringTokenizer(s);
         while (st.hasMoreTokens()) {
             token = st.nextToken(' ');
             System.out.println(token);
        }

    }
}
```

This time, the output is:

```
First, the string: this   is a  test of the string tokenizer called
JCStringTokenizer.
The whitespace between the repeated words is a tab      tab.
this
null
null
is
a
null
test
of
the
string
tokenizer
called
JCStringTokenizer.

The
whitespace
between
the
repeated
words
is
a
tab
        tab.
```

25

Thread Safety Utilities

25.1 Features of the Thread Safety Classes

`JCMessageHelper` lets you build a message dialog based on `JOptionPane`. The advantages of the `JCMessageHelper` are:

- `JCMessageHelper` invokes `JOptionPane` in a thread-safe manner.
- You can set an audible indication when the dialog appears as a parameter in the constructor.
- `JCMessageHelper` utilizes `JCSwingRunnable`, an abstract runnable class that provides the `run()` method. You can create an object of this type, and call the `runSafe` method to get it going.

25.2 Methods

JCMessageHelper

`JCMessageHelper` has static methods that resemble those in `JOptionPane`. These are:

`showError()`	Shows an error message, given a `title` and the `message` String as parameters.
`showInformation()`	Parameters are two Strings: `title` and `message`. The message appears in an information dialog.
`showWarning()`	Parameters are two Strings: `title` and `message`. The message appears in a warning dialog.

`showMessage()`	There is an optional first parameter that lets you specify a parent `Component`. The next two parameters are Strings, `title` and `message`. The next parameter is an `int` specifying the message type, and the optional last parameter is a Boolean that specifies whether to emit an audible beep when the dialog appears. The possible message types are
	`javax.swing.JOptionPane.ERROR_MESSAGE`
	`javax.swing.JOptionPane.INFORMATION_MESSAGE`
	`javax.swing.JOptionPane.WARNING_MESSAGE`
	`javax.swing.JOptionPane.QUESTION_MESSAGE`
	`javax.swing.JOptionPane.PLAIN_MESSAGE`
	
	See `javax.swing.JOptionPane` for a description of the various dialogs.

JCSwingRunnable

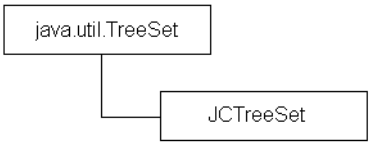
`run()`	Since this class implements `Runnable` and is used to create a thread, its `run()` method will be called to start the thread.

26

Tree Set

26.1 Features of JCTreeSet

This class adds a convenience constructor to `java.util.TreeSet`, which provides you with a way of representing an array of `Object`s as a sorted set. If the elements of the set have a defined ordering principle, it is used by default to construct the tree, or you can provide a `Comparator`. The ordering must be compatible with the conditions for a `java.util.TreeSet`. If your array contains duplicate items, `JCTreeSet` will ensure that only one of them is placed in the sorted set.



`JCTreeSet` adds this convenience constructor:

- Construct a new `JCTreeSet` containing the elements in a specified array, sorted according to the elements' natural ordering principle.

`JCTreeSet` includes the following `TreeSet` standard constructors:

- Construct an empty `TreeSet` using a specified `Collection`, sorted according to the elements' natural ordering principle.
- Construct a new `TreeSet` containing the elements in the specified `Collection`, sorted according to the elements' natural ordering principle.
- Construct a new `JCTreeSet` containing the same elements as the given `SortedSet`, sorted according to the same ordering.

26.2 Constructors and Methods

`JCTreeSet` has constructors that allow you to form a sorted set from the elements of a `Collection`, an array of `Object`s, or a `SortedSet`. A no-parameter constructor lets you

instantiate an empty tree set using natural ordering, or you can supply a `Comparator` to specify how the elements are to be sorted.

`JCTreeSet` defines no methods of its own. Its methods are inherited from `java.util.TreeSet`, `java.util.AbstractSet`, `java.util.AbstractCollection`, and `java.lang.Object`.

26.3 Examples

The example shown here illustrates passing an array of Strings to the constructor. Some Strings are duplicates, but the resulting sorted set contains no duplicates.

```
import com.klg.jclass.util.JCTreeSet;

public class TreeSetExample {

    public static void main(String args[]){
        System.out.println("Starting TreeSetExample");
        String[] items =
             {"moe", "joe", "poe", "zoe", "aoe", "poe", "joe", "moe"};
        JCTreeSet ts = new JCTreeSet(items);
        System.out.println("The number of items in the array is: "
                                                    + items.length);
        System.out.println("The number of items in JCTreeSet "
                                          + ts + " is: " + ts.size());
        System.out.println("The last element of " + ts + " is: "
                                                      + ts.last());
    }
}
```

The output of the program is:

```
Starting TreeSetExample
The number of items in the array is: 8
The number of items in JCTreeSet [aoe, joe, moe, poe, zoe] is: 5
The last element of [aoe, joe, moe, poe, zoe] is: zoe
```

27

Type Converters

27.1 Features of JCTypeConverter

There is frequently a need to convert objects to Strings and Strings to objects when you are coding a user interface. For example, a user types a String as input that you would like to convert to an object. The input String might consist of a sequence of integers, delimited by commas, that you would like to convert to an array. There are also times when you need to convert an object to a String so that you can place the text on a label or a button. The JClass type converters are a collection of the most useful conversions from String to object, and from object to String. The static methods of `JCTypeConverter` let you retrieve parameters from an application or applet and convert these parameters to particular data types.

`JCTypeConverter` performs these functions:

- Returns a trimmed String, with trailing nulls removed.
- Converts a String to an integer.
- Converts a String to a double.
- Converts a String to a Boolean.
- Converts a String to an array of `Strings`.
- Converts a String to an array of integers or `Integer` objects.
- Converts a String to an array of `Double` objects.
- Converts all occurrences of `"\n"` to newlines.
- Converts all occurrences of `char '\n'` to `String "\n"`.
- Converts a delimited list of `String` tokens to a `Vector`.
- Converts a String to an enum, or a list of enums.
- Converts an enum to a String.
- Converts an object to a String.
- Converts a `String` to a `Date`.

- Removes “escape” characters (backslashes) from the String.
- Allows parsing errors to be printed or shown in a dialog.

27.2 Features of JCSwingTypeConverter

`JCSwingTypeConverter` can perform these functions:

- Converts a String to a `Color`, or an array of `Colors`.
- Converts color to one of the `Color` enums, or RGB format.
- Converts list to a comma-separated list of tokens.
- Converts a font name to a font instance, or a `Font` to a *name-style-size* `String`, or a `String` like *Helvetica-plain-10* to a `Font`.
- Converts a String to an `Insets` instance, or creates a `String` from an AWT `Insets` value.
- Converts a String to a `Dimension` instance.
- Converts a String to a `Point` instance.

27.3 Classes

The two type converter classes are `com.klg.jclass.util.JCTypeConverter` and `com.klg.jclass.util.swing.JCSwingTypeConverter`. Both contain static methods for converting from one standard type to another. `JCTypeConverter` is for converting Java types, and `JCSwingTypeConverter` is for Swing types.

27.4 Methods

JCTypeConverter

`JCTypeConverter` contains static methods for retrieving parameters from a source file or applet, and for converting parameters to particular data types.

The methods in `JCTypeConverter` are:

`checkEnum()`	Checks the validity of an enum.
`error()`	Writes a parse error message to the standard output device.
`fromEnum()`	Converts an enum to a String.
`fromNewLine()`	Converts all occurrences of `char '\n'` to String `"\n"`
`removeEscape()`	Removes escape characters (backslashes) from the String.

`toBoolean()`	Converts a String to a Boolean. The method takes two parameters: the String representation of the Boolean, and a `boolean` default value to use if a parse error occurs.
`toDate()`	Converts a `String` to a `Date`.
`toDouble()`	Converts a String to a double. The method takes two parameters: the String representation of the number, and a `Double` default value to use if a parse error occurs.
`toDoubleList()`	Converts a String to an array of `Double` objects based on the provided delimiter. An optional third parameter is the default value, returned if a parse error occurs.
`toEnum()`	Converts a String to an enum. If the String cannot be converted, an error message is written to the console. The first three of its six parameters are the String to be converted, the enum type specified as a String, and a PARAM name for the enum (used in an error message). The next two are two-dimensional arrays that link enum types and their corresponding values. The last parameter is the value that should be returned it the Strings cannot be converted. The method has other signatures as well. See the API for details.
`toEnumList()`	Converts a String to a list of enums. If the String cannot be converted, an error message is written to the console.
`toInt()`	Converts a String to an integer. The method takes two parameters: the String representation of the integer, and an integer default value to use if a parse error occurs.
`toIntegerList()`	Converts a String to an array of `Integer`s based on the provided delimiter. An optional third parameter is the default value, returned if a parse error occurs.
`toIntList()`	Converts a String to an array of `Integer` objects based on the provided delimiter. An optional third parameter is the default value, returned if a parse error occurs.
`toNewLine()`	Converts all occurrences of "\n" to newlines.
`toString()`	Converts an object to a String. If a `String`, newlines are replaced by "\n". If a `Vector`, it is converted to a comma-separated list.
`toStringList()`	Converts a String to an array of `String`s. There are three signatures: (a) a comma-separated String, (b) a String and a delimiter, and (c) the String, delimiter, and a Boolean indicating whether the String should be trimmed. Strings are trimmed by default.

`toVector()`	Converts a delimited list of tokens to a `Vector`. The second parameter is the delimiter that separates the tokens in the String. An optional third parameter is the default value, returned if a parse error occurs.
`trim()`	Returns a trimmed String, with trailing nulls removed.

JCSwingTypeConverter

The methods in `JCSwingTypeConverter` are:

`toInsets()`	Converts a String to an `Insets` instance.
`fromInsets()`	Creates a String from an AWT `Insets` value.
`toDimension()`	Converts a String in the form "40x30" to a `Dimension` instance.
`toPoint()`	Converts a String to a `Point` instance.
`toColorList()`	Converts a String to an array of `Color`s. An optional second parameter allows you to specify a default `Color` list if an error occurs while parsing the String.
`toColor()`	Converts a String to a `Color`. An optional second parameter allows you to specify a default `Color` if an error occurs while parsing the String.
`fromColorList()`	Converts list to a comma-separated list of tokens, to one of the `Color` enums, or to RGB format.
`toFont()`	Converts a font name to a font instance, or a font name in format "name-style-size", for instance, "Helvetica-plain-10."
`fromFont()`	Returns font in format "name-style-size."

27.5 Examples

The following example gives you an indication of how the static methods in `JCSwingTypeConverter` can be used.

```
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Color;
import com.klg.jclass.util.swing.JCSwingTypeConverter;
```

```
class SwingTypeConverterExamples {

    public static void main(String[] args){

System.out.println("++++++++++++++++++++++++++++++++++++++++++++++++++");
        String s;
        Font f = new Font("System", 10, 10);
        s = JCSwingTypeConverter.fromFont(f);
        System.out.println("The name of the font is " + s);


System.out.println("++++++++++++++++++++++++++++++++++++++++++++++++++");

        String colors = "red, blue, green";
        Color[] colorarray = JCSwingTypeConverter.toColorList(colors,
                                                              null);
        for (int i=0; i<colorarray.length; i++)
            System.out.println("The array of colors is: " +
colorarray[i].toString(
System.out.println("++++++++++++++++++++++++++++++++++++++++++++++++++");

        Color[] mycolors = JCSwingTypeConverter.toColorList(new String
                                            ("black, blue, cyan"));
        for (int i=0; i<mycolors.length; i++)
            System.out.println("The Color array is: " +
                                           mycolors[i].toString());


System.out.println("++++++++++++++++++++++++++++++++++++++++++++++++++");
        Color yourcolor = JCSwingTypeConverter.toColor("darkGray",
                                                       Color.gray);
        System.out.println("The color is: " + yourcolor.toString());


System.out.println("++++++++++++++++++++++++++++++++++++++++++++++++++");
       Dimension dim = JCSwingTypeConverter.toDimension("40x30", null);
       System.out.println("The dimension is: " + dim.toString());

        }
}
```

The output of this program is:

```
++++++++++++++++++++++++++++++++++++++++++++++++++
The name of the font is System-PLAIN-10
++++++++++++++++++++++++++++++++++++++++++++++++++
The array of colors is: java.awt.Color[r=255,g=0,b=0]
The array of colors is: java.awt.Color[r=0,g=0,b=255]
The array of colors is: java.awt.Color[r=0,g=255,b=0]
++++++++++++++++++++++++++++++++++++++++++++++++++
The Color array is: java.awt.Color[r=0,g=0,b=0]
The Color array is: java.awt.Color[r=0,g=0,b=255]
The Color array is: java.awt.Color[r=0,g=255,b=255]
++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
The color is: java.awt.Color[r=64,g=64,b=64]
++++++++++++++++++++++++++++++++++++++++++++++++++++
The dimension is: java.awt.Dimension[width=40,height=30]
```

The static methods of `JCTypeConverter` are called in a similar fashion, as illustrated next.

```
import java.util.Date;
import java.text.DateFormat;
import com.klg.jclass.util.JCTypeConverter;

class TypeConverterExamples {

    public static void main(String[] args){
      System.out.println("++++++++++++++++++++++++++++++++++++++++");
        String s = "10.777";
        double dd = 10;
        double d = JCTypeConverter.toDouble(s, dd);
      System.out.println("The value of the double is: " + d);

      System.out.println("++++++++++++++++++++++++++++++++++++++++");

        s = "Abel, Ben, Curry,                    Dave";
        String[] sa = JCTypeConverter.toStringList(s, ',', true);
        for (int i=0; i<sa.length; i++)
            System.out.println("The array element is: " + sa[i]);
      System.out.println("++++++++++++++++++++++++++++++++++++++++");

        s = "1, 1, 2, 3, 5, 8, 13";
        int [] da = {1,1,1,1,1,1,1};
        int[] ii = JCTypeConverter.toIntList(s, ',', da);
        for (int i=0; i<ii.length; i++)
            System.out.println("The Integer array element is: " +
                                                          ii[i]);

       System.out.println("++++++++++++++++++++++++++++++++++++++++");
        s = "Feb 30, 2000";
        Date today = new Date("June 12, 1999");
        Date myDate = JCTypeConverter.toDate(s, today);
        System.out.println("The date is: " + myDate.toString());

        }
    }
```

Here is the output:

```
DOS:   %JAVA_HOME%\bin\java TypeConverterExamples
++++++++++++++++++++++++++++++++++++++++
The value of the double is: 10.777
++++++++++++++++++++++++++++++++++++++++
The array element is: Abel
The array element is: Ben
The array element is: Curry
The array element is: Dave
++++++++++++++++++++++++++++++++++++++++
The Integer array element is: 1
The Integer array element is: 1
The Integer array element is: 2
```

```
The Integer array element is: 3
The Integer array element is: 5
The Integer array element is: 8
The Integer array element is: 13
++++++++++++++++++++++++++++++++++++++++
The date is: Sat Jun 12 00:00:00 EDT 1999
```

28

Word Wrap

28.1 Features of JCWordWrap

`JCWordWrap` provides a static method called `wrapText()` that performs basic word-wrap logic on a `String`, given a line width in pixels and a delimiter to insert just before the line width is reached. While the delimiter is most often a newline, it can be any `String`.

The resulting String produced by `JCWordWrap` has lines no longer than the line width supplied as one of the parameters. Since the width of a line is measured in pixels, the number of words in a line depends on the `FontMetrics` currently in effect.

The other static method in this class is `replace()`. Its parameters are three `Strings`. The first parameter is the text String to be searched for occurrences of the second parameter. Any such occurrences are replaced with the third parameter.

28.2 Methods

Following is a list of the `JCWordWrap` methods:

`wrapText()`	This static method returns a word-wrapped `String`, given an input String, a `FontMetrics` object, a line width in pixels, a delimiter such as a newline, and a Boolean to indicate whether left-alignment is in effect. Word-wrap logic breaks lines by spaces, but provides no hyphenation logic. The original `String` is returned if the number of characters is less than 10.
`replace()`	Returns a String stripped of a delimiter, or replaces one String with another.

28.3 Examples

The code fragment shown here takes one rather long text String and constructs a reformatted one by adding newlines every so often. The new String `s` inserts newlines so that the lines never exceed 100 pixels, based on the current font.

```
...
FontMetrics fm = g.getFontMetrics(f);

String text = "It has flown away";
text += "The nightingale that called ";
text += "Waking me at midnight ";
text += "Yet its song seems "
text += "Still by my pillow.";

s = JCWordWrap.wrapText(text, fm, 100, delimiter, true);
Figure 57It has flown away
        The nightingale that
        called Waking me at
        midnight Yet its song
        seems Still by my
        pillow.
```

If the length for word wrapping is decreased to 50 pixels,

```
s = JCWordWrap.wrapText(text, fm, 50, delimiter, true);
```

The output String `s` is formatted as shown:

```
It has
flown
away The
nightingale
that called
Waking
me at
midnight
Yet its
song
seems
Still by my
pillow.
```

Taking this second case, use `replace()` to put the word -STOP- in place of a newline:

```
String s1 = JCWordWrap.replace(s, "\n", "-STOP-");
```

This would yield:

```
It has -STOP-flown -STOP-away The -STOP-nightengale -STOP-that called -
STOP-Waking -STOP-me at -STOP-midnight -STOP-Yet its -STOP-song -STOP-seems
-STOP-Still by my -STOP-pillow.
```

Part III

Reference Appendices

Appendix A

Bean Properties Reference

Beans in the Swing Package ■ *Beans in the com.klg.jclass.util.swing Package*

The following is a listing of the JClass Elements Bean properties and their default values. The properties are arranged alphabetically by property name. The second entry on any given row names the data type returned by the method. Note that a small number of properties are really read-only variables, and therefore only have a *get* method. These properties are marked with a "(G)" following the property name.

A.1 Beans in the Swing Package

A.1.1 Properties of JCCircularGauge

Property	Type	DefaultValue
autoTickGeneration	Boolean	True
centerColor	java.awt.Color	0,0,0
centerRadius	double	0.1
direction	int	DIRECTION_COUNTERCLOCKWISE
drawTickLabels	Boolean	True
drawTickMarks	Boolean	True
needleColor	java.awt.Color	(null)
needleInteractionType	int	INTERACTION_NONE
needleStyle	int	NEEDLE_ARROW
needleValue	double	0.0
needleWidth	double	15.0
paintCompleteBackground	Boolean	False
precision	int	0

Property	Type	DefaultValue
scaleColor	java.awt.Color	255,255,255
scaleExtent	double	1.0
scaleMax	double	100.0
scaleMin	double	0.0
snapToValue	Boolean	False
startAngle	double	0.0
stopAngle	double	360.0
tickColor	java.awt.Color	0,0,0
tickFont	java.awt.Font	(null)
tickFontColor	java.awt.Color	0,0,0
tickIncrement	double	10.0
tickInnerExtent	double	0.85
tickLabelExtent	double	0.8
tickOuterExtent	double	1.0
tickStartValue	double	0.0
tickStopValue	double	100.0
tickStyle	int	TICK_LINE
tickWidth	double	2.0
type	int	TYPE_FULL_CIRCLE
useDefaultPrecision	Boolean	True

A.1.2 Properties of JCLinearGauge

Property	Type	DefaultValue
autoTickGeneration	Boolean	True
direction	int	DIRECTION_COUNTERCLOCKWISE
drawTickLabels	Boolean	True
drawTickMarks	Boolean	True
needleColor	java.awt.Color	(null)

Property	Type	DefaultValue
needleInteractionType	int	INTERACTION_NONE
needleStyle	int	NEEDLE_ARROW
needleValue	double	0.0
needleWidth	double	15.0
precision	int	0
scaleColor	java.awt.Color	255,255,255
scaleExtent	double	1.0
scaleMax	double	100.0
scaleMin	double	0.0
snapToValue	Boolean	False
startAngle	double	0.0
stopAngle	double	360.0
tickColor	java.awt.Color	0,0,0
tickFont	java.awt.Font	(null)
tickFontColor	java.awt.Color	0,0,0
tickIncrement	double	10.0
tickInnerExtent	double	0.85
tickLabelExtent	double	0.8
tickOuterExtent	double	1.0
tickStartValue	double	0.0
tickStopValue	double	100.0
tickStyle	int	TICK_LINE
tickWidth	double	2.0
useDefaultPrecision	Boolean	True

A.1.3 Properties of JCMultiSelectList

Property	Type	Default Value
about (G)	String	com.klg.jclass.swing x.x.x

Property	Type	Default Value
alignmentX	float	0.5
alignmentY	float	0.5
background	Color	(null)
border	javax.swing.border.Border	(null)
debugGraphicsOptions	int	0
doubleBuffered	Boolean	false
enabled	Boolean	true
fixedCellHeight	int	-1
fixedCellWidth	int	-1
font	Font	(null)
foreground	Color	(null)
maximumSize	Dimension	[32767, 32767]
minimumSize	Dimension	[153, 42]
model	javax.swing.ListModel	SetValue
name	String	(null)
nextFocusableComponent	Component	(null)
opaque	Boolean	false
preferredSize	Dimension	[232, 200]
prototypeCellValue	java.lang.Object	(null)
requestFocusEnabled	Boolean	true
selectionBackground	Color	204,204,255
selectionForeground	Color	0,0,0
toolTipText	String	(null)

A.1.4 Properties of JCMDIFrame

Property	Type	Default Value
about (G)	String	com.klg.jclass.swing x.x.x

Property	Type	Default Value
alignmentX	float	0.5
alignmentY	float	0.5
background	Color	(null)
border	javax.swing.border.Border	javax.swing.plaf.metal.MetalBorders$InternalFrameBorder
debugGraphicsOptions	int	0
doubleBuffered	Boolean	false
enabled	Boolean	true
font	Font	(null)
foreground	Color	(null)
maximumSize	Dimension	[MAX_VALUE, MAX_VALUE]
minimumSize	Dimension	[120, 24]
name	String	(null)
nextFocusableComponent	Component	(null)
opaque	Boolean	false
preferredSize	Dimension	[10, 34]
requestFocusEnabled	Boolean	true
toolTipText	String	(null)

A.1.5 Properties of JCMDIPane

Property	Type	Default Value
about (G)	String	com.klg.jclass.swing x.x.x
alignmentX	float	0.5
alignmentY	float	0.5
background	Color	153,153,204
border	javax.swing.border.Border	(null)

Property	Type	Default Value
considerIconsWhen Tiling	boolean	false
debugGraphicsOptions	int	0
doubleBuffered	Boolean	false
dragMode	int	LIVE_DRAG_MODE (corresponds to DEFAULT for frameManuipulationStyle)
enabled	Boolean	true
font	Font	(null)
foreground	Color	(null)
frameManipulationStyle	int	DEFAULT
maximumSize	Dimension	[100, 100]
minimumSize	Dimension	[300, 200]
name	String	(null)
nextFocusableComponent	Component	(null)
opaque	Boolean	true
preferredSize	Dimension	[300, 200]
requestFocusEnabled	Boolean	true
toolTipText	String	(null)

A.1.6 Properties of JCTreeExplorer

Property	Type	Default Value
about (G)	String	com.klg.jclass.swing x.x.x
alignmentX	float	0.0
alignmentY	float	0.0
background	Color	(null)

Property	Type	Default Value
border	javax.swing.border.Border	javax.swing.plaf.basic.BasicBorders$SplitPaneBorder
debugGraphicsOptions	int	0
doubleBuffered	Boolean	false
enabled	Boolean	true
font	Font	(null)
foreground	Color	(null)
maximumSize	Dimension	[MAX_VALUE, MAX_VALUE]
minimumSize	Dimension	[46, 24]
name	String	(null)
nextFocusableComponent	Component	(null)
opaque	Boolean	false
preferredSize	Dimension	[515, 405]
requestFocusEnabled	Boolean	true
toolTipText	String	(null)

A.1.7 Properties of JCTreeTable

Property	Type	Default Value
about (G)	String	com.klg.jclass.swing x.x.x
alignmentX	float	0.5
alignmentY	float	0.5
autoSort	Boolean	true
background	Color	255,255,255
border	javax.swing.border.Border	(null)
debugGraphicsOptions	int	0
doubleBuffered	Boolean	false
enabled	Boolean	true

Property	Type	Default Value
font	Font	null
foreground	Color	0,0,0
maximumSize	Dimension	[0, 20]
minimumSize	Dimension	[0, 20]
name	String	(null)
nextFocusableComponent	Component	(null)
opaque	Boolean	true
preferredSize	Dimension	[0, 20]
requestFocusEnabled	Boolean	true
rootVisible	Boolean	true
scrollsOnExpand	Boolean	true
showNodeLines	int	Use Plaf
showsRootHandles	Boolean	false
sortable	Boolean	true
toolTipText	String	(null)
view	int	Tree

A.1.8 Properties of JCWizard

Property	Type	Default Value
about (G)	String	com.klg.jclass.swing x.x.x
alignmentX	float	0.5
alignmentY	float	0.5
background	Color	204,204,204
border	javax.swing.border.Border	(null)
debugGraphicsOptions	int	0
doubleBuffered	Boolean	true
enabled	Boolean	true

Property	Type	Default Value
font	Font	null
foreground	Color	0,0,0
maximumSize	Dimension	[MAX_VALUE, MAX_VALUE]
minimumSize	Dimension	[0, 0]
name	String	(null)
nextFocusableComponent	Component	(null)
opaque	Boolean	true
preferredSize	Dimension	[0, 0]
requestFocusEnabled	Boolean	true
toolTipText	String	(null)

A.1.9 Properties of JCWizardPage

Property	Type	Default Value
about (G)	String	com.klg.jclass.swing x.x.x
alignmentX	float	0.0
alignmentY	float	0.0
background	Color	204,204,204
border	javax.swing.border.Border	(null)
debugGraphicsOptions	int	0
doubleBuffered	Boolean	true
enabled	Boolean	true
font	Font	null
foreground	Color	0,0,0
maximumSize	Dimension	[33174, 131068]
minimumSize	Dimension	[407, 37]
name	String	VerticalBox0
nextFocusableComponent	Component	(null)

Property	Type	Default Value
opaque	Boolean	true
preferredSize	Dimension	[407, 39]
requestFocusEnabled	Boolean	true
toolTipText	String	(null)

A.2 Beans in the com.klg.jclass.util.swing Package

A.2.1 Properties of JCBox

Property	Type	Default Value
about	String	com.klg.jclass.util x.x.x
alignment	int	Top
alignmentX	float	0.0
alignmentY	float	0.0
background	Color	204,204,204
border	javax.swing.border.Border	(null)
debugGraphicsOptions	int	0
doubleBuffered	Boolean	True
enabled	Boolean	True
font	java.awt.Font	null
foreground	Color	0,0,0
maximumSize	Dimension	[0, 0]
minimumSize	Dimension	[0, 0]
name	String	HorizontalBox0
nextFocusableComponent	Component	(null)
opaque	Boolean	True
orientation	int	Horizontal
preferredSize	Dimension	[0,0]

Property	Type	Default Value
requestFocusEnabled	Boolean	True
toolTipText	String	(null)

A.2.2 Properties of JCBrace

Property	Type	Default Value
about	String	com.klg.jclass.util x.x.x
alignmentX	float	0.5
alignmentY	float	0.5
background	Color	(null)
doubleBuffered	Boolean	False
enabled	Boolean	True
font	java.awt.Font	(null)
foreground	Color	(null)
length	int	10
maximumSize	Dimension	[10, 32767]
minimumSize	Dimension	[10, 0]
name	String	(null)
opaque	Boolean	True
orientation	int	Horizontal
preferredSize	Dimension	[10, 0]

A.2.3 Properties of JCCheckBoxList

Property	Type	Default Value
about	String	com.klg.jclass.util x.x.x
alignmentX	float	0.5
alignmentY	float	0.5
background	Color	255,255,255

Property	Type	Default Value
border	javax.swing.border.Border	(null)
debugGraphicsOptions	int	0
doubleBuffered	Boolean	False
enabled	Boolean	True
fixedCellHeight	int	-1
fixedCellWidth	int	-1
font	java.awt.Font	null
foreground	Color	0,0,0
maximumSize	Dimension	[0, 0]
minimumSize	Dimension	[0, 0]
model	javax.swing.ListModel	SetValue
null		
name	String	(null)
nextFocusableComponent	Component	(null)
opaque	Boolean	True
preferredSize	Dimension	[0, 0]
prototypeCellValue	java.lang.Object	(null)
requestFocusEnabled	Boolean	True
selectionBackground	Color	204,204,255
selectionForeground	Color	0,0,0
selectionMode	int	Multiple Interval
toolTipText	String	(null)
visibleRowCount	int	8

A.2.4 Properties of JCExitFrame

Property	Type	Default Value
about	String	com.klg.jclass.util x.x.x

Property	Type	Default Value
alignmentX	float	0.5
alignmentY	float	0.5
background	Color	204,204,204
doubleBuffered	Boolean	False
enabled	Boolean	True
exitOnClose	Boolean	True
font	java.awt.Font	(null)
foreground	Color	(null)
maximumSize	Dimension	[MAX_VALUE, MAX_VALUE]
minimumSize	Dimension	[0, 0]
name	String	frame0
opaque	Boolean	True
preferredSize	Dimension	[0, 0]

A.2.5 Properties of JCFontChooserBar

Property	Type	Default Value
UIClassID	String	(null)
accessibleContext	javax.accessibility.AccessibleContext	(null)
alignmentX	float	(null)
alignmentY	float	(null)
autoscrolls	Boolean	(null)
background	Color	(null)
border	javax.swing.border.Border	(null)
component	(null)	null
componentCount	int	(null)
components	Component[]	(null)
debugGraphicsOptions	int	(null)

Property	Type	Default Value
doubleBuffered	Boolean	(null)
enabled	Boolean	(null)
focusCycleRoot	Boolean	(null)
focusTraversable	Boolean	(null)
font	java.awt.Font	(null)
foreground	Color	(null)
graphics	Graphics	(null)
height	int	(null)
insets	java.awt.Insets	(null)
layout	java.awt. LayoutManager	(null)
managingFocus	Boolean	(null)
maximumSize	Dimension	(null)
minimumSize	Dimension	(null)
name	String	(null)
nameList	String[]	(null)
nextFocusableComponent	Component	(null)
opaque	Boolean	(null)
optimizedDrawingEnabled	Boolean	(null)
paintingTile	Boolean	(null)
preferredSize	Dimension	(null)
registeredKeyStrokes	javax.swing. KeyStroke[]	(null)
requestFocusEnabled	Boolean	(null)
rootPane	javax.swing. JRootPane	(null)
selectedFont	java.awt.Font	(null)
toolTipText	String	(null)
topLevelAncestor	java.awt.Container	(null)

Property	Type	Default Value
underline	Boolean	(null)
validateRoot	Boolean	(null)
visible	Boolean	(null)
visibleRect	java.awt.Rectangle	(null)
width	int	(null)
x	int	(null)
y	int	(null)

A.2.6 Properties of JCFontChooserPane

Property	Type	Default Value
about	String	com.klg.jclass.util x.x.x
alignmentX	float	0.5
alignmentY	float	0.5
background	Color	(null)
border	javax.swing.border. Border	(null)
debugGraphicsOptions	int	0
doubleBuffered	Boolean	False
enabled	Boolean	True
font	java.awt.Font	(null)
foreground	Color	(null)
maximumSize	Dimension	[32767, 32767]
minimumSize	Dimension	[274, 236]
name	String	(null)
nextFocusableComponent	Component	(null)
opaque	Boolean	False
preferredSize	Dimension	[274, 369]
requestFocusEnabled	Boolean	True

Property	Type	Default Value
selectedFont	java.awt.Font	null
styleControls	int	null
toolTipEnabled	Boolean	True
toolTipText	String	(null)

A.2.7 Properties of JCHelpPane

Property	Type	Default Value
about	String	com.klg.jclass.util x.x.x
alignmentX	float	0.0
alignmentY	float	0.0
background	Color	(null)
border	javax.swing.border.Border	javax.swing.plaf.basic.BasicBorders$SplitPaneBorder@16b328a8
contentsPage	java.net.URL	(null)
debugGraphicsOptions	int	0
doubleBuffered	Boolean	False
enabled	Boolean	True
font	java.awt.Font	(null)
foreground	Color	(null)
maximumSize	Dimension	[MAX_VALUE, MAX_VALUE]
minimumSize	Dimension	[46, 24]
name	String	(null)
nextFocusableComponent	Component	(null)
opaque	Boolean	False
preferredSize	Dimension	(null)
requestFocusEnabled	Boolean	True
titlePage	java.net.URL	(null)
toolTipText	String	(null)

Property	Type	Default Value
useToolBar	Boolean	True
viewPage	java.net.URL	(null)

A.2.8 Properties of JCHTMLPane

Property	Type	Default Value
about	String	com.klg.jclass.util x.x.x
alignmentX	float	0.5
alignmentY	float	0.5
background	Color	255,255,255
border	javax.swing.border.Border	javax.swing.plaf.basic.BasicBorders $MarginBorder instance
debugGraphicsOptions	int	0
doubleBuffered	Boolean	False
enabled	Boolean	True
font	java.awt.Font	null
foreground	Color	0,0,0
maximumSize	Dimension	[MAX_VALUE, MAX_VALUE]
minimumSize	Dimension	(null)
name	String	(null)
nextFocusableComponent	Component	(null)
opaque	Boolean	True
page	java.net.URL	(null)
preferredSize	Dimension	(null)
requestFocusEnabled	Boolean	True
toolTipText	String	(null)

A.2.9 Properties of JCSortableTable

Property	Type	Default Value
about	String	com.klg.jclass.util x.x.x
alignmentX	float	0.5
alignmentY	float	0.5
autoSort	Boolean	False
background	Color	255,255,255
border	javax.swing.border.Border	(null)
debugGraphicsOptions	int	0
doubleBuffered	Boolean	False
enabled	Boolean	True
font	java.awt.Font	null
foreground	Color	0,0,0
maximumSize	Dimension	[0, 0]
minimumSize	Dimension	[0, 0]
name	String	(null)
nextFocusableComponent	Component	(null)
opaque	Boolean	True
preferredSize	Dimension	[0, 0]
requestFocusEnabled	Boolean	True
toolTipText	String	(null)

A.2.10 Properties of JCSpinBox

Property	Type	Default Value
about	String	com.klg.jclass.util x.x.x
actionCommand	String	spinBoxChanged
alignmentX	float	0.5
alignmentY	float	0.5

Property	Type	Default Value
arrowKeySpinningAllowed	Boolean	True
background	Color	(null)
border	javax.swing.border.Border	(null)
debugGraphicsOptions	int	0
doubleBuffered	Boolean	False
editable	Boolean	True
enabled	Boolean	True
font	java.awt.Font	(null)
foreground	Color	(null)
maximumSize	Dimension	[32767, 32767]
minimumSize	Dimension	[0, 0]
model	com.klg.jclass.util.swing.JCSpinBoxModel	null
name	String	(null)
nextFocusableComponent	Component	(null)
opaque	Boolean	False
preferredSize	Dimension	[24, 21]
requestFocusEnabled	Boolean	True
selectedIndex	int	-1
toolTipText	String	(null)

A.2.11 Properties of JCSpinNumberBox

Property	Type	Default Value
about	String	com.klg.jclass.util x.x.x
alignmentX	float	0.5
alignmentY	float	0.5
arrowKeySpinningAllowed	Boolean	True
background	Color	(null)

Property	Type	Default Value
border	javax.swing.border.Border	(null)
debugGraphicsOptions	int	0
doubleBuffered	Boolean	False
editable	Boolean	True
enabled	Boolean	True
font	java.awt.Font	(null)
foreground	Color	(null)
maximumSize	Dimension	[32767, 32767]
minimumSize	Dimension	[0, 0]
name	String	(null)
nextFocusableComponent	Component	(null)
opaque	Boolean	False
operation	int	Integer
preferredSize	Dimension	[173, 21]
requestFocusEnabled	Boolean	True
spinStep	java.lang.Number	1.0
toolTipText	String	(null)
value	java.lang.Number	(null)

A.2.12 Properties of JCSpring

Property	Type	Default Value
about	String	com.klg.jclass.util x.x.x
alignmentX	float	0.5
alignmentY	float	0.5
background	Color	(null)
doubleBuffered	Boolean	False
enabled	Boolean	True

Property	Type	Default Value
font	java.awt.Font	(null)
foreground	Color	(null)
horizontalElasticity	int	1
maximumSize	Dimension	[32767, 32767]
minimumSize	Dimension	[0, 0]
name	String	BidirecionalSpring0
opaque	Boolean	True
preferredSize	Dimension	[0, 0]
verticalElasticity	int	1

A.2.13 Properties of JCDateChooser

Property	Type	Default Value
about	java.lang.String	com.klg.jclass.util x.x.x Preview
alignmentX	float	0.0
alignmentY	float	0.0
background	java.awt.Color	204,204,204
border	javax.swing.border.Border	(null)
days	String[]	locale default array
debugGraphicsOptions	int	0
doubleBuffered	Boolean	True
enabled	Boolean	True
font	java.awt.Font	null
foreground	java.awt.Color	0,0,0
maximumDate	java.util.Calendar	(null)
maximumSize	java.awt.Dimension	java.awt.Dimension [width=203, height=231]
minimumDate	java.util.Calendar	(null)

Property	Type	Default Value
minimumSize	java.awt.Dimension	java.awt.Dimension [width=90, height=54]
months	String[]	locale default array
name	java.lang.String	(null)
nextFocusableComponent	java.awt.Component	(null)
opaque	Boolean	True
preferredSize	java.awt.Dimension	java.awt.Dimension [width=203, height=231]
requestFocusEnabled	Boolean	True
shortMonths	String[]	locale default array
toolTipText	java.lang.String	(null)
value	java.util.Calendar	null

Appendix B

Distributing Applets and Applications

B.1 Using 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 *jchigrid.jar*, which you may have used to develop an 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 included with the JClass DesktopViews suite of products. For more details please refer to *Quest Software's Web site*.

Appendix C
Colors and Fonts

This section provides information on common colorname values, specific RGB color values and fonts applicable to all Java programs. You may find it useful as a guide for choosing colors for cells.

C.1 Colorname Values

The following lists all the colornames that can be used within Java programs. The majority of these colors will appear the same (or similar) across different computing platforms.

- black
- blue
- cyan
- darkGray
- darkGrey
- gray
- grey
- green
- lightGray
- lightGray
- lightBlue
- magenta
- orange
- pink
- red
- white
- yellow

C.2 RGB Color Values

The following lists all the main RGB color values that can be used within JClass Elements. RGB color values are specified as three numeric values representing the red, green and blue color components; these values are separated by dashes ("-").

The following RGB color values describe the colors available to Unix systems. It is recommended that you test these color values in a JClass program on a Windows or Macintosh system before utilizing them.

The list begins with all of the variations of white, then blacks and grays, and then describes the full color spectrum ranging from reds to violets.

Example code from an HTML file:

```
<PARAM NAME=backgroundList VALUE="(4, 5 255-255-0)">
```

RGB Value	Description
255-250-250	Snow
248-248-255	Ghost White
245-245-245	White Smoke
220-220-220	Gainsboro
255-250-240	Floral White
253-245-230	Old Lace
250-240-230	Linen
250-235-215	Antique White
255-239-213	Papaya Whip
255-235-205	Blanched Almond
255-228-196	Bisque
255-218-185	Peach Puff
255-222-173	Navajo White
255-228-181	Moccasin
255 248-220	Cornsilk
255-255-240	Ivory
255-250-205	Lemon Chiffon
255-245-238	Seashell
240-255-240	Honeydew
245-255-250	Mint Cream
240-255-255	Azure
240-248-255	Alice Blue
230-230-250	Lavender
255-240-245	Lavender Blush
255-228-225	Misty Rose

RGB Value	Description
255-255-255	White
0-0-0	Black
47-79-79	Dark Slate Grey
105-105-105	Dim Gray
112- 128-144	Slate Grey
119- 136-153	Light Slate Grey
190- 190-190	Grey
211- 211-211	Light Gray
25-25-112	Midnight Blue
0-0-128	Navy Blue
100- 149 237	Cornflower Blue
72-61-139	Dark Slate Blue
106-90-205	Slate Blue
123- 104 238	Medium Slate Blue
132-112- 255	Light Slate Blue
0-0-205	Medium Blue
65-105-225	Royal Blue
0-0-255	Blue
30-144-255	Dodger Blue
0-19 -255	Deep Sky Blue
135-206-235	Sky Blue
135-206-250	Light Sky Blue
70-130-180	Steel Blue
176-196- 222	Light Steel Blue
173-216-230	Light Blue
176-224-230	Powder Blue
175-238-238	Pale Turquoise
0-206-209	Dark Turquoise
72-209-204	Medium Turquoise
64-224-208	Turquoise
0-255-255	Cyan
224-255-255	Light Cyan

RGB Value	Description
95-158-160	Cadet Blue
102-205-170	Medium Aquamarine
127-255-212	Aquamarine
0-100-0	Dark Green
85-107-47	Dark Olive Green
143-188-143	Dark Sea Green
46-139-87	Sea Green
60-179-113	Medium Sea Green
32-178-170	Light Sea Green
152-251-152	Pale Green
0-255-127	Spring Green
124-252- 0	Lawn Green
0-255-0	Green
127-255- 0	Chartreuse
0-250-154	Medium Spring Green
173-255-47	Green Yellow
50-205-50	Lime Green
154-205-50	Yellow Green
34-139-34	Forest Green
107-142-35	Olive Drab
189-183-107	Dark Khaki
240-230-140	Khaki
238-232-170	Pale Goldenrod
250-250-210	Light Goldenrod Yellow
255-255-224	Light Yellow
255-255-0	Yellow
255-215-0	Gold
238-221-130	Light Goldenrod
218-165-32	Goldenrod
184-134-11	Dark Goldenrod
188-143-143	Rosy Brown
205-92-92	Indian Red

RGB Value	Description
139-69-19	Saddle Brown
160-82-45	Sienna
205-133-63	Peru
222-184- 135	Burlywood
245-245-220	Beige
245-222-179	Wheat
244-164-96	SandyBrown
210-180-140	Tan
210-105-30	Chocolate
178-34-34	Firebrick
165-42-42	Brown
233-150-122	Dark Salmon
250-128-114	Salmon
255-160-122	Light Salmon
255-165- 0	Orange
255-140-0	Dark Orange
255-127-80	Coral
240-128-128	Light Coral
255-99-71	Tomato
255-69-0	Orange Red
255-0-0	Red
255-105-180	Hot Pink
255-20-147	Deep Pink
255-192-203	Pink
255-182-193	Light Pink
219-112-147	Pale Violet Red
176-48-96	Maroon
199-21-133	Medium Violet Red
208-32-144	Violet Red
255-0-255	Magenta
238-130-238	Violet
221-160-221	Plum

RGB Value	Description
218-112-214	Orchid
186-85-211	Medium Orchid
153-50-204	Dark Orchid
148-0-211	Dark Violet
138-43-226	Blue Violet
160- 32-240	Purple
147-112-219	Medium Purple
216-191-216	Thistle

C.3 Fonts

There are nine different logical font names that can be specified in any Java 2 program. They are:

- `Courier`
- `Dialog`
- `DialogInput`
- `Helvetica`
- `Monospaced`
- `SansSerif`
- `Serif`
- `TimesRoman`
- `ZapfDingbats`

Note: Font names are case-sensitive.

There are also four standard font style constants that can be used. The valid Java 2 font style constants are:

- `Font.BOLD`
- `Font.BOLD | Font.ITALIC`
- `Font.ITALIC`
- `Font.PLAIN`

These values are strung together with dashes ("`-`") when used with the `VALUE` attribute. You must also specify a point size by adding it to other font elements. To display a text using a 12-point italic Helvetica font, use the following:

```
Helvetica-italic-12
```

All three elements (font name, font style and point size) must be used to specify a particular font display; otherwise, the default font is used instead.

Note: Font display may vary from system to system. If a font does not exist on a system, the default font is displayed instead.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z