# Support Readiness Document
# Java 2 Standard Edition 1.3
# Swing


Sun Microsystems, Inc.
Market Development & Developer Relations
Support Readiness Education

# Support Readiness Document
# Java 2 Standard Edition 1.3
# Swing

Sun Microsystems, Inc.
Market Development & Developer Relations
901 San Antonio Road
Palo Alto, CA 94303
U.S.A.

Version: 1.3
Release Date: March 30, 2000



Sun Microsystems, Inc.
Market Development & Developer Relations
901 San Antonio Road
Palo Alto, CA 94303
U.S.A.

# Table of Contents

# Preface

This document provides Support Readiness information for Java 2 Standard Edition 1.3 Swing. The goal of Support Readiness Documents (SRDs) is to help support engineers prepare to support Software Products and Platforms Division products. SRDs are not designed to provide comprehensive product training (see the product documentation or Sun Education for this). Instead, they focus on issues immediately relevant to support, such as installation, configuration, and common user problems.

Document Format Options: PDF and PostScript

The *Java 2 Standard Edition 1.3 Swing SRD* can be viewed in PostScript or PDF format. The PDF version of the document allows navigation via a table of contents frame, and the benefit of live cross references and web links. Text that is underlined and in blue, such as the URL in this paragraph, are clickable links in the PDF version of the document. (Note: page numbers in the PDF document refer to printed pages, and will not coincide with the page numbers in the PDF reader status bar.) Although the blue color and underlining appear in the PostScript version, there are no live links when viewing that version.

Typographic Conventions

This document uses the following type conventions:

- The names of commands, files, Java™ objects, Java classes, and directories are shown in `regular monospace font`.
- Text that is a placeholder to be replaced with a real name or value appears in italic type; for example: `% unzip jsdt-1.4.zip -d` *destination directory*.
- Text that you type, when shown alongside computer output such as a command prompt, is shown in **`bold monospace font`**. The marker "`prompt>`," in regular monospace font, represents the actual command prompt you would see on your screen, which may vary depending on your specific environment, shell, or platform. For example: `Solaris prompt>` **`ls -l`**.
- The names of menu items, buttons, windows, and keyboard keys appear in regular font with initial capitals, such as the Enter key.
- URLs that are clickable web links in the PDF version of the document are shown in blue, underlined monospace font, as in `http://java.sun.com`. Although the blue color and underlining appears in the PostScript version, there are no live links when viewing that version.

- URLs that are not clickable web links are shown in regular monospace font, such as `jsdt://stard:5555/socket/Session/chatSession`.
- Cross-references to other sections of the document are shown in regular font but are blue and underlined, as in, See Section 1.0, "JSDT Overview." In the PDF version of the document, these are clickable links to the indicated section. Although the blue color and underlining appears in the PostScript version, there are no live links when viewing that version.
- New terms and book titles appear in *italic* type.

# Java 2 Standard Edition 1.3
# Swing

This document provides Support Readiness information for the Swing package. Swing is a set of graphical user interface (GUI) components in the Java™ Foundation Classes (JFC) that enable developers to incorporate Swing components such as tool bars, tables, split panes, buttons and menus into their Java applications.

Swing was released as an unbundled product to run with the Java Development Kit (JDK™) and has been added as a "core" library in the Java 2 platform.

**Note:** The Java 2 platform 1.2 was previously referred to as JDK 1.2.

## 1.0 Swing Overview

The Swing packages in the JFC provide the framework for building GUIs for Java programs. Swing includes a comprehensive set of GUI components, such as buttons, scrollbars, table, and text, which are implemented entirely in Java code and rely on the lower level infrastructure of the Abstract Window Toolkit (AWT).

### 1.1 Features, Advantages and Benefits

Swing is implemented entirely in Java code, which ensures that it behaves consistently across different hardware platforms. This feature is unlike the AWT peer components, which have inconsistencies which can be difficult to work around.

#### 1.1.1 Pluggable Look and Feel

Swing provides a Pluggable Look and Feel (PLAF) architecture, which allows the Look and Feel of a program to be selected dynamically. Swing provides the following Look and Feels:

- Windows Look and Feel – Runs on Win95/Win98/WinNT only
- Motif Look and Feel – Used on Solaris, OpenWindows, X-Windows
- Metal Look and Feel – Common Look and Feel developed for cross-platform usage, which is the default

The pluggable Look and Feel architecture also provides an Application Programming Interface (API) that allows the architecture to be extended to create custom Look and Feels.

Swing uses the Model-View-Controller paradigm to encourage a clean separation of application logic from the GUI. This API allows application programs to plug their own model implementations (the application's "data") into the components.

### 1.1.2 Components Provided with Swing

#### 1.1.2.1 AWT Components

Swing provides Java implementation versions of all of the AWT components:

- `JButton`
- `CheckBox`
- `List`
- `ComboBox`
- `TextArea`
- `TextField`
- `Label`
- `ScrollBar`
- `ScrollPane`
- `Panel`

#### 1.1.2.2 Additional High-Level Components

Swing provides the following additional high-level components that cannot be found in the AWT:

- `ToolBar`
- `ProgressBar`
- `Slider`
- `InternalFrame`
- `DesktopPane`
- `OptionPane`
- `SplitPane`
- `TabbedPane`
- `Tree`
- `Table`
- `EditorPane`
- `FileChooser`
- `ColorChooser`

#### 1.1.2.3 Menus

Swing implements its menus as real components. This implementation is unlike the AWT, where menus are treated as special cases. The components are:

- `Menu`
- `MenuBar`
- `PopupMenu`
- `MenuItem`
- `CheckBoxMenuItem`
- `RadioButtonMenuItem`
- `Separator`

#### 1.1.2.4 Text Application Program Interface

Swing also provides a sophisticated Text API which implements support for styled text, such as hypertext markup language (HTML) and rich text format (RTF).

## 1.2 New in the Java 2 Platform

The Java 2 platform is the first version of the Java software to include the Swing packages as part of the core platform. The functionality in Swing for the Java 2 platform, version 1.3, now supersedes the functionality provided in Swing 1.1.1, which was the last unbundled version for JDK1.1.X. We currently have no plans to provide any more unbundled releases of Swing for JDK1.1.X. We recommend customers migrate to Java 2 in order to get new features and bug fixes for Swing.

## 1.3 Changes in Java 2 Standard Edition 1.3

There were many minor API enhancements made in version 1.3. For a complete description of each, please consult the latest Java 2 Standard Edition 1.3 documentation. Swing changes can be found at `http://java.sun.com/products/jdk/1.3/docs/guide/swing/SwingChanges.html`.

### 1.3.1 Summary of Enhancements

#### 1.3.1.1 `Button, JCheckBox, JRadioButton`

`JCheckBox` added a "`borderPaintedFlat`" property to allow it to be more easily used as a cell renderer.

Added `DefaultButtonModel.getGroup()` method to make it easier to translate from a model to a group.

#### 1.3.1.2 `FileChooser`

`JFileChooser` added a "`isAcceptAllFileFilterUsed`" property to allow control of the `AcceptAll (*.*)` file filter, which is added to the choosable file filters combobox by default.

Support was added for removing the OK and Cancel buttons by the addition of a `controlButtonsAreShowing` property on `JFileChooser`.

#### 1.3.1.3 JFrame

`JFrame` added `EXIT_ON_CLOSE` `defaultCloseOperation` constant to make it easier for programs to automatically exit when the user closes the frame. Previously, developers had to add a `WindowListener` and do this themselves.

#### 1.3.1.4 InternalFrame

`JInternalFrame` added `setLayer(int)` method so that the beans introspector recognizes this as a property.

`BasicInternalFrameUI` added `uninstallListeners()` method to remove its listeners when no longer needed.

Changes to `JInternalFrame` "`closed`" Property:

- The default setting for `defaultCloseOperation` has been changed from `HIDE_ON_CLOSE to DISPOSE_ON_CLOSE`
- The constant `EXIT_ON_CLOSE` has been added
- `JInternalFrame`'s `doDefaultCloseAction` method has been changed from private to public

`JInternalFrame.getNormalBounds()` method was added to allow for getting the normal dimensions of an internal frame.

`JInternalFrame` added `getFocusOwner()` method.

`JInternalFrame` added `restoreSubcomponentFocus()` method.

`InternalFrameEvent` added `getInternalFrame()` method.

`MetalInternalFrameTitlePane` Class is now public.

#### 1.3.1.5 Menu (Menus)

JMenu `getPopupMenuOrigin()` method is now protected.

`BasicMenuItemUI` `installComponents()`, `uninstallComponents()` methods are now protected.

`JPopupMenu` added `isPopupTrigger()` method. Also, `javax.swing.plaf.PopupMenuUI` added an `isPopupTrigger()` method.

#### 1.3.1.6 SplitPane

`JSplitPane` added a "`resizeWeight`" property, so that when the size of a `JSplitPane` changes, the extra space is added to the right/bottom component. This gives the effect of the left/top component being fixed.

The `JSplitPane` "`dividerLocation`" property is now bound.

There is now a `Border` on the `JSplitPane`'s divider: `BasicSplitPaneDivider`. As a result, `setDividerSize` now needs to take into account the border size, which is usually an extra 2 pixels, that is, developers used to do `setDividerSize(5)`, but now it will need to be `setDividerSize(7)` (unless a developer installs a new UI with a different border size).

#### 1.3.1.7 JTabbedPane

`JTabbedPane` added a new "`toolTipTextAt`" index property to allow the tool tip text to be set for individual tabs after they are created.

#### 1.3.1.8 JTable

Improvements to column and row layout were made:

- Improved performance in tables with large numbers of columns
- Dynamic changing of individual row height
- Simplified creation of non-standard editor components
- Better handling of inter-cell spacing

As a result of the changes in this release, code written for 1.2 or for JFC/Swing 1.1 might have the following problems when run in 1.3.

Since the `TableColumn getHeaderRenderer` method now returns null by default, you can't use that method to get the default header renderer. Instead, change your code to use the `JTableHeader getDefaultRenderer` method. See "How to Use Tables" in The Java Tutorial for an example.
http://java.sun.com/docs/books/tutorial/uiswing/components/table.html

Because `JTable`'s default text editor is now smarter about data types, it gives `setValueAt` objects of the appropriate type, instead of always specifying strings. For example, if `setValueAt` is invoked for an Integer cell, then the value is specified as an Integer instead of a String. If you implemented a table model, then you might have to change its `setValueAt` method to take the new data type into account. If you implemented a class used as a data type for cells, make sure that your class has a constructor that takes a single String argument.

### 1.3.2 Summary of Changes to JTable

This is a summary of the API changes made to JTable for 1.3. For more information, see the API documentation for the following methods and fields.

**Note:** "Obsolete" means that you should avoid using the obsolete API, but that using the API isn't dangerous enough to warrant a compile-time warning.
A "Deprecated" API is not only obsolete but also results in a compile-time warning.

#### 1.3.2.1 JTable

New:

- `public void removeNotify()`
- `protected void unconfigureEnclosingScrollPane()`

- `public void changeSelection(int rowIndex, int columnIndex, boolean toggle, boolean extend)`
- `public void setRowHeight(int row, int height)`
- `public int getRowHeight(int row)`

Obsolete:

- `protected boolean cellSelectionEnabled // obsolete`

#### 1.3.2.2 `JTableHeader`

New:

- `public void setDefaultRenderer(TableCellRenderer defaultRenderer)`
- `public TableCellRenderer getDefaultRenderer()`
- `protected TableCellRenderer createDefaultRenderer()`

Obsolete:

- `protected boolean updateTableInRealTime // obsolete`
- `public boolean getUpdateTableInRealTime() // obsolete`
- `public void setUpdateTableInRealTime(boolean flag) // obsolete`

#### 1.3.2.3 `TableColumn`

New:

- `protected TableCellRenderer createDefaultHeaderRenderer()`

Obsolete:

- `public final static String COLUMN_WIDTH_PROPERTY; // obsolete`
- `public final static String HEADER_VALUE_PROPERTY; // obsolete`
- `public final static String HEADER_RENDERER_PROPERTY // obsolete`
- `public final static String CELL_RENDERER_PROPERTY // obsolete`

Deprecated:

- `transient protected int resizedPostingDisableCount // deprecated`
- `public void disableResizedPosting() // deprecated`
- `public void enableResizedPosting() //deprecated`

#### 1.3.2.4 New Class `javax.swing.AbstractCellEditor`

Existing Class `DefaultCellEditor` now extends `AbstractCellEditor`.

Before this release the height of rows in a `JTable` was always fixed. Introducing variable height rows while retaining the scalability requirements of the `JTable` (no O(N) behavior on rows) has required the new class, `javax.swing.SizeSequence`.

#### 1.3.3 `ToolBar`

Support was added for specifying the title for undocked toolbars by adding the following constructors:

- `public JToolBar(String name)`
- `public JToolBar(String name , int orientation)`

#### 1.3.4 `JTree`

`JTree` now provides a "`toggleClickCount`" property for configuring how many clicks are needed to expand or collapse a node.

`JTree` also provides two new properties for controlling how nodes are selected: "`leadSelectionPath`" and "`anchorSelectionPath`".

`JTree` provides an "`expandsSelectedPaths`" property for controlling whether or not selected nodes are made visible. Also, the `removeDescendantSelectedPaths()` method was added to be called when a node is removed or collapsed.

`DefaultTreeCellRenderer` exposed the "`hasFocus`" field as protected.

`TreeSelectionEvent` added the `isAddedPath()` method.

`DefaultTreeSelectionModel`'s `insureUniqueness()` method was made obsolete (remains for backward compatibility).

#### 1.3.5 `JViewport` (Scrolling)

`JViewport` added a "`scrollMode`" property to control the type of scrolling. J2SE 1.3 adds a new optimization, called "BLIT_SCROLL_MODE" (now the default) that copies unobscured areas of the components heavy weight ancestor to effect a scroll. Previously we had offered a related optimization that managed a per viewport backing store image. Although still supported, the `backingStoreEnabled` property has been deprecated.

#### 1.3.6 Text

A method in `HTMLEditorKit`, `insertAtBoundry()` was misspelled. It's been deprecated and `insertAtBoundary()` has been added.

The default functionality of managing the `DocumentEvent` has been moved (from `CompositeView`) and raised from package private to public. Subclassing has been made much easier as the management of the `DocumentEvent` is now distributed to the protected methods:

- `updateChildren()`
- `forwardUpdate()`

- `forwardUpdateToView()`
- `updateLayout()`

To accomplish its behavior, the methods managing the children of the view also needed to be moved from `CompositeView` to `View`:

- `removeAll()`
- `remove()`
- `append()`
- `replace()`

`BoxView` has an axis argument which subclasses previously could not get at. A subclass that sets its axis based upon i18n considerations needs this as a property. The layout of the box should also be treated separately from the requested size as in some cases (such as a table) the layout may become invalid independent of the children's preferences. The following methods have been added to `javax.swing.text.BoxView`:

- `public int getAxis()`
- `public void setAxis(int axis)`
- `public void layoutChanged(int axis)`

The `View` protocol supports building flows, but the only previous implementation had been `ParagraphView` which creates relatively simple paragraph flows. The functionality can be generalized substantially to do things like shaped flows, page breaking, and some others. Generalizing the functionality will make the creation of alternative flows much easier.

A new class called `FlowView` takes a strategy to translate the logical structure to a physical structure. The strategy is defined by a nested static class called `FlowStrategy`. `ParagraphView` extends `FlowView` adding just that behavior which is paragraph-specific, for example, line spacing, first line indent, and alignment. This change adds the following method to the `javax.swing.text.View` class:

- `public int getViewIndex(int pos, Position.Bias b)`

Class `javax.swing.text.html.FormView` was not previously localizable. There are two public static final Strings, `SUBMIT` and `RESET`, that are used to determine the text for `<form>` elements in an HTML document. As they are public static final they can not be localized. `SUBMIT` and `RESET` have been deprecated, the values are now obtained from the `UIManager` properties:

- `FormView.submitButtonText`
- `FormView.resetButtonText`

`javax.swing.text.html.parser.ParserDelegator` now implements `Serializable`.

A nullary constructor has been added to the public static inner class `HTML.Tag` to allow serialization of subclasses to work.

`AbstractWriter` is used as a base class for developers wishing to provide custom writing out of a text Document. For example, `HTMLWriter` extends `AbstractWriter`, and is used in the HTML package to output HTML. The problem with `AbstractWriter` was that most of the methods and ivars it provided were private. Subclasses had to resort to copying numerous methods and ivars to be useful. The following changes open up the API in `AbstractWriter`, making it more useful by itself (this makes `HTMLWriter` much simpler, and it won't have to copy as much from `AbstractWriter`):

- public int `getStartOffset()`
- public int `getEndOffset()`
- protected `Writer` `getWriter()`
- protected int `getLineLength()`
- protected void `setCurrentLineLength(int length)`
- protected int `getCurrentLineLength()`
- protected boolean `isLineEmpty()`
- protected void `setCanWrapLines(boolean newValue)`
- protected boolean `getCanWrapLines()`
- public void `setLineSeparator(String value)`
- public String `getLineSeparator()`
- protected int `getIndentLevel()`
- protected void `writeLineSeparator()` throws `IOException`
- protected void `write(char[] chars, int startIndex, int length)` throws `IOException`
- protected void `output(char[] content, int start, int length)` throws `IOException`

### 1.3.7 Cascading Style Sheets

The HTML package supports Cascading Style Sheets (CSS). One of the abilities of CSS is to support the cascading of style sheets (as the name implies). That is, more than one style sheet can influence the presentation simultaneously. For example, a browser usually has a style sheet that defines the default styles, a particular page might also have a style sheet that overrides those provided by the browser. The following methods expose this to developers, allowing them to link style sheets:

- public synchronized void `addStyleSheet(StyleSheet ss)`
- public synchronized void `removeStyleSheet(StyleSheet ss)`
- public `StyleSheet[] getStyleSheets()`

Previously it was not very easy for developers to insert arbitrary HTML into an existing HTML document. To accomplish this, the developer needed to have an intimate knowledge of the Swing Text Package, as well as the HTML package. Many developers

have used dynamic HTML, which provides a handful of methods that make it trivial to insert HTML into an existing page. To accommodate these users we now expose methods that closely mirror those provided by dynamic HTML:

- public void `setInnerHTML(Element elem, String htmlText)` throws `BadLocationException`, `IOException`
- public void `setOuterHTML(Element elem, String htmlText)` throws `BadLocationException`, `IOException`
- public void `insertAfterStart(Element elem, String htmlText)` throws `BadLocationException`, `IOException`
- public void `insertBeforeEnd(Element elem, String htmlText)` throws `BadLocationException`, `IOException`
- public void `insertBeforeStart(Element elem, String htmlText)` throws `BadLocationException`, `IOException`
- public void `insertAfterEnd(Element elem, String htmlText)` throws `BadLocationException`, `IOException`
- public Element `getElement(String id)`
- public Element `getElement(Element e, Object attribute, Object value)`

This constant has been added to class `HTMLEditorKit.ParserCallback`:

- public static final Object IMPLIED

The Swing Text Package uses View objects to represent a presentation of the document model for the sake of layout and rendering. If the model is large, the number of view objects that get created is large, in spite of only being able to view a small number of them. A View implementation is needed that defers creation of the objects until they are actually needed for display. This can be done by building zones with an estimated size that get replaced with the actual View objects when they are needed. This can substantially reduce the amount of memory used for large documents.

The class `ZoneView` was added which supports zones that don't consume much memory until actively viewed or edited. `WrappedPlainView` which will become more heavyweight with its support of bidi for i18n will benefit substantially from this class by changing it's superclass from `BoxView` to `ZoneView`.

The Swing Text Package uses `View` objects to represent a view of the document model to handle layout and rendering. Layout is done on the event handling thread like most other things. Layout is sometimes quite expensive in terms of cpu time, and causes the user interface to freeze while performing layout. The text package has some support of concurrency however, so this should be extended to include performing layout asynchronous to the gui event handling thread. A `View` implementation is needed that performs layout asynchronously.

To address this need, the following classes are being added to the text package:

- `javax.swing.text.AsyncBoxView`

- `javax.swing.text.LayoutQueue`

A `getGraphics()` method is included on the `javax.swing.text.View` class to fetch the `Graphics` object that will be used to render.

For I18N, `GlyphView.GlyphPainter` added a `getNextVisualPositionFrom()` method, which provides a way to determine the next visually represented model location that one might place a caret. Some views may not be visible, and they might not be in the same order found in the model, or they just might not allow access to some of the locations in the model.

### 1.3.8 Actions

Swing provides an implementation of the Command pattern which helps the application developer centralize functionality which can be accessed from multiple places in the GUI. The Action interface is used to provide a stateful `ActionListener` which can provide the implementation of functionality accessed from the toolbar, a menu item, or a keyboard binding, as examples. As the state of the Action changes, for instance when it becomes disabled, the associated controls change their state accordingly (they also become disabled).

For Actions to work as intended, the following connections need to be made (assume the Action has already been created):

- The control needs to be created.
- The Action is added as an `ActionListener` on the control.
- A `PropertyChangeListener` is created which describes how the control should be updated in response to `PropertyChangeEvents` on the Action.
- The `PropertyChangeListener` is added as a listener on the Action.
- Information about the linkage may need to be retained so it can be undone to allow garbage collection (in 1.2 this can be automatically handled with WeakRefs).

Since a typical application may have between 5 and 25 Actions, and 2-3 controls per Action, the steps above need to be done up to 75 times!

In order to relieve the developer of much of this burden, we have provided a way to have this done automatically, through helper methods on potential Containers of Actions. The three places where this is surfaced in Swing at present are:

- `JToolBar.java`
  ```
  public JButton add(Action a)
  ```
- `JMenu.java`
  ```
  public JMenuItem add(Action a)
  ```
- `JPopupMenu.java`
  ```
  public JMenuItem add(Action a)
  ```

The problems with this approach are several:

- It is highly problematic for Builders, since it overloads `Container.add()` to allow a non-Component parameter which is not itself the thing that ends up being added.
- Developers cannot participate in the configuration of the controls created without subclassing the container classes.
- Even if they do subclass, the granularity of the configuration ends up being per-Container instead of per-control added.
- It limits developers to the expected control for each Container rather than allowing the full range of `ActionEvent` sources which Action permits.

Many developers have commented that they would prefer to create their own controls which are `ActionEvent` sources and then have a method which connects them to a particular Action. The solution is along these lines, and addresses the deficiencies listed above.

The added API is initially added to `AbstractButton`, which defines the abstract superclass of `JButton`, `JMenuItem`, `JMenu`, and `JCheckBox`. The new public methods are:

- `public void setAction(Action a)`
- `public Action getAction()`
- `protected void configurePropertiesFromAction(Action a)`
- `protected PropertyChangeListener createActionPropertyChangeListener()`

In addition, constructors have been added to the `ActionEvent` sources which will allow for creating a control directly with the supplied Action.

In `JButton`:

- `public JButton(Action a)`

Equivalent constructors have been added to:

- `JCheckBox`
- `JRadioButton`
- `JToggleButton`
- `JMenuItem`
- `JMenu`
- `JCheckBoxMenuItem`
- `JRadioButtonMenuItem`

**Note:** `setAction()` is merely a helper method which performs the linkage steps described previously as a convenience to the developer.

It is not expected that developers will often switch the Action for a control on the fly. However, it is possible for them to do so, using `setAction` since it replaces the previously set action and fires a `PropertyChangeEvent`. This does not replace the

standard method of adding `ActionListeners`, note that it uses `addActionListener()` as stated previously.

The current Container APIs listed above will be reimplemented in terms of `setAction`, so that they give the same behavior as they did previously. This solution will make that code much easier to maintain.

The methods `configurePropertiesFromAction` and `createActionPropertyChangeListener` will be overridden in subclasses to provide the expected default behavior.
New factory methods allow one to control what toolbars and menus create when an action is added directly, such as, with the add method.

Addition to `JToolBar`:

- protected `JButton createActionComponent(Action a)`

Addition to `JPopupMenu`:

- protected `JMenuItem createActionComponent(Action a)`

Addition to `JMenu`:

- protected `JMenuItem createActionComponent(Action a)`

`AbstractAction` added `getKeys()` Method for serialization of Abstract Actions, and gives the developer a way to find out which keys have been set for the `AbstractAction`.

### 1.3.9 Borders

`BorderFactory` added new static methods for creating shared `EtchedBorder` instances:

- `createEtchedBorder()`

  (The lack of this method caused inconsistency in the API).

A new constructor was added to `LineBorder` to allow developers to create `LineBorders` with rounded corners:

- `public LineBorder(Color color, int thickness, boolean roundedCorners)`

### 1.3.10 Exposed Listener Lists

A `getListeners()` method has been added to the following classes:

- `javax.swing.Timer`
- `javax.swing.AbstractListModel`
- `javax.swing.DefaultBoundedRangeModel`
- `javax.swing.DefaultButtonModel`
- `javax.swing.DefaultListSelectionModel`

- javax.swing.DefaultSingleSelectionModel
- javax.swing.table.AbstractTableModel
- javax.swing.table.DefaultTableColumnModel
- javax.swing.tree.DefaultTreeModel
- javax.swing.tree.DefaultTreeSelectionModel
- javax.swing.text.AbstractDocument
- javax.swing.text.DefaultCaret
- javax.swing.event.EventListenerList

### 1.3.11 Input Verification

A new class, InputVerifier was introduced to provide a better mechanism for components to validate input.

The purpose of this class is to help clients support smooth focus navigation through GUIs with text fields. Such GUIs often want to ensure that the text entered by the user is valid (in the proper format) before allowing the user to navigate out of the text field. In order to do this, clients can implement this class, and, using JComponent's setInputVerifier method, attach an instance of it to the JComponent whose input they want to validate. Before focus is transferred to another Swing component that requests it, this class's shouldYieldFocus method is called, and focus is transferred only if that method returns true.

Without this class, clients tried to perform input validation in a FOCUS_LOST listener on the component being validated. If the input was found to be invalid, they would call requestFocus to restore focus back to the component with invalid input. One problem with this approach is that there is currently a bug (bugid #4126859) which prevents it from working. Such an API would prevent focus flicker and guarantee that focus would stay in the component with invalid input.

The API change consists of a new abstract class, InputVerifier, and two new methods added to JComponent:

- setInputVerifier()
- getInputVerifier()

### 1.3.12 Printing

JComponent previously did not override print. This meant that printing was no different than painting, resulting in the double buffer being used. This is not the desired behavior. It also makes it harder for developers to add customized printing logic. In most cases the developer would resort to completely replacing print, rewriting all the code to notify the children. To be consistent with the painting methods, these protected methods have been added to JComponent:

- printBorder()
- printChildren()
- printComponent()

### 1.3.13 Performance

One of the main reasons that Swing's startup performance was slower than desired was that as soon as any component requires a User Interface (UI) delegate, the UI Manager loads a Look And Feel, which results in loading a defaults table which includes defaults for UIs for all component classes.

In previous releases, we mistakenly believed that instance creation should be avoided, so we delayed instance creation by creating anonymous implementations of `LazyValue`, an interface which acts as a lightweight proxy that only creates its instance the first time it is retrieved from the defaults table.

Performance analysis of Java 2 Standard Edition 1.3 indicates that we were wrong in believing that instance creation was the determining factor. In fact, the overwhelming factor contributing to delay and increased footprint in this area was classloading, which was not helped by our creation of lots of anonymous interface implementations!

The general approach taken to fix this was to define a concrete `LazyValue` implementation in `UIDefaults.java` which uses reflection to create its proxied instance when asked to do so. This class is called `UIDefaultProxy`. As a result only one class is loaded, and about 90 other classloads could be avoided in a Hello World example.

In the course of replacing the existing anonymous `LazyValue` implementations and identifying other classloads that could be avoided, we came across several classes and accessor methods which were incorrectly packaged as private. Since the `UIDefaultProxy` is in the `javax.swing` package, and most of the uses are in `javax.swing.plaf.*` packages, these signatures needed to be changed so that they could be used by the proxy. These are:

- `javax.swing.plaf.basic.BasicBorders`
  `public static Border getSplitPaneDividerBorder()`
- `javax.swing.plaf.metal.MetalBorders`
  `public static class PaletteBorder`
  `public static Border getButtonBorder()`
  `public static Border getTextBorder()`
  `public static Border getTextFieldBorder()`
  `static class ToggleButtonBorder extends ButtonBorder`
  `public static Border getToggleButtonBorder()`
  `public static Border getDesktopIconBorder()`
- `javax.swing.plaf.metal.MetalIconFactory`
  `public static Icon getCheckBoxIcon()`
  `public static class PaletteCloseIcon implements Icon, UIResource, Serializable`

## 2.0 Product Distribution

Swing is shipped as part of the Java 2 Platform Standard Edition 1.3.

## 3.0 Requirements and Dependencies

This version of Swing depends on other packages in the Java 2 platform, but does not have dependencies on any outside packages.

### 3.1 Product Compatibility

#### 3.1.1 Versions

- Swing 1.0/1.0.1 – The first “final” version of the Swing product. The unbundled library was designed to be used with JDK1.1. This version shipped March 1998.
- Swing 1.0.2 – The follow-on bug fix release for Swing 1.0 (JDK1.1). This version fixed many bugs related to Applet painting. This version shipped April 1998.
- Swing 1.0.3 – The bug fix release for Swing 1.0.2. This version fixed a critical bug with mnemonics. Other than this change, it is identical to Swing 1.0.2. This version shipped June 1998.
- Swing 1.1 – This unbundled release included minor feature additions, API changes to plaf, and the package-name change. It is equivalent to the Swing release which was included in JDK1.2.0.
- Swing 1.1.1 – This unbundled release contained hundreds of bug fixes (no API changes). It is equivalent to the Swing release which was part of JDK1.2.2.

#### 3.1.2 Backward and Forward Compatibility With Other Versions

Programs written to run with Swing in Java 2, Version1.2.X should run unchanged in Java 2, Version1.3.0.

## 4.0 Key File Descriptions

### 4.1 The `rt.jar` File

All the classes in the Swing packages are contained in the `rt.jar` file. The `rt.jar` file is located in the `/jre/lib` subdirectory of the installation directory for the Java 2 platform.

### 4.2 Swing Demos

Swing provides the following demos located in the `/demo/jfc` subdirectory of the installation directory for the Java 2 platform:

- DBDemos
- FileChooserDemo

- Metalworks
- Notepad
- SampleTree
- SimpleExample
- Stylepad
- SwingApplet
- SwingSet2 (NEW for 1.3!)
- TableExample

The original `SwingSet` demo has been replaced by `SwingSet2`, which is a more cleanly written demo of Swing functionality.

## 5.0 Bugs

Please go to http://developer.java.sun.com/developer/bugParade and do a search for Swing, to find information on Swing bugs.

## 6.0 Using Swing

### 6.1 The Swing Packages

Swing includes the following packages:

**Table1. Swing Package Names and Contents**

| Swing Package Name | Contents |
|---|---|
| `javax.swing` | Top-level Swing component and utility classes and interfaces |
| `javax.swing.border` | Classes and interfaces for putting borders around components |
| `javax.swing.colorchooser` | Classes and interfaces supporting the ColorChooser |
| `javax.swing.event` | All Swing-specific event classes/interfaces |
| `javax.swing.filechooser` | Classes and interfaces supporting the File Chooser |
| `javax.swing.plaf` | Abstract classes used for pluggable Look and Feel |
| `javax.swing.plaf.basic` | Classes used as common base for standard Look and Feels |
| `javax.swing.plaf.metal` | Classes used for Metal (Java) Look and Feel |
| `javax.swing.plaf.multi` | Classes used for multiplexing Look and Feel |
| `javax.swing.table` | Classes and interfaces supporting Table |
| `javax.swing.text` | Classes and interfaces supporting Text |
| `javax.swing.text.html` | Classes and interfaces supporting HTML |
| `javax.swing.text.parser` | Classes and interfaces supporting HTML parser |

**Table1. Swing Package Names and Contents (Continued)**

| Swing Package Name | Contents |
|---|---|
| `javax.swing.text.rtf` | Classes supporting RTF |
| `javax.swing.tree` | Classes and interfaces supporting Tree |
| `javax.swing.undo` | Classes and interfaces supporting “undo” and “redo” |
| `com.sun.java.swing.plaf.windows` | Classes for Win32 Look and Feel (not “core”) |
| `com.sun.java.swing.plaf.motif` | Classes used for Motif Look and Feel (not “core) |

For detailed information about the specific classes and interfaces contained in the Swing packages, see
http://java.sun.com/products/jdk/1.3/docs/api/index.html.

The majority of basic GUI programs only need to use the classes and interfaces defined in the `javax.swing` package, which contains all the top-level component classes. These top-level classes are each prefixed with a “J,” for example `JButton`, `JTable`, `JTextField`. Programs that need to do more custom configuration of the complex components, such as `JTable`, `JTree`, `JText*`, `JFileChooser`, and `JColorChooser` will need to use the classes and interfaces defined in those respective sub packages. All of the `javax.swing.plaf` sub packages contain APIs specific to the Look and Feel implementations and most programs should never need to use these APIs. The exceptions are those programs which create or extend a Look and Feel.

### 6.1.1 Learning Swing

The best place to start learning Swing is the Java Tutorial, which now includes a great deal of Swing-specific programming instruction. For more information on this section of the Java Tutorial, see
http://java.sun.com/docs/books/tutorial/ui/index.html.

In addition, the Swing Connection Web site has numerous articles on subjects related to Swing programming. The site is updated every couple of months with new material, but all articles are archived and available at all times. To access the Swing Connection, see
http://java.sun.com/products/jfc/tsc/index.html.

## 7.0 Tuning and Troubleshooting

For the most part, techniques that apply to tuning and troubleshooting Java programs in general also apply to Swing. However, the following tips are useful when debugging problems to help isolate where the issue actually resides:

- Is the problem reproducible on both Solaris and Win32 platforms?

If so, this likely indicates a bug in the “common” Java code somewhere.

If not, then the problem is likely due to a platform-specific AWT bug (remember - Swing contains no native code, so if it behaves differently on different platforms, the differences are caused by AWT, not Swing).

- Is the problem reproducible in multiple Look and Feels?

  If the problem only shows up in a single Look and Feel—Metal, for example, then it is easier for us to isolate the debugging in finding the problem.

## 7.1 Installation and Configuration “Gotchas”

Swing is automatically installed and configured as part of the Java 2 platform installation/configuration.

## 7.2 Common User Questions

There is no official Swing FAQ, however an external FAQ has useful information: `http://users.vnet.net/wwake/swing/faq.html`.

The best way to ask Swing-related questions is to send email to the `swing-feedback@java.sun.com` alias, which is read by Swing development engineers.

Additionally, there is a mailing list dedicated to Swing technical discussions. To join, send email to: `swing-subscribe@eos.dk` for the general discussion list, or to: `advanced-swing-subscribe@eos.dk` for the advanced topic list.

The `swing@eos` mailing list is now archived at `http://www.findmail.com/list/swing/`.

Be sure to check out the ongoing Swing discussions in the `comp.lang.java.gui` and `comp.lang.java.programmer` usenet news groups.

## 7.3 Error Message Guide

In general there are no standard “error” messages in Swing because error conditions are handled using the Java Exception mechanism. Checked Exceptions are documented as part of the API; Runtime Exceptions are in many cases unpredictable and difficult to document.

## 7.4 Performance and Tuning Recommendations

An article which gives an overview of improvements made to Swing performance can be found at:
`http://java.sun.com/products/jfc/tsc/articles/performance/index.html`.

# 8.0 Reference Information

## 8.1 Technical Articles on the Swing Connection Web Site

The best documentation exists on the Swing Connection Web site at: http://java.sun.com/products/jfc/tsc/index.html. This section provides direct links to some of the key technical articles which are archived on the site:

### 8.1.1 Swing Component Architecture

A white paper explaining the Swing Model-View-Controller (MVC) and Pluggable Look and Feel (PLAF) can be found at: http://java.sun.com/products/jfc/tsc/articles/architecture/index.html.

### 8.1.2 Swing's Threading Model

http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html

http://java.sun.com/products/jfc/tsc/articles/threads/threads2.html

http://java.sun.com/products/jfc/tsc/articles/threads/threads3.html

### 8.1.3 Understanding Swing Containers

http://java.sun.com/products/jfc/tsc/articles/containers/index.html

### 8.1.4 Issues with Mixing Abstract Window Toolkit and Swing

http://java.sun.com/products/jfc/tsc/articles/mixing/index.html

### 8.1.5 Using the `JList` Component

http://java.sun.com/products/jfc/tsc/tech_topics/jlist_1/jlist.html

### 8.1.6 Using the `JTable` Class

http://java.sun.com/products/jfc/tsc/articles/jtable/index.html

### 8.1.7 Using the `JTree` Component

http://java.sun.com/products/jfc/tsc/articles/jtree/index.html

### 8.1.8 Understanding Swing Text

http://java.sun.com/products/jfc/tsc/articles/text/overview/

http://java.sun.com/products/jfc/tsc/articles/text/attributes/

http://java.sun.com/products/jfc/tsc/articles/text/element_buffer/

http://java.sun.com/products/jfc/tsc/articles/text/element_interface/

http://java.sun.com/products/jfc/tsc/articles/text/tabs/

http://java.sun.com/products/jfc/tsc/articles/text/editor_kit/index.html

http://java.sun.com/products/jfc/tsc/articles/text/concurrency/

#### 8.1.9 Swing 1.3 Keyboard Binding Mechanism

http://java.sun.com/products/jfc/tsc/special_report/kestrel/keybindings.html

#### 8.1.10 Creating `TreeTable` Components

http://java.sun.com/products/jfc/tsc/articles/treetable1/index.html

http://java.sun.com/products/jfc/tsc/articles/treetable2/index.html

#### 8.1.11 Understanding Swing Paint Architecture

http://java.sun.com/products/jfc/tsc/articles/painting/index.html

#### 8.1.12 Building Accessible GUIs

http://java.sun.com/products/jfc/tsc/articles/accessibility/index.html

#### 8.1.13 Laying Out a GUI

http://java.sun.com/products/jfc/tsc/articles/cardpanel/index.html

#### 8.1.14 Using Drag and Drop With Swing

http://java.sun.com/products/jfc/tsc/articles/dragndrop/index.html

#### 8.1.15 Internationalization and Swing

http://java.sun.com/products/jfc/tsc/articles/bidi/index.html