



[Table of Contents](#)

[Index](#)

[Reviews](#)

[Examples](#)

[Reader Reviews](#)

[Errata](#)

Java Swing, 2nd Edition

By [Brian Cole](#), [Robert Eckstein](#), [James Elliott](#), [Marc Loy](#), [David Wood](#)

Publisher : O'Reilly

Pub Date : November 2002

ISBN : 0-596-00408-7

Pages : 1278

This second edition of Java Swing thoroughly covers all the features available in Java 2 SDK 1.3 and 1.4. More than simply a reference, this new edition takes a practical approach. It is a book by developers for developers, with hundreds of useful examples, from beginning level to advanced, covering every component available in Swing. Whether you're a seasoned Java developer or just trying to find out what Java can do, you'll find Java Swing, 2nd edition an indispensable guide.



[Table of Contents](#)

[Index](#)

[Reviews](#)

[Examples](#)

[Reader Reviews](#)

[Errata](#)

Java Swing, 2nd Edition

By [Brian Cole](#), [Robert Eckstein](#), [James Elliott](#), [Marc Loy](#), [David Wood](#)

Publisher : O'Reilly

Pub Date : November 2002

ISBN : 0-596-00408-7

Pages : 1278

[Copyright](#)

[Preface](#)

[What This Book Covers](#)

[What's New in This Edition?](#)

[On the Web Site](#)

[Conventions](#)

[How to Contact Us](#)

[Acknowledgments](#)

[Chapter 1. Introducing Swing](#)

[Section 1.1. What Is Swing?](#)

[Section 1.2. Swing Features](#)

[Section 1.3. Swing Packages and Classes](#)

[Section 1.4. The Model-View-Controller Architecture](#)

[Section 1.5. Working with Swing](#)

[Section 1.6. The Swing Set Demo](#)

[Section 1.7. Reading This Book](#)

[Chapter 2. Jump-Starting a Swing Application](#)

[Section 2.1. Upgrading Your AWT Programs](#)

[Section 2.2. A Simple AWT Application](#)

[Section 2.3. Including Your First Swing Component](#)

[Section 2.4. Beyond Buttons](#)

[Section 2.5. What Is an Internal Frame?](#)

[Section 2.6. A Bigger Application](#)

[Chapter 3. Swing Component Basics](#)

[Section 3.1. Understanding Actions](#)
[Section 3.2. Graphical Interface Events](#)
[Section 3.3. Graphics Environments](#)
[Section 3.4. Sending Change Events in Swing](#)
[Section 3.5. The JComponent Class](#)
[Section 3.6. Responding to Keyboard Input](#)

[Chapter 4. Labels and Icons](#)

[Section 4.1. Labels](#)
[Section 4.2. Working with Images](#)
[Section 4.3. Support for HTML](#)
[Section 4.4. Icons](#)
[Section 4.5. Implementing Your Own Icons](#)
[Section 4.6. Dynamic Icons](#)
[Section 4.7. The ImageIcon Class](#)

[Chapter 5. Buttons](#)

[Section 5.1. The ButtonModel Interface](#)
[Section 5.2. The DefaultButtonModel Class](#)
[Section 5.3. The AbstractButton Class](#)
[Section 5.4. The JButton Class](#)
[Section 5.5. The JToggleButton Class](#)
[Section 5.6. The JToggleButton.ToggleButtonModel Class](#)
[Section 5.7. The JCheckBox Class](#)
[Section 5.8. The JRadioButton Class](#)
[Section 5.9. The ButtonGroup Class](#)

[Chapter 6. Bounded-Range Components](#)

[Section 6.1. The Bounded-Range Model](#)
[Section 6.2. The JScrollPane Class](#)
[Section 6.3. The JSlider Class](#)
[Section 6.4. The JProgressBar Class](#)
[Section 6.5. Monitoring Progress](#)

[Chapter 7. Lists, Combo Boxes, and Spinners](#)

[Section 7.1. Lists](#)
[Section 7.2. Representing List Data](#)
[Section 7.3. Handling Selections](#)
[Section 7.4. Displaying Cell Elements](#)
[Section 7.5. The JList Class](#)
[Section 7.6. Combo Boxes](#)
[Section 7.7. The JComboBox Class](#)
[Section 7.8. Spinners](#)
[Section 7.9. Spinner Models](#)
[Section 7.10. Spinner Editors](#)

[Chapter 8. Swing Containers](#)

[Section 8.1. A Simple Container](#)
[Section 8.2. The Root Pane](#)
[Section 8.3. Basic RootPaneContainers](#)
[Section 8.4. The JFrame Class](#)
[Section 8.5. The JWindow Class](#)
[Section 8.6. The JApplet Class](#)

[Chapter 9. Internal Frames](#)

[Section 9.1. Simulating a Desktop](#)
[Section 9.2. The JInternalFrame Class](#)
[Section 9.3. The JDesktopPane Class](#)
[Section 9.4. The DesktopManager Interface](#)
[Section 9.5. Building a Desktop](#)

[Chapter 10. Swing Dialogs](#)
[Section 10.1. The JDialog Class](#)
[Section 10.2. The JOptionPane Class](#)
[Section 10.3. Using JOptionPane](#)
[Section 10.4. Simple Examples](#)
[Section 10.5. Getting the Results](#)
[Section 10.6. A Comparison: Constructors Versus Static Methods](#)
[Section 10.7. Using Internal Frame Dialogs with JDesktopPane](#)

[Chapter 11. Specialty Panes and Layout Managers](#)
[Section 11.1. The JSplitPane Class](#)
[Section 11.2. The JScrollPane Class](#)
[Section 11.3. The JTabbedPane Class](#)
[Section 11.4. Layout Managers](#)
[Section 11.5. The SpringLayout Class](#)
[Section 11.6. Other Panes](#)

[Chapter 12. Chooser Dialogs](#)
[Section 12.1. The JFileChooser Class](#)
[Section 12.2. The File Chooser Package](#)
[Section 12.3. The Color Chooser](#)
[Section 12.4. The JColorChooser Class](#)
[Section 12.5. Developing a Custom Chooser Panel](#)
[Section 12.6. Developing a Custom Preview Panel](#)
[Section 12.7. Developing a Custom Dialog](#)

[Chapter 13. Borders](#)
[Section 13.1. Introducing Borders](#)
[Section 13.2. Painting Borders Correctly](#)
[Section 13.3. Swing Borders](#)
[Section 13.4. Creating Your Own Border](#)

[Chapter 14. Menus and Toolbars](#)
[Section 14.1. Introducing Swing Menus](#)
[Section 14.2. Menu Bar Selection Models](#)
[Section 14.3. The JMenuBar Class](#)
[Section 14.4. The JMenuItem Class](#)
[Section 14.5. The JPopupMenu Class](#)
[Section 14.6. The JMenu Class](#)
[Section 14.7. Selectable Menu Items](#)
[Section 14.8. Toolbars](#)

[Chapter 15. Tables](#)
[Section 15.1. The JTable Class](#)
[Section 15.2. Implementing a Column Model](#)
[Section 15.3. Table Data](#)
[Section 15.4. Selecting Table Entries](#)
[Section 15.5. Rendering Cells](#)
[Section 15.6. Editing Cells](#)

Section 15.7. Next Steps

Chapter 16. Advanced Table Examples

[Section 16.1. A Table with Row Headers](#)

[Section 16.2. Large Tables with Paging](#)

[Section 16.3. A Table with Custom Editing and Rendering](#)

[Section 16.4. Charting Data with a TableModel](#)

Chapter 17. Trees

[Section 17.1. A Simple Tree](#)

[Section 17.2. Tree Models](#)

[Section 17.3. The JTree Class](#)

[Section 17.4. Tree Nodes and Paths](#)

[Section 17.5. Tree Selections](#)

[Section 17.6. Tree Events](#)

[Section 17.7. Rendering and Editing](#)

[Section 17.8. What Next?](#)

Chapter 18. Undo

[Section 18.1. The Swing Undo Facility](#)

[Section 18.2. The UndoManager Class](#)

[Section 18.3. Extending UndoManager](#)

Chapter 19. Text 101

[Section 19.1. The Swing Text Components](#)

[Section 19.2. The JTextComponent Class](#)

[Section 19.3. The JTextField Class](#)

[Section 19.4. A Simple Form](#)

[Section 19.5. The JPasswordField Class](#)

[Section 19.6. The JTextArea Class](#)

[Section 19.7. How It All Works](#)

Chapter 20. Formatted Text Fields

[Section 20.1. The JFormattedTextField Class](#)

[Section 20.2. Handling Numerics](#)

[Section 20.3. The DefaultFormatter Class](#)

[Section 20.4. The MaskFormatter Class](#)

[Section 20.5. The InternationalFormatter Class](#)

[Section 20.6. The DateFormatter Class](#)

[Section 20.7. The NumberFormatter Class](#)

[Section 20.8. The DefaultFormatterFactory Class](#)

[Section 20.9. Formatting with Regular Expressions](#)

[Section 20.10. The InputVerifier Class](#)

Chapter 21. Carets, Highlighters, and Keymaps

[Section 21.1. Carets](#)

[Section 21.2. Highlighters](#)

[Section 21.3. Keymaps](#)

Chapter 22. Styled Text Panes

[Section 22.1. The JTextPane Class](#)

[Section 22.2. AttributeSets and Styles](#)

[Section 22.3. The Document Model](#)

[Section 22.4. Document Events](#)

[Section 22.5. Views](#)

[Section 22.6. The DocumentFilter Class](#)
[Section 22.7. The NavigationFilter Class](#)

[Chapter 23. Editor Panes and Editor Kits](#)
[Section 23.1. The JEditorPane Class](#)
[Section 23.2. Overview of the Editor Kits](#)
[Section 23.3. HTML and JEditorPane](#)
[Section 23.4. Hyperlink Events](#)
[Section 23.5. The HTMLEditorKit Class](#)
[Section 23.6. Extending HTMLEditorKit](#)
[Section 23.7. Editing HTML](#)
[Section 23.8. Writing HTML](#)
[Section 23.9. Reading HTML](#)
[Section 23.10. A Custom EditorKit](#)

[Chapter 24. Drag and Drop](#)
[Section 24.1. What Is Drag and Drop?](#)
[Section 24.2. The Drop API](#)
[Section 24.3. The Drag Gesture API](#)
[Section 24.4. The Drag API](#)
[Section 24.5. Rearranging Trees](#)
[Section 24.6. Finishing Touches](#)

[Chapter 25. Programming with Accessibility](#)
[Section 25.1. How Accessibility Works](#)
[Section 25.2. The Accessibility Package](#)
[Section 25.3. Other Accessible Objects](#)
[Section 25.4. Types of Accessibility](#)
[Section 25.5. Classes Added in SDK 1.3 and 1.4](#)
[Section 25.6. The Accessibility Utility Classes](#)
[Section 25.7. Interfacing with Accessibility](#)

[Chapter 26. Look and Feel](#)
[Section 26.1. Mac OS X and the Default Look-and-Feel](#)
[Section 26.2. How Does It Work?](#)
[Section 26.3. Key Look-and-Feel Classes and Interfaces](#)
[Section 26.4. The MultiLookAndFeel](#)
[Section 26.5. Auditory Cues](#)
[Section 26.6. Look-and-Feel Customization](#)
[Section 26.7. Creation of a Custom Look-and-Feel](#)

[Chapter 27. Swing Utilities](#)
[Section 27.1. Utility Classes](#)
[Section 27.2. The Timer Class](#)
[Section 27.3. Tooltips](#)
[Section 27.4. Rendering Odds and Ends](#)
[Section 27.5. Event Utilities](#)

[Chapter 28. Swing Under the Hood](#)
[Section 28.1. Working with Focus](#)
[Section 28.2. Multithreading Issues in Swing](#)
[Section 28.3. Lightweight Versus HeavyweightComponents](#)
[Section 28.4. Painting and Repainting](#)
[Section 28.5. Creating Your Own Component](#)

[Appendix A. Look-and-Feel Resources](#)

[Appendix B. Component Actions](#)

[Section B.1. JButton](#)

[Section B.2. JCheckBox](#)

[Section B.3. JCheckBoxMenuItem](#)

[Section B.4. JComboBox](#)

[Section B.5. JDesktopPane](#)

[Section B.6. JEditorPane](#)

[Section B.7. JFormattedTextField](#)

[Section B.8. JInternalFrame](#)

[Section B.9. JLabel](#)

[Section B.10. JList](#)

[Section B.11. JMenu](#)

[Section B.12. JMenuBar](#)

[Section B.13. JMenuItem](#)

[Section B.14. JOptionPane](#)

[Section B.15. JPasswordField](#)

[Section B.16. JPopupMenu](#)

[Section B.17. JProgressBar](#)

[Section B.18. JRadioButton](#)

[Section B.19. JRadioButtonMenuItem](#)

[Section B.20. JRootPane](#)

[Section B.21. JScrollPane](#)

[Section B.22. JScrollPane](#)

[Section B.23. JSlider](#)

[Section B.24. JSpinner](#)

[Section B.25. JSplitPane](#)

[Section B.26. JTabbedPane](#)

[Section B.27. JTable](#)

[Section B.28. JTextArea](#)

[Section B.29. JTextField](#)

[Section B.30. JTextPane](#)

[Section B.31. JToggleButton](#)

[Section B.32. JToolBar](#)

[Section B.33. JToolTip](#)

[Section B.34. JTree](#)

[Section B.35. JViewport](#)

[Section B.36. Non-JComponent Containers](#)

[Section B.37. Auditory Feedback Actions](#)

[Colophon](#)

[Index](#)

Copyright

Copyright 2003, 1998 O'Reilly & Associates, Inc.

Printed in the United States of America.

Published by O'Reilly & Associates, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly & Associates books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safari.oreilly.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly & Associates, Inc. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly & Associates, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps. The association between the image of a spider monkey and the topic of Java Swing is a trademark of O'Reilly & Associates, Inc. Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. O'Reilly & Associates, Inc. is independent of Sun Microsystems, Inc.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Preface

When Java was first released, its user interface facilities were a significant weakness. The Abstract Window Toolkit (AWT) was part of the JDK from the beginning, but it really wasn't sufficient to support a complex user interface. It supported everything you could do in an HTML form and provided free-standing frames, menus, and a few other objects, but you'd be hard-pressed to implement an application as complex as Quicken or Lotus Notes. AWT also had its share of portability problems; it relied heavily on the runtime platform's native user interface components, and it wasn't always possible to hide differences in the way these components behaved.

JDK 1.1 fixed a number of problems—most notably, it introduced a new event model that was much more efficient and easier to use—but it didn't make any major additions to the basic components. We got a ScrollPane and a PopupMenu, but that was about it. Furthermore, AWT still relied on the native components and therefore continued to have portability problems.

In April 1997, Sun's Java group (then called JavaSoft) announced the Java Foundation Classes, or JFC, which supersedes (and includes) AWT. A major part of the JFC was a set of much more complete, flexible, and portable user interface components called "Swing." (The JFC also includes a comprehensive facility for 2D graphics, printing, and Drag and Drop.) With Swing, you can design interfaces with tree components, tables, tabbed dialogs, tooltips, and a growing set of other features that computer users are accustomed to.

In addition to the new components, Swing made three major improvements to the AWT. First, Swing doesn't rely on the runtime platform's native components. It's written entirely in Java and creates its own components. This approach solved most of the portability problems since components don't inherit weird behaviors from the runtime environment or do they work against its grain. Second, because Swing is in complete control of the components, it's in control of the way components look on the screen and gives you more control over how your applications look. You can choose between several pre-built "look-and-feels" (L&Fs), or you can create your own if you want your software to show your personal style (more appropriate for games than for daily productivity software, of course). This feature is called "Pluggable Look-and-Feel," or PLAF. Third, Swing makes a very clear distinction between the data a component displays (the "model") and the actual display (the "view"). While the fine points of this distinction are appreciated mostly by computer scientists, it has important implications for all developers. This separation means that components are extremely flexible. It's easy to adapt components to display new kinds of data that their original design didn't anticipate or to change the way a component looks without getting tangled up in assumptions about the data it represents.

The first official release of Swing, for use with JDK 1.1, took place in the spring of 1998. Swing (and the rest of JFC) was built into Java 2 and revolutionized Java user interface development. The Swing components continue to evolve with Java, and Java 2 SDK 1.4 is the best version yet. This book shows you how to join the revolution.

What This Book Covers

This book gives a complete introduction to the entire Swing component set. Of course, it shows you how to use all of the components: how to display them on the screen, register for events, and get information from them. You'd expect that in any Swing book. This book goes much further. It goes into detail about the model-delegate architecture behind the components and discusses all of the data models. Understanding the models is essential when you're working on an application that requires something significantly different from the components' default behavior. For example, if you need a component that displays a different data type or one that structures data in some nonstandard way, you'll need to work with the data models. This book also discusses how to write "accessible" user interfaces and how to create your own look-and-feel.

There are a few topics this book doesn't cover, despite its girth. We assume you know the Java language. For Swing, it's particularly important to have a good grasp of inner classes (both named and anonymous), which are used by Swing itself and in our examples. We assume that you understand the JDK 1.1 event model, Java's mechanism for communicating between asynchronous threads. Swing introduced many new event types, all of which are discussed in this book, but we provide only an overview of the event mechanism as a whole. We also assume that you understand the older AWT components, particularly the Component and Container classes, which are superclasses of the Swing's JComponent. We assume that you understand the AWT layout managers, all of which are usable within Swing applications. If you are new to Java, or would like a review, you can find a complete discussion of these topics in the Java AWT Reference by John Zukowski [1] or a solid introduction in Learning Java by Pat Niemeyer and Jonathan Knudsen (both published by O'Reilly). We do not assume that you know anything about other JFC topics, like Java 2D—check out Java 2D by Jonathan Knudsen for that; all the drawing and font manipulation in this book can be done with AWT. (We do cover the JFC Accessibility API, which is supported by every Swing component, as well as the drag-and-drop facility, since this functionality is a requirement for modern user interfaces.)

[1] PDFs for the Java AWT Reference are available at this book's web site, <http://www.oreilly.com/catalog/jswing2>.

The major Swing classes fall into the following packages:

javax.accessibility

Classes that support accessibility for people who have difficulty using standard user interfaces. Covered in [Chapter 25](#)

javax.swing

The bulk of the Swing components. Covered in [Chapter 3](#)-[Chapter 14](#) and [Chapter 27](#)-[Chapter 28](#).

javax.swing.border

Classes for drawing fancy borders around components. Covered in [Chapter 13](#).

javax.swing.colorchooser

Classes providing support for the JColorChooser component. Covered in [Chapter 12](#).

javax.swing.event

Swing events. Covered throughout the book.

javax.swing.filechooser

Classes providing support for the JFileChooser component. Covered in [Chapter 12](#).

javax.swing.plaf

Classes supporting the PLAF, including classes that implement the Metal and Multi L&Fs. (Implementations of the Windows and Motif L&Fs are packaged under com.sun.java.swing.plaf, and the Macintosh Aqua L&F is under com.apple.mrj.swing.) Covered in [Chapter 26](#).

javax.swing.table

What's New in This Edition?

This second edition covers the latest developments in the Java 2 Standard Edition SDK 1.3 and 1.4. We've tried to highlight the changes from 1.2 in case you have to work with older releases for compatibility or political reasons.

For brevity's sake, we refer to Java versions by their SDK version number, describing this or that feature as having been introduced in SDK 1.3 or 1.4. Earlier versions were called Java Development Kits, so in those cases we refer to JDK 1.1 or 1.2.

This new edition incorporated your feedback from the first edition! The first edition was too heavy on the documentation side for many readers. The Javadoc for the Swing packages continues to improve, and more and more people are familiar with the patterns of Java classes and methods. With those two facts in mind, we try to focus on the parts of the API that are interesting and useful rather than just including them because they exist. We added many new examples and improved the existing examples. This book is a true and thorough revision of the first edition, not a mere update.

As a quick reference to some of the changes you'll find in the 1.3 and 1.4 releases of the SDK, [Table P-1](#) and [Table P-2](#) list any significant changes to components and briefly describe those changes. We detail these changes throughout the book as we discuss the particular components.

Table P-1. Swing changes in the Java 2 SDK 1.3

Component or feature	In chapter	Description of changes or additions
JTree	Chapter 17	Several new properties were added, including the click count to start editing and the selection path.
JTable	Chapter 15	Improved general performance and cell rendering. AbstractCellEditor is now the parent class of the DefaultCellEditor used by tables.
JSplitPane	Chapter 11	A new resizeWeight property was added, and the dividerLocationProperty is now bound.
JFileChooser	Chapter 12	You can now remove the Ok and Cancel buttons. A new property, acceptAllFileFilterUsed, was added.
JCheckBox	Chapter 5	Added new borderPaintedFlat property.
DefaultButtonModel	Chapter 5	Added new getGroup() method.
		Several fixes and newly public classes and methods. Internal frames are now

On the Web Site

The web site for this book, <http://www.oreilly.com/catalog/jswing2/>, offers some important materials you'll want to know about. All the examples in this book can be found there, as well as free utilities, PDFs of John Zukowski's Java AWT Reference (foundational for understanding Swing), and selected material from the first edition for those of you working with older SDKs.

The examples are available as a JAR file, a ZIP archive, and a compressed TAR archive. The files named *swing* were tested against J2SE SDK 1.4 for this edition. The files named *swing-1e* were tested against JDK 1.2 for the first edition of the book. The files named *swing-old* were written with the beta releases of Swing and use the com.java.swing hierarchies.

We also include a few free utilities on the site that you may want to check out:
macmetrics.jar

Lee Ann Rucker's MacMetrics theme. See [Section 26.1](#) for details on this helpful tool that enables developers without access to Mac OS X to see how their applications' interfaces will look on that platform.
oraswing.jar

Our very own utilities bundle with documentation, including:
eel.jar

The Every Event Listener utility for debugging events from the various Swing and AWT components.
relativelayout.jar

A nifty XML-based layout manager.
mapper.jar

A quick helper for discovering the InputMap and ActionMap entries (both bound and unbound) for any given component. This is the utility we used to build [Appendix B](#).

We may add other utilities as we receive feedback from readers, so be sure to check the README file on the site!

We owe a debt of gratitude to John Zukowski and O'Reilly & Associates, who have graciously allowed the classic Java AWT Reference to be placed online at our site. You can download PDFs of the entire book.

The web site also includes some expanded material that we couldn't shoehorn into this edition of the book. For those of you still working with JDK 1.2, we've included a PDF containing the "Keyboard Actions" section from Chapter 3 of the first edition—the approach changed markedly with SDK 1.3. Regardless of your version of Java, if you're planning on extending the HTMLEditorKit, you should check out the expanded material online. We cover the basics of this editor kit in [Chapter 23](#), but for those of you who want to dig in deep, you should download PDFs of the two chapters devoted to this topic..

Conventions

This book follows certain conventions for font usage, property tables, and class diagrams. Understanding these conventions up-front makes it easier to use this book.

This book uses the following font conventions:

Italic

Used for filenames, file extensions, URLs, application names, emphasis, and new terms when they are first introduced
Constant width

Used for Java class names, functions, variables, components, properties, data types, events, and snippets of code that appear in text

Constant width bold

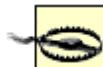
Used for commands you enter at the command line and to highlight new code inserted in a running example

Constant width italic

Used to annotate output



This icon designates a note, which is an important aside to the nearby text.



This icon designates a warning relating to the nearby text.

Properties Tables

Swing components are all JavaBeans. Properties provide a powerful way to work with JavaBeans, so we use tables throughout the book to present lists of properties. [Table P-3](#) is an example from the hypothetical JFoo class that shows how we use these tables.

Table P-3. Properties of the fictional JFoo class

Property	Data type	get	is	set	Default value
opaqueb, o, 1.4	boolean				true
bbound, o overridden, 1.4since 1.4					

See also
properties from
the JComponent

How to Contact Us

Along with O'Reilly, we have verified the information in this book to the best of our abilities, but you may find that features have changed (or even that we have made mistakes!). Please let us know about any errors you find, as well as your suggestions for future editions, by writing to:

O'Reilly & Associates, Inc. 1005 Gravenstein Highway North Sebastopol, CA 95472 (800) 998-9938 (U.S. and Canada) (707) 829-0515 (international/local) (707) 829-0104 (fax)

You can also contact O'Reilly by email. To be put on the mailing list or request a catalog, send a message to:
info@oreilly.com

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

<http://www.oreilly.com/catalog/jswing2/>

To ask technical questions or comment on the book, send email to:

bookquestions@oreilly.com

For more information about O'Reilly books, conferences, Resource Centers, and the O'Reilly Network, see O'Reilly's web site at:

<http://www.oreilly.com/>

Acknowledgments

We're particularly indebted to our technical reviewers for this second edition: Christian Hessler, John Pyeatt, Maciek Smuga-Otto, and Dave Wood.

Marc Loy

I'll start off the mutual admiration society by thanking my cohorts Jim and Brian. They came to the table after we lost Dave and Bob (from the first edition) to other books, and well, life in general. This update would not have been possible without them. Our editor Deb Cameron has the patience and diligence of some very patient and diligent god. I continue to be amazed by the support and insight I receive from my colleagues Tom Berry, Damian Moshak, and Brooks Graham. Gratitude for the inspiration to keep writing (even if it is technical) goes to Amy Hammond, my sister and confidante. A special thanks to Kathy Godeken for an early push in the right direction. Words are not enough to thank my partner Ron, so I'll not waste the space.

Brian Cole

Thanks to my family for putting up with me as an author. This goes tenfold for my partner, Beth, for that and more. Thanks to Deb, who was very understanding about deadlines, and especially to Marc and Jim, who were always willing to lend a hand despite deadlines of their own. Thanks to my employers and coworkers, who were willing to accommodate my schedule. Finally, thanks to the anonymous programmer who discovered that running java with -Dsun.java2d.noddraw=true fixes the appalling 1.3 drawing problems common on Win32 systems equipped with some popular types of video cards. You saved me a lot of time.

James Elliott

Any list of thanks has to start with my parents for fostering my interest in computing even when we were living in countries that made that a major challenge, and with my partner Joe for putting up with it today when it has flowered into a major obsession. I'd also like to acknowledge my Madison employer, Berbee, for giving me an opportunity to delve deeply into Java and build skills as an architect of reusable APIs; for letting me stay clear of the proprietary, platform-specific tar pit that is engulfing so much of the programming world; for surrounding me with such incredible colleagues; and for being supportive when I wanted to help with this book. Of course, I have Marc to thank for getting me involved in this crazy adventure in the first place, and Deb for helping make sense of it.

I wanted to be sure this edition gave good advice about how to work with Swing on Mac OS X, Apple's excellent, Unix-based environment for Java development, so I asked for some help. Lee Ann Rucker (who should also be thanked for her heroic work of single-handedly implementing the new Mac OS Look-and-Feel while on loan from Sun to Apple) shared some great ideas and approaches to writing solid, cross-platform Java applications, including the MacMetrics theme described in [Chapter 26](#). Count me among the many people wishing Sun or Apple would put her back on the Mac Java team! Eric Albert, another frequent source of insight on Apple's Java-Dev mailing list, gave me more suggestions and pointed me to his excellent chapter in Early Adopter Mac OS X Java (Wrox Press). Finally, Matt Drance at Apple's Developer Technical Support sent me an early (and helpful) version of his technical note on how to make Java applications as Mac-friendly as possible. There are many others to whom I'm indebted, but I've already used more than my fair share of space, so the rest of you know who you are!

We all want to thank the many members of O'Reilly's production department, who put in lots of work under a tight schedule.

Chapter 1. Introducing Swing

Welcome to Swing! By now, you're probably wondering what Swing is and how you can use it to spice up your Java applications. Or perhaps you're curious as to how the Swing components fit into the overall Java strategy. Then again, maybe you just want to see what all the hype is about. Well, you've come to the right place; this book is all about Swing and its components. So let's dive right in and answer the first question that you're probably asking right now, which is...

1.1 What Is Swing?

If you poke around the Java home page (<http://java.sun.com/>), you'll find Swing described as a set of customizable graphical components whose look-and-feel (L&F) can be dictated at runtime. In reality, however, Swing is much more than this. Swing is the next-generation GUI toolkit that Sun Microsystems created to enable enterprise development in Java. By *enterprise development*, we mean that programmers can use Swing to create large-scale Java applications with a wide array of powerful components. In addition, you can easily extend or modify these components to control their appearance and behavior.

Swing is not an acronym. The name represents the collaborative choice of its designers when the project was kicked off in late 1996. Swing is actually part of a larger family of Java products known as the Java Foundation Classes (JFC), which incorporate many of the features of Netscape's Internet Foundation Classes (IFC) as well as design aspects from IBM's Taligent division and Lighthouse Design. Swing has been in active development since the beta period of the Java Development Kit (JDK) 1.1, circa spring of 1997. The Swing APIs entered beta in the latter half of 1997 and were initially released in March 1998. When released, the Swing 1.0 libraries contained nearly 250 classes and 80 interfaces. Growth has continued since then: at press time, Swing 1.4 contains 85 public interfaces and 451 public classes.

Although Swing was developed separately from the core Java Development Kit, it does require at least JDK 1.1.5 to run. Swing builds on the event model introduced in the 1.1 series of JDKs; you cannot use the Swing libraries with the older JDK 1.0.2. In addition, you must have a Java 1.1-enabled browser to support Swing applets. The Java 2 SDK 1.4 release includes many updated Swing classes and a few new features. Swing is fully integrated into both the developer's kit and the runtime environment of all Java 2 releases (SDK 1.2 and higher), including the Java Plug-In.

1.1.1 What Are the Java Foundation Classes?

The FC is a suite of libraries designed to assist programmers in creating enterprise applications with Java. The Swing API is only one of five libraries that make up the JFC. The JFC also consists of the Abstract Window Toolkit (AWT), the Accessibility API, the 2D API, and enhanced support for Drag and Drop capabilities. While the Swing API is the primary focus of this book, here is a brief introduction to the other elements in the JFC:

AWT

The Abstract Window Toolkit is the basic GUI toolkit shipped with all versions of the Java Development Kit. While Swing does not reuse any of the older AWT components, it does build on the lightweight component facilities introduced in AWT 1.1.

Accessibility

The accessibility package provides assistance to users who have trouble with traditional user interfaces. Accessibility tools can be used in conjunction with devices such as audible text readers or braille keyboards to allow direct access to the Swing components. Accessibility is split into two parts: the Accessibility API, which is shipped with the Swing distribution, and the Accessibility Utilities API, which is distributed separately. All Swing components support accessibility, so this book dedicates an entire chapter ([Chapter 25](#)) to accessibility design and use.

2D API

The 2D API contains classes for implementing various painting styles, complex shapes, fonts, and colors. This Java package is loosely based on APIs that were licensed from IBM's Taligent division. The 2D API classes are not part of Swing, so they are not covered in this book.

Drag and Drop

Drag and Drop (DnD) is one of the more common metaphors used in graphical interfaces today. The user is allowed to click and "hold" a GUI object, moving it to another window or frame in the desktop with predictable results. The DnD API allows users to implement droppable elements that transfer information between Java applications and native applications. Although DnD is not part of Swing, it is crucial to a commercial-quality application. We tackle

1.2 Swing Features

Swing provides many features for writing large-scale applications in Java. Here is an overview of some of the more popular features.

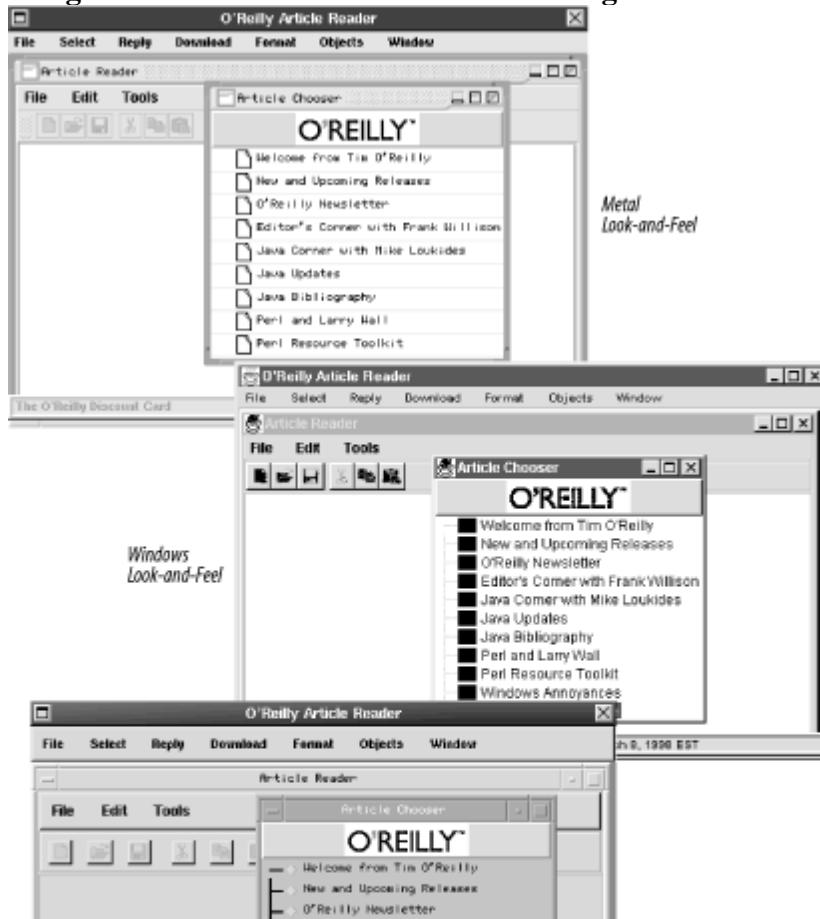
1.2.1 Pluggable Look-and-Feels

One of the most exciting aspects of the Swing classes is the ability to dictate the L&F of each of the components, even resetting the L&F at runtime. L&Fs have become an important issue in GUI development over the past 10 years. Many users are familiar with the Motif style of user interface, which was common in Windows 3.1 and is still in wide use on Unix platforms. Microsoft created a more optimized L&F in their Windows 95/98/NT/2000 operating systems. In addition, the Macintosh computer system has its own carefully designed L&F, which most Apple users feel comfortable with.

Swing is capable of emulating several L&Fs and currently supports the Windows, Unix Motif, and "native" Java Metal L&Fs. Mac OS X comes with full support for its own L&F based on Apple's Aqua Human Interface Guidelines, although you can still access Metal if you prefer. In addition, Swing allows the user to switch L&Fs at runtime without having to close the application. This way, a user can experiment to see which L&F is best for her with instantaneous feedback. (In practice, nobody really does this, but it's still pretty cool from a geeky point of view.) And, if you're feeling really ambitious as a developer (perhaps a game developer), you can create your own L&F for each one of the Swing components!

The Metal L&F combines some of the best graphical elements in today's L&Fs and even adds a few surprises of its own. [Figure 1-3](#) shows an example of several L&Fs that you can use with Swing, including the Metal L&F. All Swing L&Fs are built from a set of base classes called the Basic L&F. However, though we may refer to the Basic L&F from time to time, you can't use it on its own. If you're lucky enough to be developing applications in the Mac OS X environment, you'll be familiar with the L&F shown in [Figure 1-4](#).

Figure 1-3. Various L&Fs in the Java Swing environment



1.3 Swing Packages and Classes

Here is a short description of each package in the Swing libraries:

`javax.accessibility`

Contains classes and interfaces that can be used to allow *assistive technologies* to interact with Swing components. Assistive technologies cover a broad range of items, from audible text readers to screen magnification. Although the accessibility classes are technically not part of Swing, they are used extensively throughout the Swing components. We discuss the accessibility package in greater detail in [Chapter 25](#).

`javax.swing`

Contains the core Swing components, including most of the model interfaces and support classes.

`javax.swing.border`

Contains the definitions for the abstract border class as well as eight predefined borders. Borders are not components; instead, they are special graphical elements that Swing treats as properties and places around components in place of their insets. If you wish to create your own border, you can subclass one of the existing borders in this package, or you can code a new one from scratch.

`javax.swing.colorchooser`

Contains support for the JColorChooser component, discussed in [Chapter 12](#).

`javax.swing.event`

Defines several new listeners and events that Swing components use to communicate asynchronous information between classes. To create your own events, you can subclass various events in this package or write your own event class.

`javax.swing.filechooser`

Contains support for the JFileChooser component, discussed in [Chapter 12](#).

`javax.swing.plaf`

Defines the unique elements that make up the pluggable L&F for each Swing component. Its various subpackages are devoted to rendering the individual L&Fs for each component on a platform-by-platform basis. (Concrete implementations of the Windows and Motif L&Fs are in subpackages of com.sun.java.swing.plaf, and the Mac OS L&F is under com.apple.mrj.swing.)

`javax.swing.table`

Provides models and views for the table component, which allows you to arrange various information in a grid format with an appearance similar to a spreadsheet. Using the lower-level classes, you can manipulate how tables are viewed and selected, as well as how they display their information in each cell.

`javax.swing.text`

Provides scores of text-based classes and interfaces supporting a common design known as *document/view*. The text classes are among the more advanced Swing classes to learn, so we devote several chapters ([Chapter 19](#)-[Chapter 23](#)) to both the design fundamentals and the implementation of text applications.

`javax.swing.text.html`

Used specifically for reading and formatting HTML text through an ancillary editor kit.

`javax.swing.text.html.parser`

Contains support for parsing HTML.

`javax.swing.text.rtf`

Used specifically for reading and formatting Rich Text Format (RTF) text through an ancillary editor kit.

`javax.swing.text.rtf.parser`

1.4 The Model-View-Controller Architecture

Swing uses the *model-view-controller architecture* (MVC) as the fundamental design behind each of its components. Essentially, MVC breaks GUI components into three elements. Each of these elements plays a crucial role in how the component behaves.

Model

The model encompasses the state data for each component. There are different models for different types of components. For example, the model of a scrollbar component might contain information about the current position of its adjustable "thumb," its minimum and maximum values, and the thumb's width (relative to the range of values). A menu, on the other hand, may simply contain a list of the menu items the user can select from. This information remains the same no matter how the component is painted on the screen; model data is always independent of the component's visual representation.

View

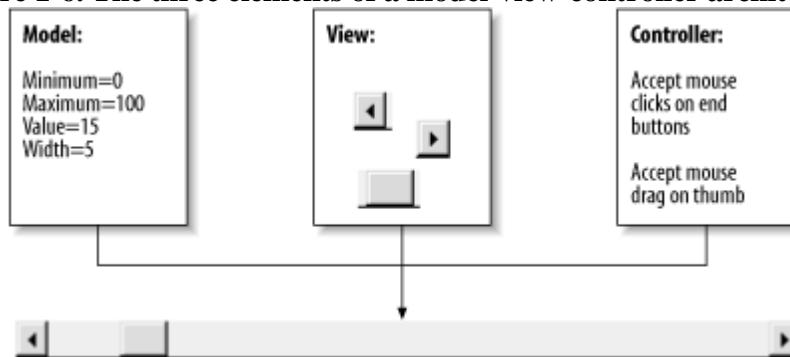
The view refers to how you see the component on the screen. For a good example of how views can differ, look at an application window on two different GUI platforms. Almost all window frames have a title bar spanning the top of the window. However, the title bar may have a close box on the left side (like the Mac OS platform), or it may have the close box on the right side (as in the Windows platform). These are examples of different types of views for the same window object.

Controller

The controller is the portion of the user interface that dictates how the component interacts with events. Events come in many forms — e.g., a mouse click, gaining or losing focus, a keyboard event that triggers a specific menu command, or even a directive to repaint part of the screen. The controller decides how each component reacts to the event—if it reacts at all.

[Figure 1-6](#) shows how the model, view, and controller work together to create a scrollbar component. The scrollbar uses the information in the model to determine how far into the scrollbar to render the thumb and how wide the thumb should be. Note that the model specifies this information relative to the minimum and the maximum. It does not give the position or width of the thumb in screen pixels—the view calculates that. The view determines exactly where and how to draw the scrollbar, given the proportions offered by the model. The view knows whether it is a horizontal or vertical scrollbar, and it knows exactly how to shadow the end buttons and the thumb. Finally, the controller is responsible for handling mouse events on the component. The controller knows, for example, that dragging the thumb is a legitimate action for a scrollbar, within the limits defined by the endpoints, and that pushing on the end buttons is acceptable as well. The result is a fully functional MVC scrollbar.

Figure 1-6. The three elements of a model-view-controller architecture



1.4.1 MVC Interaction

With MVC, each of the three elements—the model, the view, and the controller—requires the services of another element to keep itself continually updated. Let's continue discussing the scrollbar component.

1.5 Working with Swing

Our introduction to Swing wouldn't be complete unless we briefly mentioned some caveats of the Swing libraries. There are two pertinent areas: multithreading and lightweight versus heavyweight components. Being aware of these issues will help you make informed decisions while working with Swing. [Chapter 28](#) gives you in-depth guidance in these difficult areas.

1.5.1 Multithreading

Shortly before the initial release of Swing, Sun posted an article recommending that developers not use independent threads to change model states in components.[\[2\]](#) Instead, once a component has been painted to the screen (or is about to be painted), updates to its model state should occur only from the *event-dispatching queue*. The event-dispatching queue is a system thread used to communicate events to other components. It posts GUI events, including those that repaint components.

[2] Hans Muller and Kathy Walrath, "Threads and Swing," The Swing Connection,
<http://java.sun.com/products/jfc/tsc/swingdoc-archive/threads.html>.

The issue here is an artifact of the MVC architecture and deals with performance and potential race conditions. As we mentioned, a Swing component draws itself based on the state values in its model. However, if the state values change while the component is in the process of repainting, the component may repaint incorrectly—this is unacceptable. To compound matters, placing a lock on the entire model, as well as on some of the critical component data, or even cloning the data in question, could seriously hamper performance for each refresh. The only feasible solution, therefore, is to place state changes in serial with refreshes. This ensures that modifications in component state do not occur at the same time Swing is repainting any components and prevents race conditions.

1.5.2 The Z-Order Caveat: Lightweight and Heavyweight Components

One of the most frequent issues to come out of lightweight/heavyweight component use is the idea of depth, or *z-order*—that is, a well-defined method for how elements are stacked on the screen. Because of z-order, it is not advisable to mix lightweight and heavyweight components in Swing.

To see why, remember that heavyweight components depend on peer objects used at the operating system level. However, with Swing, only the top-level components are heavyweight: JApplet, JFrame, JDialog, and JWindow. Also, recall that heavyweight components are always "opaque"—they have a rectangular shape and are nontransparent. This is because the host operating system typically allocates the entire painting region to the component, clearing it first.

The remaining components are lightweight. So here is the crux of the dilemma: when a lightweight component is placed inside a heavyweight container, it shares (and actually borrows) the graphics context of the heavyweight component. The lightweight component must always draw itself on the same plane as the heavyweight component that contains it; as a result, it shares the z-order of the heavyweight component. In addition, lightweight components are bound to the clipping region of the top-level window or dialog that contains them. In effect, lightweight components are all "drawings" on the canvas of a heavyweight component. The drawings cannot go beyond the boundaries of the canvas and can always be covered by another canvas. Heavyweight components, however, are free from this restriction. Therefore, they always appear on top of the lightweight components — whether that is the intent or not.

Heavyweight components have other ramifications in Swing as well. They do not work well in scrollpanes, where they can extend beyond the clipping boundaries; they also don't work in front of lightweight menus and menu bars (unless certain precautions are taken) or inside internal frames. Some Swing classes, however, offer an interesting

1.6 The Swing Set Demo

If you're in a hurry to see all the components Swing has to offer, be sure to check out the Swing Set demonstration. The demonstration is extremely easy to set up. If you have the 1.3 or 1.4 SDK, the demonstration is included. If you have 1.2, you must first download and extract the demo classes and add them to your classpath. Then follow these steps:

1.

Change the directory to the *demo/jfc/SwingSet2* directory. (For the 1.2 release, the directory is *demo/jfc/SwingSet*.)

2.

Run the SwingSet2 (or SwingSet for 1.2) jar file:

```
% java -jar SwingSet2.jar
```

You should immediately see a splash screen indicating that the Swing Set demo is loading. When it finishes, a window appears, similar to the one in [Figure 1-9](#).

Figure 1-9. The Swing Set demo



This demo contains a series of tabs that demonstrate almost all of the components in the Swing libraries. Be sure to check out the internal frames demo and the Metal L&F. In addition, some of the Swing creators have added "Easter eggs" throughout the Swing Set demo. See if you can find some!

1.7 Reading This Book

We're well aware that most readers don't read the [Preface](#). You have our permission to skip it, provided that you look at the [Conventions](#) section. That section is particularly important because in this book we experiment with a few new techniques for explaining the Swing classes. As we said earlier, everything in Swing is a JavaBean. This means that much of an object's behavior is controlled by a set of properties, which are manipulated by accessor methods. For example, the property color is accessed by the getColor() (to find out the color) and setColor() (to change the color) methods. If a property has a boolean value, the get method is often replaced by an is method; for example, the visible property would have the isVisible() and setVisible() methods.

We found the idea of properties very powerful in helping us understand Swing. Therefore, rather than listing all of a class's accessor methods, we decided to present a table for each class, listing the class's properties and showing the property's data type, which accessor methods are present, whether the property is "bound" (i.e., changing the property generates a `PropertyChangeEvent`), when it was introduced (1.2 is the default; 1.3 and 1.4 are marked where appropriate), and the property's default value. This approach certainly saves paper (you didn't really want a 2,000-page book, did you?) and should make it easier to understand what a component does and how it is structured. Furthermore, if you're not already in the habit of thinking in terms of the JavaBeans architecture, you should get in the habit. It's a very powerful tool for understanding component design.

The conventions we use in the property tables — plus some other conventions that we use in class diagrams — are explained in the Preface. So you may ignore the rest of the Preface as long as you familiarize yourself with the conventions we're using.

The next chapter helps AWT developers get a jump on Swing by presenting a simple application; those without AWT experience may just want to skim the chapter. In [Chapter 3](#), we continue our discussion by presenting some of the fundamental classes of Swing and describing how you can use the features inherent in each of these classes to shorten your overall development time. Don't stop now—the best is yet to come!

Chapter 2. Jump-Starting a Swing Application

Now that you have an overview of Swing, let's look at a few Swing components you can put into your applications right now. This chapter shows you how to add images to buttons and how to create a rudimentary Swing application using internal frames. We won't belabor the theory and background. You'll find everything we talk about now (and tons more we don't discuss here) presented in later chapters in much greater detail. We just want to show you some of the fun stuff right away.

This chapter, and only this chapter, assumes that you have prior experience with AWT and AWT-based programs that you'd like to upgrade to use lightweight Swing components. If you are new to Java, this may not be the case; you are probably interested in learning Swing without the need to upgrade AWT applications. You can either skim this chapter or skip ahead to [Chapter 3](#), which lays a foundation for the rest of your work in Swing.

If you want to see how easily Swing components can be dropped into existing AWT applications, though, read on.

2.1 Upgrading Your AWT Programs

One of the benefits of object-oriented languages is that you can upgrade pieces of a program without rewriting the rest. While practice is never as simple as theory, with Swing it's close. You can use most of the Swing components as drop-in replacements for AWT components with ease. The components sport many fancy new features worth exploiting, but they still maintain the functionality of the AWT components you're familiar with. As a general rule, you can stick a "J" in front of your favorite AWT component and put the new class to work as a Swing component. Constructors for components such as JButton, JTextField, and JList can be used with the same arguments and generate the same events as Button, TextField, and List. Some Swing containers, like JFrame, take a bit of extra work, but not much.

Graphical buttons are essential to modern user interfaces. Nice monitors and cheap hardware have made icons almost a necessity. The AWT package in Java does not directly support image buttons. You could write an extension to support them easily enough, but why bother when Swing's JButton class provides a standard way to add image buttons?

2.2 A Simple AWT Application

You probably have some programs lying around that use regular AWT buttons that you'd love to replace with image buttons, but you don't have the time or, honestly, the necessity to produce your own image button class. Let's look at a simple application that demonstrates an upgrade path you can use on your own programs.

First, let's look at the code for this very simple application:

```
// ToolbarFrame1.java
```

2.3 Including Your First Swing Component

The first step in adding a Swing component to your application is preparing the Swing package for use. As long as you have installed SDK 1.2 or later, you don't have to take any special steps to use the Swing classes. If you're preparing an application to run with JDK 1.1, you'll need to put the *swingall.jar* file on the CLASSPATH so that the Swing components are available during compilation and at runtime.

In your source code, you include the Swing package by adding an import statement:

```
import javax.swing.*;
```

Now you're ready to replace your Button objects with JButton objects. We'll also set up the application to take advantage of Swing's L&F capabilities; we've put another row of buttons at the bottom of the frame that let you select one of the standard L&Fs:

```
// ToolbarFrame2.java
```

2.4 Beyond Buttons

Buttons are very useful, but even with great images forming the buttons, they still lack a certain glamour—every application has buttons. For the next example, let's take a look at JInternalFrame, which allows you to create free-standing frames with menus, title bars, and everything else a Frame needs right inside your application.

2.5 What Is an Internal Frame?

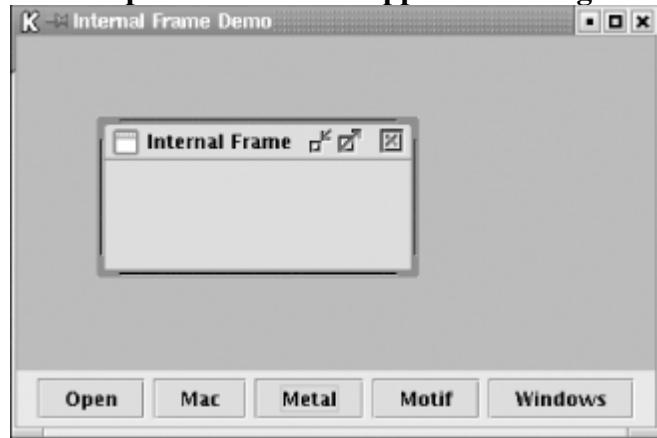
Before we start coding, here's a brief rundown of the features of an internal frame:

- Same functions as a normal Frame object, but confined to the visible area of the container it is placed in
- Can be iconified (icon stays inside main application frame)
- Can be maximized (frame consumes entire main application frame area)
- Can be closed using the standard controls for application windows
- Can be placed in a "layer," which dictates how the frame displays itself relative to other internal frames (a frame in layer 1 can never hide a frame in layer 2)

To be honest, in practice, standalone frames are often more useful than internal frames. You'll want to know about both; we have chapters dedicated to each of these topics ([Chapter 8](#) and [Chapter 9](#), respectively).

[Figure 2-6](#) shows a simple internal frame using the Metal L&F.

Figure 2-6. The SimpleInternalFrame application using the Metal L&F



For this first example, we'll add an empty internal frame to an application. Once that's working, we'll expand the simple frame to create a couple of different types of internal frames and create the framework for a simple application.

One of the prerequisites for using internal frames is that you need a window capable of managing them. The Swing package provides the JDesktopPane class for this purpose. You'll see the details of the JDesktopPane in [Chapter 9](#), but for now, here's how to get one started:

```
// Set up the layered pane.
```

2.6 A Bigger Application

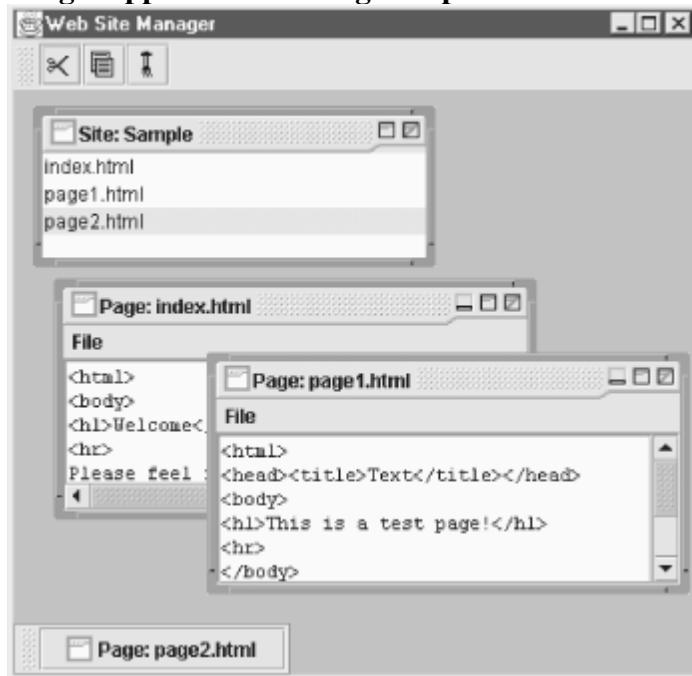
Now that you've seen how to create internal frames and played around with them a bit, let's tackle a slightly larger problem. We want to build an application that can pop up internal frames that you can actually use. This starter application is a web site manager that shows us a list of HTML pages at a site and, for any of those pages, allows us to pop up the page in a separate frame and edit it. We'll keep the main list of HTML pages in one "site" frame that contains a simple list box.

Once you have a site built up with a couple of pages, you can click on any entry in the list, and if the file exists, we'll create a new "page" frame and load the file into a JTextArea object for you to edit. You can modify the text and save the file using the File menu in the page frame.

As a bonus, we'll put those cut, copy, and paste icons to use as well. You can manipulate text in any of the open page frames. The icons work as Action objects by looking at the selected text and insertion point of the active frame. (We alluded to the Action class after our last Toolbar example. We'll demonstrate it here and discuss it thoroughly at the start of the next chapter.) If the active frame is a site frame, nothing happens.

You could certainly add a lot of features to this application and make it a real working program, but we don't want to get mired down in details just yet. (If you want to get really fancy, you could look at some of the editor kits discussed in [Chapter 23](#) and build yourself a real HTML editor.) [Figure 2-9](#) shows the finished application with a couple of open frames.

Figure 2-9. The SiteManager application running on a platform where Metal is the default L&F



We break the code for this application into three separate classes to make discussing it more manageable. The first class handles the real application frame. The constructor handles all of the interface setup work. It sets up the toolbar, as well as the Cut, Copy, and Paste buttons. It uses the default L&F for the platform on which it is run. (You could certainly attach the LnFLListener, if you wanted to.) Here's the source code:

```
// SiteManager.java
```

I1@ve RuBoard

< PREVIOUS | NEXT >

Chapter 3. Swing Component Basics

The previous chapter showed how easy it is to create some impressive programs with Swing components. Now it's time to dig in a little deeper. We begin this chapter by presenting an overview of a few key (but lower-level) helper classes such as Action, GraphicsContext, ChangeEvent, and PropertyChangeEvent, as well as the HeadlessException exception. We spend the remainder of the chapter introducing the JComponent class, the heart and soul of all Swing components.

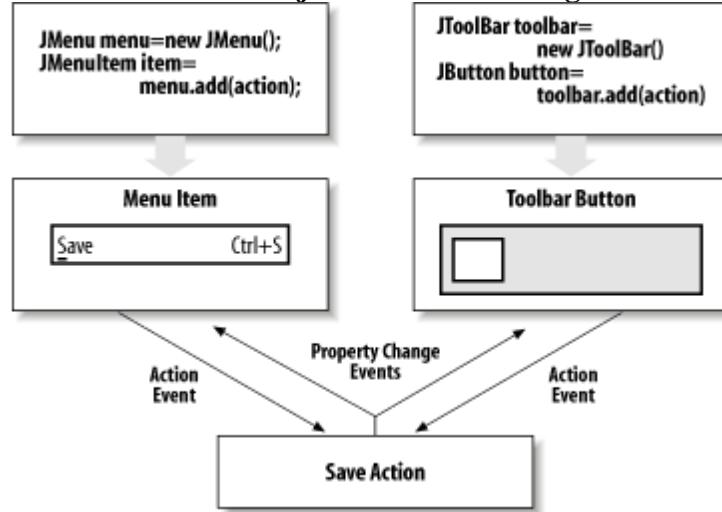
3.1 Understanding Actions

Actions are a popular addition to Swing. An action allows a programmer to bundle a commonly used procedure and its bound properties (such as its name and an image to represent it) into a single class. This construct comes in handy if an application needs to call upon a particular function from multiple sources. For example, let's say that a Swing programmer creates an action that saves data to disk. The application could then invoke this action from both the Save menu item on the File menu and the Save button on a toolbar. Both components reference the same action object, which saves the data. If the Save function is disabled for some reason, this property can be set in the action as well. The menu and toolbar objects are automatically notified that they can no longer save any data, and they can relay that information to the user.

3.1.1 Actions and Containers

Swing containers, such as JMenu, JPopupMenu, and JToolBar, can each accept action objects with their add() methods. When an action is added, these containers automatically create a GUI component, which the add() method then returns to you for customization. For example, a JMenu or a JPopupMenu creates and returns a JMenuItem from an Action while a JToolBar creates and returns a JButton. The action is then paired with the newly created GUI component in two ways: the GUI component registers as a PropertyChangeListener for any property changes that might occur in the action object, while the action object registers as an ActionListener on the GUI component. [Figure 3-1](#) shows the interactions between a menu item or toolbar and an Action.

Figure 3-1. An action in conjunction with a Swing item and toolbar



Essentially, this means that if the menu item or button is selected by the user, the functionality inside the action is invoked. On the other hand, if the action is disabled, it sends a PropertyChangeEvent to both the menu item and the toolbar, causing them to disable and turn gray. Similarly, if the action's icon or name is changed, the menu and toolbar are automatically updated.

3.1.2 The Action Interface

An action is defined by the interface it implements, in this case javax.swing.Action. Action extends the ActionListener interface from AWT; this forces concrete classes that implement Action to provide an actionPerformed() method. The programmer uses the actionPerformed() method to implement whatever behavior is desired. For example, if you are creating a Save action, you should put the code that saves the data inside of your actionPerformed() method.

When the action is added to an accepting container such as JMenu, JPopupMenu, or JToolBar, the container automatically registers the action as an ActionListener of the GUI component it creates. Consequently, if the GUI component is selected by the user, it simply invokes the actionPerformed() method of the action to do its job.

3.2 Graphical Interface Events

Whenever you interact with your application's user interface, the application receives an event from the windowing system to let it know that something happened. Some events come from the mouse, such as mouse clicks, mouse movements, and mouse drags. Other events come from the keyboard, such as key presses and key releases. Every component generates events. Different components generate different events as dictated by their purpose (and their L&F). For example, pressing a JButton generates an ActionEvent (which is really just a converted mouse event). The ActionEvent class bundles up interesting stuff like which button the event came from, when the button was pressed, whether any modifier keys (such as Shift or Ctrl) were pressed at the time of the event, and so on.

While the event-dispatching and -handling mechanism is grounded in the world of AWT (and beyond the scope of this book), we do want you to know what events the various Swing components generate—and when. The what of the events is discussed in conjunction with each of the components. As we introduce components like JTextField, JButton, and JTable, we show the events that they fire and the methods you use to attach listeners and catch the events.

The when of the events is a bit more difficult to describe. Rather than attempt to list every possible scenario for every component, we've built a small utility: EEL, the Every Event Listener. The EEL class implements every listener interface from the java.awt.event and javax.swing.event packages. It has a variety of logging mechanisms to show you the events coming from your components. You attach an EEL instance to a component (or to multiple components) using the component's add . . . Listener() method(s). You can choose to have the events sent to a file, to your console, or to an onscreen text area.

This discussion really is beyond the scope of the book. So we're posting this utility and its documentation on the web site for this book (<http://www.oreilly.com/catalog/jswing2>). Feel free to download it and use it to play with individual components or to debug an entire application. That's one of the beauties of delegation event handling: you can attach EEL to an existing component without breaking its normal interactions with the application.

3.3 Graphics Environments

SDK 1.4 recognizes a great deal of information about its environment. You can retrieve that information for your own code through the `GraphicsEnvironment`, `GraphicsDevice`, and `GraphicsConfiguration` classes from the `java.awt` package. While they aren't part of Swing proper, these classes are definitely useful for Swing applications, especially those that take full advantage of their environment.

To sum up these classes, a system keeps a local `GraphicsEnvironment` object that describes the devices on the system with an array of `GraphicsDevice` objects. Each `GraphicsDevice` contains (or at least may contain) multiple configurations of device capabilities (such as pixel formats or which visual screen you're on) bundled up in an array of `GraphicsConfiguration` objects.



The `GraphicsConfiguration` class should not be confused with the `DisplayMode` class (although it's easy to do so). The display mode is something with which most savvy computer users will be familiar. On a system that supports multisync monitors, the `DisplayMode` class encapsulates the width, height, color-depth, and refresh rate information for a given mode. The `GraphicsConfiguration` class stores things like square versus rectangular pixels. `GraphicsConfiguration` could even be used for devices such as printers. The configuration information is highly dependent on the native platform and thus varies widely from system to system. In any given system, both configurations and modes can be found through the available `GraphicsDevice` objects.

If you're curious about the various graphics configurations on your system, try out this little program, `GuiScreens.java`. It prints information on all devices and configurations. For each configuration, it also pops up a `JFrame` using that configuration.

```
// GuiScreens.java
```

3.4 Sending Change Events in Swing

Swing uses two different change event classes. The first is the standard `java.beans.PropertyChangeEvent` class. This class passes a reference to the object, sending the change notification as well as the property name, its old value, and its new value. The second, `javax.swing.event.ChangeEvent`, is a lighter version that passes only a reference to the sending object—in other words, the name of the property that changed, as well as the old and new values, are omitted.



Since the `ChangeEvent` class is not part of the JavaBeans specifications, properties that use this event are not "bound" according to the JavaBeans standard. In order to prevent confusion, properties that use a `ChangeEvent` to notify listeners of property changes have not been marked as bound in our property tables.

Because the `ChangeEvent` includes only a reference to the event originator, which never changes, you can always define a single `ChangeEvent` and reuse it over and over when firing events from your component.

3.4.1 The ChangeEvent Class

The `ChangeEvent` is a stripped-down version of the `java.beans.PropertyChangeEvent` class. This class has no methods or properties, only a constructor. This simplicity makes it a popular class for developers wanting to fire off their own events. Recipients get a reference to the source of the event but then must query the source directly to find out what just happened. It's great for quick notifications or instances in which the state of the source component is so complex it's hard to predict which pieces of information the recipient will need, but it shouldn't be used simply to save the component author a little time at the expense of runtime inefficiency if the recipient always needs to look up information that could have been part of a `PropertyChangeEvent`.

3.4.1.1 Constructor

`public ChangeEvent(Object source)`

The constructor for the `ChangeEvent` class. It takes only a single object, which represents the entity sending the event.

3.4.2 The ChangeListener Interface

Objects that intend to receive change events must implement the `com.sun.java.swing.event.ChangeListener` interface. They can then register to receive `ChangeEvent` objects from a publisher class. The `ChangeListener` interface consists of only one method.

3.4.2.1 Method

`public abstract void stateChanged(ChangeEvent e)`

Implemented in a listener object to receive `ChangeEvent` notifications.

3.5 The JComponent Class

JComponent is an abstract class that almost all Swing components extend; it provides much of the underlying functionality common throughout the Swing component library. Just as the java.awt.Component class serves as the guiding framework for most of the AWT components, the javax.swing.JComponent class serves an identical role for the Swing components. We should note that the JComponent class extends java.awt.Container (which in turn extends java.awt.Component), so it is accurate to say that Swing components carry with them a great deal of AWT functionality as well.

Because JComponent extends Container, many Swing components can serve as containers for other AWT and Swing components. These components may be added using the traditional add() method of Container. In addition, they can be positioned with any Java layout manager while inside the container. The terminology remains the same as well: components that are added to a container are said to be its *children*; the container is the *parent* of those components. Following the analogy, any component that is higher in the tree is said to be its *ancestor*, while any component that is lower is said to be its *descendant*.

Recall that Swing components are considered "lightweight." In other words, they do not rely on corresponding peer objects within the operating system to render themselves. As we mentioned in [Chapter 1](#), lightweight components draw themselves using the standard features of the abstract Graphics object, which not only decreases the amount of memory each component uses but allows components to have transparent portions and take on nonrectangular shapes. And, of course, lightweight components are free of a dedicated L&F.

It's not out of the question to say that a potential benefit of using lightweight components is a decrease in testing time. This is because the functionality necessary to implement lightweight components in the Java virtual machine is significantly less than that of heavyweight components. Heavyweight components must be individually mapped to their own native peers. On the other hand, one needs to implement only a single lightweight peer on each operating system for all the Swing components to work correctly. Hence, there is a far greater chance that lightweight components will execute as expected on any operating system and not require rounds of testing for each platform.



Because all Swing components extend Container, you should be careful that you don't add() to Swing components that aren't truly containers. The results range from amusing to destructive.

In JDK 1.2, JComponent reuses some of the functionality of the java.awt.Graphics2D class. This consists primarily of responsibilities for component painting and debugging.

3.5.1 Inherited Properties

Swing components carry with them several properties that can be accessed through JComponent but otherwise originate with AWT. Before we go any further, we should review those properties of java.awt.Container and java.awt.Component that can be used to configure all Swing components. This discussion is relatively brief; if you need a more thorough explanation of these AWT classes, see Java AWT Reference by John Zukowski (O'Reilly), which can be downloaded from this book's web site, <http://www.oreilly.com/catalog/jswing2/>. [Table 3-5](#) lists the properties that JComponent inherits from its AWT superclasses.

Table 3-5. Properties inherited from the AWT Component and Container classes

Property	Data type	get	is	set	Default value (if applicable)

3.6 Responding to Keyboard Input

Swing provides a flexible framework for keyboard-based control, which can be used by any component. The rest of the chapter explains this mechanism.

3.6.1 The InputMap Class

InputMap maps keystrokes to logical action names. When the user types a key combination, it's looked up in the input map of the focused component (and perhaps other components in the active window, as described earlier). If a match is found, the resulting object is used as a key in the corresponding component's ActionMap to look up the concrete Action class to be invoked. The platform-specific L&F implementations provide InputMaps consistent with the key-binding conventions for their platforms.

When looking for values in an InputMap, a `java.awt.KeyStroke` is always used as the key. `KeyStroke` is a simple, immutable class that represents a particular keyboard action (including any modifier keys). `KeyStrokes` are intended to be unique (that is, if two `KeyStroke` variables represent the same action, they should reference the same `KeyStroke` instance). To ensure uniqueness, you can't create `KeyStrokes` directly; you must obtain them through the static `getKeyStroke()` factory methods in the `KeyStroke` class.

Although the result of looking up a `KeyStroke` in an InputMap is an arbitrary object, and any object can be used as a key for looking up an action in an ActionMap, in practice the values are `Strings`. By convention, their content is a descriptive name for the action to be performed (such as `copy`, `print`, `save`, or the like). This allows InputMaps to be largely self-documenting (it's easy to print their contents as a "cheat sheet" showing the keys that invoke particular commands) and also improves the readability of code that requests actions programmatically. The most common way this string is obtained is by calling `getName()` on the Action to be added to the map.

InputMaps can be chained together so that common functionality can be shared in a basic InputMap; specialized components can add custom keystrokes to their own InputMap and delegate the common cases to the shared map via the `parent` property.

3.6.1.1 Property

The single property defined by `InputMap` is shown in [Table 3-10](#). The `parent` property establishes a fallback `InputMap` that is consulted if a key mapping is not found in the current map, much as the inheritance chain is followed when looking up the members of Java classes. If you create a cycle in the parent chain (for example, by setting two `InputMaps` to be parents of each other), many of the method calls crash with a `StackOverflowError`.

Table 3-10. `InputMap` property

Property	Data type	get	is	set	Default value
<code>parent</code>	<code>InputMap</code>				<code>null</code>

3.6.1.2 Constructor

`public InputMap()`

The default constructor is the only constructor available. It creates an empty `InputMap` with no parent.

3.6.1.3 Methods

`public KeyStroke getKeyStroke()`

Chapter 4. Labels and Icons

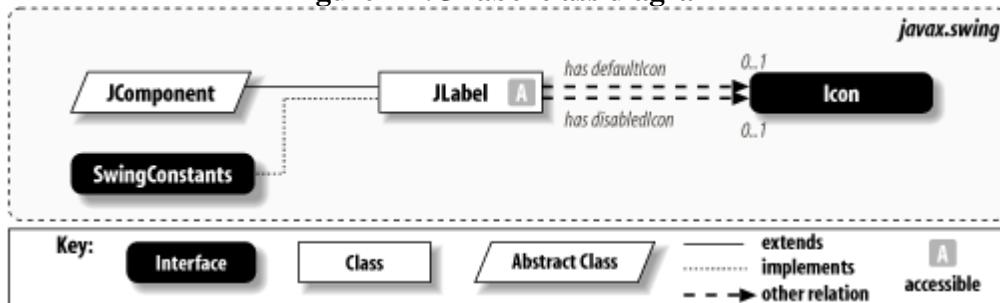
We'll begin our look at the Swing components with the `JLabel` class. In addition, we'll look at Swing's `Icon` interface and an implementation of this interface called `ImageIcon`. With just these few constructs, you'll begin to see how Swing aids in the sophisticated UI development in Java.

4.1 Labels

Swing allows you to create labels that can contain text, images, or both. We'll begin this chapter with a look at the `JLabel` class.

The `JLabel` class allows you to add basic, noninteractive labels to a user interface. Because of its inherent simplicity, there is no model class for `JLabel`. [Figure 4-1](#) shows a class diagram for `JLabel`. We'll get into the two relationships to `Icon` a little later.

Figure 4-1. `JLabel` class diagram



`JLabel` objects may consist of both text and graphics (icons), but for simple text-only labels, the interface with `JLabel` is very similar to that of `java.awt.Label`. The code to create and display a very simple text label looks like this:

```
// SimpleJLabelExample.java
```

4.2 Working with Images

JLabels make it very simple to add graphics to your user interface. Images used in JLabels (and also in other Swing components, such as buttons) are of type javax.swing.Icon, an interface described in detail in the next section.

These two lines of code show how simple it is to create a label containing an image:

```
ImageIcon icon = new ImageIcon("images/smile.gif");
```

4.3 Support for HTML

The use of HTML is supported by most Swing components. For example, it is possible to use HTML markup to create multiline and multifont labels:

```
JLabel label = new JLabel("<html>line 1<p><font color=blue size=+2>"
```

4.4 Icons

Swing introduces the concept of an icon for use in a variety of components. The Icon interface and ImageIcon class make dealing with simple images extremely easy.

The Icon interface is very simple, specifying just three methods used to determine the size of the Icon and display it. Implementations of this interface are free to store and display the image in any way, providing a great deal of flexibility. In other words, icons don't have to be bitmaps or GIF images, but are free to render themselves any way they choose. As we'll see later, an icon can simply draw on the component if that's more efficient. The examples at the end of this section show a couple of different ways the interface might be implemented.

4.4.1 Properties

The Icon interface defines the properties listed in [Table 4-3](#). The iconHeight and iconWidth properties specify the size of the Icon in pixels.

Table 4-3. Icon properties

Property	Data type	get	is	set	Default value
iconHeight	int				
iconWidth	int				

4.4.2 Method

`public void paintIcon(Component c, Graphics g, int x, int y)`

Paint the Icon at the specified location on the given Graphics. For efficiency reasons, the Graphics object will (probably) not be clipped, so be sure not to draw "outside the lines." You must make sure to keep your horizontal position between x and x + getIconWidth() - 1, and your vertical position between y and y + getIconHeight() - 1 while painting. The Component is provided to allow its properties (such as foreground or background color) to be used when painting or so it can be used as an image observer (see [Section 4.7](#) later in this chapter).

4.5 Implementing Your Own Icons

Here's a class that implements the Icon interface and uses ovals as simple icons:

```
// OvalIcon.java
```

4.6 Dynamic Icons

Icons are under no obligation to paint themselves the same way every time they are displayed. It's perfectly reasonable (and often quite useful) to have an icon that uses some sort of state information to determine how to display itself. In the next example, we create two sliders (JSlider is explained in detail in [Chapter 6](#)) that can be used to change the width and height of a dynamic icon:

// DynamicIconExample.java

I1@ve RuBoard

4.7 The ImageIcon Class

Swing provides a concrete implementation of the Icon interface that is considerably more useful than our OvalIcon class. ImageIcon uses a java.awt.Image object to store and display any graphic and provides synchronous image loading (i.e., the Image is loaded completely before returning), making ImageIcons very powerful and easy to use. You can even use an ImageIcon to display an animated GIF^{89a}, making the ubiquitous "animation applet" as simple as this:

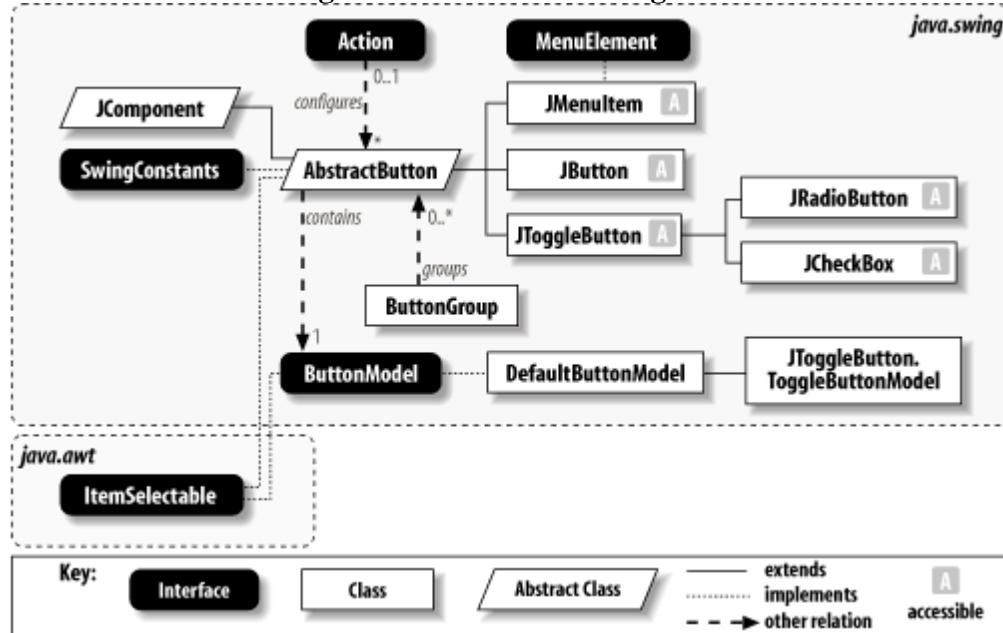
```
// AnimationApplet.java  
I1@ve RuBoard
```

Chapter 5. Buttons

Buttons are simple UI components used to generate events when the user presses them. Swing buttons can display icons, text, or both. In this section, we'll introduce the `Action` interface and `DefaultButtonModel` class (which define the state of the button). Next, we'll look at the `AbstractButton` class (which defines much of the functionality for all button types). Finally, we'll look at four concrete subclasses of `AbstractButton` and see how they can be grouped together using a `ButtonGroup`.

[Figure 5-1](#) shows the class hierarchy, with significant relationships between the button-related Swing classes. As we discussed in the introductory chapters, each button (`AbstractButton`) keeps a reference to a `ButtonModel`, which represents its state.

Figure 5-1. Button class diagram



The `JMenuItem` class shown here (and its subclasses, not shown) is not covered in this chapter; see [Chapter 14](#) for details.

5.1 The ButtonModel Interface

The state of any Swing button is maintained by a ButtonModel object. This interface defines methods for reading and writing a model's properties and for adding and removing various types of event listeners.

5.1.1 Properties

The properties for the ButtonModel interface are listed in [Table 5-1](#). The actionCommand property specifies the name of the command to be sent as part of the ActionEvent that is fired when the button is pressed. This can be used by event handlers that are listening to multiple buttons to determine which button is pressed.

Table 5-1. ButtonModel properties

Property	Data type	get	is	set	Default value
actionCommand	String				
armed	boolean				
enabled	boolean				
group	ButtonGroup				
mnemonic	int				
pressed	boolean				
rollover	boolean				
selected	boolean				
See also java.awt.ItemSelectable.					

If no actionCommand is specified, an ActionEvent takes the button's text for its command string, so it is usually not necessary to specify an explicit actionCommand. You may find it useful to do so for buttons that have icons but no text or for multiple buttons with the same text. actionCommand properties can also be handy for internationalization. For example, if you need to change a button's text from "Hot" to "Caliente", you won't have to change any event-handling code if you set the actionCommand to "Hot".

5.2 The DefaultButtonModel Class

Swing provides an implementation of the ButtonModel interface called DefaultButtonModel. This class is used directly by the AbstractButton class and indirectly by the other button classes. JToggleButton defines an inner class extending DefaultButtonModel that it and its descendants use to manage their state data.

5.2.1 Properties

The DefaultButtonModel class gets most of its properties from ButtonModel. The default values set by this class are shown in [Table 5-3](#).

Table 5-3. DefaultButtonModel properties

Property	Data type	get	is	set	Default value
actionCommando	String				null
armedo	boolean				false
enabledo	boolean				true
groupo, *	ButtonGroup				null
mnemonico	int				KeyEvent.VK_UNDEFINED
pressedo	boolean				false
rollovero	boolean				false
selectedo	boolean				false
selectedObjects	Object[]				null
ooverridden, *getter was introduced in SDK 1.3					

The only property here that does not come from the ButtonModel interface is the selectedObjects property.

5.3 The AbstractButton Class

AbstractButton is an abstract base class for all button components (JButton, JToggleButton, JCheckBox, JRadioButton, and JMenuItem and its subclasses). Since it provides functionality common to all types of buttons, we'll cover it here before getting to the concrete button classes.

AbstractButton provides much of the functionality associated with the interaction between the various concrete button classes and their ButtonModel objects. As we mentioned earlier, buttons in Swing can be made up of an image (Icon), text, or both. The relative positions of the text and icon are specified exactly as they are with the JLabel class.

Image buttons may specify as many as seven different images, allowing the button to be displayed differently depending on its current state. The seven icons are described in [Table 5-5](#), with the other properties defined by AbstractButton.

5.3.1 Properties

The AbstractButton class defines the properties shown in [Table 5-5](#).

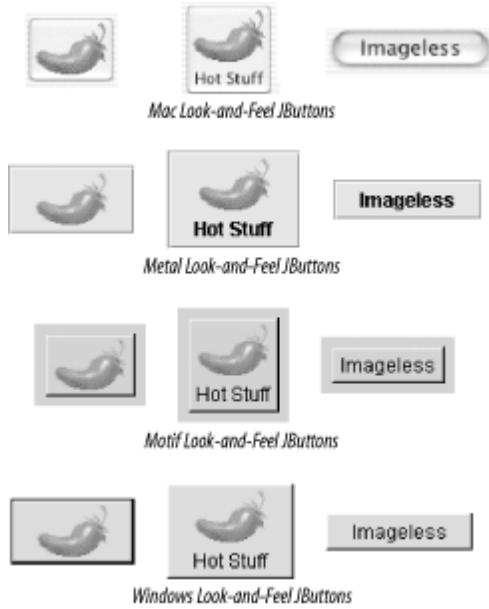
Table 5-5. AbstractButton properties

Property	Data type	get	is	set	Default value
action1.3	Action				null
actionCommand	String				null
borderPaintedb	boolean				true
contentAreaFilledb	boolean				true
disabledIconb	Icon				null
disabledSelectedIconb	Icon				null
displayedMnemonicIndex1.4	int				-1
enabledo	boolean				true
focusPaintedb	boolean				true

5.4 The JButton Class

JButton is the simplest of the button types, adding very little to what is provided by the AbstractButton class. JButtons are buttons that are not toggled on and off but instead act as push buttons, which invoke some action when clicked. [Figure 5-2](#) shows what these buttons look like in four of the Swing L&Fs.

Figure 5-2. JButtons in four L&Fs



5.4.1 Properties

The JButton class inherits most of its properties and default values from its superclasses. The exceptions to this are shown in [Table 5-9](#). The model property is set to a new instance of DefaultButtonModel when a JButton is created.

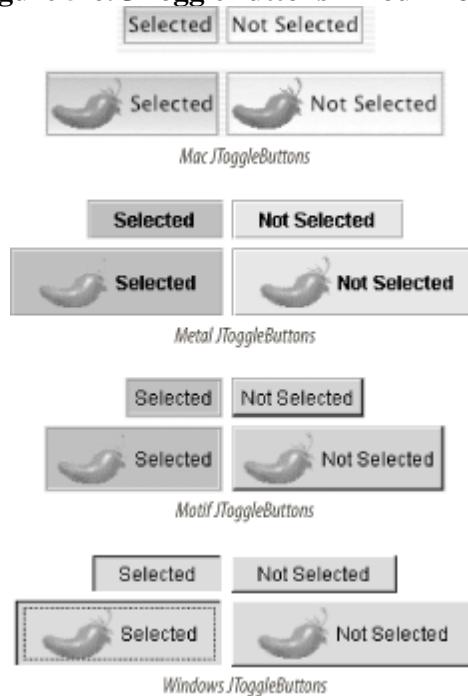
Table 5-9. JButton properties

Property	Data type	get	is	set	Default value
accessibleContext	AccessibleContext				JButton.AccessibleJButton()
defaultButton	boolean				false
defaultCapable	boolean				true
modelo	ButtonModel				DefaultButtonModel()
UIClassIDo	String				"ButtonUI"

5.5 The JToggleButton Class

JToggleButton is an extension of AbstractButton and is used to represent buttons that can be toggled on and off (as opposed to buttons like JButton which, when pushed, "pop back up"). It should be noted that while the subclasses of JToggleButton (JCheckBox and JRadioButton) are the kinds of JTogglesButtons most commonly used, JToggleButton is not an abstract class. When used directly, it typically (though this is ultimately up to the L&F) has the appearance of a JButton that does not pop back up when pressed (see [Figure 5-6](#)).

Figure 5-6. JToggleButtons in four L&Fs



5.5.1 Properties

The JToggleButton class inherits all of its properties and most of its default values from its superclass. The exceptions are shown in [Table 5-10](#). The model property is set to a new instance of ToggleButtonModel when a JToggleButton is created. ToggleButtonModel (described in the next section) is a public inner class that extends DefaultButtonModel.

Table 5-10. JToggleButton properties

Property	Data type	get	is	set	Default value
accessibleContext	AccessibleContext				JToggleButton.AccessibleJToggleButton()
modelo	ButtonModel				ToggleButtonModel()
UIClassIDo	String				"ToggleButtonUI"

5.6 The JToggleButton.ToggleButtonModel Class

As we mentioned earlier, JToggleButton does not use the DefaultButtonModel class as its model. ToggleButtonModel, a public static inner class that extends DefaultButtonModel, is used instead.

5.6.1 Properties

`ToggleButtonModel` modifies the methods for working with the properties listed in [Table 5-11](#). New implementations of `isSelected()` and `setSelected()` use the button's `ButtonGroup` (if defined) to keep track of which button is selected, ensuring that even if multiple selected buttons are added to a group, only the first one is considered selected (since the group keeps track of the "officially" selected button). In addition, the `setPressed()` method is redefined to call `setSelected()` when the button is released (if it is armed)

Table 5-11. `JToggleButton.ToggleButtonModel` properties

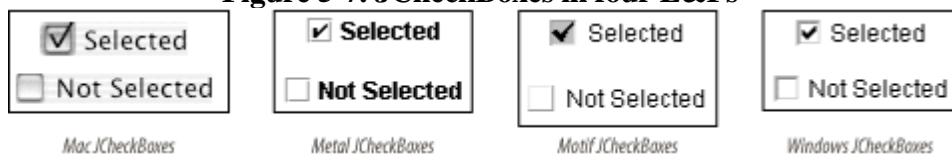
Property	Data type	get	is	set	Default value
<code>pressedo</code>	<code>boolean</code>				false
<code>selectedo</code>	<code>boolean</code>				false
<code>overridden</code> See also properties from DefaultButtonMod el (Table 5-3).					

5.7 The JCheckBox Class

The JCheckBox^[5] class is shown in various L&Fs in [Figure 5-7](#). JCheckBox is a subclass of JToggleButton and is typically used to allow the user to turn a given feature on or off or to make multiple selections from a set of choices. A JCheckBox is usually rendered by showing a small box into which a "check" is placed when selected (as shown in [Figure 5-7](#)). If you specify an icon for the checkbox, this icon replaces the default box. Therefore, if you specify an icon, you should always also supply a selected icon—otherwise, there is no way to tell if a checkbox is selected.

[5] Note that the java.awt.Checkbox class differs in capitalization from javax.swing.JCheckBox.

Figure 5-7. JCheckBoxes in four L&Fs



5.7.1 Properties

The JCheckBox class inherits most of its properties from its superclasses. The exceptions are shown in [Table 5-12](#). By default, no border is painted on JCheckboxes, and their horizontalAlignment is to the leading edge (which means to the left in the default locale, in which text reads left to right).^[6] Setting the borderPaintedFlat property to true is a hint to the L&F that the checkbox should be drawn more plainly than usual. (This is used primarily by cell renderers for tables and trees.)

[6] This locale sensitivity was introduced in SDK 1.4; previously, checkboxes were always aligned to the left.

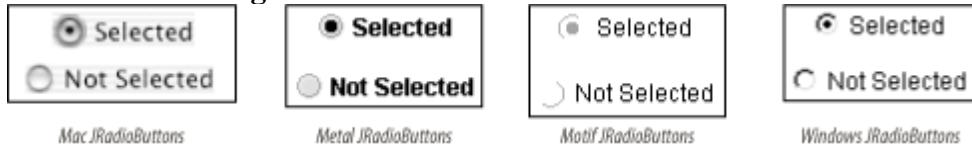
Table 5-12. JCheckBox properties

Property	Data type	get	is	set	Default value
accessibleContext o	AccessibleContext				AccessibleJCheck Box
borderPaintedo	boolean				false
borderPaintedFlat 1.3, b	boolean				false
horizontalAlignme nto	int				LEADING1.4
UIClassIDo	String				"CheckBoxUI"

5.8 The JRadioButton Class

JRadioButton is a subclass of JToggleButton, typically used with other JRadioButtons, that allows users to make a single selection from a set of options ([Figure 5-8](#)). Because radio buttons form a set of choices, JRadioButtons are usually used in groups, managed by a ButtonGroup (described in the next section). If you specify an icon for the radio button, you should also specify a selected icon so it will be visually apparent if a button is selected.

Figure 5-8. JRadioButtons in four L&Fs



5.8.1 Properties

The JRadioButton class inherits all its properties and most of its default values from its superclass. The only exceptions are shown in [Table 5-14](#). By default, no border is painted on JRadioButtons, and their horizontalAlignment is set to the leading edge (to the left in the default locale, in which text reads left to right).[\[7\]](#)

[7] This locale sensitivity was introduced in SDK 1.4; previously, radio buttons were always aligned to the left.

Table 5-14. JRadioButton properties

Property	Data type	get	is	set	Default value
accessibleContext o	AccessibleContext				JRadioButton.AccessibleJRadioButton()
borderPaintedo	boolean				false
horizontalAlignme nto	int				LEADING1.4
UIClassIDo	String				"RadioButtonUI"
1.4since 1.4, overridden					
See also properties from JToggleButton (
Table 5-10).					

5.9 The ButtonGroup Class

The ButtonGroup class allows buttons to be logically grouped, guaranteeing that no more than one button in the group is selected at any given time. In fact, once one of the buttons is selected, the ButtonGroup ensures that exactly one button remains selected at all times. Note that this allows for an initial state (in which no button is selected) that can never be reached again once a selection is made, except programmatically.

As mentioned earlier, ButtonGroups typically hold JRadioButtons (or JRadioButtonMenuItems, discussed in [Chapter 14](#)), but this is purely a convention and is not enforced by ButtonGroup. ButtonGroup's add() method takes objects of type AbstractButton, so any button type may be added—even a mix of types. Of course, adding a JButton to a ButtonGroup would not be very useful since JButtons do not have selected and deselected states. In fact, JButtons added to ButtonGroups have no effect on the state of the other buttons if they are pressed.

ButtonGroup objects do not have any visual appearance; they simply provide a logical grouping of a set of buttons. You must add buttons in a ButtonGroup to a Container and lay them out as though no ButtonGroup were being used.

It's worth noting that some methods in the ButtonGroup class deal with AbstractButton objects and some deal with ButtonModel objects. The add(), remove(), and getElements() methods all use AbstractButton, while the getSelection(), isSelected(), and setSelected() methods use ButtonModel objects.

5.9.1 Properties

ButtonGroup defines the properties listed in [Table 5-15](#). The buttonCount property is the number of buttons in the group. The elements property is an Enumeration of the AbstractButton objects contained by the group. The selection property contains the ButtonModel of the currently selected button.

Table 5-15. ButtonGroup properties

Property	Data type	get	is	set	Default value
buttonCount	int				0
elements	Enumeration				Empty
selection	ButtonModel				null

5.9.2 Voting with a Button Group

The following example demonstrates the use of a ButtonGroup to ensure that only a single selection is made from a list of choices. Listeners are added to the buttons to show which events are fired each time a new button is selected.

```
// SimpleButtonGroupExample.java
```

Chapter 6. Bounded-Range Components

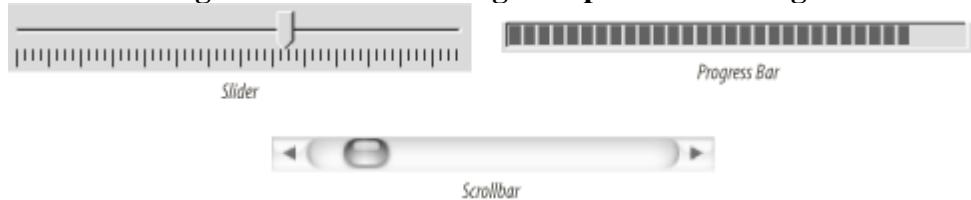
This chapter groups several Swing components together by the model that drives them: the *bounded-range model*. Bounded-range components in Swing include JSlider, JProgressBar, and JScrollPane. In addition, we discuss two classes that use progress bars: ProgressMonitor and ProgressMonitorInputStream. These classes display status dialogs using a JOptionPane that you can assign to a variety of tasks.

6.1 The Bounded-Range Model

Components that use the bounded-range model typically consist of an integer value that is constrained within two integer boundaries. The lower boundary, the *minimum*, should always be less than or equal to the model's current *value*. In addition, the model's value should always be less than the upper boundary, the *maximum*. The model's value can cover more than one unit; this size is referred to as its *extent*. With bounded range, the user is allowed to adjust the value of the model according to the rules of the component. If the value violates any of the rules, the model can adjust the values accordingly.

The javax.swing.BoundedRangeModel interface outlines the data model for such an object. Objects implementing the BoundedRangeModel interface must contain an adjustable integer value, an extent, a minimum, and a maximum. Swing contains three bounded-range components: JScrollBar, JSlider, and JProgressBar. These components are shown in [Figure 6-1](#).

Figure 6-1. Bounded-range components in Swing



6.1.1 Properties

[Table 6-1](#) shows the properties of the BoundedRangeModel interface.

Table 6-1. BoundedRangeModel properties

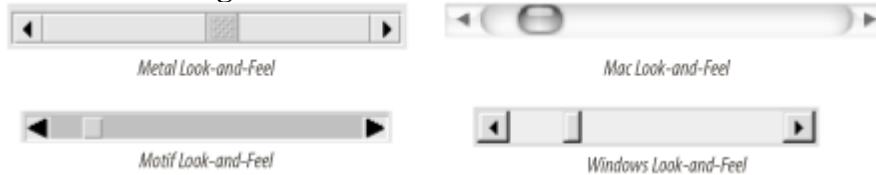
Property	Data type	get	is	set	Default value
extent	int				
maximum	int				
minimum	int				
value	int				
valueIsAdjusting	boolean				

The *minimum*, *maximum*, and *value* properties form the actual bounded range. The *extent* property can give the *value* its own subrange. Extents can be used in situations where the model's value exceeds a single unit; they can also be changed dynamically. For example, the sliding "thumbs" of many scrollbars resize themselves based on the percentage of total information displayed in the window. If you wish to emulate this behavior with Swing, you could declare a bounded-range scrollbar and set the *extent* property to grow or shrink as necessary.

6.2 The JScrollBar Class

JScrollBar is the Swing implementation of a scrollbar. The JScrollBar class is shown in various L&Fs in [Figure 6-3](#).

Figure 6-3. Scrollbars in several L&Fs



To program with a scrollbar, it is important to understand its anatomy. Scrollbars are composed of a rectangular tab, called a *slider* or *thumb*, located between two arrow buttons. The arrow buttons on either end increment or decrement the slider's position by an adjustable number of units, generally one. In addition, clicking in the area between the thumb and the end buttons (often called the *paging area*) moves the slider one *block*, or 10 units by default. The user can modify the value of the scrollbar in one of three ways: by dragging the thumb in either direction, by pushing on either of the arrow buttons, or by clicking in the paging area.

Scrollbars can have one of two orientations: horizontal or vertical. [Figure 6-4](#) provides an illustration of a horizontal scrollbar. JScrollBar uses the bounded-range model to represent the scrollbar's data. The assignment of each bounded-range property is also shown in [Figure 6-5](#). The minimum and maximum of the scrollbar fall on the interior edges of the arrow buttons. The scrollbar's value is defined as the left (or top) edge of the slider. Finally, the extent of the scrollbar defines the width of the thumb in relation to the total range. (The older Adjustable interface from the java.awt package referred to the extent as the "visible amount.") Note that horizontal scrollbars increment to the right and vertical scrollbars increment downward.

Figure 6-4. Anatomy of a horizontal scrollbar

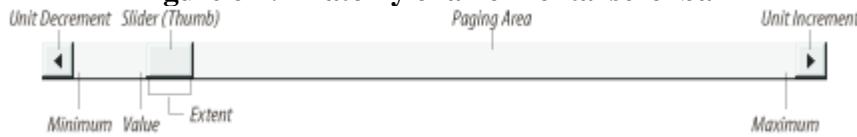
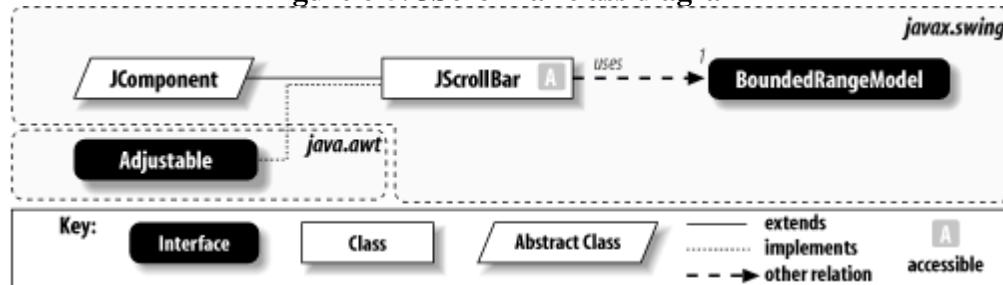


Figure 6-5. JScrollBar class diagram



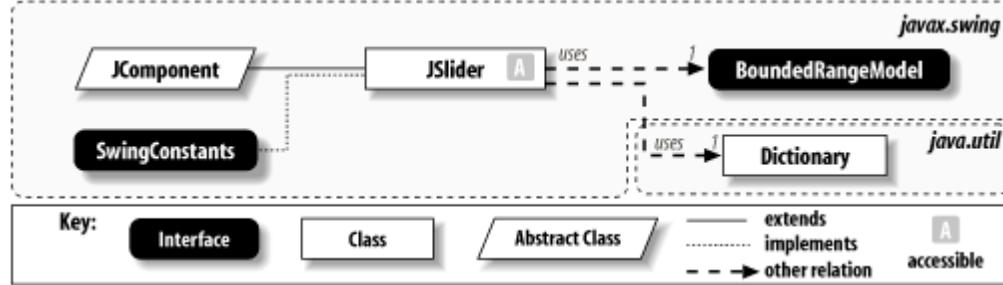
6.2.1 Properties

[Table 6-3](#) shows the properties of the JScrollBar component. Most of these properties come from the java.awt.Adjustable interface. The orientation property gives the direction of the scrollbar, either JScrollBar.HORIZONTAL or JScrollBar.VERTICAL. The unitIncrement property represents the integer amount by which the bounded-range value changes when the user clicks on either of the arrow buttons. The blockIncrement property represents the integer amount by which the scrollbar value changes when the user clicks in either of the paging areas. The enabled property indicates whether the scrollbar can generate or respond to events. The minimum, maximum, value, and valueIsAdjusting properties match the equivalent properties in the BoundedRangeModel of the scrollbar. The visibleAmount property matches the extent property in the model; it indicates the thickness of the thumb. The minimumSize and maximumSize properties allow the scrollbar to behave appropriately when it is resized.

6.3 The JSlider Class

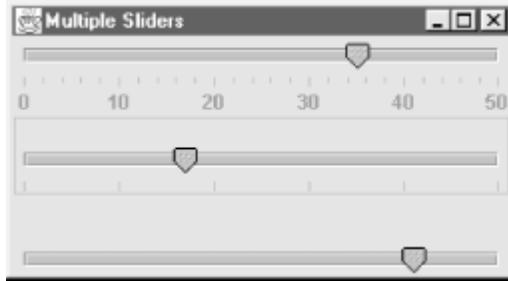
The JSlider class represents a graphical slider. Like scrollbars, sliders can have either a horizontal or vertical orientation. With sliders, however, you can enhance their appearance with tick marks and labels. The class hierarchy is illustrated in [Figure 6-8](#). In most instances, a slider is preferable to a standalone scrollbar. Sliders represent a selection of one value from a bounded range. Scrollbars represent a range of values within a bounded range and are best used in things like the JScrollPane.

Figure 6-8. JSlider class diagram



The JSlider class allows you to set the spacing of two types of tick marks: major and minor. Major tick marks are longer than minor tick marks and are generally used at wider intervals. [Figure 6-9](#) shows various sliders that can be composed in Swing.

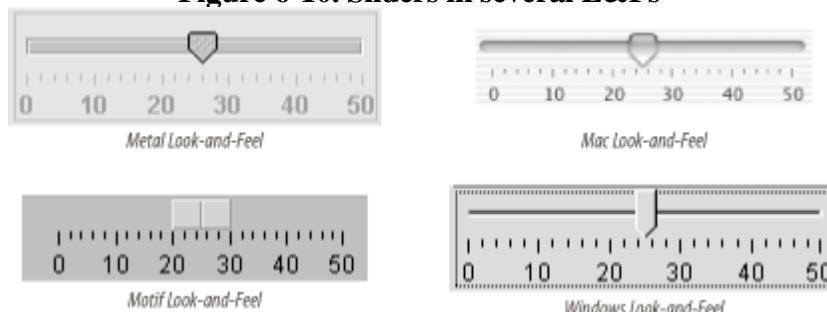
Figure 6-9. Various sliders in Swing



The `setPaintTicks()` method sets a boolean, which is used to activate or deactivate the slider's tick marks. In some L&Fs, the slider changes from a rectangular shape to a pointer when tick marks are activated. This is often done to give the user a more accurate representation of where the slider falls.

You can create a Dictionary of Component objects to annotate the slider. Each entry in the Dictionary consists of two fields: an Integer key, which supplies the index to draw the various components, followed by the component itself. If you do not wish to create your own label components, you can use the `createStandardLabels()` method to create a series of JLabel objects for you. In addition, if you set the `paintLabels` property to true and give a positive value to the `majorTickSpacing` property, a set of labels that matches the major tick marks is automatically created. [Figure 6-10](#) shows what a JSlider looks like in four L&Fs.

Figure 6-10. Sliders in several L&Fs

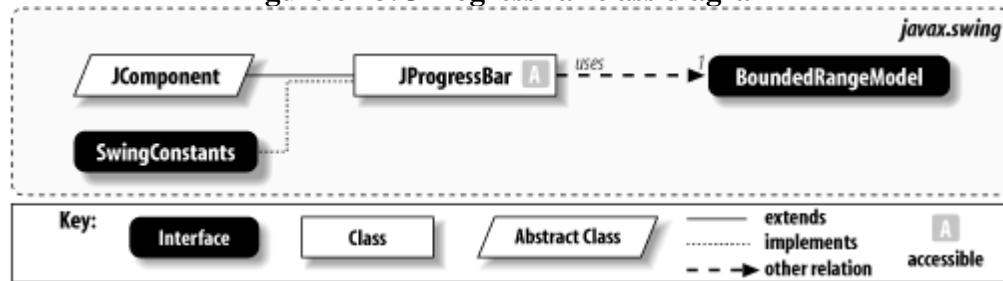


6.3.1 Properties

6.4 The JProgressBar Class

Swing makes it easy to create progress bars. Applications typically use progress bars to report the status of time-consuming jobs, such as software installation or large amounts of copying. The bars themselves are simply rectangles of an arbitrary length, a percentage of which is filled based on the model's value. Swing progress bars come in two flavors: horizontal and vertical. If the orientation is horizontal, the bar fills from left to right. If the bar is vertical, it fills from bottom to top. SDK 1.4 added the ability to show indeterminate progress (progress when you don't know the total). The class hierarchy is illustrated in [Figure 6-13](#).

Figure 6-13. JProgressBar class diagram



Different L&Fs can contain different filling styles. Metal, for example, uses a solid fill, while the Windows L&F uses an LCD style, which means that the bar indicates progress by filling itself with dark, adjacent rectangles instead of with a fluid line (at the opposite extreme, the Mac's is so fluid that it even contains moving ripples). The JProgressBar class also contains a boolean property that specifies whether the progress bar draws a dark border around itself. You can override this default border by setting the border property of the JComponent. [Figure 6-14](#) shows a Swing progress bar with the different L&Fs.

Figure 6-14. Progress bars in various L&Fs



6.4.1 Properties

The basic properties of the JProgressBar object are listed in [Table 6-5](#). The orientation property determines which way the progress bar lies; it must be either JProgressBar.HORIZONTAL or JProgressBar.VERTICAL. The minimum , maximum, and value properties mirror those in the bounded-range model. If you don't really know the maximum, you can set the indeterminate value to true. That setting causes the progress bar to show an animation indicating that you don't know when the operation completes. (Some L&Fs might not support this feature.) The boolean borderPainted indicates whether the component's border should appear around the progress bar. Borders are routinely combined with progress bars—they not only tell the user where its boundaries lie, but also help to set off the progress bar from other components. An important note about the JProgressBar class: there are no methods to access the extent variable of its bounded-range model. This property is irrelevant in the progress bar component.

Table 6-5. JProgressBar properties

Property	Data type	get	is	set	Default value
accessibleContext	AccessibleContext				JProgressBarAccessibleJProgressBar()

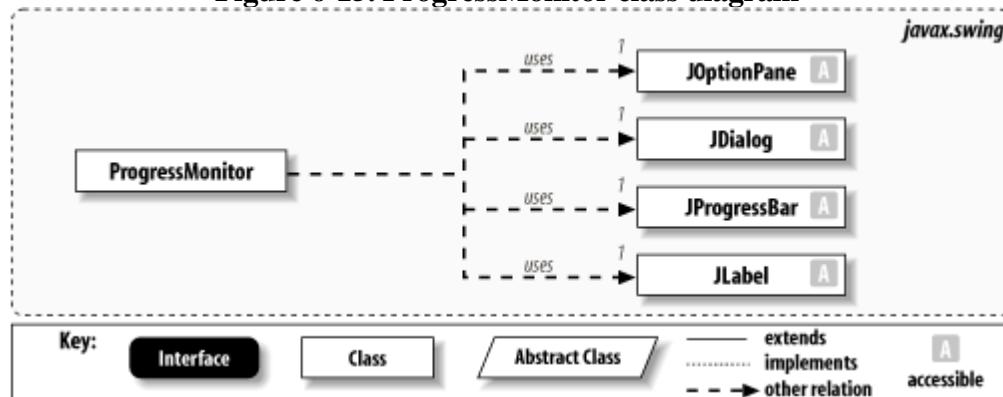
6.5 Monitoring Progress

By themselves, progress bars are pretty boring. Swing, however, combines progress bars with the dialog capabilities of JOptionPane to create the ProgressMonitor and ProgressMonitorInputStream classes. You can use ProgressMonitor to report on the current progress of a potentially long task. You can use ProgressMonitorInputStream to automatically monitor the amount of data that has been read in with an InputStream. With both, you can define various strings to be posted in the progress monitor dialogs to offer a better explanation of the task at hand.

6.5.1 The ProgressMonitor Class

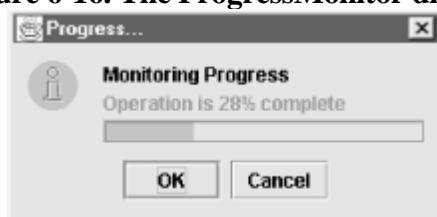
The ProgressMonitor class is a generic progress dialog box that can be used for practically anything. There are two string descriptions that can be set on a ProgressMonitor dialog box. The first is a static component that can never change; it appears on the top of the dialog and is set in the constructor. The second is a variable string-based property that can be reset at any time. It appears below the static string, slightly above the progress bar. [Figure 6-15](#) shows the structure for this class.

Figure 6-15. ProgressMonitor class diagram



Once instantiated, the ProgressMonitor dialog (shown in [Figure 6-16](#)) does not pop up immediately. The dialog waits a configurable amount of time before deciding whether the task at hand is long enough to warrant the dialog. If it is, the dialog is displayed. When the current value of the progress bar is greater than or equal to the maximum, as specified in the constructor, the progress monitor dialog closes. If you need to close the progress monitor early, you can call the `close()` method. The user can close this dialog as well by pressing OK or Cancel; you can test the `canceled` property to see if the user wanted to cancel the operation or simply did not care to watch the progress.

Figure 6-16. The ProgressMonitor dialog



The ProgressMonitor class does not fire any events indicating that it is complete or that the operation was canceled. You should test the `isCancelled()` method each time you call `setProgress()` to see if the user has canceled the dialog.

6.5.1.1 Properties

Chapter 7. Lists, Combo Boxes, and Spinners

This chapter deals with three similar components: lists, combo boxes, and spinners. All three present a catalog of choices to the user. A list allows the user to make single or multiple selections. A combo box permits only a single selection but can be combined with a text field that allows the user to type in a value as well. From a design standpoint, both lists and combo boxes share similar characteristics, and both can be extended in ways that many Swing components cannot. SDK 1.4 introduced spinners, which are compact components that allow you to click or "spin" through a set of choices one at a time.

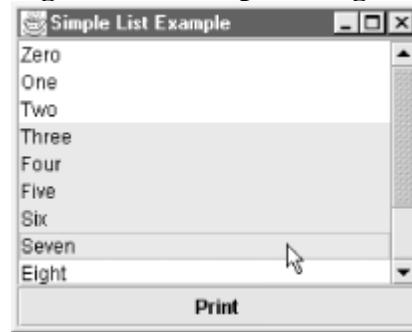
7.1 Lists

A *list* is a graphical component that presents the user with choices. Lists typically display several items at a time, allowing the user to make either a single selection or multiple selections. In the event that the inventory of the list exceeds the space available to the component, the list is often coupled with a scrollpane to allow navigation through the entire set of choices.

The Swing JList component allows elements to be any Java class capable of being rendered—which is to say anything at all because you can supply your own renderer. This offers a wide range of flexibility; list components can be as simple or as complex as the programmer's needs dictate.

Let's get our feet wet with a simple list. The following example uses the Swing list class, JList, to create a single-selection list composed only of strings. [Figure 7-1](#) shows the result.

Figure 7-1. A simple Swing list



// SimpleList.java

I1@ve RuBoard

7.2 Representing List Data

Swing uses one interface and two classes to maintain a model of the list elements. When programming with lists, you often find that you can reuse these classes without modification. Occasionally, you may find it necessary to extend or even rewrite these classes to provide special functionality. In either case, it's important to examine all three in detail. Let's start with the easiest: ListModel.

7.2.1 The ListModel Interface

ListModel is a simple interface for accessing the data of the list. It has four methods: one method to retrieve data in the list, one method to obtain the total size of the list, and two methods to register and unregister change listeners on the list data. Note that the ListModel interface itself contains a method only for retrieving the list elements — not for setting them. Methods that set list values are defined in classes that implement this interface.

7.2.1.1 Properties

The ListModel interface defines two properties, shown in [Table 7-1](#). elementAt is an indexed property that lets you retrieve individual objects from the list; size tells you the total number of elements.

Table 7-1. ListModel properties

Property	Data type	get	is	set	Default value
elementAt	Object				
size	int				
indexed					

7.2.1.2 Events

The ListModel interface also contains the standard addListDataListener() and removeListDataListener() event subscription methods. These methods accept listeners that notify when the contents of the list have changed. A ListDataEvent should be generated when elements in the list are added, removed, or modified. ListDataEvent and the ListDataListener interface are discussed later in this chapter.

public abstract void addListDataListener(ListDataListener l) public abstract void removeListDataListener(ListDataListener l)

Add or remove a specific listener for ListDataEvent notifications.

7.2.2 The AbstractListModel Class

The AbstractListModel class is a skeletal framework to simplify the life of programmers who want to implement the ListModel interface. It provides the required addListDataListener() and removeListDataListener() event registration methods. It also provides three protected methods that subclasses can use to fire ListDataEvent objects. These

7.3 Handling Selections

The `JList` class in Swing depends on a second model, this one to monitor the elements that have been selected by the user. As with the list data model, the programmer is given many places in which standard behavior can be altered or replaced when dealing with selections. Swing uses a simple interface for models that handle list selections (`ListSelectionModel`) and provides a default implementation (`DefaultListSelectionModel`).

7.3.1 The ListSelectionModel Interface

The `ListSelectionModel` interface outlines the methods necessary for managing list selections. Selections are represented by a series of ranges, where each range is defined by its endpoints. For example, if the elements One, Two, Three, Six, Seven, and Nine were selected in the opening example of the chapter, the list selection model would contain three entries that specified the ranges {1,3}, {6,7}, and {9,9}. All selection indices are zero-based, and the ranges are closed, meaning both endpoint indices are included within the selection. If only one element is present in a range, such as with Nine, both endpoints are identical.

7.3.1.1 Properties

[Table 7-5](#) shows the properties of the `ListSelectionModel` interface. The first four properties of the list selection model can be used to retrieve various indices that are currently selected in the list. The `anchorSelectionIndex` and `leadSelectionIndex` properties represent the anchor and lead indices of the most recent range of selections, as illustrated in [Figure 7-2](#). The `maxSelectionIndex` and `minSelectionIndex` properties return the largest and smallest selected index in the entire list, respectively.

Table 7-5. `ListSelectionModel` properties

Property	Data type	get	is	set	Default value
<code>anchorSelectionIndex</code>	int				
<code>leadSelectionIndex</code>	int				
<code>maxSelectionIndex</code>	int				
<code>minSelectionIndex</code>	int				
<code>selectionEmpty</code>	boolean				
<code>selectionMode</code>	int				

7.4 Displaying Cell Elements

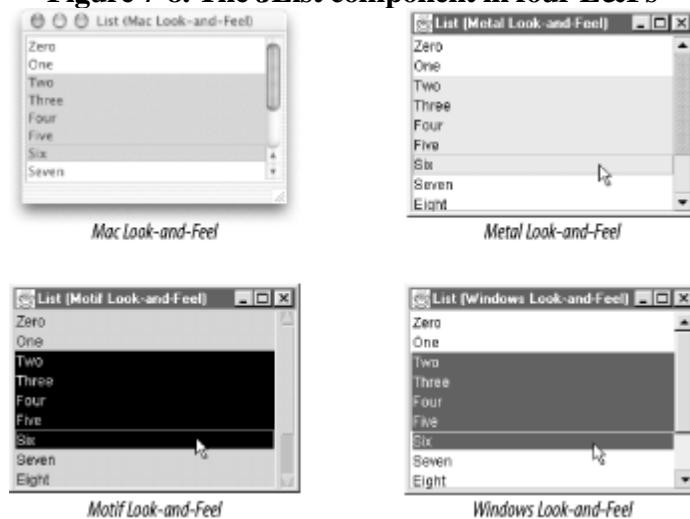
Swing gives the programmer the option to specify how each element in the list (called a *cell*) should be displayed on the screen. The list itself maintains a reference to a *cell renderer*. Cell renderers are common in Swing components, including lists and combo boxes. Essentially, a cell renderer is a component whose `paint()` method is called each time the component needs to draw or redraw an element. To create a cell renderer, you need only to register a class that implements the `ListCellRenderer` interface. This registration can be done with the `setCellRenderer()` method of `JList` or `JComboBox`:

```
JList list = new JList( );
```


7.5 The JList Class

The JList class is the generic Swing implementation of a list component. In the default selection mode, you can make multiple selections by clicking with the mouse while holding down the modifier key defined by the current L&F (generally Shift for a single, contiguous range and Ctrl or Command for noncontiguous selections). The JList class does not provide scrolling capabilities, but it can be set as the viewport of a JScrollPane to support scrolling. [Figure 7-8](#) shows the JList component in four different L&Fs.

Figure 7-8. The JList component in four L&Fs



7.5.1 Properties

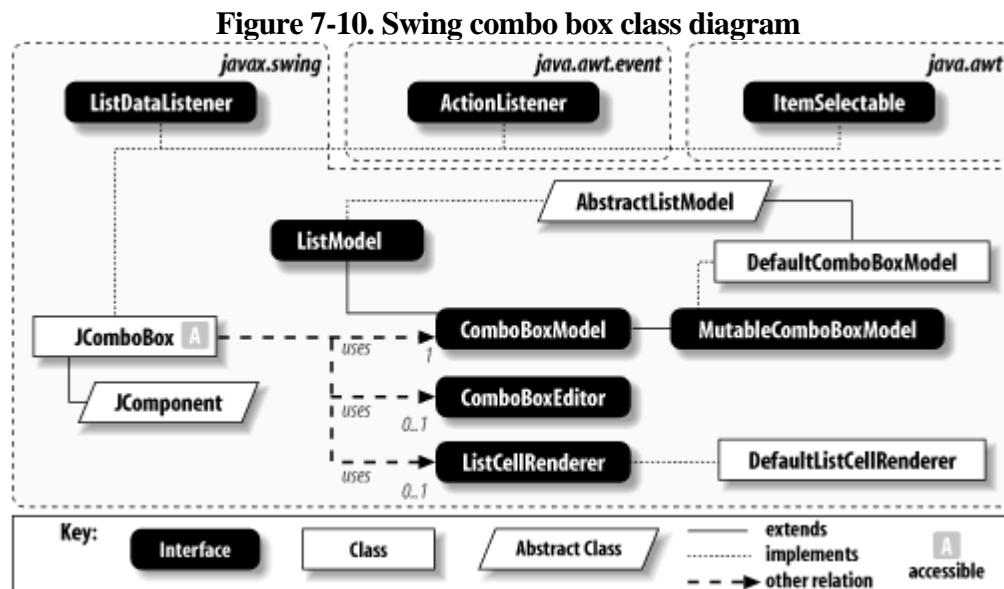
The JList class essentially combines the features of the data model, the selection model, and the cell renderer into a single Swing component. The properties of the JList class are shown in [Table 7-9](#).

Table 7-9. JList properties

Property	Data type	get	is	set	Default value
accessibleContext o	AccessibleContext				JList.AccessibleJList
anchorSelectionIn dex	int				
cellRendererb	ListCellRenderer				From L&F
dragEnabled1.4	boolean				false
firstVisibleIndex	int				
fixedCellHeighth	int				

7.6 Combo Boxes

A combo box component is actually a combination of a Swing list (embedded in a pop-up window) and a text field. Because combo boxes contain a list, many of the classes discussed in the first part of this chapter are used here as well. Unlike lists, a combo box allows the user only one selection at a time, which is usually copied into an editable component at the top, such as a text field. The user can also manually enter a selection (which does not need to be on the list). [Figure 7-10](#) shows a high-level class diagram for Swing's combo box classes.



Like lists, the combo box component uses a data model to track its list data; the model is called `ComboBoxModel`.

7.6.1 The `ComboBoxModel` Interface

The `ComboBoxModel` interface extends the `ListModel` interface and is used as the primary model for combo box data. It adds two methods to the interface, `setSelectedItem()` and `getSelectedItem()`, thus eliminating the need for a separate selection model. Since a `JComboBox` allows only one selected item at a time, the selection "model" is trivial and is collapsed into these two methods.

Because the data of the `ComboBoxModel` is stored in an internal list, the `ComboBoxModel` also reuses the `ListDataEvent` to report changes in the model state. However, with the addition of methods to monitor the current selection, the model is now obligated to report changes in the selection as well, which it does by firing a modification `ListDataEvent` with both endpoints as -1. Again, you should always query the event source to determine the resulting change in the elements.

You can create your own `ComboBoxModel` or use the default provided with the `JComboBox` class. The default model is an inner class of `JComboBox`. If you need to create your own, it is (as before) a good idea to extend the `AbstractListModel` class and go from there.

7.6.1.1 Property

[Table 7-11](#) shows the property defined by the `ComboBoxModel` interface. The `selectedItem` property lets you set or retrieve the currently selected object.

Table 7-11. `ComboBoxModel` property

Property	Data type	get	is	set	Default value
----------	-----------	-----	----	-----	---------------

7.7 The JComboBox Class

JComboBox combines a button or editable field and a drop-down list. It is very similar to the AWT Choice component and even implements the ItemSelectable interface for backward compatibility. By default, the JComboBox component provides a single text edit field adjacent to a small button with a downward arrow. When the button is pressed, a pop-up list of choices is displayed, one of which can be selected by the user. If a selection is made, the choice is copied into the edit field, and the pop up disappears. If there was a previous selection, it is erased. You can also remove the pop up by pressing Tab (or Esc, depending on the L&F) while the combo box has the focus. [Figure 7-11](#) shows combo boxes as they appear in four different L&Fs.

Figure 7-11. The JComboBox component in four L&Fs



The text field in the JComboBox component can be either editable or not editable. This state is controlled by the `editable` property. If the text field is editable, the user is allowed to type information into the text box (which may not correspond to anything in the list), as well as make selections from the list. If the component is not editable, the user can only make selections from the list.

Unless you specify a set of objects in the constructor, the combo box comes up empty. You can use the `addItem()` method to add objects to the combo box list. Conversely, the `removeItem()` and `removeItemAt()` methods remove a specified object from the list. You also have the ability to insert objects at specific locations in the combo box list with the `insertItemAt()` method. If you wish to retrieve the current number of objects in the list, use the `getItemCount()` method, and if you wish to retrieve an object at a specific index, use the `getItemAt()` method.

Note that the list component inside the JComboBox is not part of the component itself but rather part of its UI delegate. Hence, there is no property to access the list component directly. However, you should be able to get any information you need through the component properties or the `ComboBoxModel`.

As with regular pop-up menus, you have the ability to specify whether the pop up in the JComboBox component should be drawn as a lightweight or a heavyweight component. Lightweight components require less memory and computing resources. However, if you are using any heavyweight components, you should consider forcing the combo box to use a heavyweight pop up, or else the pop up could be obscured behind your heavyweight components. This can be done by setting the `lightWeightPopupEnabled` property to false. If the property is set to true, the combo box uses a lightweight pop up when appropriate.

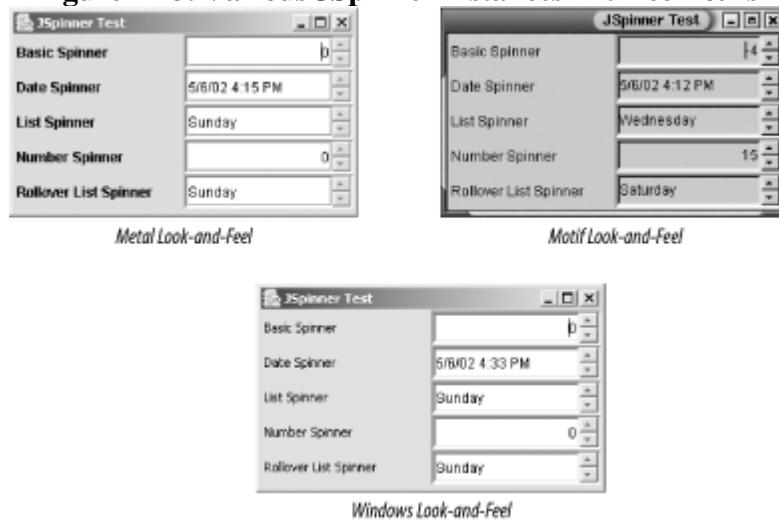
Combo boxes use the same `ListCellRenderer` as the `JList` component (discussed earlier in this chapter) to paint selected and nonselected items in its list.

7.7.1 The Key Selection Manager

7.8 Spinners

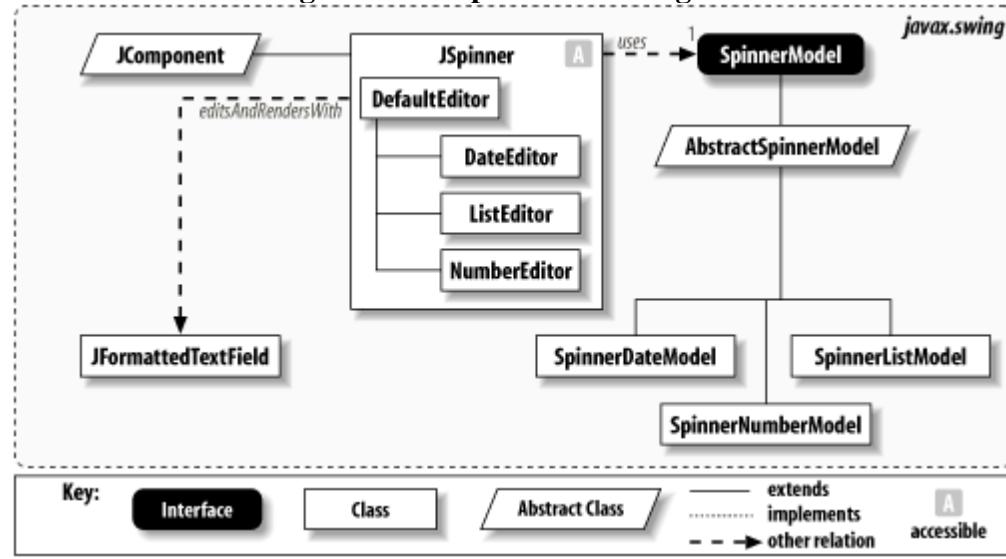
You might be wondering just what a spinner is. It's a new 1.4 component similar to the JComboBox, but it shows only one item. It includes up and down arrows to "scroll" through its set of values. A JFormattedTextField is used to edit and render those values. Spinners are quite flexible. They work nicely with a set of choices (such as the months of the year) as well as with unbounded ranges such as a set of integers. [Figure 7-13](#) shows several examples of spinners in different L&Fs. The Mac L&F is missing from this figure because the SDK 1.4 was not available on OS X at the time we went to press.

Figure 7-13. Various JSpinner instances in three L&Fs



The classes involved in spinners are shown in [Figure 7-14](#).

Figure 7-14. JSpinner class diagram



7.8.1 Properties

JSpinner has several properties related to the values it displays (see [Table 7-16](#)). Most of the properties are easy to understand from their names alone. The currently selected value is available through the read/write value property.

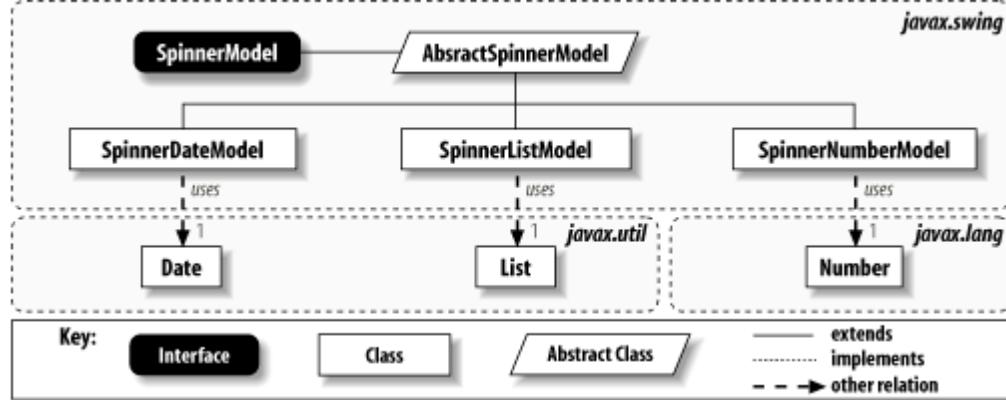
Table 7-16. JSpinner properties

Property	Data type	get	is	set	Default value

7.9 Spinner Models

The javax.swing package includes several pre-built models for many common data types suited to spinners. [Figure 7-15](#) shows the hierarchy of these models.

Figure 7-15. SpinnerModel class diagram



7.9.1 The SpinnerModel Interface

The SpinnerModel interface includes methods required to successfully store and retrieve spinner data. It includes a read/write value and next and previous properties, and it forces implementing models (such as AbstractSpinnerModel) to support a ChangeListener.

7.9.1.1 Properties

Not surprisingly, the properties for SpinnerModel are centered on the value being shown in the spinner. Notice in [Table 7-17](#) that the model stores only the current value and the next/previous values. The actual list (or other object) behind these values is not part of the model.

Table 7-17. SpinnerModel properties

Property	Data type	get	is	set	Default value
nextValue	Object				
previousValue	Object				
value	Object				

7.9.1.2 Events

Any changes to the selected value should be reported through ChangeEvent objects.

```
public void addChangeListener(ChangeListener l)
public void removeChangeListener(ChangeListener l)
```

Add or remove a specific listener for ChangeEvent notifications from the component.

7.10 Spinner Editors

You probably noticed that the JSpinner class also includes several inner classes. These inner classes provide basic editors (and renderers) for spinners for each of the major model types. While you'll typically rely on the editor picked by your spinner when you create it, you can override that decision if you like. Here's a simple example of a modified DateEditor. This spinner displays an mm/yy date, and the step size is one month. [Figure 7-16](#) shows such a spinner.

Figure 7-16. A customized DateEditor used in a JSpinner



Here's the source code that built this editor:

```
// MonthSpinner.java
```

Chapter 8. Swing Containers

In this chapter, we'll take a look at a number of components Swing provides for grouping other components together. In AWT, such components extended `java.awt.Container` and included `Panel`, `Window`, `Frame`, and `Dialog`. With Swing, you get a whole new set of options, providing greater flexibility and power.

8.1 A Simple Container

Not everything in this chapter is more complex than its AWT counterpart. As proof of this claim, we'll start the chapter with a look at the JPanel class, a very simple Swing container.

8.1.1 The JPanel Class

JPanel is an extension of JComponent (which, remember, extends java.awt.Container) used for grouping together other components. It gets most of its implementation from its superclasses. Typically, using JPanel amounts to instantiating it, setting a layout manager (this can be set in the constructor and defaults to a FlowLayout), and adding components to it using the add() methods inherited from Container.

8.1.1.1 Properties

JPanel does not define any new properties. [Table 8-1](#) shows the default values that differ from those provided by JComponent.

Table 8-1. JPanel properties

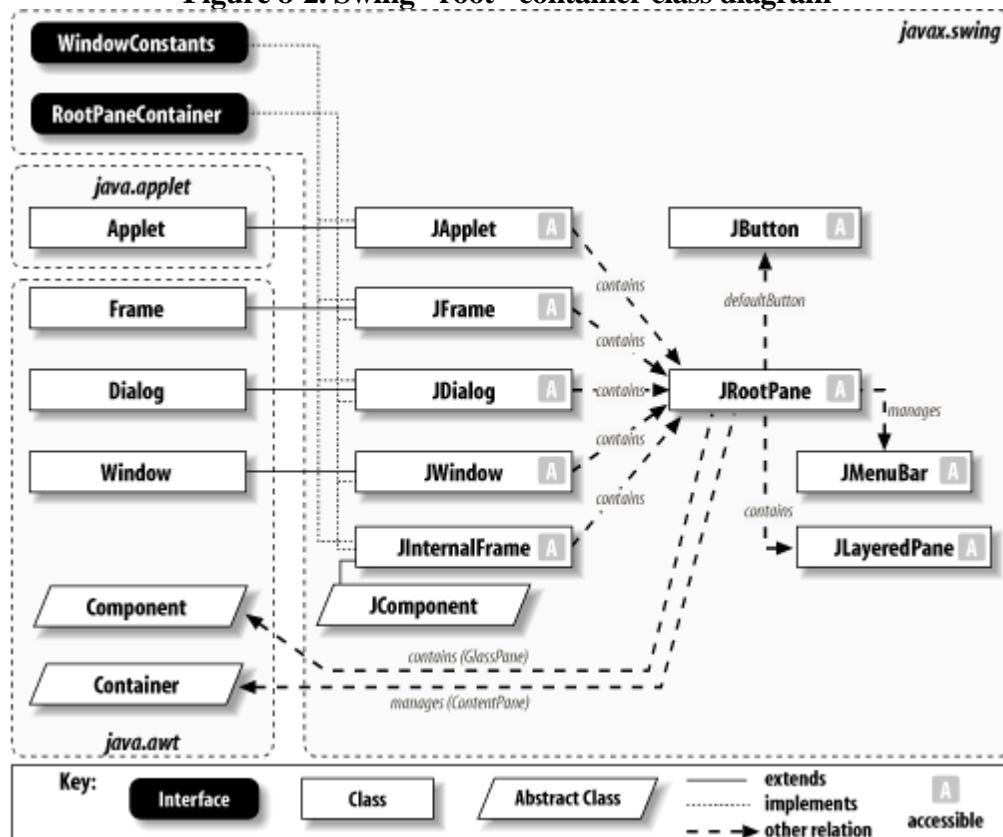
Property	Data type	get	is	set	Default value
accessibleContext	AccessibleContext				JPanel.AccessibleJPanel()
doubleBuffered	boolean				true
layout	LayoutManager				FlowLayout()
opaque, b	boolean				true
UI1.4	PaneUI				From L&F
UIClassID	String				"PanelUI"
1.4since 1.4, bbound, o overridden					
See also properties from the JComponent class (Table 3-6).					

8.2 The Root Pane

Now that we've seen the simplest example of a Swing container, we'll move on to something a bit more powerful. Most of the other Swing containers (JFrame, JApplet, JWindow, JDialog, and even JInternalFrame) contain an instance of another class, JRootPane, as their only component, and implement a common interface, RootPaneContainer. In this section, we'll look at JRootPane and RootPaneContainer, as well as another class JRootPane uses, JLAYEREDPANE.

Before jumping into the descriptions of these classes, let's take a look at how the classes and interfaces that make up the Swing root containers fit together. [Figure 8-2](#) shows that JApplet, JFrame, JDialog, and JWindow do not extend JComponent as the other Swing components do. Instead, they extend their AWT counterparts, serving as top-level user interface windows. This implies that these components (unlike the lightweight Swing components) have native AWT peer objects.

Figure 8-2. Swing "root" container class diagram



Notice that these Swing containers (as well as JInternalFrame) implement a common interface, RootPaneContainer. This interface gives access to the JRootPane's properties. Furthermore, each of the five containers uses a JRootPane as the "true" container of child components managed by the container. This class is discussed later in this chapter.

8.2.1 The JRootPane Class

JRootPane is a special container that extends JComponent and is used by many of the other Swing containers. It's quite different from most containers you're probably used to using. The first thing to understand about JRootPane is that it contains a fixed set of components: a Component called the glass pane and a JLAYEREDPANE called, logically enough, the layered pane. Furthermore, the layered pane contains two more components: a JMenuBar and a Container called the content pane. [\[1\]](#) [Figure 8-3](#) shows a schematic view of the makeup of a JRootPane.

[1] In general, JLAYEREDPANES can contain any components they wish. This is why [Figure 8-2](#) does not show JLAYEREDPANE containing the menu bar and content pane. In the case of the JRootPane, a JLAYEREDPANE is used to hold these two specific components.

8.3 Basic RootPaneContainers

For the rest of this chapter, we'll look at some basic containers (`JFrame`, `JWindow`, and `JApplet`) that implement `RootPaneContainer` and use `JRootPane`. First, we'll take a quick look at a simple interface called `WindowConstants`.

8.3.1 The WindowConstants Interface

`WindowConstants` is a simple interface containing only constants. It is implemented by `JFrame`, `JDialog`, and `JInternalFrame`.

8.3.1.1 Constants

The constants defined in `WindowConstants` specify possible behaviors in response to a window being closed. These values are shown in [Table 8-7](#).

Table 8-7. `JWindowConstants` constants

Constant	Type	Description
<code>DISPOSE_ON_CLOSE</code>	<code>int</code>	Disposes window when closed
<code>DO NOTHING ON CLOSE</code>	<code>int</code>	Does nothing when closed
<code>EXIT_ON_CLOSE</code> ^{1.4, *}	<code>int</code>	Exits the virtual machine when closed
<code>HIDE_ON_CLOSE</code>	<code>int</code>	Hides window when closed
1.4since 1.4		
*This constant was added in 1.4, although a matching constant was defined in the 1.3 <code>JFrame</code> class.		

In the next section, we'll look at a strategy for exiting the application in response to a frame being closed .

8.4 The JFrame Class

The most common Swing container for Java applications is the JFrame class. Like java.awt.Frame, JFrame provides a top-level window with a title, border, and other platform-specific adornments (e.g., minimize, maximize, and close buttons). Because it uses a JRootPane as its only child, working with a JFrame is slightly different than working with an AWT Frame. An empty JFrame is shown in [Figure 8-9](#).

Figure 8-9. Empty JFrame instances on Unix, Mac, and Windows platforms



The primary difference is that calls to add() must be replaced with calls to getContentPane().add(). In fact, the addImpl() method is implemented so that a call made directly to add() throws an Error. (The error message tells you not to call add() directly.)

8.4.1 Properties

JFrame defines the properties shown in [Table 8-8](#). The accessibleContext property is as expected. ContentPane , glassPane, layeredPane, and JMenuBar are really properties of JRootPane (described earlier in the chapter). JFrame provides direct access to these panes, as required by the RootPaneContainer interface.

Table 8-8. JFrame properties

Property	Data type	get	is	set	Default value
accessibleContext	AccessibleContext				JFrame.Accessible -JFrame()
background	Color				UIManager.getColor("control")
contentPane	Container				From rootPane
defaultCloseOperation	int				HIDE_ON_CLOSE
glassPane	Component				From rootPane
JMenuBar	JMenuBar				From rootPane
layeredPane	JLayeredPane				From rootPane

8.5 The JWindow Class

JWindow is an extension of java.awt.Window that uses a JRootPane as its single component. Other than this core distinction, JWindow does not change anything defined by the Window class.

In AWT, one common reason for using the Window class was to create a pop-up menu. Since Swing explicitly provides a JPopupMenu class (see [Chapter 14](#)), there is no need to extend JWindow for this purpose. The only time you'll use JWindow is if you have something that needs to be displayed in its own window without the adornments added by JFrame. Remember, this means that the window can only be moved or closed programmatically (or via the user's platform-specific window manager controls, if available).

One possible use for JWindow would be to display a splash screen when an application is starting up. Many programs display screens like this, containing copyright information, resource loading status, etc. Here's such a program:

```
// SplashScreen.java
```

8.6 The JApplet Class

JApplet is a simple extension of java.applet.Applet to use when creating Swing programs designed to be used in a web browser (or appletviewer). As a direct subclass of Applet, JApplet is used in much the same way, with the init(), start(), and stop() methods still playing critical roles. The primary thing JApplet provides that Applet does not is the use of a JRootPane as its single display component. The properties and methods described below should look a lot like those described in the previous sections on JFrame and JWindow. [Figure 8-11](#) shows a JApplet running in appletviewer.

Figure 8-11. A JApplet running in the SDK appletviewer



Here's the code for this simple applet:

```
// SimpleApplet.java
```

Chapter 9. Internal Frames

[Section 9.1. Simulating a Desktop](#)

[Section 9.2. The JInternalFrame Class](#)

[Section 9.3. The JDesktopPane Class](#)

[Section 9.4. The DesktopManager Interface](#)

[Section 9.5. Building a Desktop](#)

9.1 Simulating a Desktop

Some GUI applications keep their entire interface in a single root window, which looks like a desktop environment and contains internal "windows" and icons, elements you'd find on the actual desktop. This style of interface was first used in early versions of the Windows operating system because, at that time, the operating system didn't support multiple, overlapping real windows for each application, so there was no alternative.

While there are still a few special circumstances in which this kind of interface is actually desirable (for example, if you're creating an emulation of another computer or a virtual environment and want to keep that world clearly distinct in its own window), most applications would be better off using real windows on the actual desktop.

Applications that stick to a virtual desktop interface out of habit do their users a disservice in a number of ways. Because their internal frames (simulated windows) are restricted to exist within the application's root window, the user has to compromise between making that window very large so there is room to position the application's windows in a convenient way, and keeping it small so that the windows of other applications can be seen and accessed conveniently. It's usually impossible to come up with a happy medium. The user is also prevented from overlapping windows of this application with windows of any other application, even if this would provide a better workflow.

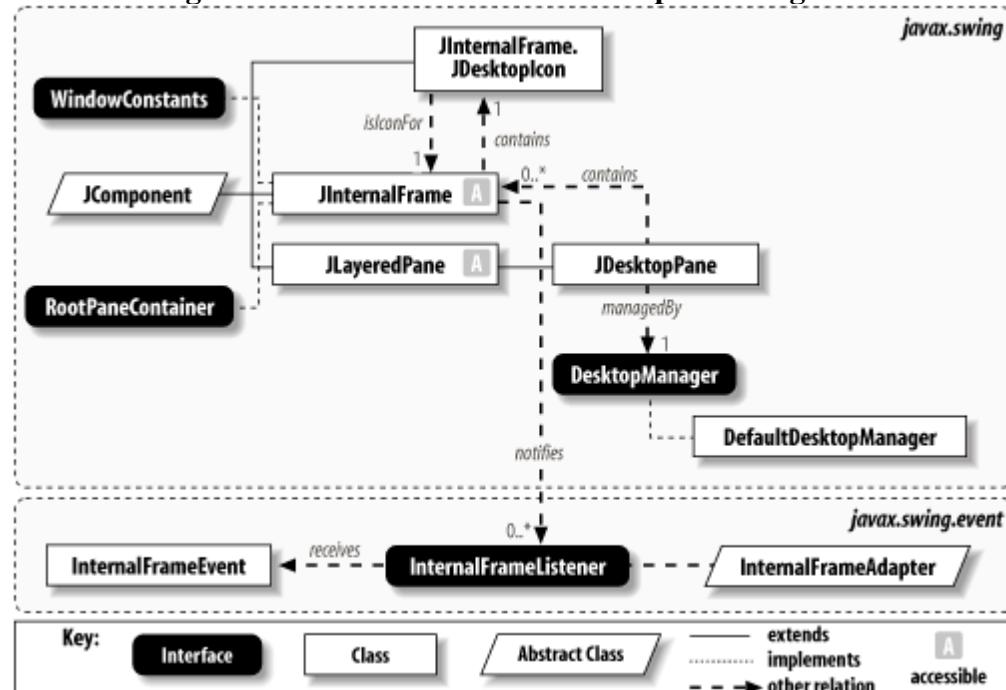
Most of the perceived advantages of using a simulated desktop environment in a typical application can be achieved in a better way by thoughtful use of application palettes and the positioning of separate windows. If (despite all these caveats) it turns out that you do need to create and manage your own desktop in a window, Swing can accommodate you.

In this chapter, we'll look at a collection of classes Swing provides to allow you to create this type of application in Java. At the end of the chapter, we'll provide a large sample program that shows how to implement a variety of useful features.

9.1.1 Overview

Before looking at each of the classes involved in the Swing desktop/internal frame model, we'll take a moment for an overview of how they all work together. [Figure 9-1](#) shows the relationships between the classes we'll be covering in this chapter.

Figure 9-1. Internal frame and desktop class diagram



9.2 The JInternalFrame Class

JInternalFrame provides the ability to create lightweight frames that exist inside other components. An internal frame is managed entirely within some other Java container, just like any other component, giving the program complete control over iconification, maximization, resizing, etc. Despite looking like "real" windows, the underlying windowing system knows nothing of the existence of internal frames.^[1] Figure 9-2 shows what internal frames look like in various L&Fs.

[1] Note that JInternalFrame extends JComponent, not JFrame or Frame, so this statement should seem logical.

Figure 9-2. JInternalFrames in four L&Fs



There's quite a lot to discuss about JInternalFrames, but most of their power comes when they are used inside a JDesktopPane. This section provides a quick overview of the properties, constructors, and methods available in JInternalFrame, and a more detailed discussion of using internal frames follows.

9.2.1 Properties

JInternalFrame defines the properties and default values shown in [Table 9-1](#). The background and foreground properties are delegated to the frame's content pane.

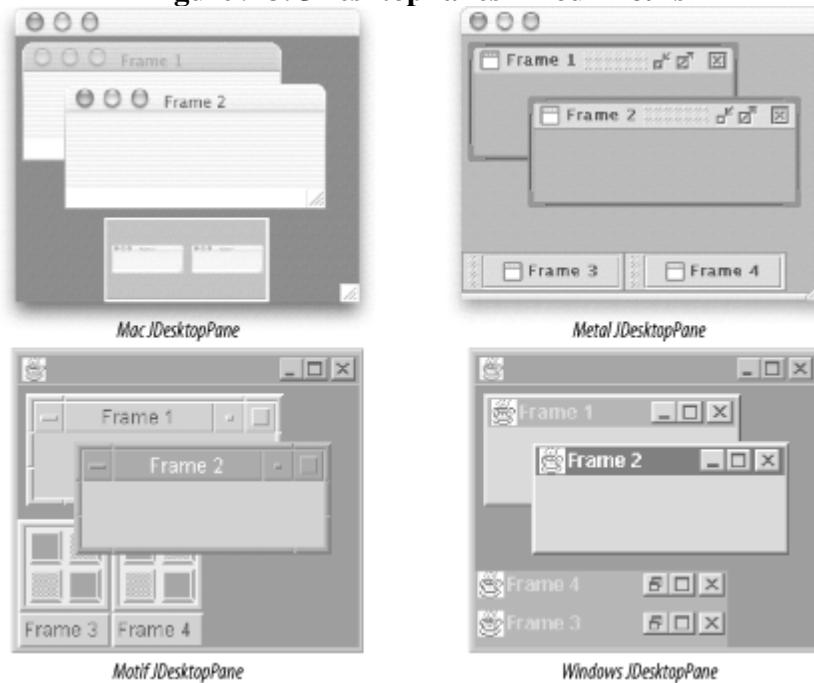
Table 9-1. JInternalFrame properties

Property	Data type	get	is	set	Default value
accessibleContext	AccessibleContext				JInternalFrame.AccessibleJInternalFrame()
closable	boolean				false
iconifiable	boolean				false

9.3 The JDesktopPane Class

JDesktopPane is an extension of JLAYEREDPANE, which uses a DesktopManager to control the placement and movement of frames. [Figure 9-3](#) shows what JDesktopPane looks like in several L&Fs. Like its superclass, JLAYEREDPANE has a null layout manager. Components added to it must be placed at absolute locations with absolute sizes because it is intended to house JInternalFrames, which rely on the user to determine their placement.

Figure 9-3. JDesktopPanes in four L&Fs



Another reason for using JDesktopPane is to allow pop-up dialog boxes to be displayed using JInternalFrames. This is discussed in detail in the next chapter.

9.3.1 Properties

[Table 9-4](#) shows the properties defined by JDesktopPane. The allFrames property provides access to all JInternalFrames contained by the desktop. The desktopManager property holds the DesktopManager object supplied by the pane's L&F. (We'll cover the responsibilities of the DesktopManager in the next section.) The opaque property defaults to true for JDesktopPanes, and isOpaque() is overridden so that it always returns true. UI contains the DesktopPaneUI implementation, and UIClassID contains the class ID for JDesktopPane.

Table 9-4. JDesktopPane properties

Property	Data type	get	is	set	Default value
accessibleContext o	AccessibleContext				JDesktopPane.AccessibleJDesktopP ane()
allFrames	JInternalFrame[]				Empty array
desktopManager	DesktopManager				From L&F

9.4 The DesktopManager Interface

This interface is responsible for much of the management of internal frames contained by JDesktopPanes. It allows an L&F to define exactly how it wants to manage things such as frame activation, movement, and iconification. Most of the methods in InternalFrameUI implementations should delegate to a DesktopManager object. As described earlier, DesktopManagers are contained by JDesktopPane objects and are intended to be set by the L&F. You can also create your own variations on the supplied implementations to provide custom behavior, as shown in the example that concludes this chapter.

9.4.1 Methods

The majority of the methods in this interface act on a given JInternalFrame. However, those methods that could be applied to other types of components do not restrict the parameter unnecessarily (they accept any JComponent), despite the fact that they are typically used only with JInternalFrames. If you implement your own DesktopManager or other L&F classes, you may find a need for this flexibility.

public abstract void activateFrame(JInternalFrame f)

Called to indicate that the specified frame should become active (is gaining focus).

public abstract void beginDraggingFrame(JComponent f)

Called to indicate that the specified frame is now being dragged. The given component is normally a JInternalFrame.

public abstract void beginResizingFrame(JComponent f, int direction)

Called to indicate that the specified frame will be resized. The direction comes from SwingConstants and must be NORTH, SOUTH, EAST, WEST, NORTH_EAST, NORTH_WEST, SOUTH_EAST, or SOUTH_WEST, representing the edge or corner being dragged (although this value is currently ignored by all provided implementations). The given component is normally a JInternalFrame. When resizing is complete, endResizingFrame() is called.

public abstract void closeFrame(JInternalFrame f)

Called to indicate that the specified frame should be closed.

public abstract void deactivateFrame(JInternalFrame f)

Called to indicate that the specified frame is no longer active (has lost focus).

public abstract void deiconifyFrame(JInternalFrame f)

Called to indicate that the specified frame should no longer be iconified.

public abstract void dragFrame(JComponent f, int newX, int newY)

Called to indicate that the specified frame should be moved from its current location to the newly specified coordinates. The given component is normally a JInternalFrame.

public abstract void endDraggingFrame(JComponent f)

Called to indicate that the specified frame is no longer being dragged. The given component is normally a JInternalFrame.

public abstract void endResizingFrame(JComponent f)

Called to indicate that the specified frame is no longer being resized. The given component is normally a JInternalFrame.

public abstract void iconifyFrame(JInternalFrame f)

Called to indicate that the specified frame should be iconified.

public abstract void maximizeFrame(JInternalFrame f)

9.5 Building a Desktop

In this section, we'll pull together some of the things we've discussed in the previous section to create an application using JDesktopPane, JInternalFrame, and a custom DesktopManager. The example will show:

- The effect of adding frames to different layers of the desktop
- How to display a background image ("wallpaper") on the desktop
- How to keep frames from being moved outside of the desktop
- How to deiconify, move, and resize internal frames by frame " tiling"
- How to take advantage of JInternalFrame's constrained properties by requiring that there be at least one noniconified frame on the desktop

[Figure 9-4](#) shows what the application looks like when it's running. Here, we see the desktop with three frames, plus a fourth that has been iconified. The frames titled "Lo" are in a lower layer than the "Up" frames. No matter which frame is active or how the frames are arranged, the "Up" frame always appears on top of the others. Frames in the same layer can be brought to the front of that layer by clicking on the frame. This display also shows the use of a background image (what good is a desktop if you can't put your favorite image on the background, right?). This image is added to a very low layer (the lowest possible Java int, actually) to ensure that it is always painted behind anything else in the desktop. [Figure 9-5](#) shows the same display after the frames have been "tiled."

Figure 9-4. SampleDesktop layered frames and background image

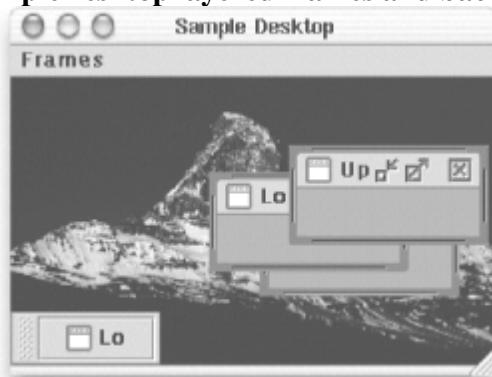


Figure 9-5. SampleDesktop with tiled frames



Chapter 10. Swing Dialogs

In most applications, information occasionally needs to be displayed for a brief period of time, often just long enough for the user to read it and click OK or perhaps enter a value, such as a password. Swing provides the JOptionPane class to make creating such simple dialog boxes extremely easy — in many cases requiring just one line of code.

Applications may also serve more complex dialog needs, such as providing a property editor in which a set of related values can be modified, with an appropriate interface. Swing's JDialog class supports such general-purpose dialogs. JDialogs can also be non-modal,[\[1\]](#) which means the user does not need to close the dialog before interacting with other application windows. When possible, implementing such an interface yields a more pleasant and productive user experience.

[1] A modal dialog forces the user to interact only with that dialog until it is dismissed.

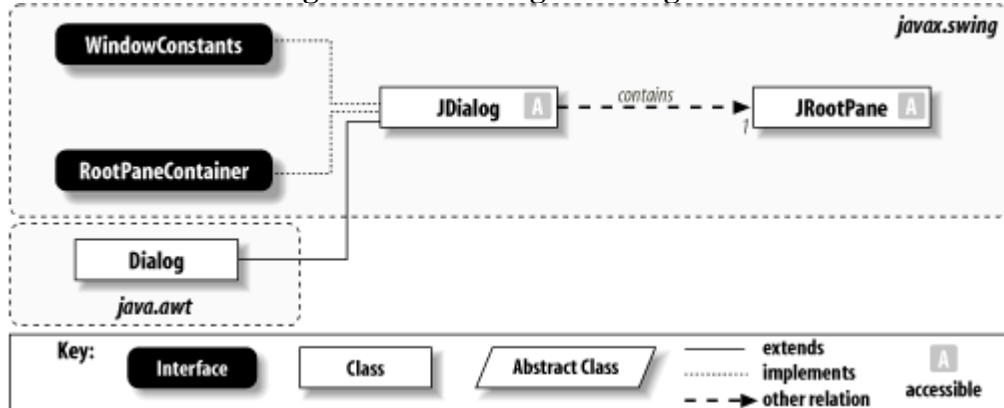
Even though JOptionPane makes it very easy (for the programmer) to pop up a dialog, bear in mind that this will disrupt the flow of activity for users and force them to deal with the dialog before they can proceed with their underlying task. While this is sometimes unavoidable or even appropriate, it is usually worth trying to find less disruptive alternatives (direct manipulation, a non-modal floating notification, or some other non-modal approach). This may require more work on the part of the developer but will result in a better application. And if the application is widely adopted, the benefits are multiplied across the entire user base.

10.1 The JDialog Class

JDialog is the Swing version of its superclass, java.awt.Dialog. It provides the same key features described in [Chapter 8\[2\]](#) in the discussion of JWindow, JFrame, and JApplet: it uses a JRootPane as its container, and it provides default window-closing behavior. Since JDialog extends java.awt.Dialog, it has a heavyweight peer and is managed by the native windowing system. [Figure 10-1](#) shows how JDialog fits into the class hierarchy.

[2] Certain parts of this chapter assume that you have read at least part of [Chapter 8](#).

Figure 10-1. JDialog class diagram



10.1.1 Properties

JDialog defines the properties and default values listed in [Table 10-1](#). The content-Pane, glassPane, JMenuBar, and layeredPane properties are taken from rootPane, which is set to a new JRootPane by the constructor.

Table 10-1. JDialog properties

Property	Data type	get	is	set	Default value
accessibleContext	AccessibleContext				JDialog.Accessible JDialog()
contentPane	Container				From rootPane
defaultCloseOperation	int				HIDE_ON_CLOSE
defaultLookAndFeelDecorated, 1.4	boolean				Depends on L&F, often false
glassPane	Component				From rootPane

10.2 The JOptionPane Class

JOptionPane is a utility class used to create complex JDialogs and JInternalFrames (the latter of which is used for lightweight dialogs). [Figure 10-2](#) shows where JOptionPane fits into the class hierarchy; [Figure 10-3](#) shows JOptionPane in four L&Fs. It provides a range of convenient ways to create common pop-up modal dialog boxes, which significantly reduces the amount of code you are required to write, at the expense of forcing the user to drop whatever she's doing and react to the pop up.

Figure 10-2. JOptionPane class diagram

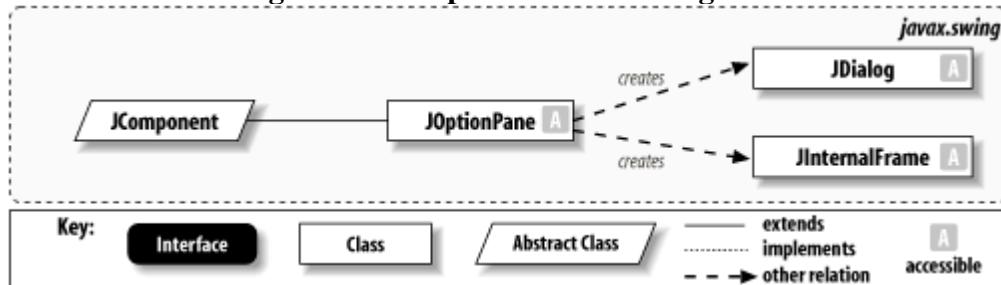
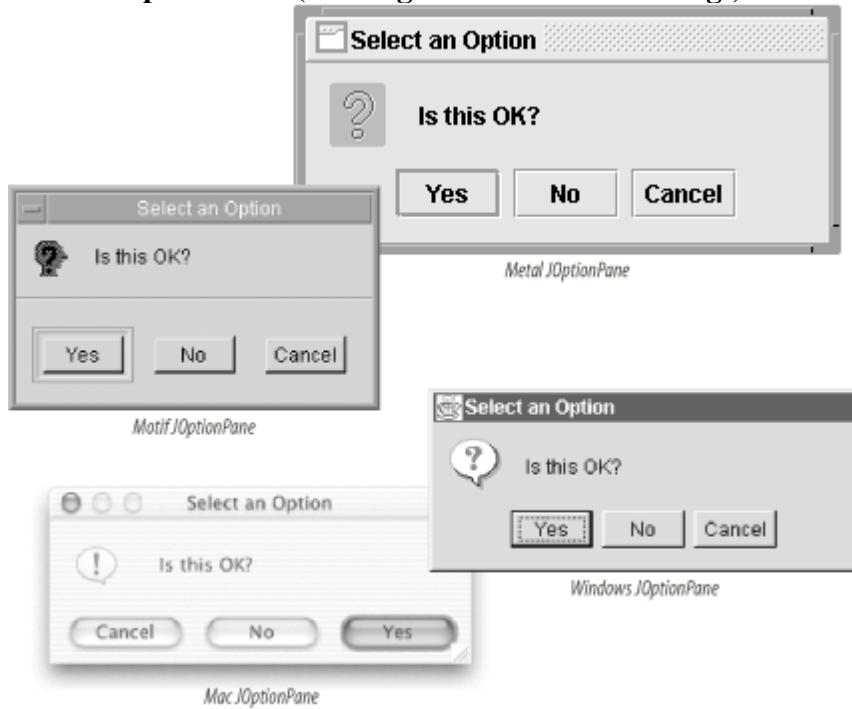


Figure 10-3. JOptionPanees (showing internal confirm dialogs) in four L&Fs



For example, to create a very simple dialog window with the text "Click OK after you read this" and an OK button without JOptionPane, you'd have to write something like this:

```
public void showSimpleDialog(JFrame f) {
```


10.3 Using JOptionPane

There are basically two ways to use JOptionPane. The simplest, demonstrated by the example at the beginning of this section, is to invoke one of the many static methods of the class. These methods all result in a JDialog or JInternalFrame being displayed immediately for the user. The methods return when the user clicks one of the buttons in the dialog.

The other way to use JOptionPane is to instantiate it using one of the many constructors and then call createDialog() or createInternalFrame(). These methods give you access to the JDialog or JInternalFrame and allow you to control when and where they are displayed. In most cases, the static methods will do everything you need. As with JDialog constructors, the static methods can (since 1.4) potentially throw a HeadlessException if the graphics environment is operating in a "headless" mode, meaning that there is no display, keyboard, or mouse.

It's worth noting that JOptionPane extends JComponent. When you instantiate JOptionPane, you actually have a perfectly usable component, laid out with the structure we described earlier. If you wanted to, you could display the component directly, but it typically makes sense to use it only in a dialog or internal frame.



All of the methods in JOptionPane that result in the creation of a JDialog create modal dialogs, but the methods that display JInternalFrames do not enforce modality. When the internal frame dialog is displayed, the other windows in your application can still receive focus. Typically, you will create modal dialogs. The situations in which you might use internal frames are discussed in the preceding chapter. See [Section 10.7](#) later in this chapter for more information about working with dialogs as internal frames.

10.3.1 Events

JOptionPane fires a PropertyChangeEvent any time one of the bound properties listed in [Table 10-6](#) changes value. This is the mechanism used to communicate changes from the JOptionPane to the JOptionPaneUI, as well to the anonymous inner class listeners JOptionPane creates to close the dialog when the value or inputValue property is set by the L&F.

10.3.2 Constants

Tables [Table 10-3](#) through [Table 10-6](#) list the many constants defined in this class. They fall into four general categories:

Message types

Used to specify what type of message is being displayed
Option types

Used to specify what options the user should be given
Options

Used to specify which option the user selected
Properties

Contains the string names of the pane's bound properties

Table 10-3. JOptionPane constants for specifying the desired message type

10.4 Simple Examples

The following are a few examples that show some of things you can do with JOptionPane static methods and constructors.

Here's an input dialog with more than 20 selection values. This results in the creation of a JList ([Figure 10-8](#)):

```
JOptionPane.showInputDialog(null, "Please choose a name", "Example 1",
```

I1@ve RuBoard

10.5 Getting the Results

Now that we've seen how to create all sorts of useful dialog boxes, it's time to take a look at how to retrieve information about the user's interaction with the dialog. [Table 10-7](#) showed the return types of the various methods. Here's a quick summary of what the returned values mean.

Input Dialogs

The versions that do not take an array of selection values return a String. This is the data entered by the user. The methods that do take an array of selection values return an Object reflecting the selected option. It's up to the L&F to determine the component used for presenting the options. Typically, a JComboBox is used if there are fewer than 20 choices, and a JList is used if there are 20 or more.[\[10\]](#) In any case, if the user presses the Cancel button, null is returned.

[10] Prior to SDK 1.4, the JList allowed the user to highlight multiple selections even though only the first could be detected. This has been fixed to allow only a single selection to be highlighted.

Confirm Dialogs

These methods return an int reflecting the button pressed by the user. The possible values are: YES_OPTION, NO_OPTION, CANCEL_OPTION, and OK_OPTION. CLOSED_OPTION is returned if the user closes the window without selecting anything.

Message Dialogs

These methods have void return types because they do not request a user response.

Option Dialogs

If no options are specified, this method returns one of the constant values YES_OPTION, NO_OPTION, CANCEL_OPTION, and OK_OPTION. If options are explicitly defined, the return value gives the index to the array of options that matches the button selected by the user. CLOSED_OPTION is returned if the user closes the window without selecting anything.

Getting a value from a JOptionPane you've instantiated directly is also very simple. The value is obtained by calling the pane's getValue() method. This method returns an Integer value using the same rules as those described for option dialogs with two small variations. Instead of returning an Integer containing CLOSED_OPTION, getValue() returns null if the dialog is closed. Also, if you call getValue() before the user has made a selection (or before displaying the dialog at all, for that matter), it will return UNINITIALIZED_VALUE. To get the value of user input (from a JTextField, JComboBox, or JList), call getInputValue(). This will return the entered String or the selected Object (which may also be a String). Note that, just as with the static "show" methods, there's no way to find out about multiple selections the user may have made when there are more than 20 choices.

The following example contains code to retrieve results from JOptionPanees.

10.6 A Comparison: Constructors Versus Static Methods

We've talked quite a bit about the two fundamental ways to create dialogs using JOptionPane: instantiate a JOptionPane and ask it to put itself into a JDialog or JInternalFrame, which you then display, or create and display the dialog in a single step by invoking one of the many static "show" methods.

The basic trade-off is this: using the static methods is a bit easier, but using a constructor allows you to hold onto and reuse the JOptionPane instance, a tempting feature if the pane is fairly complex and you expect to display it frequently (if you use the static methods, the option pane is recreated each time you call). The significance of this difference depends largely on the complexity of the pane. Because of lingering issues that make reusing JOptionPane problematic, it's still best to avoid this feature (see the note in the discussion following this example program for details).

The following example shows the differences between using JOptionPane's static methods and its constructors. It allows both internal and noninternal dialogs to be created, showing how each is done.

```
// OptPaneComparison.java
```

10.7 Using Internal Frame Dialogs with JDesktopPane

In order to get the best results when using internal frame dialogs created by JOptionPane, the dialogs need to be placed in a JDesktopPane. However, this may not be convenient if your application does not use a JDesktopPane. In this section, we'll show how you can easily adapt your application to use a JDesktopPane so that you can use internal frame dialogs.

Recall that JDesktopPane has a null layout manager, leaving the management of the location of its contents up to the DesktopManager and the user. This makes JDesktopPane a poor choice when you just need a container in which to build your main application. As a result, if you want to have an internal frame dialog displayed in a "normal" container, you need a solution that gives you the features of both JDesktopPane and a more layout-friendly container.

This is actually a pretty straightforward goal to achieve. You need to create a JDesktopPane and add your application container to it so that it fills an entire layer of the desktop. When there are no internal frames displayed, this looks the same as if you were displaying the application container alone. The benefit is that when you need to add an internal frame dialog, you have a desktop to add it to.

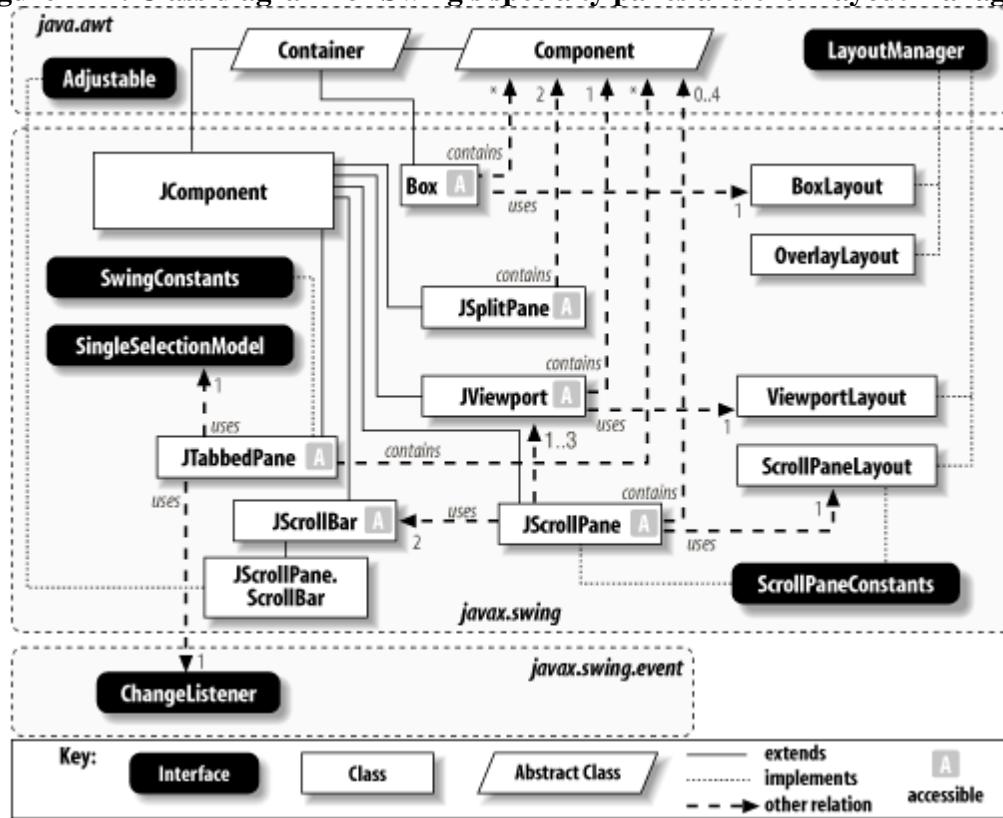
Here's a simple example that shows how this works. It also shows how you can make sure your container fills the desktop, even if the desktop changes size (since there's no layout manager, this won't happen automatically).

```
// DialogDesktop.java
```

Chapter 11. Specialty Panes and Layout Managers

With all the spiffy Swing components out there, you might expect to see a few new layout managers to help place them, and you wouldn't be disappointed. The Swing package includes several layout managers. However, most of these managers are designed for specific containers—JScrollPane has its own ScrollPaneLayout manager, for example. The Swing package also includes several new convenience containers that handle things such as scrolling and tabs. (We'll take a close look at these containers and their associated layout managers in this chapter.) [Figure 11-1](#) shows a class diagram of Swing's specialty panes and their layout managers. The OverlayLayout and SpringLayout general layout managers can be used with any containers. We tackle them separately at the end of this chapter. (SpringLayout was added in SDK 1.4.)

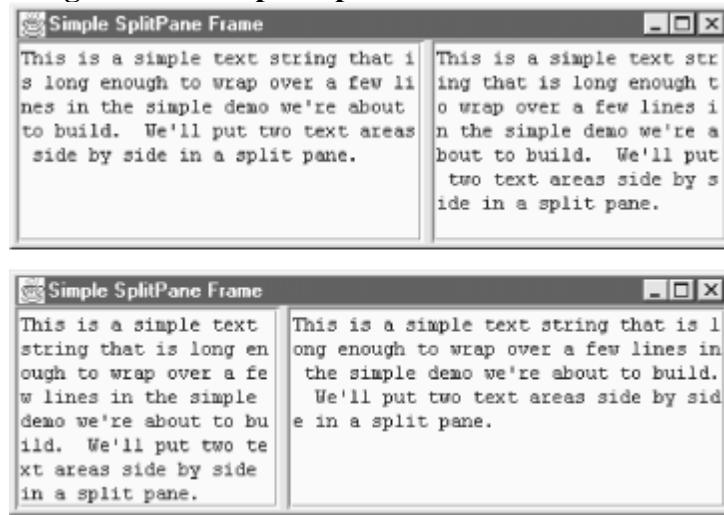
Figure 11-1. Class diagram for Swing's specialty panes and their layout managers



11.1 The JSplitPane Class

The JSplitPane component allows you to place two (and only two) components side by side in a single pane. You can separate the pane horizontally or vertically, and the user can adjust this separator graphically at runtime. You have probably seen such a split pane approach in things like a file chooser or a news reader. The top or left side holds the list of directories or news subject lines while the bottom (or right side) contains the files or body of the currently selected directory or article. To get started, [Figure 11-2](#) shows a simple split pane example that shows two text areas with a horizontal split. You can adjust the width of the split by grabbing the divider and sliding it left or right.

Figure 11-2. Simple JSplitPane with two text areas



Even with the code required to make the text areas behave (more on that in [Chapter 19](#)), the following example is still fairly simple. If you are looking to get up and running with a quick split pane, this is the way to go.

```
// SimpleSplitPane.java
```

11.2 The JScrollPane Class

The JScrollPane class offers a more flexible version of the ScrollPane class found in the AWT package. Beyond the automatic scrollbars, you can put in horizontal and vertical headers as well as active components in the corners of your pane. ([Figure 11-6](#) shows the exact areas available in a JScrollPane, which is managed by the ScrollPaneLayout class.)

Many Swing components use JScrollPane to handle their scrolling. The JList component, for example, does not handle scrolling on its own. Instead, it concentrates on presenting the list and making selection easy, assuming you'll put it inside a JScrollPane if you need scrolling. [Figure 11-3](#) shows a simple JScrollPane in action with a JList object.

Figure 11-3. JScrollPane showing two portions of a list that is too long for one screen



This particular example does not take advantage of the row or column headers. The scrollpane adds the scrollbars automatically, but only "as needed." If we were to resize the window to make it much larger, the scrollbars would become inactive. Here's the code that builds this pane:

```
// ScrollList.java
```

11.3 The JTabbedPane Class

The tabbed pane is now a fixture in applications for option displays, system configuration displays, and other multiscreen UIs. In the AWT, you have access to the CardLayout layout manager, which can be used to simulate the multiscreen behavior, but it contains nothing to graphically activate screen switching—you must write that yourself. [Figure 11-7](#) shows that with the JTabbedPane, you can create your own tabbed pane, with tab activation components, very quickly.

Figure 11-7. A simple tabbed pane with three tabs in several L&Fs



Here's the code that generated this simple application. We use the tabbed pane as our real container and create new tabs using the addTab() method. Note that each tab can contain exactly one component. As with a CardLayout-managed container, you quite often add a container as the one component on the tab. That way, you can then add as many other components to the container as necessary.

```
// SimpleTab.java
```


11.4 Layout Managers

Beyond these specialty panes with their dedicated layout managers, the Swing package also includes some general layout managers you can use with your own code. You can use the new BoxLayout to make things like toolbars and OverlayLayout to make things like layered labels.

The BoxLayout class is a manager that gives you one row or column to put everything in. It's great for toolbars and button ribbons. It also comes with its very own convenience container called Box. The Box class is a lightweight container that requires a BoxLayout manager. While you can certainly use the BoxLayout class to control your own panel, frame, or other container, the Box class provides several shortcuts for dealing with components in a boxed layout. You'll often find that using a Box is easier than creating a panel or frame that you control with a BoxLayout manager.

11.4.1 The Box Class

Let's start with a look at the convenience container that puts the BoxLayout manager to use. The Box class is a lightweight container object whose primary purpose is to let you add components to a horizontal or vertical box without having to think about getting the constraints right. (As of SDK 1.4, Box extends JComponent.) You use the normal Container.add() method to place components in the box. Components are placed left to right (or top to bottom) in the order you add them.

11.4.1.1 Properties

[Table 11-11](#) shows the properties of the Box class. You are not allowed to change a box's layout manager, so the setLayout accessor always throws an AWTError.

Table 11-11. Box properties

Property	Data type	get	is	set	Default value
accessibleContext o	AccessibleContext				box.AccessibleBo x
layouto	layoutManager				BoxLayout
ooverridden See also properties from the JComponent class (Table 3-6).					

11.4.1.2 Constructor

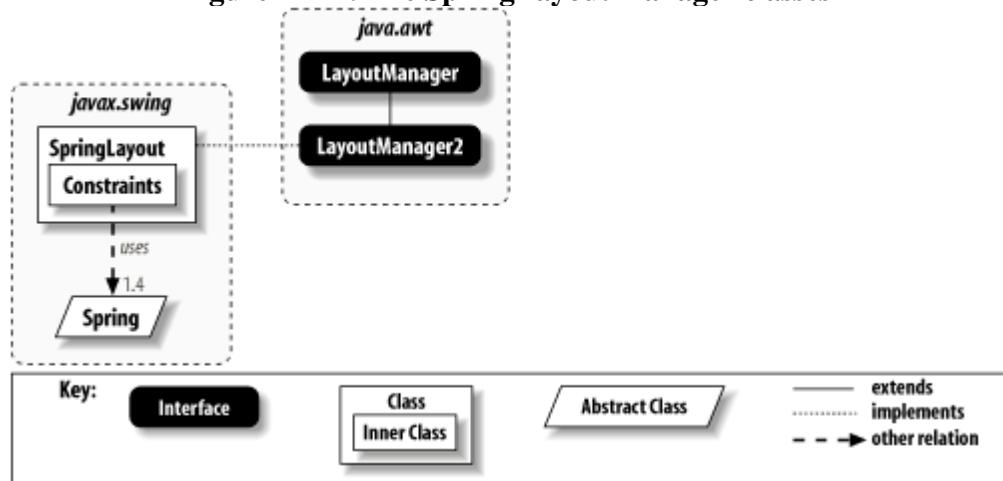
public Box(int alignment)

Create a container with a BoxLayout manager using the specified alignment. The possible values for the alignment are BoxLayout.X_AXIS (a horizontal box) and BoxLayout.Y_AXIS (a vertical box). (Refer to [Figure 11-13](#) for an

11.5 The SpringLayout Class

With SDK 1.4, a new—but not really new—layout manager was added. The SpringLayout manager uses the notion of springs and struts to keep everything in place. A version of SpringLayout existed in the early alpha and betas of the Swing package, but it was not included because the Swing team felt it still needed too much work. While it still needs a bit of work, it has come a long way. Its inclusion in SDK 1.4 is a testament to that progress. The class diagram for SpringLayout and its helpers is shown in [Figure 11-17](#).

Figure 11-17. The SpringLayout manager classes



Before you dive too deeply into this layout manager, you should know that its purpose in life is to aid GUI builders and other code-generating tools. It can certainly be hand-coded—and we have the examples to prove it—but you'll often leave this layout manager to the aforementioned tools. (If you want a flexible replacement for the GridBagLayout, you might want to take a look at the RelativeLayout manager written by our own Jim Elliott. The complete package with docs, tutorial, and source code can be found on this book's web site, <http://www.oreilly.com/catalog/jswing2/>.)

11.5.1 Springs and Struts

Now that you're here for the long haul, let's look at the core of the SpringLayout manager's approach to component layout: spring and struts. A *spring* is effectively a triplet representing a range of values. It contains its minimum, preferred, and maximum lengths. A *strut* is a spring with all the spring removed—its minimum, preferred, and maximum lengths are identical. With SpringLayout at the helm, you use springs and struts to specify the bounds (x, y, width, height) of all your components. (You can mimic the null layout manager by using only struts.)

The not-so-obvious big win in this layout manager is that springs can be anchored between the edges of components and will maintain their relationship even when the container is resized. This makes it possible to create layouts that would be difficult in other managers. While you could probably use a grand GridBagLayout to do the trick, SpringLayout should provide better performance once it's all fixed up and finalized.

[Figure 11-18](#) shows a simple application that uses SpringLayout. We position directional buttons over a large picture for navigation. Notice how the North button stays horizontally centered and anchored to the top edge of the application after we resize the frame. The other buttons behave similarly. Just to reiterate, you could certainly accomplish this with nested containers or a properly constructed GridBagLayout; the SpringLayout should simply prove to be the most maintainable over the long haul. We'll look at the source code for this example after we examine the API in more detail.

Figure 11-18. A SpringLayout-managed container at two different sizes



11.6 Other Panes

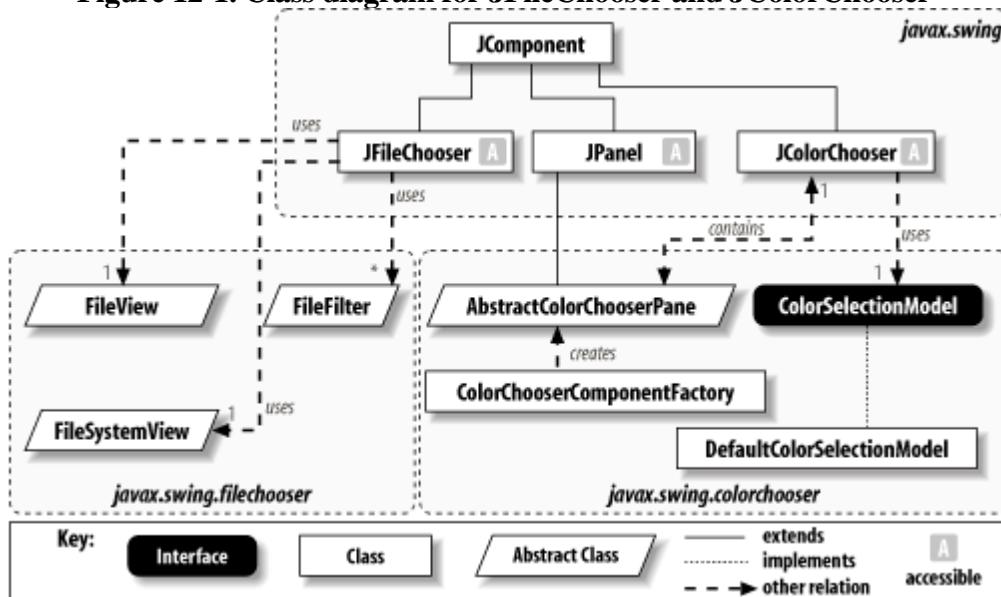
[Chapter 10](#) showed you the basic "panes" such as JOptionPane. With the containers and layout managers shown in this chapter, you can create just about any pane you like. However, there are some very common application panes that we have not yet discussed. [Chapter 12](#) describes some of the other panes available, including a file chooser and a color chooser.

Chapter 12. Chooser Dialogs

Just about every application you write these days needs to have a mechanism for opening and saving files. In the AWT, you can use the `FileDialog` class, but this is a heavyweight dialog that lacks the flexibility of the Swing components we've seen so far. The `JFileChooser` is Swing's answer to the `FileDialog`. The Swing package also contains a helper dialog for choosing colors (a common task during application configuration). We'll look at both of these dialogs in this chapter.

To get things started, [Figure 12-1](#) shows the class hierarchy of the pieces we'll be looking at in this chapter.

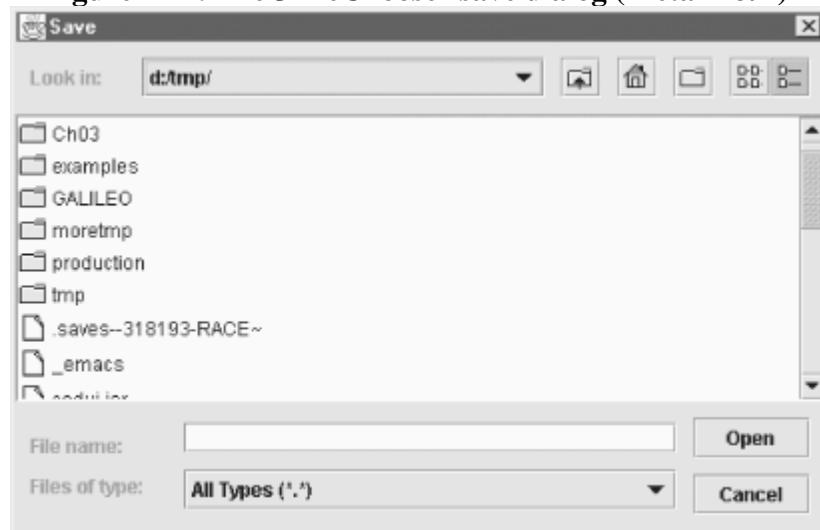
Figure 12-1. Class diagram for `JFileChooser` and `JColorChooser`



12.1 The JFileChooser Class

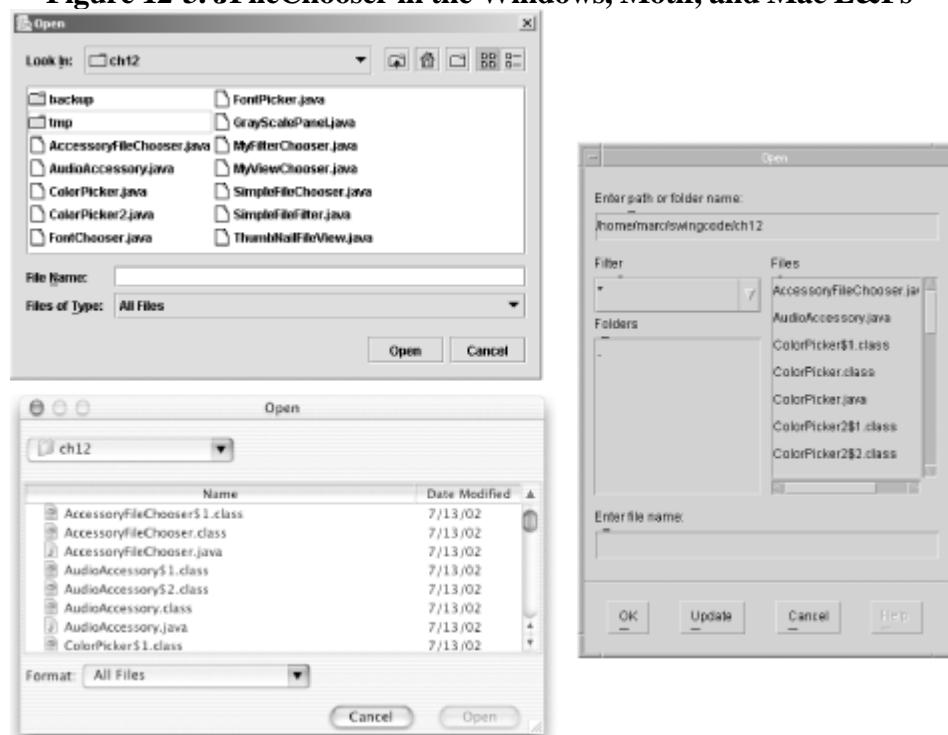
Since it plays such an integral role in just about every commercial application, let's look at the file chooser first. The `JFileChooser` class bundles a directory pane and typical selection buttons into a handy interface. [Figure 12-2](#) shows the dialog window you get when you select the Save option of a simple application. As you might expect, other L&Fs can also be applied to this chooser.

Figure 12-2. The `JFileChooser` save dialog (Metal L&F)



[Figure 12-3](#) shows sample file choosers for open and save dialogs in the Windows, Motif, and Mac OS X L&Fs.

Figure 12-3. `JFileChooser` in the Windows, Motif, and Mac L&Fs



The application itself reports only which file (or files, if you use the Open option) you chose to open or save. Our application has a Pick Directory button that restricts the chooser to directories. The event handlers for each button do most of the interesting work. In each case, we create a new `JFileChooser` object, make any changes to the default properties that we need for the particular action, and then show the dialog. As you will see from the constants discussed later, the int returned from the `showDialog()` method indicates whether the user accepted a file selection or canceled the dialog. If we have a successful selection, our example application puts the name of the file into a display label.

12.2 The File Chooser Package

Under javax.swing, you'll find a package of helper classes for the JFileChooser. The javax.swing.filechooser package contains several classes for displaying and filtering files.

12.2.1 The FileFilter Class

The FileFilter class can be used to create filters for JFileChooser dialogs. The class contains only two abstract methods. It's important to note that extensions are not the only way to judge a file's fitness for a particular filter. The Mac filesystem, for example, can understand the creator of a file regardless of the file's name. On Unix systems, you might write a filter to display only files that are readable by the current user.

12.2.1.1 Constructor

public FileFilter()

The FileFilter class receives this default constructor at compile time; it is not defined in the class itself.

12.2.1.2 Filter methods

public abstract boolean accept(File f)

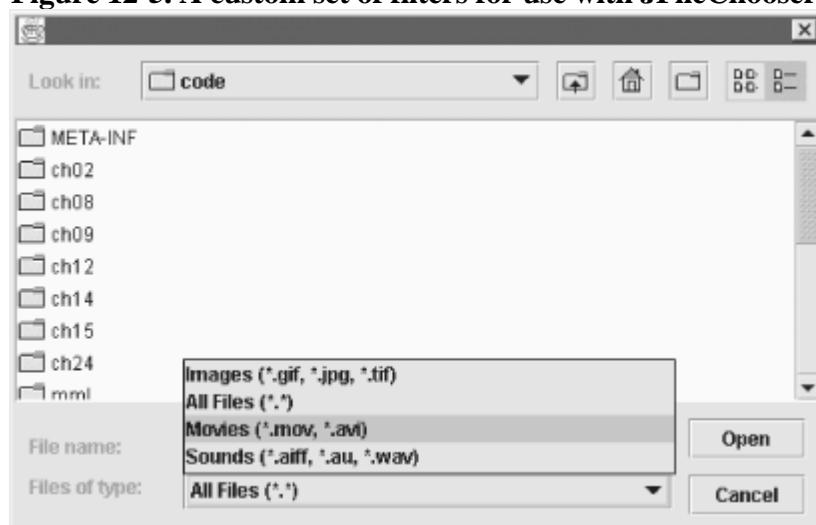
Return true if file f matches this filter. Note that you must explicitly accept directories (f.isDirectory() == true) if you want to allow the user to navigate into any subfolders.

public abstract String getDescription()

Return a short description to appear in the filters pulldown on the chooser. An example would be "Java Source Code" for any .java files.

[Figure 12-5](#) shows a file chooser with custom filters for multimedia file types.

Figure 12-5. A custom set of filters for use with JFileChooser



Here's the code for the application. Before we make this chooser visible, we create and insert the three new filters for our media types. Other than that, it's pretty much the same code as our previous applications. The Swing demos included in the SDK provide access to a similar extension-based file filter class. However, we use this example anyway, as it illustrates the inner workings of a filter that should seem familiar to most programmers.

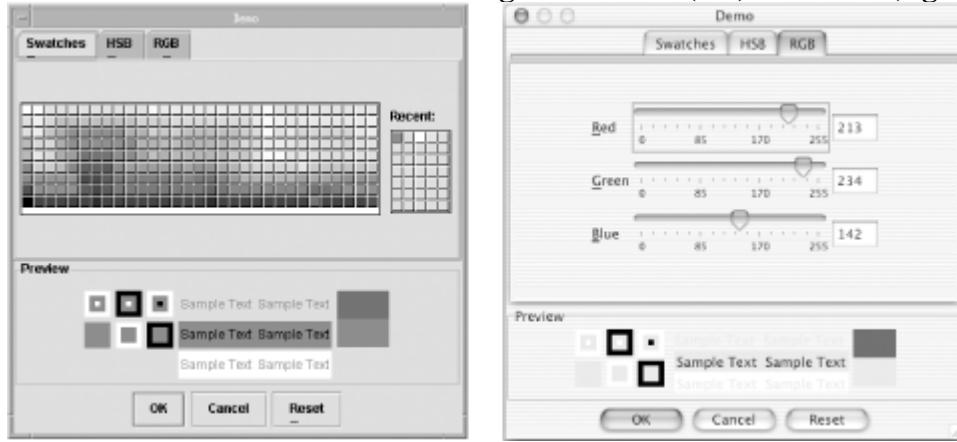
```
// MyFilterChooser.java
```

11@ve RuBoard

12.3 The Color Chooser

As the name indicates, the JColorChooser component is designed to allow users to pick a color. If your application supports customized environments (like the foreground, background, and highlight colors for text), this control might come in handy. You can pick a color from a palette and then look at that color in a preview panel that shows you how your color looks with black and white. The dialog also has an RGB mode that allows you to pick the exact amounts of red, blue, and green using sliders. The standard color chooser window looks like [Figure 12-7](#).

Figure 12-7. The default JColorChooser dialog in Swatches (left) and RGB (right) modes



The JColorChooser class provides a static method for getting this pop up going quickly. Here's the code that produced the screens in [Figure 12-7](#):

```
// ColorPicker.java
```

I1@ve RuBoard

12.4 The JColorChooser Class

The JColorChooser class allows you to create a standard dialog with a color palette from which users can select a color.

12.4.1 Properties

In addition to the typical UI properties of Swing components, the color chooser has the properties listed in [Table 12-6](#). The chooserPanels property contains an array of all the chooser panels currently associated with this color chooser. You can get and set the entire array at once or, more commonly, you can add and remove chooser panels using some of the methods described later. The color property contains the currently selected color in the chooser. (This property is just a convenient access point for the selectedColor property of the selectionModel.) The previewPanel property contains the JComponent subclass that previews your color choice. (You can see an example of the default preview panel in [Figure 12-7](#).) The selectionModel property dictates which selection model the chooser uses. The dragEnabled property allows you to drag colors from the chooser to another part of your application, but no transferHandler (inherited from JPanel) is in place to support this feature yet. You would need to supply a handler to make this property meaningful.

Table 12-6. JColorChooser properties

Property	Data type	get	is	set	Default value
accessibleContext	AccessibleContext				JColorChooser.AccessibleJColorChooser()
chooserPanels	AbstractColorChooser-Panel[]				null
color	Color				Color.white
dragEnabled	boolean				false
previewPanel	JComponent				null
selectionModel	ColorSelectionModel				DefaultColorSelectionModel
UI	ColorChooserUI				From L&F
UIClassID	String				"ColorChooserUI"

12.5 Developing a Custom Chooser Panel

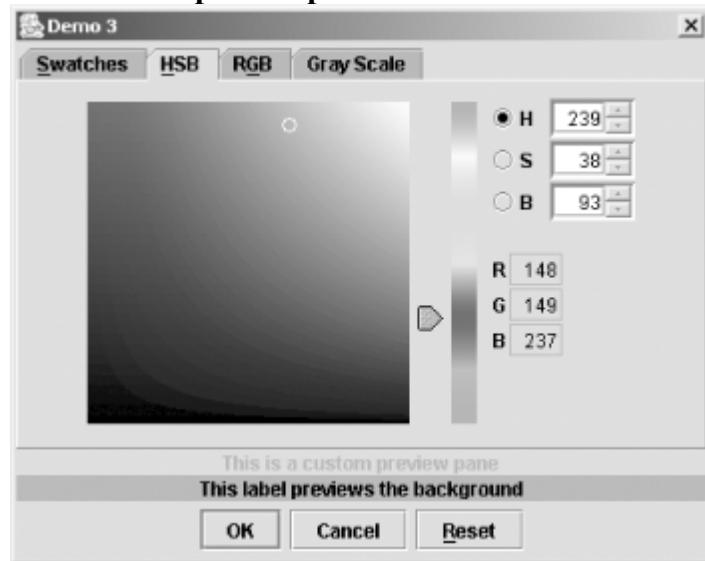
If you look at the JColorChooser component, you'll realize that it is really just a tabbed pane with a color previewer below it. You can have as many chooser panels in it as you like. Let's take a brief look at a panel that can be added to a color chooser. We'll create a simple panel for selecting a shade of gray with one slider rather than pushing each slider for red, green, and blue to the same value. [Figure 12-8](#) shows the resulting panel; its source code is presented here.

```
// GrayScalePanel.java  
I1@ve RuBoard
```

12.6 Developing a Custom Preview Panel

In addition to creating custom color chooser panels, you can create your own preview panel. You can use any `JComponent`—just set the `previewPanel` property. As you update the color in the chooser, the `foreground` property of the preview panel changes. You can extend a container like `JPanel` and override the `setForeground()` method to gain more control over what parts of the pane are updated. [Figure 12-9](#) shows a simple custom preview pane. We add two labels: one to show the foreground color (this label says, "This is a custom preview pane") and one to show the background color.

Figure 12-9. A custom preview panel added in a `JColorChooser` object



Remember that some L&Fs don't allow you to set the foreground or background colors of some components. If you're on a Mac OS X system, for example, you can run `ColorPicker3` this way:

```
% java -Dswing.defaultlaf=javax.swing.plaf.metal.MetalLookAndFeel ColorPicker3
```

This custom pane is added to our `ColorPicker3` chooser by setting the `previewPanel` property after we create the chooser object:

```
chooser.setPreviewPanel(new CustomPane());
```

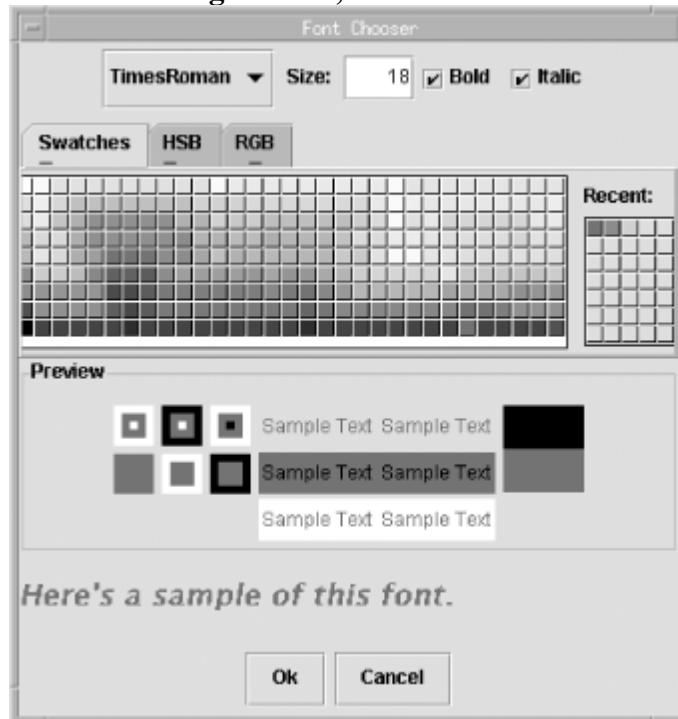
Here's the code for the `CustomPane` class. We build it as an inner class.

```
public class CustomPane extends JPanel {
```

12.7 Developing a Custom Dialog

While you might rely entirely on the standard color chooser dialog, it is possible to create a color chooser component and use it inside your own dialogs or applications. Let's take a look at a fancy font chooser that lets you pick the face, style, and color. [Figure 12-10](#) shows an example of such a dialog.

Figure 12-10. A custom dialog window, with a JColorChooser as one piece of it



It looks like a lot is going on in the code that built this dialog window, but it's not really that bad. The first part of the code is devoted to the tedious business of setting up the graphical-interface pieces. Notice that we create a regular `JColorChooser` object and never call either the `showDialog()` or `createDialog()` methods. You can also see the piece of code required to catch color updates in that section. We attach a `ChangeListener` to the `ColorSelectionModel` for the chooser. The event handler for that listener simply calls `updatePreviewColor()` to keep our custom previewer in sync with the color shown in the chooser.

You'll notice that we're storing our font information in a `SimpleAttributeSet` object. This object is used with the `JTextPane` class (and you can find out more about it in [Chapter 22](#)). For right now, just know that it has some convenient methods for storing text attributes, such as the font name, bold/italic, and size.

Here's the startup code:

```
// FontChooser.java
```

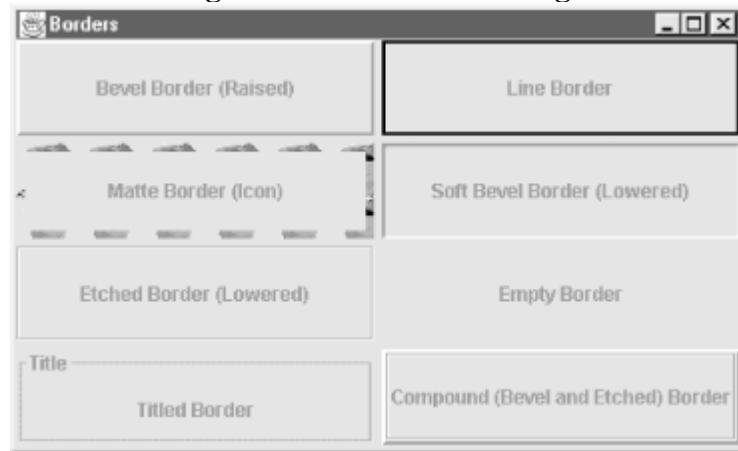
Chapter 13. Borders

Swing provides eight unique styles of borders and allows you to combine them to form more elaborate versions. This chapter introduces you to the Swing borders and shows you how to work with and configure them. At the end of the chapter, we show you how to create a border of your own.

13.1 Introducing Borders

[Figure 13-1](#) shows the standard borders that Swing provides. There are eight border styles: bevel, soft bevel, empty, etched, line, matte, titled, and compound. The MatteBorder gives you two borders in one: the border area can be filled with either a solid color or an icon. (This figure shows only the icon; you can see a better example of both in [Figure 13-11](#).)

Figure 13-1. Borders in Swing



You can place a border around any Swing component that extends JComponent. The JComponent class contains a border property that is inherited by all Swing components. (Top-level components that don't inherit from JComponent, like JFrame and JDialog, can't have borders.) By default, the border property is null (no border), but you can access and modify it. Once you've set a component's border, the component paints itself using that border from that point on, and the insets of the border replace the component's default insets.

Here's how to set a component's border:

```
JLabel label = new JLabel("A Border");  
I1@ve RuBoard
```

13.2 Painting Borders Correctly

The golden rule of creating borders is: "Never paint in the component's region." Here's a border that violates this rule:

```
public class WrongBorder extends AbstractBorder {
```

I1@ve RuBoard

13.3 Swing Borders

The following sections discuss Swing's built-in border classes in detail.

13.3.1 The BevelBorder and SoftBevelBorder Classes

A *bevel* is another name for a slanted edge. The BevelBorder class can be used to simulate a raised or lowered edge with a slant surrounding the component, similar to the appearance of a button. The default bevel edge is two pixels wide on all sides. [Figure 13-4](#) shows two bevel borders, the first raised and the second lowered.

Figure 13-4. Raised and lowered bevel borders



Notice how the border creates the illusion of three dimensions. The bevel border simulates a light source above and to the left of the object (this light source location must be consistent for all 3D components in order to be effective). The border is then drawn with four colors: an outer and inner *highlight* color and an outer and inner *shadow* color. The highlight colors represent the two surfaces of the bevel facing toward the light while the shadow colors represent the surfaces facing away from the light. [Figure 13-5](#) shows how a bevel border uses the highlight and shadow colors.

Figure 13-5. The four colors of a bevel border



When the bevel is raised, the top and left sides of the border are highlighted, and the bottom and right sides of the border are shadowed. This presents the appearance of the surface protruding above the background. When the bevel is lowered, the highlighted and shadowed surfaces are reversed, and the border appears to sink into the background. A bevel border is two pixels wide on all sides. The inner color represents the inner pixels for the border; the outer color represents the outer pixels.

The beveled border in Swing has a subclass, SoftBevelBorder, that can be used to simulate a subtle raised or lowered edge around a component. In fact, the only difference from the regular BevelBorder is that the soft beveled edge is slightly thinner on two of its four sides and provides for small rounded corners. [Figure 13-6](#) shows a pair of soft bevel borders; if your eyes are really good, you may be able to tell the difference between these and the plain bevel borders.

Figure 13-6. Soft bevel borders in Swing



13.3.1.1 Properties

[Table 13-2](#) shows the properties of BevelBorder and SoftBevelBorder. The bevelType property shows whether the

13.4 Creating Your Own Border

Creating your own border is simple when you extend the `AbstractBorder` class. You need to define three things: how to draw the border, whether it is opaque, and what its insets are. To accomplish this, you must implement `paintBorder()`, both `isBorderOpaque()` methods, and `getBorderInsets()`. The hard part of coming up with your own border is doing something creative with the `Graphics` primitives in the `paintBorder()` method. A reminder: make sure that you paint only in the insets region that you define for yourself. Otherwise, you could be painting over the component you intend to border.

Let's take a look at a simple border:

```
// CurvedBorder.java
```

Chapter 14. Menus and Toolbars

This chapter discusses Swing menus and toolbars. Menus are the richer and more flexible of the two, so they encompass most of the chapter. They tend to be the first thing users explore in learning a new application, so it's fitting that Swing provides a great deal of freedom in laying out menu components.

Toolbars allow you to group buttons, combo boxes, and other elements together in repositionable panels; these tools can assist the user in performing many common tasks. You can add any component to a Swing toolbar, even non-Swing components. In addition, Swing allows the toolbar to be dragged from the frame and positioned inside a child window for convenience.

14.1 Introducing Swing Menus

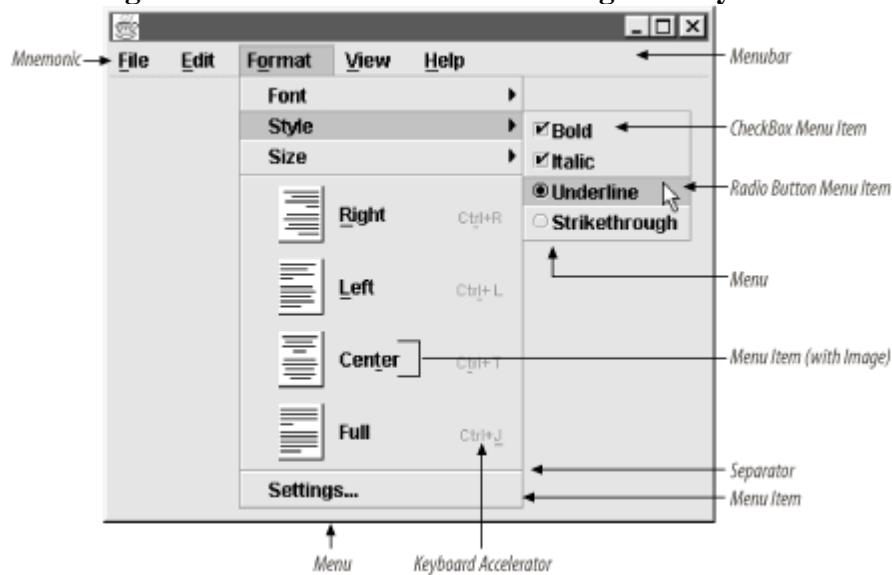
Swing menu components are subclasses of JComponent. Consequently, they have all the benefits of a Swing component, and you can treat them as such with respect to layout managers and containers.

Here are some notable features of the Swing menu system:

- Icons can augment or replace menu items.
- Menu items can be radio buttons.
- Keyboard accelerators can be assigned to menu items; these appear next to the menu item text.
- Most standard Swing components can be used as menu items.

Swing provides familiar menu separators, checkbox menu items, pop-up menus, and submenus for use in your applications. In addition, Swing menus support keyboard accelerators and "underline" style (mnemonic) shortcuts, and you can attach menu bars to the top of Swing frames with a single function that adjusts the frame insets accordingly. On the Macintosh, your application can be configured so that this method places the menu bar at the top of the screen, where users expect to find it. [Figure 14-1](#) defines the various elements that make up the menu system in Swing.

Figure 14-1. The elements of the Swing menu system



Note that not all platforms support underline-style mnemonics. Notably, on the Macintosh (which has never provided this sort of user interface) mnemonics do not appear at all in the system menu bar, and though they are visible in the actual menus, they do not work in either place. If your application uses mnemonics, you should consider grouping the code to set them up into a separate method that is invoked only when running on a platform that supports them.

All platforms do support accelerators (shortcuts) but have different conventions about the key used to invoke them. You can take advantage of the Toolkit method getMenuShortcutKeyMask to always use the right key.

14.2 Menu Bar Selection Models

In all GUI environments, menu components allow only one selection to be made at a time. Swing is no exception. Swing provides a data model that menu bars and menus can use to emulate this behavior: the SingleSelectionModel.

14.2.1 The SingleSelectionModel Interface

Objects implementing the SingleSelectionModel interface do exactly what its name suggests: they maintain an array of possible selections and allow one element in the array to be chosen at a time. The model holds the index of the selected element. If a new element is chosen, the model resets the index representing the chosen element and fires a ChangeEvent to each of the registered listeners.

14.2.1.1 Properties

Objects implementing the SingleSelectionModel interface contain the properties shown in [Table 14-1](#). The selected property is a boolean that tells if there is a selection. The selectedIndex property is an integer index that represents the currently selected item.

Table 14-1. SingleSelectionModel properties

Property	Data type	get	is	set	Default value
selected	boolean				
selectedIndex	int				

14.2.1.2 Events

Objects implementing the SingleSelectionModel interface must fire a ChangeEvent (not a PropertyChangeEvent) when the object modifies its selectedIndex property, i.e., when the selection has changed. The interface contains the standard addChangeListener() and removeChangeListener() methods for maintaining a list of ChangeEvent listeners.

void addChangeListener(ChangeListener listener) void removeChangeListener(ChangeListener listener)

Add or remove the specified ChangeListener from the list of listeners receiving this model's change events.

14.2.1.3 Method

The SingleSelectionModel interface contains one other method:

public void clearSelection()

Clear the selection value, forcing the selected property to return false.

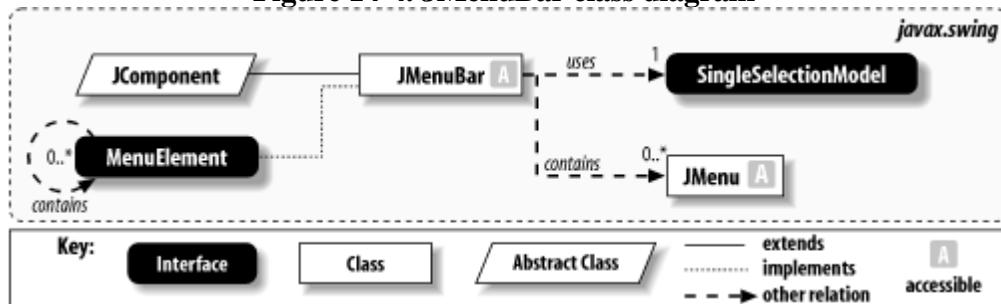
14.2.2 The DefaultSingleSelectionModel Class

Swing provides a simple default implementation of the SingleSelectionModel interface in the

14.3 The JMenuBar Class

Swing's JMenuBar class supersedes the AWT MenuBar class. This class creates a horizontal menu bar component with zero or more menus attached to it. JMenuBar uses the DefaultSingleSelectionModel as its data model because the user can raise, or *activate*, only one of its menus at a given time. Once the mouse pointer leaves that menu, the class removes the menu from the screen (or *cancels* it, in Swing lingo), and all menus again become eligible to be raised. [Figure 14-4](#) shows the class hierarchy for the JMenuBar component.

Figure 14-4. JMenuBar class diagram



You can add JMenu objects to the menu bar with the `add()` method of the JMenuBar class. JMenuBar then assigns an integer index based on the order in which the menus were added. The menu bar displays the menus from left to right on the bar according to their assigned index. In theory, there is one exception: the help menu. You are supposed to be allowed to mark one menu as the help menu; the location of the help menu is up to the L&F. In practice, trying to do this results in JMenuBar throwing an Error.

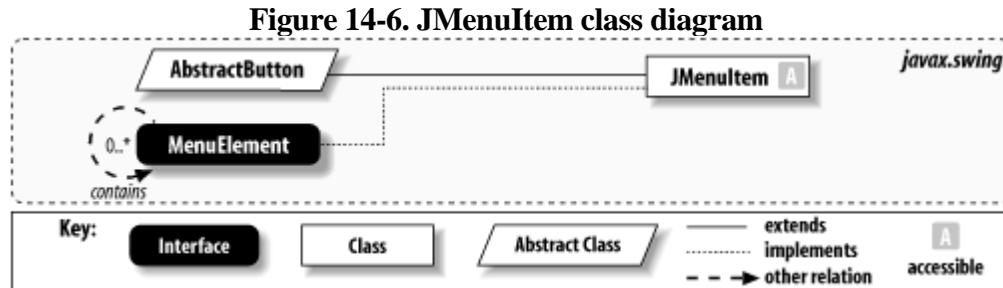
14.3.1 Menu Bar Placement

You can attach menu bars to Swing frames or applets in one of two ways. First, you can use the `setJMenuBar()` method of JFrame, JDialog, JApplet, or JInternalFrame:

```
JFrame frame = new JFrame( "Menu" );
```


14.4 The JMenuItem Class

Before discussing menus, we should introduce the JMenuItem class. [Figure 14-6](#) shows the class diagram for the JMenuItem component.

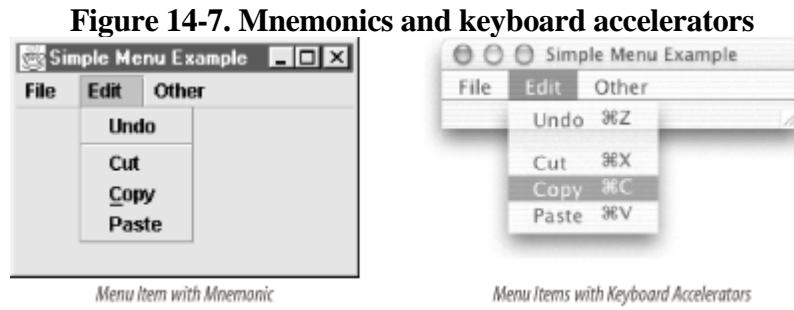


A JMenuItem serves as a wrapper for strings and images to be used as elements in a menu. The JMenuItem class is essentially a specialized button and extends the AbstractButton class. Its behavior, however, is somewhat different from standalone buttons. When the mouse pointer is dragged over a menu item, Swing considers the menu item to be *selected*. If the user releases the mouse button while over the menu item, it is considered to be chosen and should perform its action.

There is an unfortunate conflict in terminology here. Swing considers a menu item selected when the mouse moves over it, as updated by the MenuSelectionManager and classes that implement the MenuElement interface. On the other hand, Swing considers a button selected when it remains in one of two persistent states, such as a checkbox button remaining in the checked state until clicked again. So when a menu item is selected, its button model is really armed. Conversely, when a menu item is deselected, its button model is disarmed. Finally, when the user releases the mouse button over the menu item, the button is considered to be clicked, and the AbstractButton's doClick() method is invoked.

14.4.1 Menu Item Shortcuts

Menu items can take both keyboard accelerators and (on some platforms) mnemonics. Mnemonics are an artifact of buttons; they appear as a single underline below the character that represents the shortcut. Keyboard accelerators, on the other hand, are inherited from JComponent. With menu items, they have the unique side effect of appearing in the menu item. (Their exact appearance and location is up to the L&F.) [Figure 14-7](#) shows both mnemonics and keyboard accelerators.



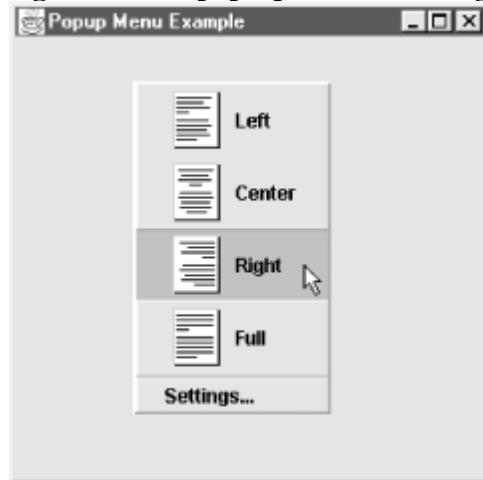
Keyboard accelerators and mnemonics perform the same function: users can abbreviate common GUI actions with keystrokes. However, a mnemonic can be activated only when the button (or menu item) it represents is visible on the screen. Menu item keyboard accelerators can be invoked any time the application has the focus—whether the menu item is visible or not. Also, as noted, accelerators work on all platforms and all L&Fs while mnemonics are less universal. Menus may be assigned both at once.

Let's look at programming both cases. Keyboard accelerators typically use a variety of keystrokes: function keys, command keys, or an alphanumeric key in combination with one or more modifiers (e.g., Shift, Ctrl, or Alt). All of

14.5 The JPopupMenu Class

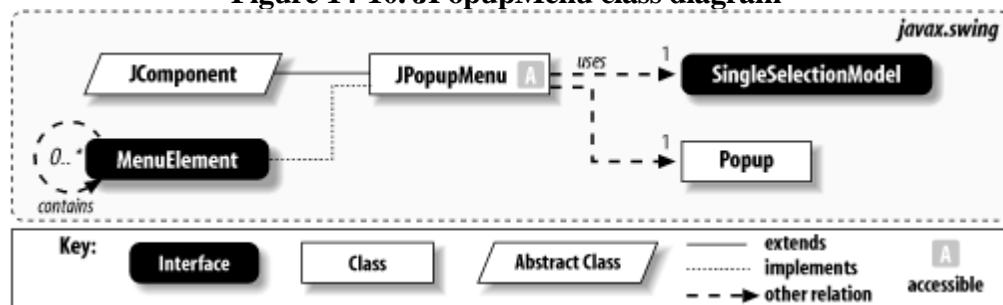
Pop-up menus are an increasingly popular user-interface feature. These menus are not attached to a menu bar; instead, they are free-floating menus that associate themselves with an underlying component. This component is called the *invoker*. Linked to specific interface elements, pop-up menus are nicely context-sensitive. They are brought into existence by a platform-dependent pop-up trigger event that occurs while the mouse is over the invoking component. In AWT and Swing, this trigger is typically a mouse event. Once raised, the user can interact with the menu normally. [Figure 14-9](#) is an example of a pop-up menu in Swing.

Figure 14-9. A pop-up menu in Swing



You can add or insert JMenuItem, Component, or Action objects to the pop-up menu with the add() and insert() methods. The JPopupMenu class assigns an integer index to each menu item and orders them based on the layout manager of the pop-up menu. In addition, you can add separators to the menu by using the addSeparator() method; these separators also count as an index. [Figure 14-10](#) shows the class diagram for the JPopupMenu component. Starting with SDK 1.4, pop-up menus use the Popup class to actually draw themselves. This class is also used for other briefly displayed interface elements like tooltips.

Figure 14-10. JPopupMenu class diagram



14.5.1 Displaying the Pop-up Menu

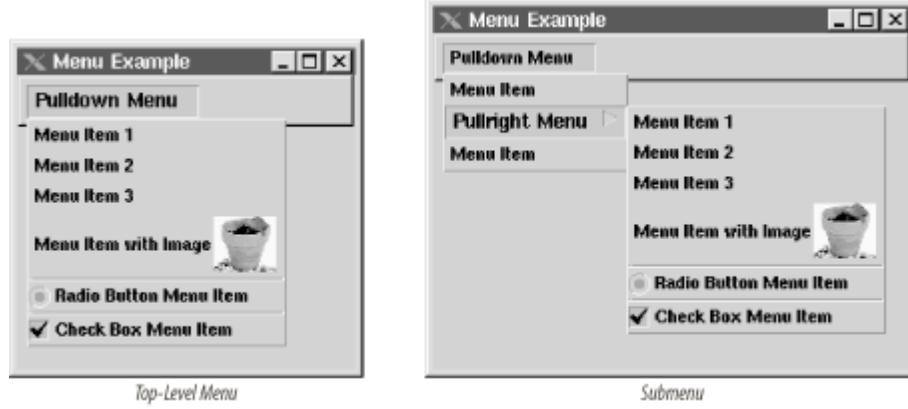
Pop-up menus are usually raised by invoking the show() method in response to a platform-specific pop-up trigger. The show() method sets the location and invoker properties of the menu before making it visible. Pop ups are automatically canceled by a variety of events, including clicking a menu item; resizing an invoking component; or moving, minimizing, maximizing, or closing the parent window. (You won't need to worry about canceling pop-up menus.) You raise the pop-up menu at the right time by checking all your MouseEvents to see if they're the pop-up trigger. A word to the wise: if a MouseEvent is the pop-up trigger, be sure not to pass it on to your superclass, or Swing could cancel the pop-up menu immediately after raising it! Also, be sure to check both pressed and released events because some platforms use one or the other. The easiest way to do that is to check all mouse events. Here's a processMouseEvent() method that raises a pop-up menu upon receiving the appropriate trigger:

```
public void processMouseEvent(MouseEvent e) {
```


14.6 The JMenu Class

The JMenu class represents the anchored menus attached to a JMenuBar or another JMenu. Menus directly attached to a menu bar are called *top-level* menus. Submenus, on the other hand, are not attached to a menu bar but to a menu item that serves as its title. This menu item title is typically marked by a right arrow, indicating that its menu appears alongside the menu item if the user selects it. See [Figure 14-11](#).

Figure 14-11. Top-level menu and submenu



JMenu is a curious class. It contains a MenuUI delegate, but it uses a ButtonModel for its data model. To see why this is the case, it helps to visualize a menu as two components: a menu item and a pop-up menu. The menu item serves as the title. When it is pressed, it signals the pop-up menu to show itself either below or directly to the right of the menu item. JMenu actually extends the JMenuItem class, which makes it possible to implement the title portion of the menu. This, in effect, makes it a specialized button. On some platforms you can use the mnemonic property of the JMenuItem superclass to define a shortcut for the menu's title and, consequently, the menu. In addition, you can use the enabled property of JMenuItem to disable the menu if desired.

As with pop-up menus, you can add or insert JMenuItem, Component, or Action objects in the pop-up portion of the menu by calling the add() and insert() methods. You can also add a simple string to the menu; JMenu creates the corresponding JMenuItem object for you internally. The JMenu class assigns an integer index to each menu item and orders them based on the layout manager used for the menu. You can also add separators to the menu by using the addSeparator() method.

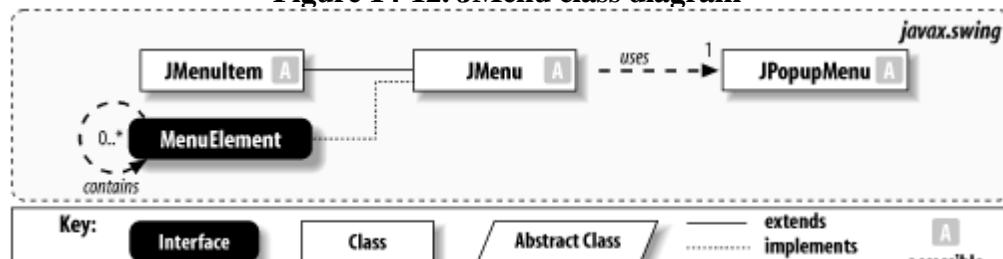


You cannot use keyboard accelerators with JMenu objects (top-level or submenu), because accelerators trigger actual program actions, not simply the display of a menu from which actions can be chosen. On some platforms you can use the setMnemonic() method to set a shortcut to bring up the menu, but the only universal, reliable approach is to assign keyboard accelerators to the non-submenu JMenuItem objects that trigger program actions.

You can programmatically cause the submenu to pop up on the screen by setting the popupMenuVisible property to true. Be aware that the pop up does not appear if the menu's title button is not showing.

[Figure 14-12](#) shows the class diagram for the JMenu component.

Figure 14-12. JMenu class diagram



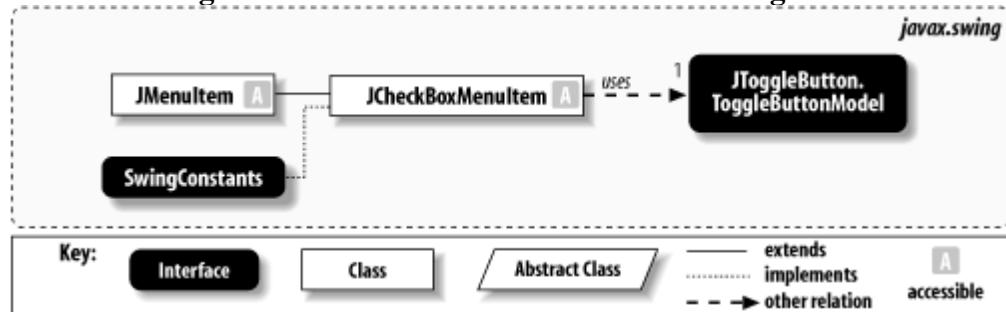
14.7 Selectable Menu Items

So far, we've covered traditional menu items that produce a simple, text-oriented label associated with an action. But that's not the only type of item to which users are accustomed. Swing provides for two selectable menu items: the checkbox menu item and the radio button menu item.

14.7.1 The JCheckBoxMenuItem Class

Checkbox menu items are represented by the `JCheckBoxMenuItem` class. As you might have guessed, this object behaves similarly to the `JCheckBox` object. By clicking on a checkbox menu item, you can toggle a UI-defined checkmark that generally appears to the left of the menu item's label. There is no mutual exclusion between adjoining `JCheckBoxMenuItem` objects—the user can check any item without affecting the state of the others. [Figure 14-14](#) shows the class diagram for the `JCheckBoxMenuItem` component.

Figure 14-14. `JCheckBoxMenuItem` class diagram



14.7.1.1 Properties

[Table 14-9](#) shows the properties of the `JCheckBoxMenuItem` class. `JCheckBoxMenuItem` inherits the `JMenuItem` model (`ButtonModel`) and its accessors. The `JCheckBoxMenuItem` class also contains two additional component properties. The `state` property has the value `true` if the menu item is currently in the checked state, and `false` if it is not. The `selectedObjects` property contains an `Object` array of size one, consisting of the text of the menu item if it is currently in the checked state. If it is not, `getSelectedObjects()` returns `null`. The `getSelectedObjects()` method exists for compatibility with AWT's `ItemSelectable` interface.

Table 14-9. `JCheckBoxMenuItem` properties

Property	Data type	get	is	set	Default value
accessibleContext	AccessibleContext				<code>JCheckBoxMenuItem.AccessibleJCheckBoxMenuItem()</code>
selectedObjects	Object[]				
state	boolean				<code>false</code>
					<code>CheckBoxMenuItem</code>

14.8 Toolbars

Toolbars are another approach to providing access to commonly used application features. They are more likely than menus to use graphical representations of commands. Because they remain on-screen at all times (unlike menus, which drop down only when activated) they can provide a useful "dashboard" for indicating the current state of the application. On the other hand, they take up more room than menu bars, so it's good to let the user decide whether they should be visible at all.

Toolbars have the ability to "tear" themselves from their location within a frame and embed their components in a moveable standalone window. This gives the user the freedom to drag the toolbar anywhere on the screen. In addition, toolbars can "dock" in locations where the layout manager can support them.

14.8.1 The JToolBar Class

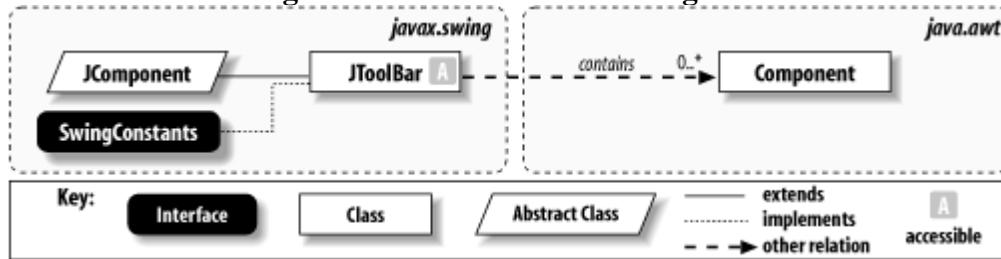
Like the menu bar, the JToolBar class is a container for various components. You can add any component to the toolbar, including buttons, combo boxes, and even additional menus. Like menus, the toolbar is easiest to work with when paired with Action objects.

When a component is added to the toolbar, it is assigned an integer index that determines its display order from left to right. While there is no restriction on the type of component that can be added, the toolbar generally looks best if it uses components that are the same vertical height. Note that toolbars have a default border installed by the L&F. If you don't like the default, you can override the border with one of your own using the setBorder() method. Alternatively, you can deactivate the drawing of the border by setting the borderPainted property to false.

JToolBar has its own separator that inserts a blank space on the toolbar; you can use the addSeparator() method to access this separator. Separators are useful if you want to add space between groups of related toolbar components. The separator for toolbars is actually an inner class. Be sure not to confuse this separator with the JSeparator class.

[Figure 14-20](#) shows the class diagram for the JToolBar component.

Figure 14-20. JToolBar class diagram



14.8.1.1 Floating toolbars

Although toolbars can be easily positioned in Swing containers, they do not have to stay there. Instead, you can "float" the toolbar by holding the mouse button down while the cursor is over an empty section of the toolbar (that is, not over any of its components) and dragging. This places the toolbar in a moveable child window; you can position it anywhere in the viewing area. Toolbars can then reattach themselves to specific locations, or *hotspots*, within the frame. Letting go of the toolbar while dragging it over a hotspot anchors the toolbar back into the container. [Figure 14-21](#) is an example of a floating toolbar.

Figure 14-21. A floating toolbar

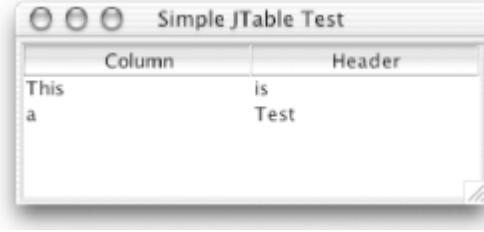


Chapter 15. Tables

Tables represent one of the most common formats for viewing data. Database records are easy to sort and choose from a table. Statistics on disk usage can be displayed for several computers or several time periods all at once. Stock market quotes can be tracked. And where would sales presentations be without tables? Well, the JTable class in the Swing package now gives you access to a single component that can handle all of the preceding examples and more.

Without getting fancy, you can think of tables as an obvious expression of two-dimensional data. In fact, the JTable class has a constructor that takes an Object[][] argument and displays the contents of that two-dimensional array as a table with rows and columns. For example, [Figure 15-1](#) shows how a table of string objects falls out very quickly.

Figure 15-1. A simple JTable with a two-dimensional array of strings for data



This program was generated with very little code. All we did was set up a JTable object with a String[][] argument for the table data and a String[] argument for the table's headers. Rather than adding the table itself directly to our window, we enclose it in a scrollpane:

```
// SimpleTable.java
```


15.1 The JTable Class

Before we get ahead of ourselves, let's look at the `JTable` class and its supporting cast members.

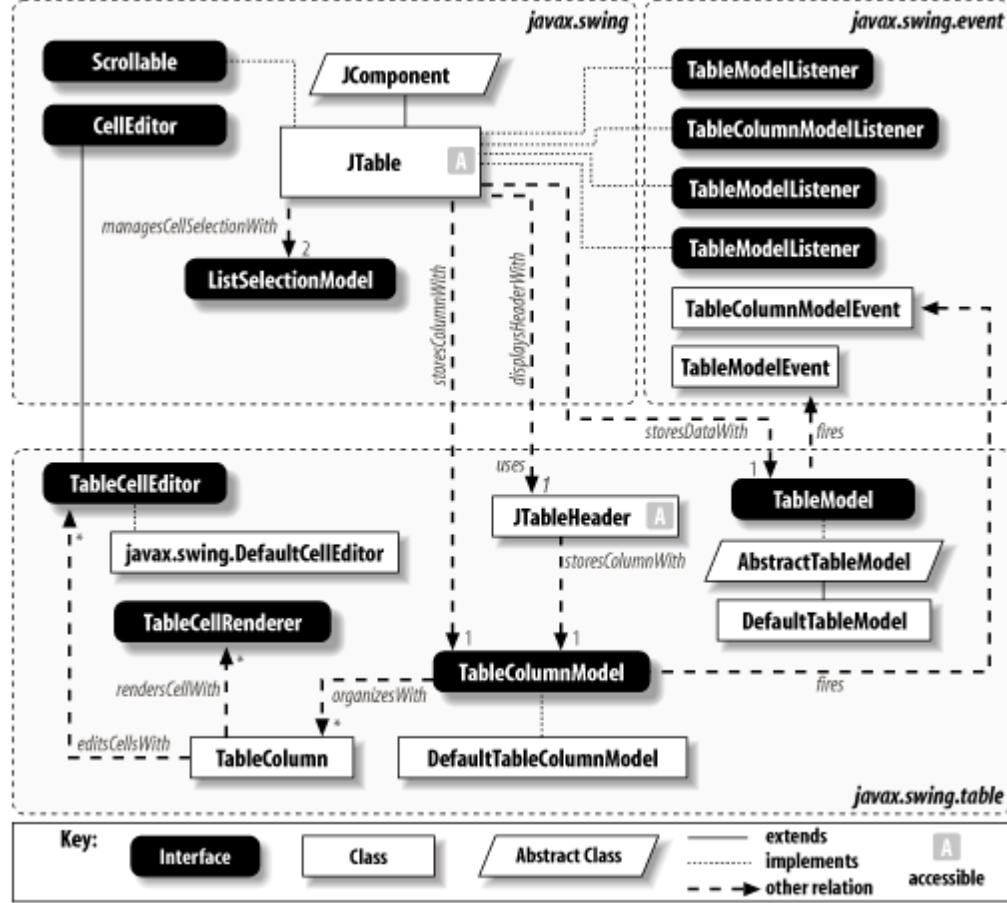
15.1.1 Table Columns

With Swing tables, the basic unit is not an individual cell but a column. Most columns in real-world tables represent a certain type of information that is consistent for all records. For example, a record containing a person's name is a String and might be the first column of the table. For every other record (row), the first cell is always a String. The columns do not need to all have the same data type. The same record could hold not only a person's name, but whether or not they owned a computer. That column would hold Boolean values, not String values. The models supporting JTable reflect this view of the world. There is a TableModel that handles the contents of each cell in the table. You will also find a TableColumnModel that tracks the state of the columns in the table (how many columns, the total width, whether or not you can select columns, etc.).

The ability to store different types of data also affects how the table draws the data. The table column that maps to the "owns a computer" field could use a JCheckBox object for the cells of this column while using regular JLabel objects for the cells of other columns. But again, each column has one data type and one class responsible for drawing it.

Now, as the `JTable` class evolves, you may find alternate ways to think about tables without relying so heavily on columns. You'll want to keep an eye on the API in future releases of the Swing package. [Figure 15-2](#) shows how the classes of the `JTable` package fit together.

Figure 15-2. JTable class diagram



Well, we made it this far without officially discussing the `JTable` class itself. Dynamic data and database queries are handled entirely by the table model underneath the display. So what can you do with a `JTable` object? The `JTable` class gives you control over the appearance and behavior of the table. You can control the spacing of columns, their

15.2 Implementing a Column Model

Here's a custom column model that keeps all of its columns in alphabetical order as they are added:

// SortingColumnModel.java

I1@ve RuBoard

15.3 Table Data

We've seen the TableColumnModel, which stores a lot of information about the structure of a table but doesn't contain the actual data. The data that's displayed in a JTable is stored in a TableModel. The TableModel interface describes the minimum requirements for a model that supplies the information necessary to display and edit a table's cells and to show column headers. The AbstractTableModel fills out most of the TableModel interface, but leaves the methods for retrieving the actual data undefined. The DefaultTableModel extends AbstractTableModel and provides an implementation for storing data as a vector of vectors. We'll look at both the abstract and default table models in more detail later in this chapter.

15.3.1 The TableModel Interface

All of the table models start with this interface. A table model must be able to give out information on the number of rows and columns in the table and have access to the values of the cells of the table. The TableModel interface also has methods that can be used to encode information about the columns of the table (such as a localized name or class type) separate from the column model.

15.3.1.1 Properties

The TableModel interface supports the properties shown in [Table 15-9](#). The columnCount is the number of columns in the data model. This does not have to match the number of columns reported by the column model. Likewise, rowCount is the number of rows in the data model. columnName and columnClass are indexed properties that let you retrieve the name of the column and the class of objects in the column. The name used in the table model is distinct from anything used in the TableColumn class. For both properties, remember that the index refers to the table model, regardless of where the column appears on the screen.

Table 15-9. TableModel properties

Property	Data type	get	is	set	Default value
columnCount	int				
rowCount	int				

15.3.1.2 Events

As you may have come to expect from other models in the Swing package, the TableModel has its own event type, TableModelEvent, generated whenever the table changes. A full discussion of the TableModelEvent class and the TableModelListener appears later in this chapter.

```
public void addTableModelListener(TableModelListener l) public void  
removeTableModelListener(TableModelListener l)
```

Add or remove listeners interested in receiving table model events.

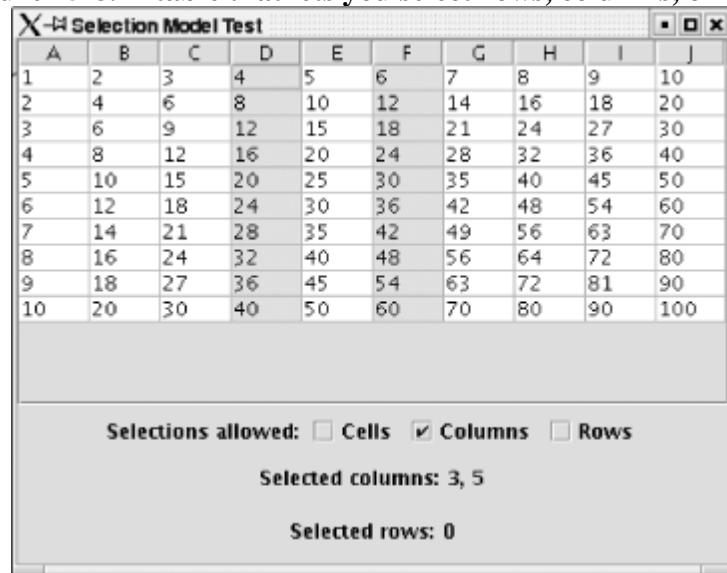
15.3.1.3 Cell methods

15.4 Selecting Table Entries

All this, and all we can do is render and edit data in a table. "What about selecting data?" you ask. Yes, we can do that, too. And, as you might expect, the ListSelectionModel (discussed in [Chapter 7](#)) drives us through these selections. Unlike most of the other components, however, the two-dimensional JTable has two selection models, one for rows and one for columns.

[Figure 15-8](#) shows an application that allows you to turn on and off the various selections allowed on a table (cell, row, and column). As you select different rows and columns, two status labels show you the indices of the selected items.

Figure 15-8. A table that lets you select rows, columns, or cells



Let's look at the code for this example. Most of the work is getting the interface you see running. Once that's done, we attach our two reporting labels as listeners to the row selection and column selection models. The interesting part of the code is the ListSelectionListener, written as an inner class. This class tracks any ListSelectionModel and updates a label with the currently selected indices every time it changes. (Those indices are retrieved using the getSelectedIndices() method we wrote ourselves.) Since we rely on only the list selection model, we can use the same event handler for both the row and the column selections.

```
// SelectionExample.java
```


15.5 Rendering Cells

You can build your own renderers for the cells in your table. By default, you get renderers for Boolean types (JCheckBox for display and editing), ImageIcon types, Number types (right-justified JTextField), and Object types (JTextField). However, you can specify a particular renderer for a class type or for a particular column, or even for a particular cell.

15.5.1 The TableCellRenderer Interface

This interface provides access to a rendering component without defining what the component does. This works because a renderer functions by rubber-stamping a component's image in a given location. The only method this interface defines initializes and returns just such a component:

```
public abstract Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected,
boolean hasFocus, int row, int column)
```

This model takes a value, which can also be retrieved by getting the cell at row, column of table, and returns a component capable of drawing the value in a table cell (or anywhere, really). The resulting drawing can be affected by the selection state of the object and whether it currently has the keyboard focus.

15.5.2 The DefaultTableCellRenderer Class

The javax.swing.table package includes a default renderer that produces a JLabel to display text for each cell in the table. The JTable class uses this renderer to display Numbers, Icons, and Objects. JTable creates a new default renderer and then aligns it correctly and attaches an appropriate icon, depending on the type of data. Object objects are converted to strings using `toString()` and are shown as regular labels. Number objects are shown right-aligned, and Icons are shown using centered labels. Boolean values do not use DefaultTableCellRenderer; instead, they use a private renderer class that extends JCheckBox. Go back and take a look at [Figure 15-5](#) for an example of how this renderer works on different types of data.

15.5.2.1 Properties

The DefaultTableCellRenderer modifies three properties of the JLabel class, as shown in [Table 15-15](#). The color values are used as the "unselected" foreground and background colors for text. You might recall that the selected foreground and background colors are governed by the JTable class. If you set either of these properties to null, the foreground and background colors from JTable are used.

Table 15-15. DefaultTableCellRenderer properties

Property	Data type	get	is	set	Default value
background	Color				null
foreground	Color				null
opaque	boolean				true

15.6 Editing Cells

In addition to custom renderers, you can also create custom editors for your table cells. (Actually, the basic stuff in this section also applies to the JTree class.) You have several options ranging from straightforward to completely homegrown.

15.6.1 The CellEditor Interface

This interface governs the basic functionality required of an editor. It has methods for retrieving a new value and determining when to start and stop editing. The basic process for editing is:

- The user clicks the required number of times on the cell (varies from editor to editor).
- The component (usually JTree or JTable) replaces the cell with its editor.
- The user types or chooses a new value.
- The user ends the editing session (e.g., pressing Enter in a text field).
- The editor fires a change event to interested listeners (usually the tree or table containing the cell), stating that editing is finished.
- The component reads the new value and replaces the editor with the cell's renderer.

15.6.1.1 Events

The CellEditor interface requires methods for adding and removing cell editor listeners, which are objects interested in finding out whether editing is finished or canceled. The CellEditorListener class is discussed later in the chapter.

```
public abstract void addCellEditorListener(CellEditorListener l)
public abstract void removeCellEditorListener(CellEditorListener l)
```

15.6.1.2 Methods

```
public Object getCellEditorValue()
```

Access the only property of a cell editor, which is the cell's current value. After successful editing, a table or tree calls this method to retrieve the new value for the cell.

```
public abstract boolean isCellEditable(EventObject anEvent)
```

Should return true if anEvent is a valid trigger for starting this kind of editor. For example, if you want the user to double-click on a field to invoke the editor, this method would test whether anEvent is a double-click mouse event. If it was only a single-click, you could return false. If it was a double-click, you could return true.

```
public abstract boolean shouldSelectCell(EventObject anEvent)
```

This method should return true if the cell to be edited should also be selected. While you usually want to select the cell, there are some situations in which not selecting the cell is preferable. For example, you might be implementing a table that lets the user edit cells that are part of an ongoing selection. Since you want the selection to remain in place,

15.7 Next Steps

There are many other things you can do with JTable and its various supporting models. While we don't have time or space to present all of them, we will take a look at a few more interesting examples of JTable features in [Chapter 16](#).

Chapter 16. Advanced Table Examples

In this chapter, we're going to take a different approach. Tables are extremely flexible, useful gadgets. Here, we're going to show you how to put tables to work in more advanced situations. Most of these examples require working on the TableModel itself or the TableColumnModel. But once you know what you're doing, subclassing these models is fairly easy and gives you a lot of flexibility.

We will look at four examples:

- A scrollable table with row headers. Remember that a JTable understands column headers but doesn't have any concept of a row header. Also, remember that a JScrollPane understands both column and row headers. In this example, we'll show you how to add row headers to a JTable and make them work properly within a JScrollPane.
- A table that has an extremely large number of rows. Scrolling stops working well when you have more than a few hundred rows. We'll build a table with 10,000 rows, let you page up and down to select a range of 100 rows within the table, and then scroll back and forth within that more limited range.
- A table with a custom editor and renderer for working with cells that contain something other than just text. We'll represent a numeric value with a slider. The user can also move the slider to edit the value.
- A TableChart component that builds pie charts based on the TableModel class used by JTable. In this example, the JTable is almost superfluous, although it provides a convenient way to edit the data in the pie chart. The real point is that the TableModel is a powerful abstraction that can be put to use even when there's no table around.

16.1 A Table with Row Headers

As we promised, this is a table with headers for both rows and columns. The JTable handles the column headers itself; we need to add machinery for the rows. [Figure 16-1](#) shows the resulting table. It shows column labels, plus two data columns from a larger table. Scrolling works the way you would expect. When you scroll vertically, the row headers scroll with the data. You can scroll horizontally to see other data columns, but the row headers remain on the screen.

Figure 16-1. A table with both row and column headers

Row #	Column 2	Column 3	Colu
509	09	c509	d509
510	10	c510	d510
511	11	c511	d511
512	12	c512	d512
513	13	c513	d513
514	14	c514	d514
515	15	c515	d515
516	16	c516	d516
517	17	c517	d517
...

The trick is that we really have two closely coordinated tables: one for the row headers (a table with only one column) and one for the data columns. There is a single TableModel, but separate TableColumnModels for the two parts of the larger table. In the figure, the gray column on the left is the row header; it's really column 0 of the data model.

To understand what's going on, it helps to remember how a Swing table models data. The TableModel itself keeps track of all the data for the table, i.e., the values that fill in the cells. There's no reason why we can't have two tables that share the same table model—that's one of the advantages of the model-view-controller architecture. Likewise, there's no reason why we can't have data in the table model that isn't displayed; the table model can keep track of a logical table that is much larger than the table we actually put on the screen. This is particularly important in the last example, but it's also important here. The table that implements the row headers uses the first column of the data model and ignores everything else; the table that displays the data ignores the first column.

The TableColumnModel keeps track of the columns and is called whenever we add, delete, or move a column. One way to implement tables that use or ignore parts of our data is to build table column models that do what we want, which is add only a particular column (or group of columns) to the table. That's the approach we've chosen. Once we have our models, it is relatively simple to create two JTables that use the same TableModel, but different TableColumnModels. Each table displays only the columns that its column model allows. When we put the tables next to each other, one serves as the row header, and the other displays the body. Depending on the data you put in your tables, you could probably automate many of these steps. Here is the code for our not-so-simple table:

```
// RowHeaderTable.java
```

16.2 Large Tables with Paging

Working conveniently with very large tables can be a pain. Scrolling up and down is fine as long as the table is only a few hundred lines long, but when it gets larger, a tiny movement in the scrollbar can change your position by a few thousand rows. One way to solve this is by combining paging with scrolling. We'll create a table with 10,000 rows (large enough to make scrolling through the entire table a hassle) and add buttons to page up and down 100 rows at a time. Within any group of 100 rows, you can use the scrollbar as usual to move around. [Figure 16-2](#) shows the result.

Figure 16-2. A paging (and scrolling) table

Record Number	Batch Number	Reserved
716	893336849410	Reserved
717	893336849410	Reserved
718	893336849410	Reserved
719	893336849410	Reserved
720	893336849410	Reserved
721	893336849410	Reserved
722	893336849410	Reserved
723	893336849410	Reserved

In this example, we're using a simple trick. There are really two tables to worry about: a logical table that contains all 10,000 rows, which might represent records that were read from a database, and the physical table that's instantiated as a JTable object and displayed on the screen. To do this trick, we implement a new table model, PagingModel, which is a subclass of AbstractTableModel. This table model keeps track of the data for the entire logical table: all 10,000 rows. However, when a JTable asks it for data to display, it pretends it knows only about the 100 rows that should be on the screen at this time. It's actually quite simple. (And we don't even need to worry about any column models; the default column model is adequate.)

Here's the PagingModel code used to track the 10,000 records:

```
// PagingModel.java
```

16.3 A Table with Custom Editing and Rendering

Recall from the previous chapter that you can build your own editors and renderers for the cells in your table. One easy customization is altering the properties of the DefaultTableCellRenderer, which is an extension of JLabel. You can use icons, colors, text alignment, borders, and anything that can change the look of a label.

You don't have to rely on JLabel, though. Developers come across all types of data. Some of that data is best represented as text—some isn't. For data that requires (or at least enjoys the support of) alternate representations, your renderer can extend any component. To be more precise, it can extend Component, so your options are boundless. We look at one of those options next.

16.3.1 A Custom Renderer

[Figure 16-3](#) shows a table containing audio tracks in a mixer format, using the default renderer. We have some track information, such as the track name, its start and stop times, and two volumes (left and right channels, both using integer values from 0 to 100) to control.

Figure 16-3. A standard table with cells drawn by the DefaultTableCellRenderer

Track	Start	Stop	Left Volume	Right Volume
Bass	0:00:000	1:00:000	56	56
Strings	0:00:000	0:52:010	72	52
Brass	0:08:000	1:00:000	99	0
Wind	0:08:000	1:00:000	0	99

Try the sliders!

We'd really like to show our volume entries as sliders. The sliders give us a better indication of the relative volumes. [Figure 16-4](#) shows the application with a custom renderer for the volumes.

Figure 16-4. A standard table with the volume cells drawn by VolumeRenderer

Track	Start	Stop	Left Volume	Right Volume
Bass	0:00:000	1:00:000		
Strings	0:00:000	0:52:010		
Brass	0:08:000	1:00:000		
Wind	0:08:000	1:00:000		

Try the sliders!

The code for this example involves two new pieces and a table model for our audio columns. First, we must create a new renderer by implementing the TableCellRenderer interface ourselves. Then, in the application code, we attach our new renderer to our volume columns. The model code looks similar to the models we have built before. The only real difference is that we now have two columns using a custom Volume class. The following Volume class encodes an integer. The interesting thing about this class is the setVolume() method, which can parse a String, Number, or other Volume object.

```
// Volume.java
```


16.4 Charting Data with a TableModel

Our last example shows that the table machinery isn't just for building tables; you can use it to build other kinds of components (like the pie chart in [Figure 16-6](#)). If you think about it, there's no essential difference between a pie chart, a bar chart, and many other kinds of data displays; they are all different ways of rendering data that's logically kept in a table. When that's the case, it is easy to use a TableModel to manage the data and build your own component for the display.

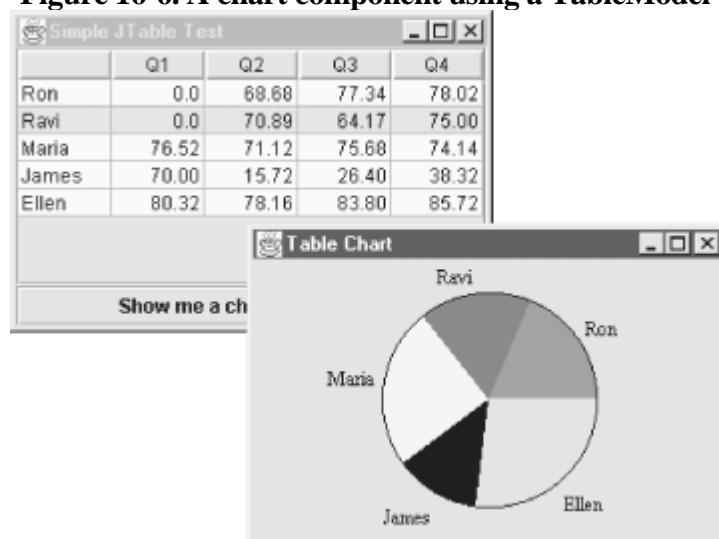
With AWT, building a new component was straightforward: you simply created a subclass of Component. With Swing, it's a little more complex because of the distinction between the component itself and the user-interface implementation. But it's not terribly hard, particularly if you don't want to brave the waters of the Pluggable L&F. In this case, there's no good reason to make pie charts that look different on different platforms, so we'll opt for simplicity. We'll call our new component a TableChart; it extends JComponent. Its big responsibility is keeping the data for the component updated; to this end, it listens for TableModelEvents from the TableModel to determine when changes have been made.

To do the actual drawing, TableChart relies on a delegate, PieChartPainter. To keep things flexible, PieChartPainter is a subclass of ChartPainter, which gives us the option of building other kinds of chart painters (bar chart painters, etc.) in the future. ChartPainter extends ComponentUI, which is the base class for user interface delegates. Here's where the model-view-controller architecture comes into play. The table model contains the actual data, TableChart is a controller that tells a delegate what and when to paint, and PieChartPainter is the view that paints a particular kind of representation on the screen.

Just to prove that the same TableModel can be used with any kind of display, we also display an old-fashioned JTable using the same data—which turns out to be convenient because we can use the JTable's built-in editing capabilities to modify the data. If you change any field (including the name), the pie chart immediately changes to reflect the new data.

The TableChart class is particularly interesting because it shows the "other side" of table model event processing. In the PagingModel of the earlier example, we had to generate events as the data changed. Here, you see how those events might be handled. The TableChart has to register itself as a TableModelListener and respond to events so that it can redraw itself when you edit the table. The TableChart also implements one (perhaps unsightly) shortcut: it presents the data by summing and averaging along the columns. It would have been more work (but not much more) to present the data in any particular column, letting the user choose the column to be displayed. (See [Figure 16-6](#).)

Figure 16-6. A chart component using a TableModel



Here's the application that produces both the pie chart and the table. It includes the TableModel as an anonymous inner class. This inner class is very simple, much simpler than the models we used earlier in this chapter; it provides an array for storing the data, methods to get and set the data, and methods to provide other information about the table.

Chapter 17. Trees

One crucial component that found its way into the Swing set is the tree. Tree components help you visualize hierarchical information and make traversal and manipulation of that information much more manageable. A tree consists of nodes, which can contain either a user-defined object along with references to other nodes, or a user-defined object only. (Nodes with no references to other nodes are commonly called leaves.) In modern windowing environments, the directory list is an excellent example of a tree. The top of the component is the root directory or drive, and under that is a list of subdirectories. If the subdirectories contain further subdirectories, you can look at those as well. The actual files found in any directory in this component are the leaves of the tree.

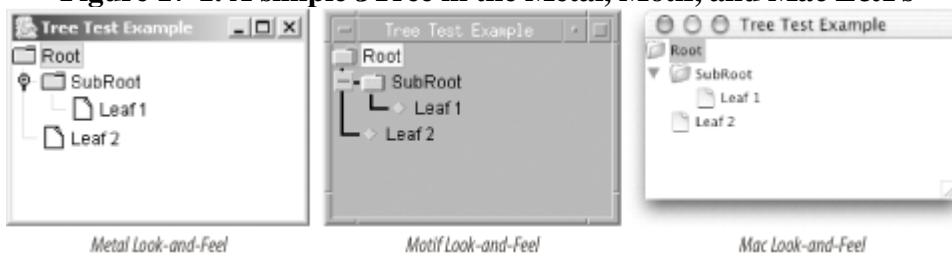
Any data with parent-child relationships can be displayed as a tree. Another common example is an organizational chart. In such a chart, every management position is a node, with child nodes representing the employees under the manager. The organizational chart's leaves are the employees who are not in management positions, and its root is the president or CEO. Of course, real organizations don't always adhere to a strict tree structure. In a tree, each node has exactly one parent node, with the exception of the root node, which cannot have a parent (so trees are not cyclic). This means—in the world of trees—that two managers cannot manage the same employee.

In short, whenever you have a clearly defined hierarchy, you can express that hierarchy as a tree. Swing implements trees with the JTree class and its related models. With trees, as with tables, it's particularly important to understand the models. The JTree itself merely coordinates the tree's display.

17.1 A Simple Tree

Before we look at the models supporting the JTree class, let's look at a very simple example of a tree built with some of the various L&Fs ([Figure 17-1](#)). The javax.swing.DefaultMutableTreeNode class serves as our node class. You don't have to worry about specifically making a node a leaf. If the node has no references to other nodes by the time you display it, it's a leaf.

Figure 17-1. A simple JTree in the Metal, Motif, and Mac L&Fs



This example works by building up a series of unconnected nodes (using the DefaultMutableTreeNode class) and then connecting them. As long as we stick to the default classes provided with the tree package, we can build a regular model out of our nodes quite quickly. In this example, we build the model based on an empty root node, and then populate the tree by attaching the other nodes to the root or to each other. You can also build the tree first, and then create the model from the root node. Both methods have the same result. With a valid tree model in place, we can make a real JTree object and display it.

```
// TestTree.java
```


17.2 Tree Models

Looking at [Figure 17-4](#) you can get an overview of where all the tree pieces come from. As with many of the Swing components you've seen already, the models supporting the data for trees play a crucial role in making the component run. Two interfaces are particularly important: TreeModel, which describes how to work with tree data, and TreeSelectionModel, which describes how to select nodes.

17.2.1 The TreeModel Interface

To get started, you need a tree model. The TreeModel interface is the starting point for your model. You don't have to start from scratch; there is a default implementation (DefaultTreeModel) that you can subclass or just look at for ideas. (We'll look at this class later in the chapter.)

17.2.1.1 Property

The TreeModel has one root property, listed in [Table 17-1](#). This read-only property designates the root of a tree: by definition, the node that has no parent. All other nodes in your tree are descendants of this node.

Table 17-1. TreeModel property

Property	Data type	get	is	set	Default value
root	Object				

17.2.1.2 Events

The tree model uses the TreeModelEvent class defined in the javax.swing.event package. A TreeModelEvent indicates that the tree changed: one or more nodes were added, modified, or deleted. You will find a more detailed discussion in [Section 17.6](#), later in this chapter.

```
public void addTreeModelListener(TreeModelListener l) public void  
removeTreeModelListener(TreeModelListener l)
```

Add or remove listeners interested in receiving tree model events.

17.2.1.3 Miscellaneous methods

Several miscellaneous methods are defined in this model for querying the tree and its structure. Although the actual data structures are defined by the classes that implement the model, the model works as if every node maintains an array of its children, which in turn can be identified by an index into the array.

```
public Object getChild(Object parent, int index)
```

Given a parent object, return the child node at the given index . In many cases, if you specify an invalid index or try to get a child from a leaf, you should receive an ArrayIndexOutOfBoundsException. Of course, this is user-definable. It would also be possible to create a model that simply returned a null or default child.

```
public int getChildCount(Object parent)
```

Given a parent node, return the number of children this node has. Leaves return a value of 0.

```
public int getChildIndex(Object parent, Object child)
```


17.3 The JTree Class

Now that you've seen all the tree models and some of the default implementations, let's look at the visual representation we can give them. The `JTree` class can build up trees out of several different objects, including a `TreeModel`. `JTree` extends directly from `JComponent` and represents the visual side of any valid tree structure.

As another example of hierarchical data, let's look at a tree that displays XML documents. (We'll leave the details of XML to Brett McLaughlin and his excellent Java and XML book. Of course, as our own Bob Eckstein also wrote the XML Pocket Reference, we'll include a shameless plug for that, too.) Here's an entirely contrived XML document that contains several layers of data:

```
<?xml version="1.0"?>
```


17.4 Tree Nodes and Paths

You probably noticed that the DefaultTreeModel class depends on TreeNode and TreePath objects. In a tree, a TreeNode represents an individual piece of data stored at a particular point in a tree, and a path represents a collection of these pieces that are directly related to each other (in an ancestor/descendant relationship). Let's look at the classes that make up the typical nodes and paths.

17.4.1 The TreeNode Interface

A TreeNode is the basic unit of a tree. This interface defines the minimum properties and access routines a typical tree model expects to see in its nodes.

17.4.1.1 Properties

The TreeNode interface contains the properties listed in [Table 17-6](#). The TreeNode properties are straightforward and deal with the structure of the node. The parent property holds a valid value for every node in a tree, except the root. The childAt property lets you access a particular child in the tree. The childCount property contains the number of children associated with this node, if it allows children. If the node does not allow children, it is probably a leaf, but it is also possible to have a mutable tree node that has no children, or does not allow children, and yet is not a leaf. (An empty directory with no write permissions would be an example of such a node.)

Table 17-6. TreeNode properties

Property	Data type	get	is	set	Default value
allowsChildren	boolean				
childAt	TreeNode				
childCount	int				
leaf	boolean				
parent	TreeNode				
indexed					

Notice that the children are not properties of a TreeNode. This is not to say that a TreeNode does not have children, but rather the accessor methods do not fit the "property" definition.

17.4.1.2 Child access methods

public int getIndex(TreeNode node) public Enumeration children()

17.5 Tree Selections

After the tree is built and looks the way you want it to, you need to start working with selections so it does something useful. The JTree class introduced many of the selection manipulation methods already, but let's take a closer look at the model for selecting paths in a tree and the DefaultSelectionModel provided in the javax.swing.tree package. If you're comfortable with selection models, you probably won't find anything surprising here and may want to skip to [Section 17.7](#).

Selections are based on rows or paths. It's important to realize the distinction between a "row" and a "path" for trees. A path contains the list of nodes from the root of the tree to another node. Paths exist regardless of whether or not you plan to display the tree.

Rows, however, are completely dependent on the graphical display of a tree. The easiest way to think about a row is to think of the tree as a JList object. Each item in the list is a row on the tree. That row corresponds to some particular path. As you expand and collapse folders, the number of rows associated with the tree changes. It's the RowMapper object's job to relate a row number to the correct path.

Depending on your application, you may find rows or paths more efficient. If your program deals mostly with the user object data, paths are a good choice. If you're working with the graphical interface (automatically expanding folders and the like), rows may be more useful.

17.5.1 The RowMapper Interface

Tree selections make extensive use of the RowMapper interface. (In the absence of a RowMapper, you simply cannot retrieve information on rows in the tree from classes like TreeSelectionModel. TreePath information is still available, and the tree, its model, and the selection model will all continue to work just fine.) It is a simple interface with one method:

```
public int[] getRowsForPaths(TreePath paths[])
```

The UI for your tree should implement this to return a list of row indices matching the supplied paths. If any of the paths are null or not visible, -1 should be placed in the return int array. While this may seem like an obvious task, you must account for the expanded or collapsed state of the nodes in the tree; remember that there's no such thing as a collapsed row. This is one reason why the JTree class cannot simply use a ListSelectionModel.

17.5.2 The TreeSelectionModel Interface

Now for the heart of selections. The TreeSelectionModel interface determines what a tree selection can look like.

17.5.2.1 Properties

TreeSelectionModel contains the properties listed in [Table 17-11](#). The selection model properties deal primarily with the current selection on the tree. The notion of a "lead" selection stems from the fact that a selection can happen as a process, not only as a single event. The lead selection is the most recently added cell in the selection. It might be the only path selected, but it might also be the most recent selection out of several in a range or discontiguous group of selections. If the selection contains more than one path, the `getSelectionPath()` method returns the first selection in the path, which may or may not be the same thing as `getLeadSelectionPath()`. It's also good to remember that, if it has children, selecting a "folder" node in a tree does not imply selecting all the nodes underneath it.

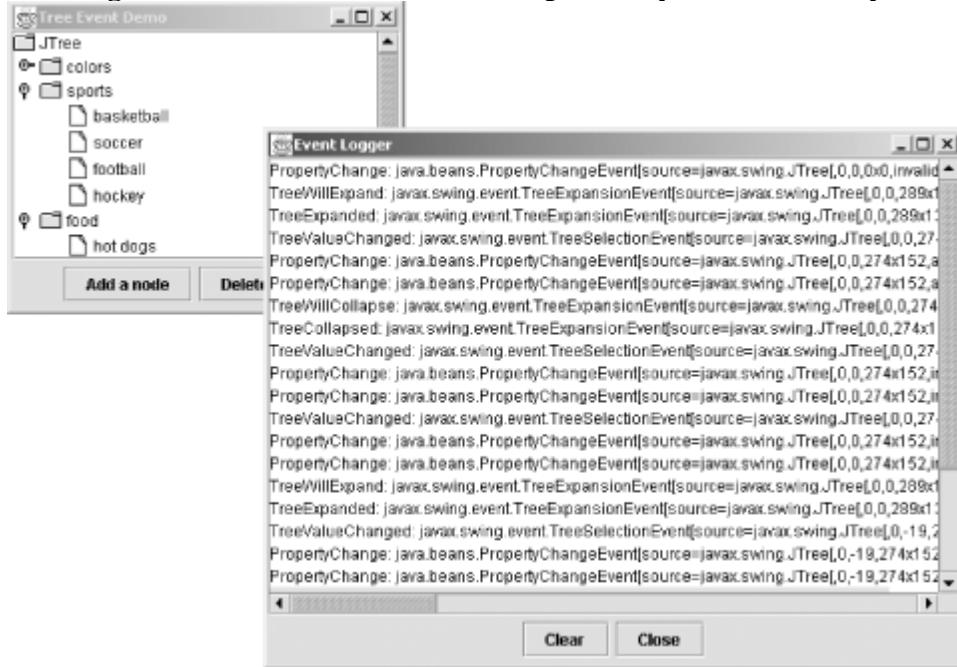
Having said that, the rest of the properties are fairly self-explanatory. `minSelectionRow` and `maxSelectionRow` let you get the smallest and largest selected row numbers. `rowMapper` holds a utility that manages the mapping between paths and row numbers. `selectionPaths` and `selectionRows` let you access the rows or paths currently selected.

17.6 Tree Events

Trees generate three types of events worth mentioning. Apart from the obvious selection events (`TreeSelectionEvent`, `TreeSelectionListener`), you can receive expansion events (`TreeExpansionEvent`, `TreeExpansionListener`, `TreeWillExpandListener`) from the graphical side, and you can catch structural changes to the model itself (`TreeModelEvent`, `TreeModelListener`).

[Figure 17-9](#) shows a simple program that uses the Every Event Listener (EEL) class (described in [Chapter 3](#)) to display all events that come from selecting, expanding, and editing a tree. (We use a tree built by the default `JTree` constructor.) For editing, you can change the text of a node or add and remove nodes so that you can monitor model events.

Figure 17-9. The JTree events as reported by our EEL utility



Here's the source code required to hook up all the various events:

```
// TreeEvents.java
```


17.7 Rendering and Editing

As with the table cells covered in previous chapters, you can create your own tree cell renderers and editors. The default renderers and editors usually do the trick, but you're probably reading this because they don't do the trick for you, so forge onward! If you went through building your own renderers and editors for tables, you'll find this material quite familiar. The tree uses renderers and editors in much the same way that tables do. In fact, you might recall that the DefaultCellEditor class can return both table and tree cell editors.

17.7.1 Rendering Nodes

Why would you want to render a node? Good question. One reason is that you want to modify the L&F of a tree without writing a whole UI package for trees. If you had some special way of presenting the "selected" look, for example, you could write your own tree renderer and still use the default L&F for your other components. You might want to render something other than a string with an icon for the nodes of your tree. Or, as we mentioned above, you might want tooltips that vary based on the particular node you rest your cursor on. "Because I can" is also a good reason.

17.7.1.1 But I just want to change the icons!

Before we tackle creating our own renderers, we should point out that the Metal L&F lets you modify the set of icons used by a tree for the leaves and folders. To change the icons, use the UIManager class and the L&F icons for trees. You can also use the client property JTree.lineStyle to affect the type of lines drawn from folders to leaves. [Chapter 26](#) has much more detail on L&Fs, but this short example should get you started for the tree-specific properties.

Call the `putClientProperty()` method on your instance of the tree to set its line style. Your choices of styles are:

- Horizontal

Thin horizontal lines drawn above each top-level entry in the tree (the default)
Angled

The Windows-style, right-angle lines from a folder to each of its leaves
None

No lines at all

Call the `UIManager.put()` method to modify the icons used by all trees. The icons you can replace are:

- Tree.openIcon

Used for opened folders
Tree.closedIcon

Used for closed folders
Tree.leafIcon

Used for leaves
Tree.expandedIcon

Used for the one-touch expander when its node is expanded
Tree.collapsedIcon

17.8 What Next?

With this control over the look of a tree and its contents, you can create some impressive interfaces for a wide variety of data and your own network management software that lets you browse domains and subdomains and computers and users. Any hierarchy of information you can think of can be shown graphically.

It is worth pointing out, however, that you are not restricted to graphical applications with these models. You can use DefaultTreeModel to store regular, hierarchical data, even if you have no intention of displaying the data on a screen. The model is flexible and provides a good starting point if you don't have any tree data structures of your own lying around.

Chapter 18. Undo

In many applications (word processors, spreadsheets, and board games, to name a few), the user is given the opportunity to undo changes made to the state of the application. In a word processor you can undo deletions. In a chess game, you're often allowed to take back undesirable moves (typically after realizing your queen has just been banished from the board). Without support, providing these undo capabilities can be a lot of work for the programmer, especially if you want to provide a powerful undo system that keeps a history of undoable operations and allows them to be undone and redone indefinitely.

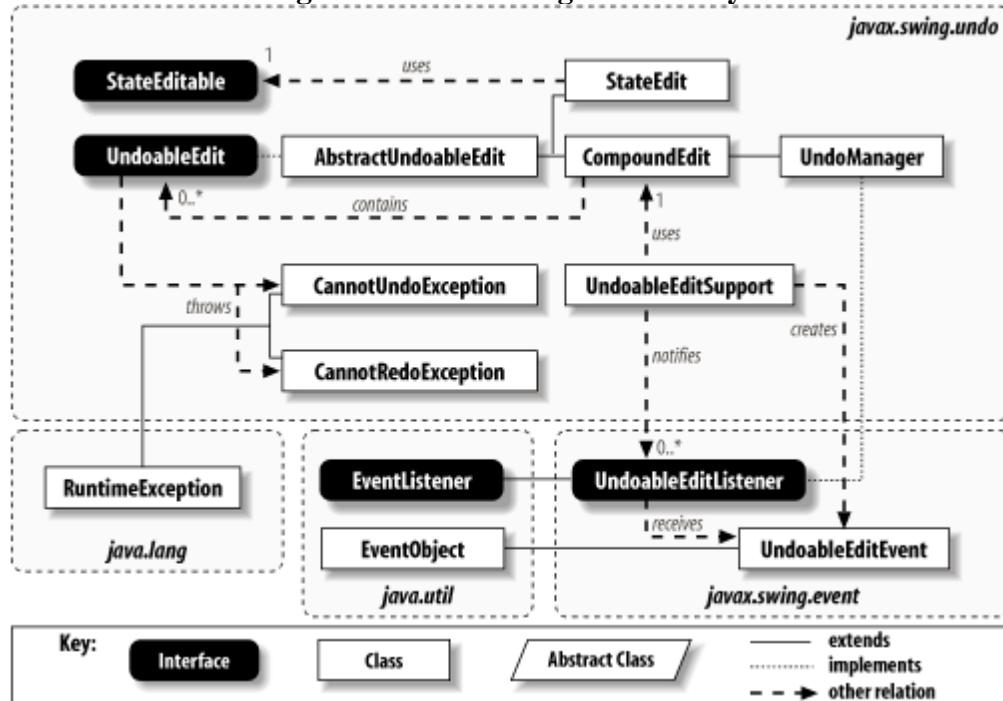
Thankfully, Swing provides a collection of classes and interfaces that support this advanced undo functionality. Within the Swing packages, only the classes in the javax.swing.text package currently use these facilities, but you are free to use undo in any component you create or extend. You can even use it for undoing things that may not be directly associated with a UI component (like a chess move). It's important to realize that the undo facility is not tied in any way to the Swing components themselves. One could easily argue that the package might be more logically called "java.util.undo." None of the classes or interfaces in the javax.swing.undo package use any other Swing object.

In this chapter, we'll look at everything Swing provides to support undo, but we won't get into the details of how the text components use this facility ([Chapter 22](#) does the honors in that department).

18.1 The Swing Undo Facility

The javax.swing.undo package contains two interfaces and seven classes (two of which are exception classes). These, along with a listener interface and event class from the javax.swing.event package, comprise the undo facility shown in [Figure 18-1](#).

Figure 18-1. The Swing undo facility



Here is a brief overview of each class:

UndoableEdit

The base interface for just about everything else in the undo package. It serves as an abstraction for anything in an application that can be undone.

AbstractUndoableEdit

The default implementation of UndoableEdit provides a starting point for building new UndoableEdit classes. It provides a simple set of rules for determining whether undo and redo requests are allowable (based on whether the edit has already been undone, redone, or killed). Despite its name, it is not an abstract class. This is really just a technicality since the default implementation is not at all useful as it is, so you'd never want to instantiate one.

CompoundEdit

This extension of AbstractUndoableEdit allows multiple edits to be grouped together into a single edit. Those familiar with the classic Design Patterns by Erich Gamma et al. (Addison-Wesley) will recognize this construct as a basic implementation of the Composite pattern.

UndoableEditEvent

This event class can be used to notify listeners that an undoable edit has been made.

UndoableEditListener

The listener interface to which UndoableEditEvents are sent. It contains a single method called `undoableEditHappened()`.

UndoManager

An extension of CompoundEdit that can manage a list of edits to be undone or redone in sequence. UndoManager implements UndoableEditListener, so it can be added to many components that generate UndoableEditEvents, allowing it to manage edits from multiple sources in a single undo list.

18.2 The UndoManager Class

UndoManager is an extension of CompoundEdit that can track a history of edits, allowing them to be undone or redone one at a time. Additionally, it implements UndoableEditListener by calling addEdit() each time an UndoableEditEvent is fired. This allows a single UndoManager to be added as a listener to many components that support undo, providing a single place to track all edits and populate an undo menu for the entire application.

It may seem a bit strange that UndoManager extends CompoundEdit. We'll explain why shortly, but first it's important to understand the primary ways in which UndoManager acts differently than CompoundEdit. For starters, when you add an edit to an UndoManager, it is placed in a list of edits available for undo. When you call undo(), only the first (significant) edit is undone. This is different from the behavior of CompoundEdit, in which a call to undo() results in a call to undo() on all of the added edits.

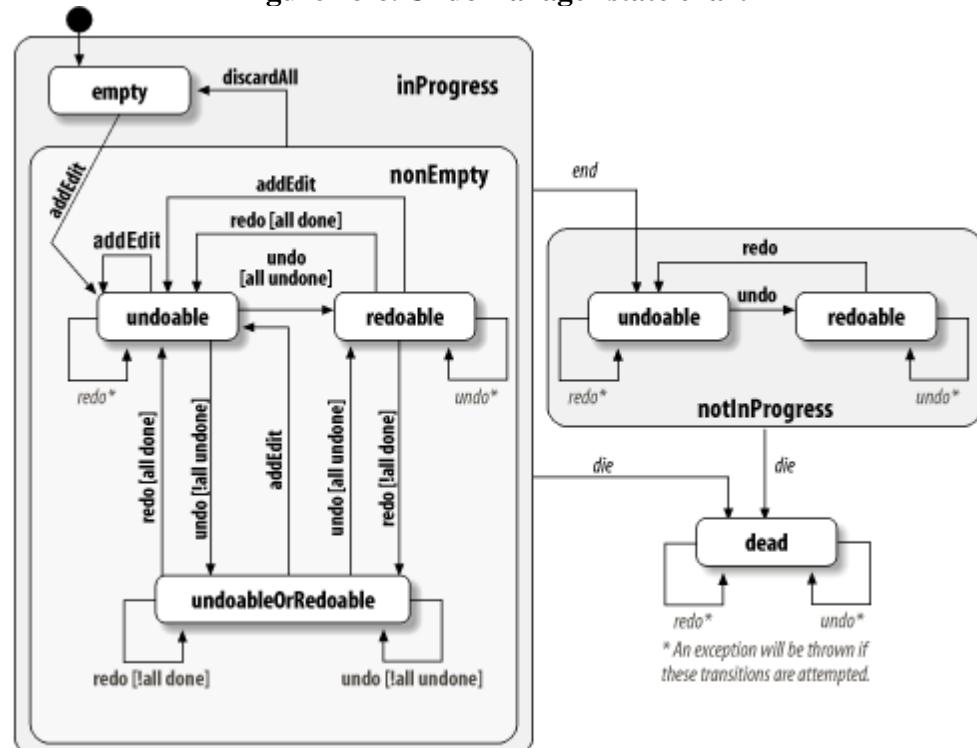
Another major difference between UndoManager and its superclass is the semantics of the inProgress property. In CompoundEdit, we could add new edits only when we were inProgress, and only after calling end() could undo() or redo() be called. In contrast, UndoManager allows undo() and redo() to be called while it is inProgress. Furthermore, when end() is called, it stops supporting sequential undo/redo behavior and starts acting like CompoundEdit (undo() and redo() call their superclass implementations when the UndoManager is not inProgress).

For the strong-hearted,^[3] Figure 18-6 shows a state chart for the UndoManager class.^[4] For several reasons, this chart is considerably more complicated than the ones in the previous sections. First, as mentioned earlier, UndoManager has the curious behavior that once end() is called, it begins to act (for the most part) like a CompoundEdit. This is why we have the transition from the inProgress state to a new superstate (notInProgress, for lack of a better name), the contents of which look just like the CompoundEdit state chart (see Figure 18-4).

[3] All others might want to skip ahead to the description of [Figure 18-7](#).

[4] This chart assumes that all edits are significant. For details on why this is important, see the descriptions of the editToBeUndone() and editToBeRedone() methods later in this section.

Figure 18-6. UndoManager state chart



* An exception will be thrown if these transitions are attempted.

18.3 Extending UndoManager

Now that we've looked at all of the classes and interfaces in the undo framework, we'll look at a few ideas for extending the functionality it provides.

In this example, we'll extend UndoManager to add a few extra features. The first thing we'll add is the ability to get a list of the edits stored in the manager. This is a simple task of returning the contents of the edits vector inherited from CompoundEdit. We also provide access to an array of significant undoable edits and an array of significant redoable edits. These might be useful in a game like chess, in which we want to provide a list of past moves.

The next major feature we add is support for listeners. At this writing, the current UndoManager does not have any way of notifying you when it receives edits. As we saw in an earlier example, this means that you have to listen to each edit-generating component if you want to update the user interface to reflect new undoable or redoable edits as they occur. In our manager, we simply add the ability to add and remove undoable edit listeners to the undo manager itself. Each time an edit is added, the undo manager fires an UndoableEditEvent to any registered listeners. This way, we can just add the undo manager as a listener to each edit-generating component and then add a single listener to the undo manager to update the UI.

The methods of our new undo manager can be divided into two groups, each supporting one of the two features we're adding. We'll split the source code listing along these lines so we can talk about each set of methods independently. Here's the first half:

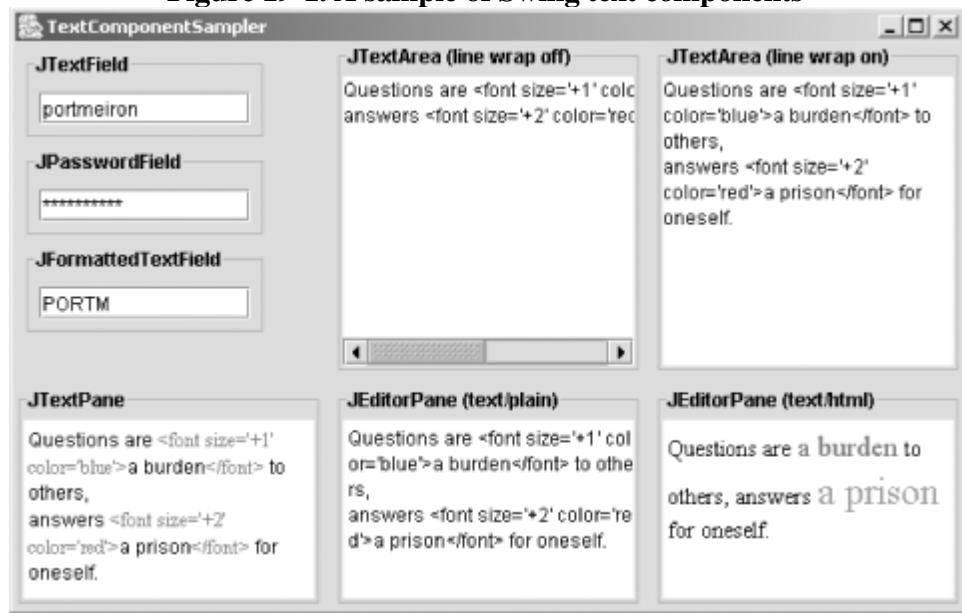
```
// ExtendedUndoManager.java
```

Chapter 19. Text 101

Swing provides an extensive collection of classes for working with text in user interfaces. In fact, because there's so much provided for working with text, Swing's creators placed most of it into its own package: javax.swing.text. This package's dozens of interfaces and classes (plus the six concrete component classes in javax.swing) provide a rich set of text-based models and components complex enough to allow endless customization yet simple to use in the common case.

In this chapter we'll look at JTextField, the base class for all of the text components shown in [Figure 19-1](#), and then discuss JPasswordField, and JTextArea. Then we'll introduce what's going on behind the scenes. We save the more complex model, event, and view classes for later, but we occasionally refer to things you may want to investigate further in the next four chapters.

Figure 19-1. A sample of Swing text components



JFormattedTextField is an extension of JTextField with formatting and object-parsing abilities. We'll devote [Chapter 20](#) to JFormattedTextField and its related classes.

Swing text components allow you to customize certain aspects of the L&F without much work. This includes the creation of custom caret (cursors), custom highlighting, and custom key bindings to associate Actions with special key combinations. These features are covered in [Chapter 21](#).

We describe JTextPane in [Chapter 22](#) and discuss styles, the Document model and Views. Style features include structured text supporting multiple fonts and colors, and even embedded Icons and Components.

Finally, we turn to JEditorPane in [Chapter 23](#) and see how all of this is tied together by something called an EditorKit. EditorKits allow you to define which view objects should be used, which special actions your editor will support, and how your documents can be input and output via streams. You can even register EditorKits for specific content types to enable JEditorPanes to handle those content types automatically.

19.1 The Swing Text Components

Despite all the complexity and power Swing's text components provide, it's still pretty simple to do most things. [Figure 19-1](#) shows each of the six Swing text components, plus an extra JTextArea (to show a different wrapping style) and an extra JEditorPane (to show a different EditorKit).

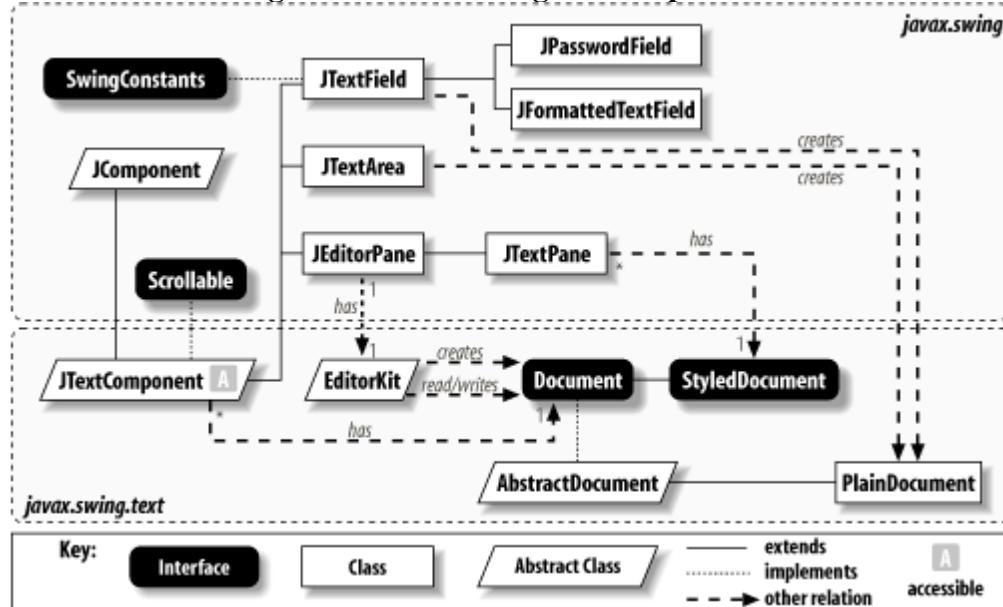
// TextComponentSampler.java

I1@ve RuBoard

19.2 The JTextComponent Class

The six concrete Swing text component classes have quite a bit in common. Consequently, they share a common base class, `JTextComponent`. [Figure 19-2](#) shows the class hierarchy for the Swing text components. As you can see, the concrete text components are in the `javax.swing` package with the rest of the Swing component classes, but `JTextComponent` and all its supporting classes can be found in `javax.swing.text`.

Figure 19-2. The Swing text components



`JTextComponent` is an abstract class that serves as the base class for all text-based Swing components. It defines a large number of properties and methods that apply to its subclasses. In this introductory chapter, we'll pass quickly over many of these properties, as they require an understanding of the underlying model and view aspects of the text framework.

19.2.1 Properties

`JTextComponent` defines the properties and default values shown in [Table 19-1](#). `document` is a reference to the Document model for the component, where the component's content data is stored. (We'll discuss the details of the Document interface in [Chapter 22](#).) The UI property for all text components is a subclass of `javax.swing.plaf.TextUI`.

Table 19-1. `JTextComponent` properties

Property	Data type	get	is	set	Default value
accessibleContext	AccessibleContext				AccessibleJTextC omponent
actions	Action[]				From the UI's EditorKit
caretb	Caret				null
caretColorb	Color				null

19.3 The JTextField Class

JTextField allows the user to enter a single line of text, scrolling the text if its size exceeds the physical size of the field. A JTextField fires an ActionEvent to any registered ActionListeners (including the Action set via the setAction() method, if any) when the user presses the Enter key.

JTextFields (and all JTextComponents) are automatically installed with a number of behaviors appropriate to the L&F, so cut/copy/paste, special cursor movement keys, and text-selection gestures should work without any extra intervention on your part.

The following program presents a JTextField for the user to edit (shown in [Figure 19-3](#)). The JTextField is initially right-justified, but the justification changes each time the Enter key is pressed.

```
// JTextFieldExample.java
```

19.4 A Simple Form

One of the most common user-interface constructs is the basic form. Typically, forms are made up of labels and fields, with the label describing the text to be entered in the field. Here's a primitive TextFormField class that shows the use of mnemonics, tooltips, and basic accessibility support. Note that we call setLabelFor() to associate each label with a text field. This association allows the mnemonics to set the focus and, together with setToolTipText(), supports accessibility (see [Chapter 25](#)).

// TextFormField.java

19.5 The JPasswordField Class

A JPasswordField is a text field in which an echo character (*) by default) is displayed in place of the characters typed by the user. This feature is generally used when entering passwords to avoid showing the password on the screen. Except for the quirky display, JPasswordField behaves like an ordinary JTextField, though some steps have been taken to enhance the password field's security.

One reason that JPasswordField is a separate class from JTextField in Swing (which is not the case in the analogous AWT classes) is so the L&F can treat them differently by specifying different UI delegates. The UI delegate is responsible for hiding the input characters in a JPasswordField.

19.5.1 Properties

[Table 19-5](#) shows the properties defined by JPasswordField. JPasswordField has its own unique UIClassID value. The value for accessibleContext property is AccessibleJPasswordField, an inner class that extends the JTextField.AccessibleJTextField class. The echoChar property specifies the character to be displayed in the field each time a key is pressed. This character is used to hide the actual input characters, though the L&F may choose to ignore it and hide the input characters some other way.

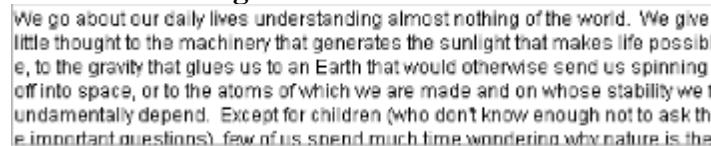
Table 19-5. JPasswordField properties

Property	Data type	get	is	set	Default value
accessibleContext	AccessibleContext				AccessibleJPasswordField
echoChar	char				'*'
password	char[]				
text0	string				
UIClassIDo	String				"PasswordFieldUI"
ooverridden					
See also properties from the JTextComponent class (Table 19-1).					

19.6 The JTextArea Class

The JTextArea class displays multiple lines of text in a single font and style. Its default behavior is not to wrap lines of text, but line-wrapping can be enabled on word or character boundaries. [Figure 19-5](#) shows a JTextArea.

Figure 19-5. JTextArea



We go about our daily lives understanding almost nothing of the world. We give little thought to the machinery that generates the sunlight that makes life possible, to the gravity that glues us to an Earth that would otherwise send us spinning off into space, or to the atoms of which we are made and on whose stability we fundamentally depend. Except for children (who don't know enough not to ask the important questions), few of us spend much time wondering what nature is like.

Like all Swing JTextComponents (but unlike java.awt.TextArea), JTextArea lacks integrated scrollbars. Fortunately, it is easy to embed a JTextArea inside a JScrollPane for seamless scrolling. (JTextArea implements the Scrollable interface, so JScrollPane can be intelligent about scrolling it.)

JTextArea handles newlines in a cross-platform way. Line separators in text files can be newline (\n), carriage return (\r), or carriage return newline (\r\n), depending on the platform. Swing's text components remember which line separator was originally used, but always use a newline character to represent one in memory. So always use \n when working with the content of a text component. When writing the content back to disk (or to whatever destination you give the write() method), the text component translates newlines back to the remembered type. If there is no remembered type (because the content was created from scratch), newlines are translated to the value of the line.separator system property.

19.6.1 Properties

JTextArea defines properties shown in [Table 19-6](#). AccessibleJTextArea is an inner class that extends JTextComponent.AccessibleJTextComponent.

Table 19-6. JTextArea properties

Property	Data type	get	is	set	Default value
accessibleContext	AccessibleContext				AccessibleJTextArea
columns	int				0
font, color	Font				From superclass
lineCount	int				From document
lineWrap	boolean				false
preferredScrollableViewportSize	Dimension				See comments below

19.7 How It All Works

The modularity of the Swing text components can be confusing. Fortunately, most of the time it doesn't matter how it works as long as it does work. However, some understanding of what's going on behind the scenes is necessary for what is to come in the next four chapters.

Let's take a look at what needs to happen for a text component to be displayed. These behaviors describe the responsibilities of a JTextArea, but they are similar for other JTextComponents:

- The text component retrieves its UI delegate from the L&F and installs it. For JTextArea, this might be javax.swing.plaf.basic.BasicTextAreaUI.
- The UI delegate may set properties such as font, foreground, and selection color. The UI delegate may also set the caret, highlighter, InputMaps and ActionMap. The maps allow text components to respond to L&F-specific keyboard commands for actions such as cut/copy/paste, select-all, caret-to-end-of-line, page-down, and so on.
- The UI delegate also instantiates an EditorKit. For JTextArea this might be javax.swing.text.DefaultEditorKit. Most of the Actions in the text component's array come from the EditorKit.
- If the text component's constructor didn't receive a Document, it creates one. JTextArea creates its Document (a PlainDocument) directly, but other text components delegate this to the EditorKit.
- The Document is responsible for storing the component's text content. It does this by breaking it into a hierarchy of one or more Elements. Each Element can hold part of the Document's content and some style information. JTextArea creates one Element for each line of text and ignores style information. Other text components can be more sophisticated.
- The text component registers itself as a listener for events it needs to track. This includes registering as a DocumentListener so it can update itself in response to any changes that occur in its Document.
- The text component may delegate or partially delegate preferredSize, minimumSize, and maximumSize to the UI delegate. JTextArea does this, but if its rows and columns properties are nonzero, it enforces a minimum on preferredSize.
- The UI delegate is responsible for painting the component, but it uses the EditorKit to paint the text content. The EditorKit does this by way of a ViewFactory. The ViewFactory creates a hierarchy of one or more View objects from the hierarchy of Element objects. The View hierarchy is then painted to the screen. A one-to-one correspondence from Element to View is typical but not required of the ViewFactory.

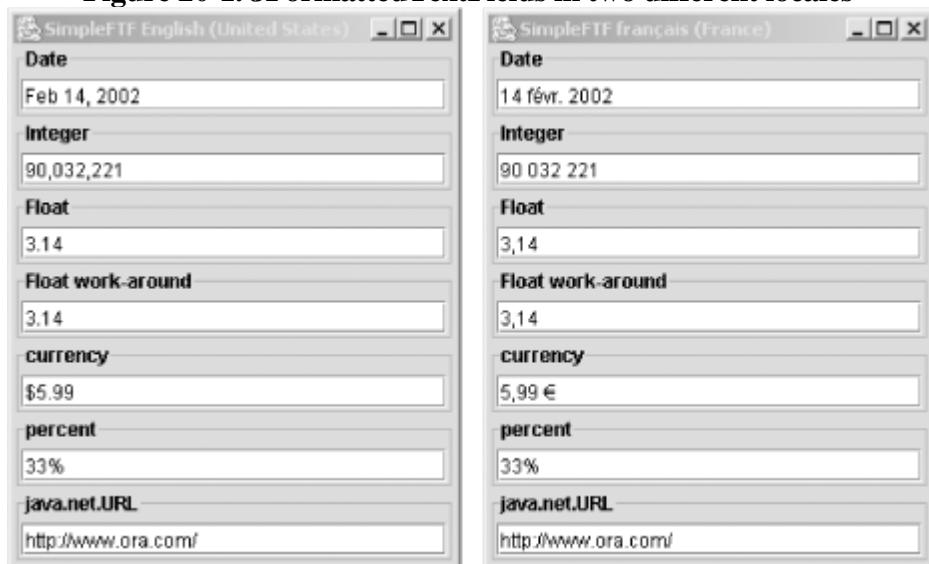
In this chapter, we've shown how easy it is to do simple things with the Swing text framework. However, if you want to do more than we've demonstrated in this chapter, Swing has a lot to offer. In the next four chapters, we'll examine the rest of the Swing text package, building many interesting and powerful sample programs as we go.

Chapter 20. Formatted Text Fields

Swing provides extended functionality for text fields through the `JFormattedTextField` class introduced in SDK 1.4. A `JFormattedTextField` can display its value in a friendly (and locale-specific) way, enforce restrictions on its value, be used to edit non-String objects, and permit its value (or part of its value) to be incremented or decremented with the keyboard.

[Figure 20-1](#) shows several `JFormattedTextField`s, but for the full effect you may wish to run the `SimpleFTF` program and play with it a bit. Most of the fields show locale-specific formatting. The Integer field puts delimiters between millions and thousands and between thousands and units. It changes its appearance (by dropping the delimiters) temporarily when it gains focus. An invalid value either adjusts to the closest valid value or reverts to the most recent valid value when a field loses focus, depending on the field. (For example, try changing the date to February 34.) Also, be sure to notice how elements of the Date field can be incremented and decremented with the up arrow and down arrow keys. (The L&F specifies keys for incrementing and decrementing, but existing L&Fs use the up arrow and down arrow. In addition, the Enter and Escape keys usually commit and cancel an edit, respectively.)

Figure 20-1. `JFormattedTextFields` in two different locales



You might also notice some nonintuitive behavior. Attempting to edit the first Float field drops all but the first digit after the decimal point, and text input in the URL field defaults to overwrite mode (not the expected insert mode). As a workaround for the former, see the second Float field in `SimpleFTF`. For the latter, see the `DefaultFormatter` and its `setOverwriteMode` property later in this chapter.

Here's the code for `SimpleFTF`:

```
// SimpleFTF.java
```


20.1 The JFormattedTextField Class

JFormattedTextField extends JTextField primarily by having a value property in addition to its content (accessed by the text property). The user may manipulate the field's content, but its value doesn't (necessarily) change until the user commits the edit. If the user cancels the edit, the content reverts back to the most recent valid value.

The field's value may be of any Object type, but it is displayed and edited as a String. The field uses its formatter to translate between Object and String representations.

JFormattedTextField works by maintaining an AbstractFormatterFactory (defined as a public abstract inner class). Whenever the field receives focus, it obtains an AbstractFormatter (another public abstract inner class) from the factory to oversee editing until it loses focus. It also queries the factory for a formatter at other times, such as when it loses focus or when setValue() is called.

The factory does not generally create new objects (as other factory classes typically do), but usually just hands out an appropriate existing formatter instance for the field to use. The factory is used to permit different formatters to be used for viewing versus editing, not to support a variety of different types. JFormattedTextField does support a variety of types, but its constructor creates a custom factory for the type passed in. So if you create a new JFormattedTextField(new java.util.Date()), every formatter it gets from its factory is for dates, [1] and it rejects user input that looks like a java.net.URL or some other class. [2]

[1] Unless you replace the factory, of course, using setFormatterFactory().

[2] Subclasses of java.lang.Number are an exception of sorts. By default, JFormattedTextField creates the same factory for any subclass of java.lang.Number. So, for example, a simple formatted text field created with a java.lang.Integer accepts floating-point values. See [Section 20.2](#) later in this chapter.

20.1.1 Properties

[Table 20-1](#) shows the properties defined by JFormattedTextField. It has its own unique UIClassID value but does not override the inherited accessibleContext. The document property is listed in the table only because JFormattedTextField overrides setDocument() to register itself as a listener for the new document and remove itself as a listener for the previous one.

Table 20-1. JFormattedTextField properties

Property	Data type	get	is	set	Default value
actions	Action[]				From superclass plus CommitAction, CancelAction
document, o	Document				PlainDocument
editValid	boolean				true
focusLostBehavior	int				COMMIT_OR_REPLACE

20.2 Handling Numerics

To a certain extent, JFormattedTextField treats the types Float, Double, Integer, Long, Short, and Byte (all subclasses of java.lang.Number) as interchangeable. This can sometimes be surprising. For example, with this field:

```
JFormattedTextField ftf = new JFormattedTextField(new Integer(4));
```

you might expect to be able to retrieve the value like this:

```
int val = ((Integer)ftf.getValue()).intValue(); // Incorrect
```

This code is likely to throw a ClassCastException because ftf.getValue() might return one of the other numeric types if the user has edited the field.[\[5\]](#)

[5] By default, it tends to return either a Long or a Double, but it's not so simple even when knowing that. A field showing 5.008 may return a Double value, but if the user shortens it to 5.00 with the Backspace key, it may now return a Long value (despite the presence of the decimal point).

A safer way to retrieve the value is:

```
int val = ((Number)ftf.getValue()).intValue(); // Correct
```

Casting to Number like this always works because the methods floatValue(), doubleValue(), integerValue(), longValue(), shortValue(), and byteValue() are all defined in the java.lang.Number class.

If for some reason you want to force getValue() to return a specific numeric type, this can be done by instantiating a NumberFormat, calling its setValueClass() method, and passing the NumberFormat into the JFormattedTextField constructor.

20.2.1 The JFormattedTextField.AbstractFormatter Class

AbstractFormatter is an abstract inner class of JFormattedTextField that defines the basic API for formatters. Usually, there is no reason to extend AbstractFormatter directly since DefaultFormatter (discussed in the next section) provides a more complete starting point.

20.2.1.1 Public methods

```
public abstract Object stringToValue(String text) throws java.text.ParseException
public abstract String valueToString(Object value) throws java.text.ParseException
```

These two methods are the heart of the formatter API. They are used by the field to convert between the field's value (which may be any Object type) and the String representation displayed in the field. If conversion is impossible, a java.text.ParseException is thrown. In particular, the stringToValue() method throws a ParseException if its argument is not valid. Returning without throwing a ParseException indicates that the input was deemed valid.

```
public void install(JFormattedTextField ftf)
```

Immediately after a JFormattedTextField obtains a formatter from its formatterFactory, it calls this method so the formatter can initialize itself for a new field. AbstractFormatter's implementation of this method stores ftf for later use (by the getFormattedTextField() method, for example) and sets the text content of the field. It also installs any values returned by getActions(), getDocumentFilter(), and getNavigationFilter() on ftf. (AbstractFormatter returns null in all three of those methods but is prepared for subclasses not to.) Subclasses may override this method if they wish to add listeners to the field or its Document or to modify the field's selection or caret position.

```
public void uninstall()
```

JFormattedTextField calls this method on its formatter just before it obtains a new formatter from its

20.3 The DefaultFormatter Class

DefaultFormatter is a concrete implementation of AbstractFormatter that provides enough functionality for many purposes. It can be used for classes with a constructor that takes a single String. To support other classes, you may have to override DefaultFormatter's `stringToValue()` method and its `valueToString()` method if the class's `toString()` method is not adequate.

DefaultFormatter maintains the field's `editValid` property, checking to see if the current edit is valid after every user keystroke and calling `setEditValid()` as appropriate.

20.3.1 Properties

[Table 20-3](#) shows the properties defined by DefaultFormatter.

Table 20-3. DefaultFormatter properties

Property	Data type	get	is	set	Default value
allowsInvalid	boolean				true
commitsOnValidEdit	boolean				false
overwriteMode	boolean				true
valueClass	Class				null

The `allowsInvalid` property controls whether the field's content may be temporarily invalid during an edit. Consider an integer field with a current value of 25 that the user wants to change to 19. The user may decide to do this by pressing the Backspace key twice, then the 1 key and the 9 key. After the first backspace, the content of the field is 2. After the second backspace, the content of the field is the empty string, but if `allowsInvalid` were false, this would not be allowed (since the empty string is not a valid integer), and the field would refuse to allow the 2 to be deleted. Setting this property to false can be effective in certain cases but should be done only after careful consideration.

`commitsOnValidEdit` controls how often a field's value is set during an edit. If this property is true, then the field attempts to commit its content after every keystroke. The default value is false, which means the field does not commit until something special happens, such as when the field loses focus or the user presses the Enter key. Consider again the situation from the previous paragraph. After the first backspace, the field shows 2. If `commitsOnValidEdit` is false, nothing is committed, and the field's value remains 25. If the user later cancels the edit (using the Escape key, for example), the content of the field reverts to 25. If `commitsOnValidEdit` is true, the 2 is committed immediately, and the field's value becomes 2.

When the `overwriteMode` property is true, characters entered by the user (and even pastes from the clipboard) replace characters in the field. When it is false (insert mode), new characters are inserted without deleting any of the existing characters. The default value is true, which is often not what you want. Subclasses of DefaultFormatter often call `setOverwriteMode(false)` in their constructors.

20.4 The MaskFormatter Class

MaskFormatter is a subclass of DefaultFormatter that formats strings by matching them against a mask. The mask is a string of literals and nonliterals. The nonliterals (listed and described in [Table 20-4](#)) are wildcards that match a family of characters. Literals match only themselves. A single quote preceding a nonliteral (or another single quote) turns it into a literal. So, for example, the mask "ABa'A#'Hb" consists of the nonliteral A, the literal B, the literal a, the literal A, the nonliteral #, the literal ', the nonliteral H, and the literal b. The string "1BaA1'1b" matches this mask.

Table 20-4. Mask nonliterals (case-sensitive)

char	Matches	Notes
*	Any character	
A	Any alphanumeric character	Tested by Character.isLetterOrDigit()
?	Any alphabetic character	Tested by Character.isLetter()
U	Uppercase alphabetic	Like ? but lowercase is mapped to uppercase
L	Lowercase alphabetic	Like ? but uppercase is mapped to lowercase
#	Any numeric character	Tested by Character.isDigit()
H	Any hexadecimal numeric	Like # but includes abcdefABCDEF
'		Precedes any character in this table to create a literal

The JFormattedTextField in [Figure 19-1](#) (and the code that follows it) uses a simple MaskFormatter with mask "UUUUU".

A MaskFormatter installed on a JFormattedTextField controls the caret so that by default it skips over literals and lands on nonliterals, which is nice. It can also be configured (by setting the valueContainsLiteralCharacters property to false) to have getValue() skip the literals, so the above string would be returned as "111" instead of "1BaA1'1b".

MaskFormatter works with Unicode characters, so, for example, the nonliteral # matches any Unicode DECIMAL_DIGIT_NUMBER, not just ASCII 0-9. (If you think this might be a problem, take a look at the validCharacters property.) In general, one character in the mask matches exactly one character of the input string, but with Unicode there may be a few languages where this is not always the case.

20.5 The InternationalFormatter Class

InternationalFormatter is a subclass of DefaultFormatter that delegates most of its work to an object of type java.text.Format (which is always a subclass since java.text.Format itself is abstract). Many of these subclasses provide internationalization support through awareness of locales, hence the name InternationalFormatter.

InternationalFormatter is rarely used directly. It is a repository for the common functionality of its two subclasses, DateFormatter and NumberFormatter.

20.5.1 Properties

[Table 20-6](#) shows the properties defined by InternationalFormatter.

Table 20-6. InternationalFormatter properties

Property	Data type	get	is	set	Default value
format	java.text.Format				null
minimum	Comparable				null
maximum	Comparable				null
overwriteModeo	boolean				false
ooverridden					
See also properties from the DefaultFormatter class (Table 20-3).).					

The format property is key to InternationalFormatter and is used by the valueToString() and stringToValue() methods to translate between the field's Object and String representations. It can be set in the constructor or via the setFormat() method.

The minimum and maximum properties can constrain the value of the field. If minimum is not null, value may not be less than minimum. If maximum is not null, value may not be greater than maximum. These may be of any type that implements the Comparable interface. (Provided it hasn't already been set, calling setMinimum() or setMaximum() also sets the formatter's valueClass property.)

InternationalFormatter overrides the overwriteMode property, setting it to false.

20.6 The DateFormatter Class

DateFormatter is a subclass of InternationalFormatter that uses an object of type java.text.DateFormat as its format. JFormattedTextField instantiates a DateFormatter for itself if you pass a java.util.Date or a java.text.DateFormat into its constructor. With a java.util.Date, the format is localized to the default locale. This is usually what you want, but if you want your program to use the same date format no matter where in the world it is run, construct your JFormattedTextField with a specific java.text.DateFormat.

Most of the DateFormatter implementation is concerned with providing support to increment and decrement subfields of any date from the keyboard. It does a nice job with this. If you haven't done so yet, you might want run the SimpleFTF program and play with the date field.

20.6.1 Properties

DateFormatter does not define any properties beyond those it inherits (see [Table 20-6](#)). The minimum and maximum properties can be handy if you want the field to be restricted to a specific range of dates.

20.6.2 Constructors

public DateFormatter()

Create a new DateFormatter and call java.text.DateFormat.getDateInstance() to set the format to the formatting style of the current locale.

public DateFormatter(java.text.DateFormat format)

Create a new DateFormatter with the specified format.

20.7 The NumberFormatter Class

NumberFormatter is a subclass of InternationalFormatter that uses an object of type java.text.NumberFormat as its format. JFormattedTextField instantiates a NumberFormatter for itself if you pass a subclass of Number or a java.text.NumberFormat into its constructor. With a subclass of Number, the format is localized to the current locale. Again, this is usually what you want, but if you want your program to use the same number format no matter where in the world it is run, construct your JFormattedTextField with a specific java.text.NumberFormat.

20.7.1 Properties

NumberFormatter does not define any properties beyond those it inherits (see [Table 20-6](#)). The minimum and maximum properties can be handy if you want the field to be restricted to a specific range.

20.7.2 Constructors

public NumberFormatter()

Create a new NumberFormatter and call java.text.NumberFormat.getNumberInstance() to set the format, which is a general-purpose format for the current locale.

public NumberFormatter(java.text.NumberFormat format)

Create a new NumberFormatter with the specified format.

20.7.3 Public Method

public Object stringToValue(String text) throws java.text.ParseException

NumberFormatter does not technically override this method, but (through private methods) does alter how it works slightly. This method should succeed if valueClass has been set to Integer, Long, Short, Byte, Float, or Double, even though those classes don't have constructors that take a single String argument.

20.7.4 The JFormattedTextField.AbstractFormatterFactory Class

AbstractFormatterFactory is an abstract inner class of JFormattedTextField that defines the basic API (a single method) for formatter factories. There is usually no reason to extend AbstractFormatterFactory directly. Instantiate a DefaultFormatterFactory instead.

Each JFormattedTextField has its own factory. See the description of formatter factories in [Section 20.1](#) earlier in this chapter.

20.7.4.1 Public method

public abstract JFormattedTextField.AbstractFormatter getFormatter(JFormattedTextField tf)

This method returns a formatter for tf to use. Simple factories return the same formatter instance (not even a clone) each time. Sophisticated factories might examine tf and return different formatters depending on whether it has focus, whether it is enabled, etc.

20.8 The DefaultFormatterFactory Class

DefaultFormatterFactory is Swing's only concrete implementation of AbstractFormatterFactory. It holds one or more formatters and decides which one to give to the field depending on whether the field has focus and whether the field's value is null.

20.8.1 Properties

[Table 20-7](#) shows the formatter properties defined by DefaultFormatterFactory.

Table 20-7. DefaultFormatterFactory properties

Property	Data type	get	is	set	Default value
defaultFormatter	JFormattedTextField.AbstractFormatter				null
displayFormatter	JFormattedTextField.AbstractFormatter				null
editFormatter	JFormattedTextField.AbstractFormatter				null
nullFormatter	JFormattedTextField.AbstractFormatter				null

defaultFormatter is used if one of the other formatter properties is null. It is common for defaultFormatter to be the only non-null formatter property, in which case the field uses the defaultFormatter exclusively.

displayFormatter is intended for use when the field does not have focus. It may be null, in which case defaultFormatter is used instead.

editFormatter is intended for use when the field has focus. It may be null, in which case defaultFormatter is used instead.

nullFormatter is intended for use when the field's content is null. It may be null, in which case displayFormatter or editFormatter (depending on whether the field has focus) is used instead. (If displayFormatter/editFormatter is also null, defaultFormatter is used.)

20.8.2 Constructors

public DefaultFormatterFactory()

20.9 Formatting with Regular Expressions

If you are a fan of regular expressions, you might be wondering whether Swing provides any direct support for using regular expressions with JFormattedTextFields. The answer is no. There is no direct support, but it's easy to write your own formatter for regular expressions.

```
// RegexPatternFormatter.java -- formatter for regular expressions
```

20.10 The InputVerifier Class

InputVerifier is an abstract class introduced in SDK 1.3. Subclasses of InputVerifier can be attached to JComponents to control whether they are willing to give up focus. Typically, this is done to force the user to put a component in a "valid" state before allowing the user to transfer focus to another component.

A brief example of using an InputVerifier with a JFormattedTextField appeared earlier in this chapter. (See the focusLostBehavior property of the JFormattedTextField class.)

20.10.1 Public Methods

public abstract boolean verify(JComponent input)

Return true if the component should give up focus, or false to keep focus. You should implement this method to examine the component and make the determination, but you should not produce any side effects on the component.

public boolean shouldYieldFocus(JComponent input)

This method is called when the user attempts to transfer focus from the given component elsewhere. Return true if the component should give up focus, or false to keep focus. This method is permitted to produce side effects.

InputVerifier's implementation simply defers to the verify() method, but implementing classes may override either.

Typically, subclasses provide only a verify() method unless they intend to produce side effects.

Chapter 21. Carets, Highlighters, and Keymaps

Like some of the other Swing components (JTree, for example), the text components allow you to do a certain amount of customization without having to implement your own L&F. Certain aspects of these components' behavior and appearance can be modified directly through properties of JTextSCComponent. This chapter explains how to modify three such components: carets, highlighters, and keymaps.

With the more flexible text components (JEditorPane and anything that extends it, including JTextPane), you can control the View objects created to render each Element of the Document model. In this chapter, we'll concentrate on the classes and interfaces related to modifying text components without dealing with View objects. [Chapter 23](#) discusses custom View classes.

JTextComponent has three UI-related properties that you can access and modify directly. These properties are defined by the following interfaces:

Caret

Keeps track of where the insertion point is located and defines how it is displayed. This includes its size and shape, its blink rate (if any), etc. (Don't confuse this with java.awt.Cursor, which tracks the mouse, not the insertion point.)

Highlighter

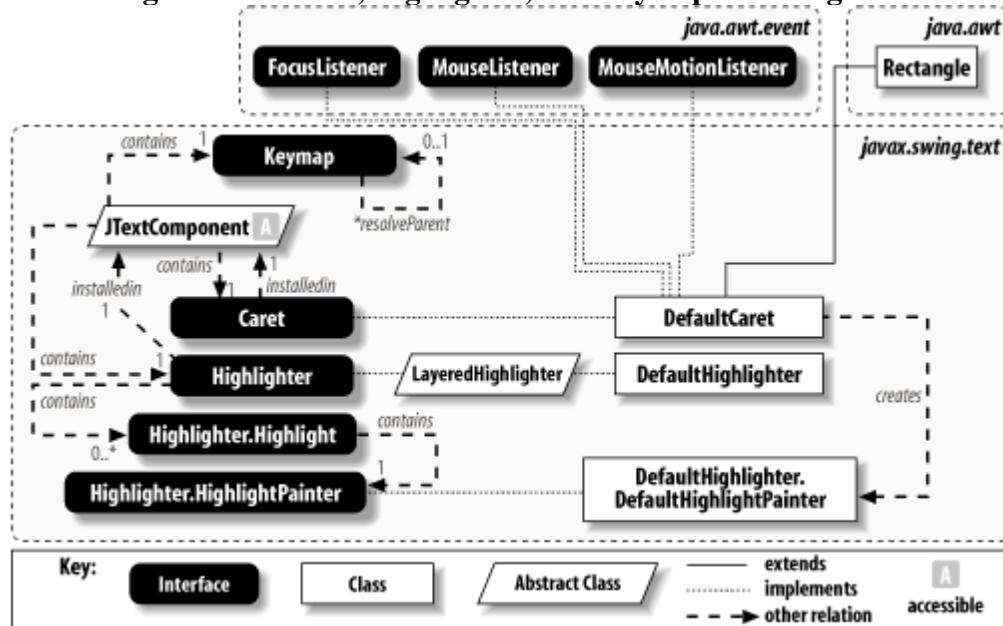
Keeps track of which text should be highlighted and how that text is visually marked. Typically, this is done by painting a solid rectangle "behind" the text, but this is up to the implementation of this interface.

Keymap

Defines a hierarchy of Actions to be performed when certain keys are pressed. For example, pressing Ctrl-C may copy some text, or Command-V may paste at the current caret location. This is considered an L&F feature because different native L&Fs have different default keymaps.

As you might expect, Swing provides default implementations of these interfaces. [Figure 21-1](#) shows these classes and interfaces and the relationships between them. Note that each Caret and Highlighter is associated with a single JTextComponent (set by its install() method) while Keymap has no direct relation to any JTextComponent, and therefore can be used by multiple components.

Figure 21-1. Caret, Highlighter, and Keymap class diagram



In the next few sections, we'll take a closer look at these interfaces as well as the default implementations for Caret and Highlighter. The default implementation of Keymap is an inner class of JTextComponent, which we can't subclass.

21.1 Carets

Carets represent the location where new text is inserted.

21.1.1 The Caret Interface

The Caret interface provides a number of useful features for dealing with text insertion and selection.

21.1.1.1 Properties

The Caret interface defines the properties shown in [Table 21-1](#). The blinkRate property specifies the number of milliseconds between Caret blinks. A value of 0 indicates that the Caret shouldn't blink at all.

Table 21-1. Caret properties

Property	Data type	get	is	set	default Value
blinkRate	int				
dot	int				
magicCaretPosition	Point				
mark	int				
selectionVisible	boolean				
visible	boolean				

The dot property is the current Caret position as an offset into the Document model. The mark is the other end of the current selection. If there is no selection, the value of mark is the same as dot. The selectionVisible property designates whether the current selection (if any) should be decorated by the component's Highlighter.

The visible property indicates whether the Caret itself should be visible. This is almost always true when the Caret's text component is editable and has focus but may not be in other situations.

magicCaretPosition is a Point used when moving among lines with uneven end positions to ensure that the up and down arrow keys produce the desired effect. For example, consider the following text:

Line 1 is long

11@ve RuBoard

21.2 Highlighters

Highlighters determine how text is marked to make it stand out. The order in which we discuss the highlighter interfaces may seem counterintuitive. The basic `Highlighter` interface is so straightforward that you'll rarely need to work with it directly, so we will describe it later. At this point, we discuss the interface you're most likely to use first: the `Highlighter.HighlightPainter` interface.

21.2.1 The `Highlighter.HighlightPainter` Interface

This is an inner interface of the `Highlighter` interface. If you want to change the way that highlights are drawn in your text component, this is the interface you'd implement.

Implementations of `Highlighter.HighlightPainter` are returned by `Caret` implementations and passed to `Highlighter` implementations (described later in this section; there are a lot of interfaces working together), which use them to decorate the area "behind" a selection. The only concrete implementation that's provided in Swing is `DefaultHighlighter.DefaultHighlightPainter`, which paints highlights as a solid background rectangle of a specified color.

This interface consists of a single `paint()` method. Unlike the `paint()` method of `Caret`, this method is called before the text itself is rendered, so there's no need to worry about obscuring text or XOR mode.

21.2.1.1 Method

`public void paint(Graphics g, int p0, int p1, Shape bounds, JTextComponent c)`

Render a highlighter behind the text of the specified component. The `p0` and `p1` parameters specify offsets into the document model defining the range to be highlighted. The `bounds` parameter defines the bounds of the specified component.

21.2.2 A Custom `HighlightPainter`

Here's a sample implementation of the `Highlighter.HighlightPainter` interface that paints highlights as thick underlines instead of as the usual solid rectangle. To use this highlight painter, we need to set a `Caret` that returns an instance of our highlight painter in its `getSelectionPainter()` method. The `main()` method in this example (provided for demonstration purposes only since the highlight painter would be complete without it) shows one way of doing this.

```
// LineHighlightPainter.java
```


21.3 Keymaps

A Keymap contains mappings from KeyStrokes^[2] to Actions and provides a variety of methods for accessing and updating these mappings.

[2] KeyStroke is discussed in more detail in [Chapter 27](#). Basically, it's just a representation of a key being typed, containing both the key code and any key modifiers (Ctrl, Alt, etc.).

21.3.1 The Keymap Interface

One last interface you can use to customize your application without implementing your own L&F is Keymap.

Normally, an L&F defines a set of meaningful keystrokes. For example, Windows users expect Ctrl-C to copy text and Ctrl-V to paste while the Mac uses the Command key instead. Ctrl-Insert and Shift-Insert perform the same tasks for Motif users. These key sequences work as expected in Swing text components because of the Keymap installed by the L&F. This interface lets you change or augment this behavior.



The Keymap interface has been available to Swing text components since the beginning. SDK 1.3 introduced the more flexible and universal keyboard event system based on InputMaps and ActionMaps (described in [Chapter 3](#)). The techniques illustrated in this section remain useful: Keymap support was reimplemented using the new mechanism for backward compatibility. This approach is still quite convenient if all you need is to add a couple of keyboard commands to a text component. Also be sure to learn how to use InputMap and ActionMap directly—they give you more capabilities, in many more situations.

21.3.1.1 Properties

The Keymap interface defines the properties shown in [Table 21-8](#). The boundAction and boundKeyStrokes properties contain the Actions and KeyStrokes (both part of the javax.swing package) local to the Keymap.

Table 21-8. Keymap properties

Property	Data type	get	is	set	Default value
boundActions	Action[]				
boundKeyStrokes	KeyStroke[]				
defaultAction	Action				
name	String				
resolveParent	Keymap				

Chapter 22. Styled Text Panes

In this chapter we'll discuss Swing's most powerful text component, JTextPane, as well as the Document model and the many classes and interfaces that go along with it. All text components use the Document interface to interact with their models. JTextPane uses the StyledDocument interface, an extension of the Document interface with style-manipulation methods.

This is a long chapter because there are so many classes and interfaces used by Document and its ilk. You may have no immediate need to learn about some of these classes, especially those in the latter half of the chapter. Don't feel that you must read the chapter straight through.

22.1 The JTextPane Class

JTextPane is a multiline text component that can display text with multiple fonts, colors, and even embedded images. It supports named hierarchical text styles and has other features that can help implement a word processor or a similar application.

Technically, JTextPane is a subclass of JEditorPane, but it usually makes more sense to think of JTextPane as a text component in its own right. JEditorPane uses "editor kits" to handle text in various formats (such as HTML or RTF) in a modular way. Use a JEditorPane when you want to view or edit text in one of these formats. Use a JTextPane when you want to handle the text yourself. We'll cover JEditorPane and editor kits (including JTextPane's editor kit, StyledEditorKit) in [Chapter 23](#).

22.1.1 Properties

JTextPane defines the properties shown in [Table 22-1](#). document and styledDocument are both names for the same property. The model returned by getDocument() always implements StyledDocument (which is an interface that extends Document). Attempting to call setDocument() with a Document that is not a StyledDocument throws an IllegalArgumentException. (The Document and StyledDocument interfaces are described in detail later in this chapter.)

Table 22-1. JTextPane properties

Property	Data type	get	is	set	Default value
characterAttributes	AttributeSet				
documento	Document				Value of styledDocument
editorKit, o	EditorKit				StyledEditorKit
inputAttributes	MutableAttributeSet				
logicalStyle	Style				
paragraphAttributes	AttributeSet				
styledDocument	StyledDocument				DefaultStyledDocument

22.2 AttributeSets and Styles

AttributeSet and its relatives are used to hold collections of attributes that can be used by styled text components (including JTextPane). For example, an AttributeSet might comprise an attribute for font size, an attribute for foreground color, and an attribute for indentation. Each attribute is simply a key/value pair. The Document model keeps track of which attribute sets apply to which blocks of text.

The interfaces and classes that are used for attribute sets are shown in [Figure 22-2](#). We'll discuss each one in detail, but first we'll provide a brief overview of what they do and how they relate. At the end of this section, we'll develop a Style-based text editor example.

AttributeSet

This interface defines basic methods for accessing a read-only set of attributes. An AttributeSet may have a "resolving parent," which (if it exists) is consulted when property lookups can't be resolved by the current set.

MutableAttributeSet

This interface extends AttributeSet with methods that allow attributes to be added, given new values, or deleted from the set.

Style

This interface extends MutableAttributeSet to add two things: an optional name for the style and support for adding and removing ChangeEventListeners.

SimpleAttributeSet

A basic implementation of the MutableAttributeSet interface.

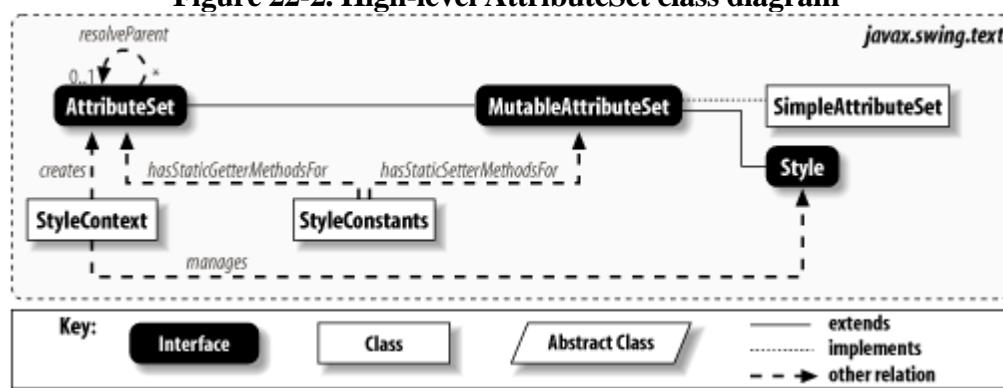
StyleConstants

This class defines the standard attribute keys used by Swing's text components. It also defines some static utility methods for getting and setting attribute values from attribute sets.

StyleContext

This utility class provides two services: it can create new AttributeSets in a space-efficient manner, and it manages a shared pool of named Style and Font information.

Figure 22-2. High-level AttributeSet class diagram

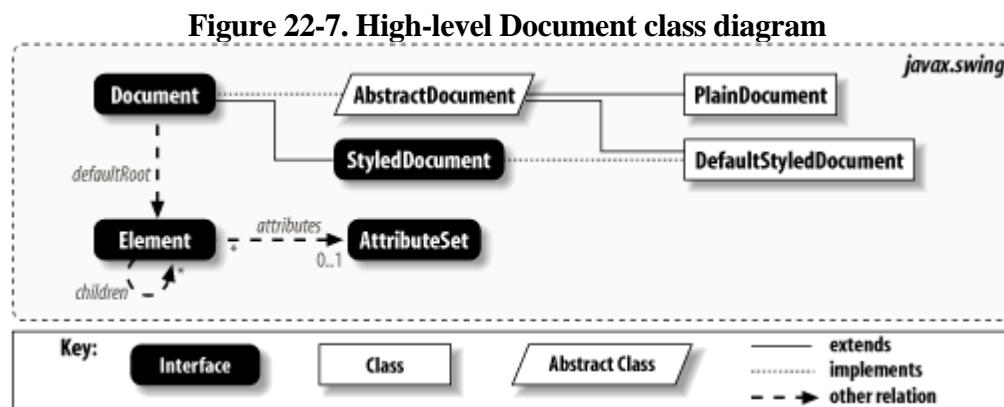


Since Swing does not provide a top-level public implementation of the Style interface, when you need a Style, you must request it from a StyleContext.[\[4\]](#) If you need a MutableAttributeSet, you can instantiate a SimpleAttributeSet using one of its constructors. If you want an AttributeSet, you can ask the default StyleContext for one or you can instantiate a SimpleAttributeSet.

[4] Which StyleContext do you use? The static `StyleContext.getDefaultStyleContext()` method returns the system's default StyleContext, but it is probably better to ask your JTextPane, which has style methods that delegate (eventually) to its StyleContext. For an example, see [Section 22.2.7](#) later in this chapter.

22.3 The Document Model

The Document is the M part of the MVC (Model-View-Controller) architecture for all of Swing's text components. It is responsible for the text content of the component as well as relevant style information for text components that support styled text. The Document model must be simple enough to be used by JTextField, but powerful and flexible enough to be used by JTextPane and JEditorPane. Swing accomplishes this by providing the classes and interfaces shown in [Figure 22-7](#).



Basically, a Document partitions its content into small pieces called Elements. Each Element is small enough that its style information can be represented by a single AttributeSet. The Elements are organized into a tree structure [\[22\]](#) with a single root.

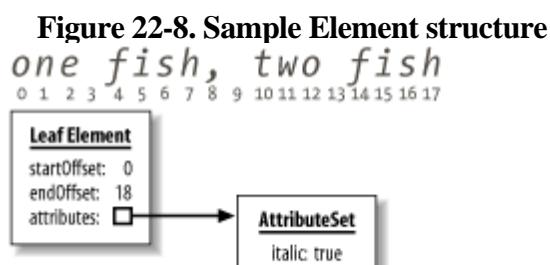
[22] The term Element applies to the interior nodes of the tree as well as to the leaf nodes. So it is more accurate to say that the Document partitions its content into small pieces called LeafElements. The interior nodes are called BranchElements, and each may have an arbitrary number of child Elements.

Swing provides the Document interface, which doesn't support styled text, and the StyledDocument interface, which does. But note that there is no StyledElement interface. Swing provides a single Element interface that does support style. The simpler document types (such as PlainDocument, which JTextField and JTextArea use by default) use Elements but don't assign any style information to them.

22.3.1 The Element Interface

The Element interface is used to describe a portion of a document. But note that an Element does not actually contain a portion of the document; it just defines a way of structuring a portion. It maintains a start offset and end offset into the actual text content, which is stored elsewhere by the Document.

Each Element may have style information, stored in an AttributeSet, that applies to the entire Element. [Figure 22-8](#) gives an example of how a Document that supports styled text must change a single Element representing a phrase in italics into a subtree of Elements when a word in the middle is changed from italic to bold.



22.4 Document Events

When changes are made to a Document, observers of the Document are notified by the event types DocumentEvent and UndoableEditEvent, defined in the javax.swing.event package. UndoableEditEvent and its associated listener interface are discussed in [Chapter 18](#). In this section, we'll look at DocumentEvent and its relatives.

22.4.1 The DocumentListener Interface

Document uses this interface to notify its registered listeners of changes. It calls one of three methods depending on the category of change and passes a DocumentEvent to specify the details.

22.4.1.1 Methods

public void changedUpdate(DocumentEvent e)

Signal that an attribute or set of attributes has changed for some of the Document's content. The DocumentEvent specifies exactly which part of the Document is affected.

public void insertUpdate(DocumentEvent e)

Signal that text has been inserted into the Document. The DocumentEvent specifies which part of the Document's content is new.

public void removeUpdate(DocumentEvent e)

Signal that text has been removed from the Document. The DocumentEvent specifies where the text was located in the Document before it was deleted.

Suppose we want the parentheses matcher we wrote in the last section to update its colors "live" instead of waiting for the user to click on a button. All we have to do is register with the pane's Document as a DocumentListener. Whenever we're notified that text has been inserted or deleted, we recolor the parentheses. It's that easy.

We do have to be careful not to call run() directly from insertUpdate() or removeUpdate(). This results in an IllegalStateException when our code attempts to obtain the Document's write lock. The solution is to avoid coloring the parentheses until the Document is finished with its event dispatching, which is easily done with SwingUtilities.invokeLater().

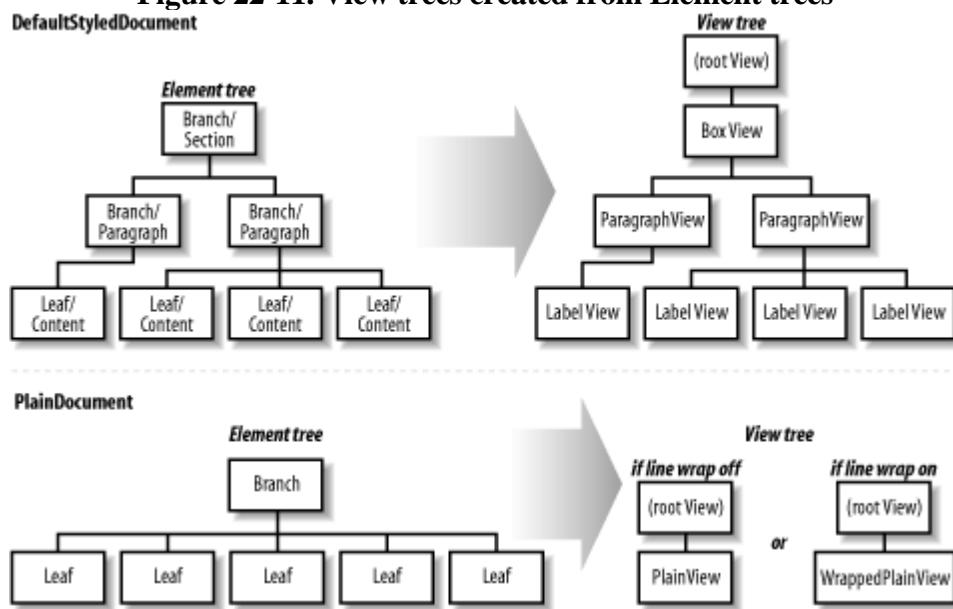
```
// LiveParenMatcher.java
```


22.5 Views

In our discussion of how Swing represents styled text, we haven't mentioned how it is actually drawn on the screen. That's where the View classes come in. They form most of the V part of the MVC architecture for text components and are responsible for rendering the text.

The way this works is a tree of View objects is created from the Document's Element tree. (Examples of this are shown in [Figure 22-11](#).) For DefaultStyledDocuments in a JTextPane, the View tree is modeled closely on the structure of the Element tree, with almost a one-to-one mapping from Elements to Views. A PlainDocument in a JTextArea is handled more simply, with a single View object that paints the entire Element tree.

Figure 22-11. View trees created from Element trees



Notice that the View trees have a root View above what could be considered the "natural" root. This was done to ease the implementation of the other View classes, which can now all assume that they have a non-null parent in the View tree. So that each child doesn't have to register as a DocumentListener, the root View also takes care of dispatching DocumentEvents to its children. The actual type of the root View is a package-private inner class. (You are free to create a View tree with any root you like, so this does not always need to be the case. The implementations of the TextUI.getRootView() method that Swing provides do return a package-private View class.)

Creation of the View tree is often handled by the text component's TextUI, but if its EditorKit returns a non-null value from its getViewFactory() method, this task is delegated to the factory. This gives the more complex text components (such as JEditorPane, discussed in [Chapter 23](#)) a powerful tool for customizing their appearance.

22.5.1 The ViewFactory Interface

Implementations of the ViewFactory interface determine the kind of View object that needs to be created for each Element passed in to the factory. We'll see more of the ViewFactory interface in the next chapter.

22.5.1.1 Method

```
public View create(Element elem)
```

Return a View for the given Element.

22.5.2 The View Class

22.6 The DocumentFilter Class

DocumentFilter is a class that oversees calls to insertString(), remove(), and replace() on any subclass of AbstractDocument. It can allow the edit to occur, substitute another edit, or block it entirely. Because all of Swing's Document classes inherit from AbstractDocument, this means you can attach a DocumentFilter to pretty much any Swing text component. The Document interface does not define a setDocumentFilter() method though, so you have to cast the object returned by the getDocument() method to AbstractDocument before you can set the DocumentFilter. DocumentFilter was introduced in SDK 1.4.

Here's how DocumentFilter works. AbstractDocument's insertString(), remove(), and replace() methods check for the existence of a DocumentFilter. If there is one, they forward the call to the like-named method of the DocumentFilter object. But the DocumentFilter methods are passed an extra parameter called the FilterBypass. The FilterBypass object has its own insertString(), remove(), and replace() methods, and these actually change the Document's content.

It's important that DocumentFilter methods call FilterBypass methods because an endless calling loop occurs if the Document methods are called. The delegation to the DocumentFilter is "bypassed" in FilterBypass. FilterBypass also provides a getDocument() method so the DocumentFilter can examine the contents of the Document before deciding whether to allow an edit.

22.6.1 Constructor

public DocumentFilter()

Instantiate a DocumentFilter that allows all edits to occur.

22.6.1.1 Methods

DocumentFilter's methods allow all edits to occur by default, so you override only the ones you want to be more restrictive.

```
public void insertString(DocumentFilter.FilterBypass fb, int offset, String text, AttributeSet attr) throws
BadLocationException
public void remove(DocumentFilter.FilterBypass fb, int offset, int length) throws
BadLocationException
public void replace(DocumentFilter.FilterBypass fb, int offset, int length, String text,
AttributeSet attr) throws BadLocationException
```

22.6.1.2 DocumentFilter.FilterBypass methods

```
public Document getDocument() public void insertString(int offset, String text, AttributeSet attr) throws
BadLocationException
public void remove(int offset, int length) throws BadLocationException
public void replace(int offset, int length, String text, AttributeSet attr) throws BadLocationException
```

Here's a simple example of a DocumentFilter that doesn't allow lowercase letters in the Document. All it does is convert the input String to uppercase before delegating to the FilterBypass.

```
// UpcaseFilter.java
```


22.7 The NavigationFilter Class

The NavigationFilter class is similar to the DocumentFilter class except that it oversees caret positioning instead of edits to the Document. Like DocumentFilter, NavigationFilter was introduced in SDK 1.4. Unlike DocumentFilter, you can easily install a NavigationFilter on any Swing text component without having to perform a cast. Just pass your NavigationFilter into the component's setNavigationFilter() method, and the component's Caret will filter all movement through it. NavigationFilters can also be specified by subclasses of AbstractFormatter (see [Chapter 20](#)).

Two of NavigationFilter's methods work just like DocumentFilter's methods. The Caret's moveDot() and setDot() methods check for the existence of a NavigationFilter. If there is one, they forward the call to the like-named method of the NavigationFilter object. But the NavigationFilter methods are passed an extra parameter called the FilterBypass. The FilterBypass object has its own moveDot() and setDot() methods, and these actually change the position of the Caret. FilterBypass also provides access to the Caret via its getCaret() method.

A third method, getNextVisualPositionFrom(), is called by the default caret movement Actions[28] when they are acting on a component that has a NavigationFilter. Thus, a NavigationFilter can completely control how the caret reacts to the arrow keys. (If no NavigationFilter exists, the default caret movement Actions call the like-named method of the appropriate View object instead. See [Section 22.5.2](#) earlier in this chapter.) There is no FilterBypass involved with this method.

[28] These Actions are defined in DefaultEditorKit (see [Chapter 23](#)) and are bound by the existing L&Fs to the arrow keys (to move the caret) and Shift-arrow keys (to extend the current selection).

22.7.1 Constructor

public NavigationFilter()

Instantiate a NavigationFilter that reproduces the default caret movement behavior.

22.7.1.1 Methods

NavigationFilter's methods reproduce the default caret movement behavior, so you override only the ones you want to behave differently.

*public void setDot(NavigationFilter.FilterBypass fb, int dot, Position.Bias bias)
public void moveDot(NavigationFilter.FilterBypass fb, int dot, Position.Bias bias)*

The bias parameter is not present in the corresponding methods in the Caret interface. Except for forwarding it to the FilterBypass methods when necessary, you can ignore it.

*public int getNextVisualPositionFrom(JTextComponent comp, int pos, Position.Bias bias, int direction,
Position.Bias[] biasRet) throws BadLocationException*

See the like-named method in [Section 22.5.2](#) for a description of this method. The default implementation delegates (by way of the component's TextUI) to the appropriate View object. If you override this method, you can call super.getNextVisualPositionFrom(comp, pos, bias, biasRet) to determine where the caret would go if there is no NavigationFilter. Note that the caret is not guaranteed to actually appear at the offset returned by this method because the NavigationFilter's moveDot() and setDot() methods may not allow it.

22.7.1.2 NavigationFilter.FilterBypass methods

*public Caret getCaret()
public void setDot(int dot, Position.Bias bias)
public void moveDot(int dot, Position.Bias bias)*

This was a long chapter, but now you know all about Swing's handling of styled text. You have only one more chapter to go, and then the Swing text components will no longer hold any mysteries for you. In the next chapter we cover the last of the Swing text components: the flexible and powerful JEditorPane.

Chapter 23. Editor Panes and Editor Kits

Over the last four chapters we've covered just about all the classes and interfaces that make up the Swing text framework. In this chapter, we'll look at a class that ties everything together: `EditorKit`. An `EditorKit` pulls together the document model, document view, document editing actions, and document I/O strategy, serving as a central reference point for a given document type.

In addition to looking at `EditorKit` and its subclasses, this chapter introduces the `TextAction` class (an abstract extension of `AbstractAction`) and the many useful concrete action classes available as inner classes of the `EditorKit` subclasses. These actions include basic functions such as copying and pasting text as well as style-oriented tasks such as changing font characteristics.

Throughout the chapter, we build simple but powerful editors for working with increasingly complex content types, moving from plain text to styled text to HTML. Finally, we discuss the process for creating your own editor kit.

23.1 The JEditorPane Class

JEditorPane is an extension of JTextComponent capable of displaying various types of content, such as HTML and RTF. It is not intended to be used as a full-featured web browser, but it can be used to view simple HTML and is ideal for integrating online help into Java applications.

JEditorPanes work closely with EditorKit objects. An EditorKit plugs into the editor pane to customize it for a particular content type. Without an EditorKit telling it how to work, a JEditorPane can't function. We discuss EditorKit in the next section.

[Figure 23-1](#) shows the JEditorPane in action, displaying a portion of the Javadoc for the JEditorPane class. Here's the code:

Figure 23-1. JEditorPane showing an HTML page



// HTMLExample.java

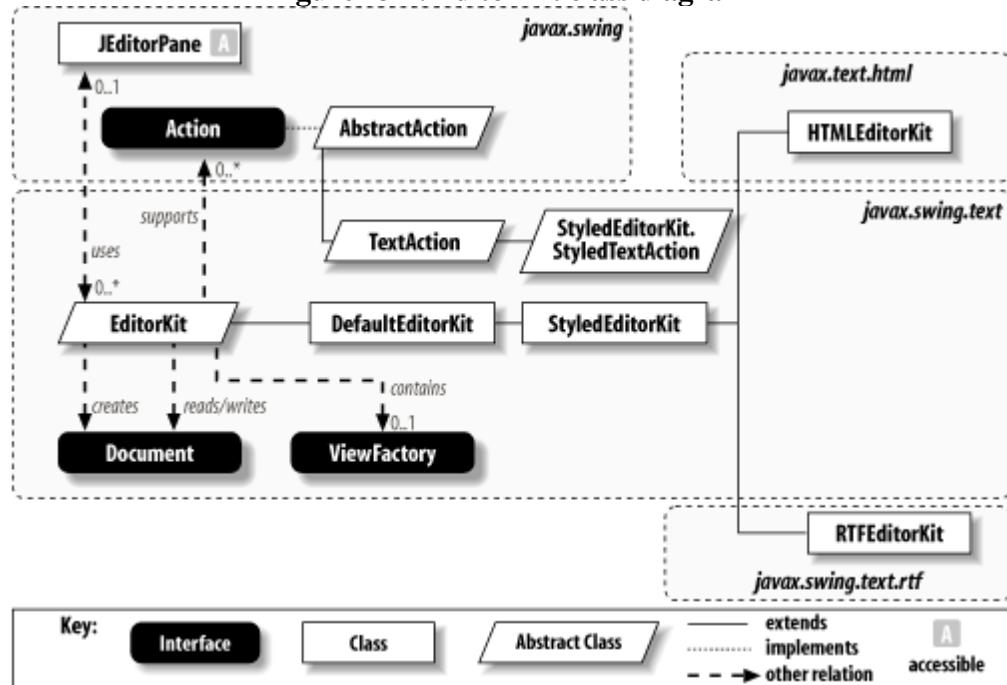
23.2 Overview of the Editor Kits

The following sections provide an overview of the various editor kits. With minor variations, this information applies to all editor kits.

23.2.1 The EditorKit Class

The EditorKit class is the abstract base class for all editor kits. It has a number of methods that define the model (e.g., `createDefaultDocument()`), the view (e.g., `getViewFactory()`), the capabilities (`getActions()`), and the I/O strategy (`read()` and `write()`) for a given type of document content. [Figure 23-2](#) shows the EditorKit class and the many classes and interfaces it interacts with.

Figure 23-2. EditorKit class diagram



This figure shows several important things about the EditorKit class. First, each EditorKit instance is typically associated with a single JEditorPane (though in some cases, it doesn't care). The EditorKit defines how to create a default Document as well as how to read and write the Document to a stream. In addition, each EditorKit may define a ViewFactory responsible for creating View objects for each Element in the Document. Finally, the diagram shows that an EditorKit may define a set of Actions that it supports.

The other classes shown in the diagram are the subclasses of EditorKit and AbstractAction. We'll look at each of these classes, as well as some inner classes not shown on the diagram, throughout this chapter.

23.2.1.1 Properties

EditorKit defines the properties shown in [Table 23-3](#). The `actions` property defines the set of actions that can be used on a text component, which uses a model and a view produced by this EditorKit. `contentType` indicates the MIME type of the data that this kit supports. The `viewFactory` property is the ViewFactory object used to create View objects for the Elements of the document type produced by the kit. The accessors for all three of these properties are abstract.

Table 23-3. EditorKit properties

Property	Data type	get	is	set	Default value
----------	-----------	-----	----	-----	---------------

23.3 HTML and JEditorPane

As we discussed in [Chapter 19](#), Swing provides (somewhat spotty) support for working with the Web's most common markup language, HTML. Fortunately, as Swing matures, the HTML support in the JEditorPane class improves.

The support for reading, writing, and displaying HTML through the JEditorPane class is provided by the HTMLEditorKit and associated classes. This class is an extension of the generic DefaultEditorKit class and is devoted (not surprisingly) to HTML files. You might recall from the previous discussion of EditorKit that there are three basic parts of an editor kit: its parser, its file writer, and its association with a ViewFactory. While we have looked briefly at these pieces before, here we focus on their implementation in HTMLEditorKit.

Along the way, we'll see how to extend some of these pieces to do custom work in the context of an HTML document. We decided that a detailed discussion of the classes that extend the editor kit would not be interesting to a majority of our readers, so we concentrate on the more immediately useful classes. However, if you want to see the details and play with the internals of the HTMLEditorKit, we have two entire chapters devoted to this topic on the book's web site (<http://www.oreilly.com/catalog/jswing2>). Throughout this discussion, we'll point you to that material if you want more detail than we've provided here.

Much of the information in those chapters describes how you can extend HTML with custom tags. Adding custom tags to HTML is not really a good idea in most cases. You are better off using XML, which is by definition extensible, for that purpose. A great book to start with is Java and XML by Brett McLaughlin—from O'Reilly, of course.

If you're just looking for basic HTML editing and rendering, you'll be in good shape after reading the rest of this chapter. If you need to support custom tags or write your own ViewFactory, grab the online chapters.

23.3.1 A Quick Browser Example

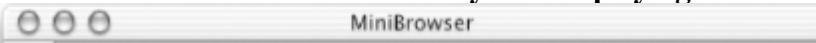
Before we get down to business on the API, here's an example of the HTMLEditorKit's power. In a very small amount of code, we can create a web browser. This minibrowser contains nothing more than two labels, a text field, and a JEditorPane controlled by the HTMLEditorKit. The browser is similar to the very first example in this chapter, but we've added the ability to enter URLs, and we show the destination URL in the status bar whenever you mouse over a hyperlink. (We also added support for hyperlinks inside HTML frames.)

We'll cover all the parts in more detail, but if you can't wait to get started with your own browser, pay special attention to the *SimpleLinkListener.java* file. That's where most of the real work in this example occurs.[\[7\]](#) If all you need to do is display HTML text in your application, this is the example to look at (or perhaps the even simpler one at the start of the chapter).

[7] The original version of this browser appeared in Marc's article, "Patch the Swing HTMLEditorKit," JavaWorld, January 1999 (<http://www.javaworld.com/javaworld/jw-01-1999/jw-01-swing.html>). Our derivative code is used with permission. Thanks also to Mats Forslöf (at Marcwell, in Sweden) for his improvements to the original version. Feel free to check out the article, but be aware that the patch mentioned fixes a pre-1.3 bug.

Here's the code for this mini-browser. The first file, *MiniBrowser.java*, sets up the basic Swing components you can see in [Figure 23-6](#), namely a text field for entering new URLs, a JEditorPane for displaying the current page, and a label at the bottom of the frame to function as the status bar. (The status bar reflects the "go to" URL when you place your mouse cursor over a hyperlink.) The second file, *SimpleLinkListener.java*, handles the hyperlink events.

Figure 23-6. MiniBrowser on a Mac OS X system displaying www.oreilly.com



23.4 Hyperlink Events

JEditorPanes fire a type of event called a HyperlinkEvent. Typically, this event is fired when the user clicks on a hyperlink in the currently displayed document; the program normally responds by loading a new page. To support this event type, a related event class and listener interface are available in the javax.swing.event package.

23.4.1 The HyperlinkListener Interface

The HyperlinkListener interface (found in javax.swing.event) defines a single method, used to respond to hyperlink activations:

```
public abstract void hyperlinkUpdate(HyperlinkEvent e)
```

Called to indicate that a hyperlink request has been made. Typical implementations of this method obtain the new URL from the event and call setPage() on the associated JEditorPane. See the JEditorPane example earlier in the chapter to learn how this method can be used.

23.4.2 The HyperlinkEvent Class

The HyperlinkEvent class (found in javax.swing.event) describes a hyperlink request.

23.4.2.1 Properties

HyperlinkEvent defines the properties shown in [Table 23-9](#). The URL property contains a java.net.URL object that can be used to retrieve URL content represented by the event. The description property allows a description of the link (typically, the text of the hyperlink) to be supplied. This can be useful when the URL property is null, such as when the hyperlink can't be parsed well enough to create a URL object. The eventType property defines the type of event that has occurred.

Table 23-9. HyperlinkEvent properties

Property	Data type	get	is	set	Default value
description	String				null
eventType	HyperlinkEvent.EventType				From constructor
sourceElement ^{1.4}	Element				null
URL	java.net.URL				From constructor
1.4since 1.4					
See also the					

23.5 The HTMLEditorKit Class

The API for the HTMLEditorKit is more or less what you might expect from an editor kit if you made it through [Chapter 19](#)-[Chapter 22](#) on Swing text components. We'll take a brief look at the API and then go through examples of how to work with this editor kit.

23.5.1 Inner Classes

As you can see in [Figure 23-7](#), HTMLEditorKit defines several inner classes. These inner classes are integral to the display and editing of HTML content. While we don't want to spend too much time on the details of these classes, you should know about them. We list them here as a quick reference.

public static class HTMLEditorKit.HTMLFactory

The view factory implementation for HTML.

public static class HTMLEditorKit.HTMLTextAction

An Action for inserting HTML into an existing document.

public static class HTMLEditorKit.InsertHTMLTextAction

An extension of HTMLTextAction that allows you to insert arbitrary HTML content. This one is quite handy. For example, we can create an action for a toolbar or menu that inserts a copyright:

```
private final static String COPY_HTML =
```

23.6 Extending HTMLEditorKit

As a quick example of how we might extend this class to add some functionality of our own, let's look at an editor kit that spits out debugging information as we load documents. This allows us to see the steps involved in extending an editor kit, and it leaves us with a very useful tool for implementing other extensions (such as custom tags and attributes).

The first step, of course, is to create our extended editor kit. In this example, we create a debugging editor kit that spits out the styles loaded and the individual tags it passes by. Here's the code:

```
// DebugHTMLEditorKit.java
```

23.7 Editing HTML

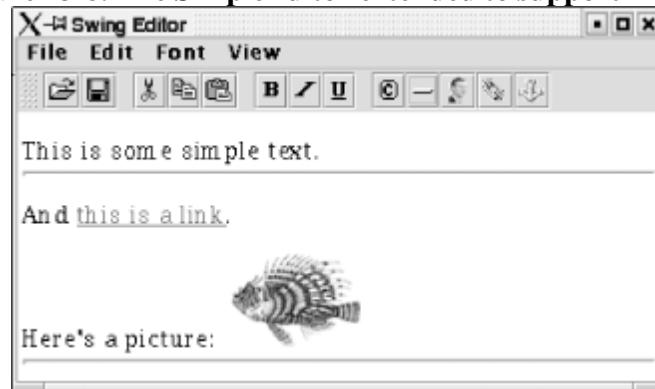
Generally, editor kits serve as a collection point for everything you would need to read, write, edit, and display some type of content. The HTMLEditorKit in particular serves as just such a collection point for HTML content. In addition to the HTMLEditorKit class proper, several supporting classes aid in the process of reading HTML, displaying it in the framework of JEditorPane, and writing it to a stream. Specifically, we look at the javax.swing.text.html.parser package and the HTMLWriter class. The APIs for this section remain a bit opaque, but as we mentioned earlier, each new release of the SDK comes with increased support, functionality, and openness.

If you're not interested in the hows and wheres of reading and writing HTML, but still want to be able to edit HTML documents, go ahead and dive into this example. We extend the SimpleEditor from earlier in this chapter to support two things:

- More HTML actions, including horizontal rules, images, and hyperlinks
- A Save menu item to write the document as HTML

To get started, [Figure 23-8](#) is a screenshot of our editor in action. The document you see was created from scratch using the editor.

Figure 23-8. The SimpleEditor extended to support HTML



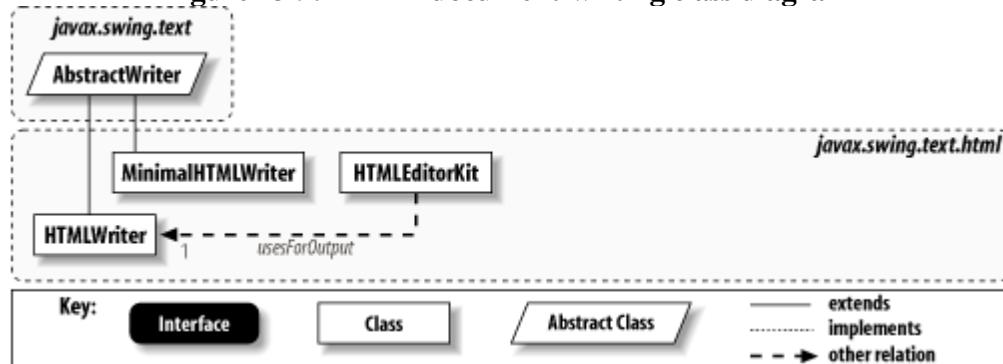
Here's the HTML generated when you save the document:

```
<html>
```


23.8 Writing HTML

The write() method from JEditorPane takes advantage of the writer installed as part of the HTMLEditorKit. In our previous example, that writer is the HTMLWriter class. Starting with a more generic styled document, you could also write HTML with MinimalHTMLWriter. The classes described in this section both extend from the AbstractWriter class. (See [Figure 23-9](#).)

Figure 23-9. HTML document-writing class diagram



23.8.1 The AbstractWriter Class

In this chapter, we've talked about a variety of strategies for saving document content. As of SDK 1.2, a new class provides some assistance in creating a rendition of an in-memory document structure suitable for saving as human-readable text. It relies on the ElementIterator class. AbstractWriter supports indentation to clarify the document structure as well as maximum line length to keep the generated output easy to read.

ElementIterator is a simple iterator class (somewhat obviously) devoted to working with Element objects. It has the usual next() and previous() methods of any bidirectional iterator. Both return objects of type Element. Unlike the new iterators in the Collections API, there is no hasNext() method. Instead, next() or previous() return null to signal the "end" of the stream. As with the constructors for AbstractWriter, an ElementIterator can be built on a Document or start from a particular Element. This class is covered in more depth in [Chapter 22](#).

23.8.1.1 Properties

AbstractWriter defines the properties shown in [Table 23-14](#).

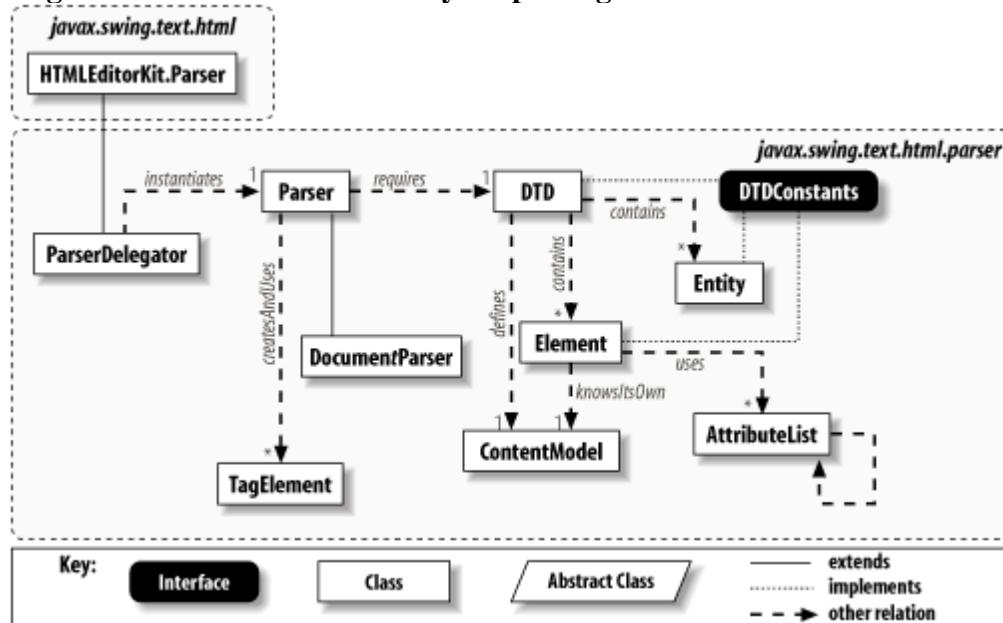
Table 23-14. AbstractWriter properties

Property	Data type	get	is	set	Default value
canWrapLines ^{1,3} , p	boolean				true
currentLineLength ^{1,3} , p	int				
documentp	Document				

23.9 Reading HTML

[Figure 23-11](#) shows the hierarchy breakdown for the classes involved in reading and parsing an HTML document with the HTMLEditorKit.

Figure 23-11. The class hierarchy for parsing HTML via HTMLEditorKit



23.9.1 Document Parsers

The first function involved in loading and displaying an HTML document is parsing it. The HTMLEditorKit class has hooks for returning a parser to do the job. The classes in the javax.swing.text.html.parser package implement a DTD-based[\[8\]](#) parser for this purpose.

[8] DTD stands for Document Type Definition. The editor kit uses its own compiled DTD based on HTML 3.2 rather than one of the public standard versions—another factor that complicates efforts to extend the tags supported.

But since we're here, let's look at the flow of an incoming HTML document. The editor kit instantiates a parser to read the document. ParserDelegator does what its name implies and delegates the actual parsing duties to another class—DocumentParser, in this case. ParserDelegator also handles loading the DTD used to create the real parser. Ostensibly, you could load your own DTD, but this whole process is rather tightly coupled to the HTML DTD supplied by the good folks at Sun. Once the parser is in place, you can send it a document and a ParserCallback instance and start parsing. As the parser finds tokens and data, it passes them off to the callback instance that does the real work of building the document.

You can display the document as it is built, or you can wait for the entire document to be loaded before displaying it. The tokenThreshold property from HTMLDocument determines exactly when the display work begins. See the discussion following [Table 23-12](#) for more details on the token threshold.

23.10 A Custom EditorKit

Although Swing's HTML support is less than ideal, it is sufficient to handle inline help systems and aid in rapid prototyping. Each release of the SDK improves support and usability. It will continue to get better. In the meantime, if you're desperate for serious markup language support, you really should check out XML.

If you're interested in doing your own EditorKit work, look up the more detailed HTMLEditorKit chapters online. You should also check out the javax.swing.text.rtf package. It serves the same basic purpose as the HTML package but reads and writes RTF files. However, be aware that RTF seems to be even more plagued with "acceptable variants" than HTML. Make sure you test your output on an intended target system before rolling out your new commercial editor!

To round out this final section, we'll review the steps involved in creating your own editor kit. These steps include:

- Creating the EditorKit class
- Defining the Document type
- Defining new Actions
- Creating custom View classes
- Creating a ViewFactory
- Creating a "reader" and a "writer"
- Telling JEditorPane about your new kit

23.10.1 Create the EditorKit Class

First, create your EditorKit class. Depending on the type of documents you're going to support, you can subclass the abstract EditorKit base class or extend any of the existing kits we've covered in this chapter.

Much of the work in creating this class is covered in the steps that follow. In addition to those more complex issues, you should implement the following EditorKit methods:

public abstract Object clone()

Strangely, the Swing implementations of this method just return a new instance of the editor kit subclass. It is probably a good idea to return a "true clone" here, although this method is not currently used within Swing.

public void install(JEditorPane c)

Chapter 24. Drag and Drop

Until the Java 2 platform hit the streets, Drag and Drop support (specifically, support for interacting with the native windowing system underneath the JVM) was lacking. The ability to let users drag a file from their file chooser into your application is almost a requirement of a modern, commercial user interface. The `java.awt.dnd` package gives you and your Java programs access to that support. You can now create applications that accept information dropped in from an outside source. You can also create Java programs that compile draggable information that you export to other applications. And, of course, you can add both the drop and drag capabilities to a single application to make its interface much richer and more intuitive.

"But wait!" you cry. "I recognize that package name. That's an AWT package!" You're right. Technically, Drag and Drop support is provided under the auspices of the AWT, not as a part of Swing. However, one driving force behind Swing is that it provides your application with a more mature, sophisticated user interface. Because Drag and Drop directly affects that maturity, we figured that you'd like to hear about it—even if it is not a part of Swing. And to try and hide that fact, we'll be using Swing components in all of the examples. However, we should note for completeness that Drag and Drop support can be added just as easily to good, old-fashioned AWT components—they just don't look as nice.

24.1 What Is Drag and Drop?

If you have ever used a graphical file system manager to move a file from one folder to another, you have used Drag and Drop (often abbreviated DnD). The term "drag and drop" refers to a GUI action in which the end user "picks up" an object such as a file or piece of text by moving the cursor over the object, clicking the mouse button, and, without releasing the button, "dragging" the object to a particular area of the screen and releasing the mouse button to "drop" the object. This process is meant to extend the desktop metaphor. Just like your real desktop, you can rearrange things by picking them up, moving them, and dropping them in a filing cabinet, a trash can, an in-box, a shoebox, or the floor.

Some programmers have added DnD functionality to individual components. For example, you might want to have a graphically rearrangeable JTree. Even without the DnD package, you can accomplish this with a clever bit of programming and a good deal of time. With the DnD package, however, not only do you not need the clever bit of programming, you are not limited to one component. You can drag information from one component to another in the same application, in two different Java applications (using separate JVMs), or even between your Java application and the native windowing system.

24.1.1 DnD and SDK 1.4

SDK 1.4 introduced several new features that make minimal DnD functionality easy to program for simple cases. On the drag side, many components now have a new boolean property called `dragEnabled`. If you set this property to true, you can "export" information by dragging it away from the component. [Table 24-1](#) shows the Swing components that support the `dragEnabled` property and the format of the information they export.

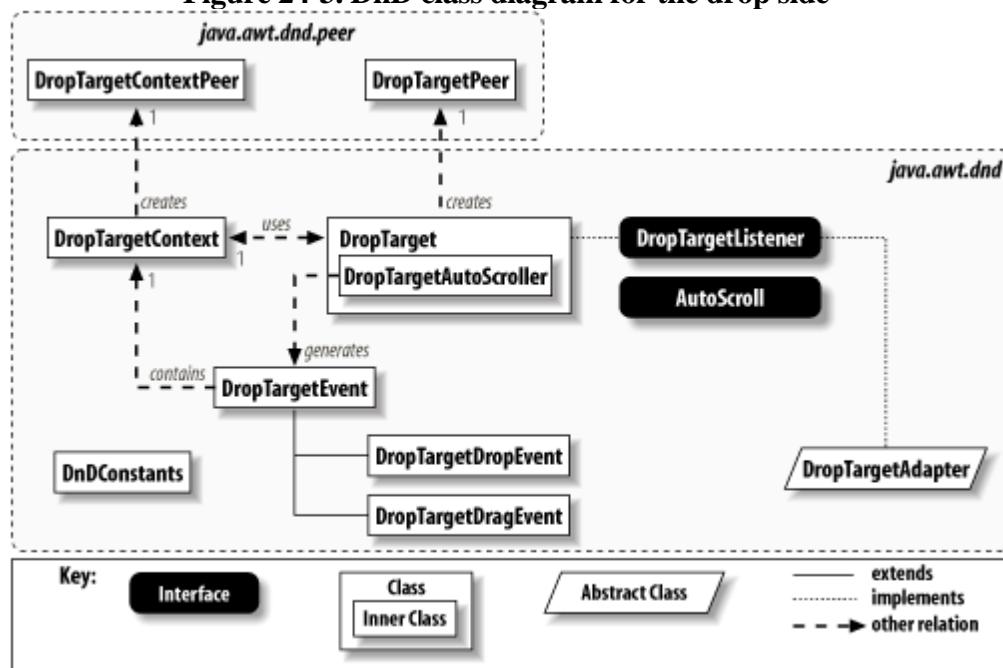
Table 24-1. Export capabilities of various Swing components

Component	Export data type(s)	Description
JColorChooser	application/x-java-jvm-local-objectref	Exports a reference to a <code>java.awt.Color</code> object. Imports a <code>color TransferHandler</code> property so you can accept such a drag.
JEditorPane	text/plain, other	If the content type for the editor is not plain text, the export contains both a plain-text version and an "other" version determined by the <code>write()</code> method of the editor pane.
JFileChooser	application/x-java-file-list	The files dragged out of a <code>JFileChooser</code> are set up in the same manner as those of a native file chooser. If you drop them on a Java drop site, the type is <code>x-java-file-list</code> .
JFormattedTextField	text/plain	Recall that the "format" in this text field refers to text validation, not visual formatting such as that found in HTML.
	text/plain	Similar to the JFormattedTextField, but for floating-point numbers.

24.2 The Drop API

This example uses several of the classes found in the `java.awt.dnd` package. Here's how they fit together (see [Figure 24-3](#)). Remember that we're just looking at the drop side of DnD. We'll explore the drag side in the next section and autoscrolling at the end of this chapter.

Figure 24-3. DnD class diagram for the drop side



24.2.1 The DropTarget Class

An obvious starting point for playing with this API is the `DropTarget` class. This class encapsulates the functionality you add to regular GUI components so that they can respond to drop events. You can make just about anything a drop target. Well, not anything, but any Component can be a drop target. Typically, you pick the component that responds to the drop—such as the text area in our example—but you can also pick a proxy component that simply helps produce all the right events while your event handler plays with the dropped data.

Note that just because the `DropTarget` class implements the `DropTargetListener` interface, this does not mean that the target is already handling drop events for you. The `DropTarget` class implementation of the interface supports the internal mechanisms for responding to events; it isn't used by us as programmers. We still have to create our own listener.

24.2.1.1 Properties

[Table 24-4](#) lists five properties of the `DropTarget` class that govern the context of this target and its data transfer capabilities.

Table 24-4. `DropTarget` properties

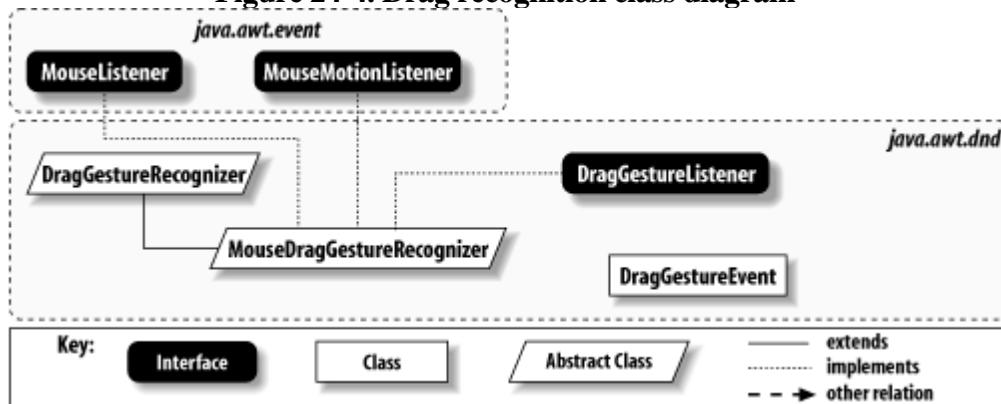
Property	Data type	get	is	set	Default value
active	boolean				true

24.3 The Drag Gesture API

Now that the drop side is working, how do we make the drag side go? First, we need a bit of background information. To successfully accomplish a drag in Java, we must first recognize what constitutes a drag gesture. A *drag gesture* is an action the user takes to indicate that she's starting a drag. Typically, this is a mouse drag event, but it is not hard to imagine other gestures that could be used. For example, a voice-activated system might listen for the words "pick up" or something similar.

The API for drag gestures is fairly simple. Four DnD classes and interfaces make up the core: DragGestureRecognizer, MouseDragGestureRecognizer, DragGestureListener, and DragGestureEvent, as shown in [Figure 24-4](#).

Figure 24-4. Drag recognition class diagram



24.3.1 The DragGestureRecognizer Class

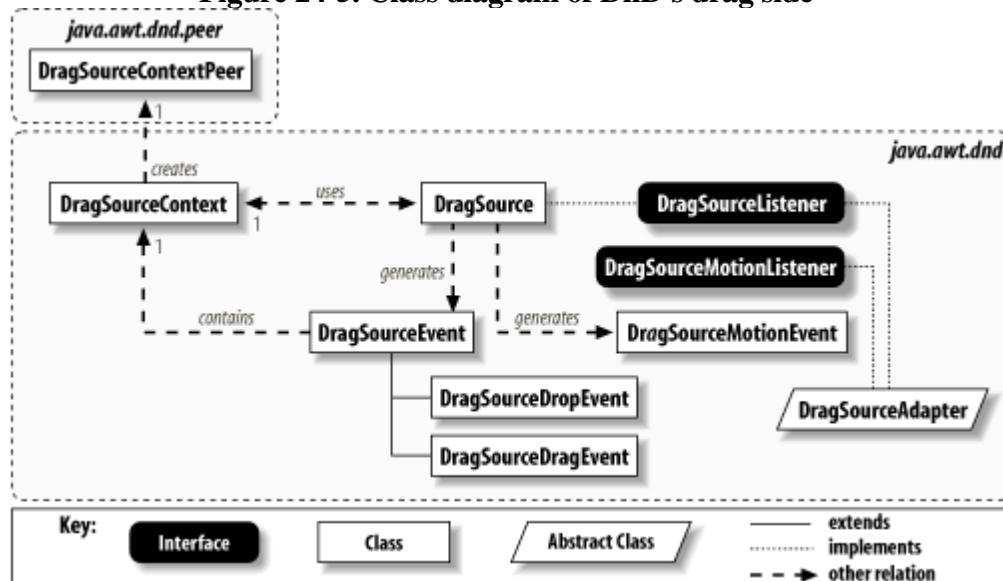
At the heart of this API is the DragGestureRecognizer class. Basically, the recognizer is responsible for recognizing a sequence of events that says "start dragging." This can be quite different for different platforms, so the DragGestureRecognizer class is abstract and must be subclassed to create an appropriate implementation. (The MouseDragGestureRecognizer takes a few more steps to complete the class and makes assumptions about using a mouse as the trigger for drag events. You can use the toolkit on your system to retrieve a concrete version.) This sounds more complex than it is. The idea is that you attach a drag gesture recognizer to a component with code similar to this:

```
JLabel jl = new JLabel( "Drag Me ! " );
I1@ve RuBoard
```


24.4 The Drag API

Once you recognize a drag gesture, you can start an actual drag. The drag source receives many of the same types of events as the drop target. Like a drop target, a drag source has a native peer in the form of a `DragSourceContextPeer` and receives events as the user drags his object over the source and finally drops it on a drop target. As shown in [Figure 24-5](#), the class diagram for the Drag API is similar to that of the Drop API.

Figure 24-5. Class diagram of DnD's drag side



24.4.1 The DragSource Class

The `DragSource` class, the flip side of `DropTarget`, provides support for creating draggable information. Recall from the beginning of this chapter that unlike the `DropTarget` class, a regular GUI component is not used as a base. Rather, the base component for starting a drag is encapsulated in the `DragGestureRecognizer`. The recognizer can in turn start a drag for the `DragSource`. The `DragSource` class encompasses the context of the drag and provides a focal point for maintaining the visual state of the drag. For example, several predefined cursors that you can use during the drag operation are built into this class.

24.4.1.1 Properties

The `DragSource` class has only three properties, two of which are static and apply to the overall DnD system. These properties are shown in [Table 24-14](#).

Table 24-14. DragSource properties

Property	Data type	get	is	set	Default value
defaultDragSource	DragSource				
dragImageSupport	boolean				
dragSourceListeners	DragSourceListener[]				Empty array

24.5 Rearranging Trees

The JTree class is a ripe candidate for the DnD system. Adding nodes to a tree, moving nodes from one tree to another, or even just rearranging the nodes within a single tree can all be accomplished quite easily with DnD gestures. Unfortunately, JTree does not have any built-in support for these actions. The next few examples take a look at extending JTree to include such features.

24.5.1 A TransferHandler Example

At the beginning of the chapter we mentioned that 1.4 made several parts of the DnD API obsolete—at least for simple tasks. That point is worth reiterating. If you have a simple drop target or a simple drag source that requires custom handling, you can use the TransferHandler class and the transferHandler property of JComponent to do most of your work.

One common feature of graphical trees is the ability to add nodes by dropping them into an existing tree. While the TransferHandler approach cannot handle all of the intricacies of adding and rearranging trees, it can make basic imports very straightforward.

Here's an example of a tree that can accept incoming lists of files from the native windowing system. You can drag icons from your desktop or file manager and drop them into the tree. They are appended to the root folder.

24.6 Finishing Touches

With the DnD API, we can add a few more features to help us achieve the goal of user interface maturity. Since we can't drop our objects on other leaves, we really should keep the drag cursor consistent. If we're over a folder, we can accept the drag and show the "ok to drop here" cursor. If we're anywhere else, we'll reject the drag and show the "no drop" version. The current implementation of our handlers allows only nodes to be moved. We can include support for "copy or move" drags. (We can also keep our cursor in sync with the new support so users know whether they're copying or moving.) Once all this is working, we should also set up autoscrolling on our tree to give the user access to any off-screen parts of the tree.

24.6.1 Dynamic Cursors

If your application can use the new DnD support built into the 1.4 release, you shouldn't have to worry about cursor management. If you're using one of the older SDKs or building fancier custom support, read on.

The DnD package ships with some standard cursors for indicating "ok to drop" and "not ok to drop." These cursors are displayed automatically when you accept() or reject() an item as it is dragged over potential drop targets. Here's a look at the dragEnter() and dragOver() event handlers that check to see whether the mouse is over a folder in the tree. If it's not, we reject the drag. This rejection causes the cursor to update its appearance. (Note that with the new cursor features in Java 2, you can define your own cursors and use them here if you don't like the cursors supplied by your windowing system.)

```
private TreeNode getNodeForEvent(DropTargetDragEvent dtde) {  
    I1@ve RuBoard
```

Chapter 25. Programming with Accessibility

Accessibility is a Java feature that allows you to extend your programs by interfacing an application's components with *assistive technologies* that have special permission to use them. Assistive technologies are, in the narrowest sense, specialized tools that people can use to assist in interacting with your application; examples include voice-recognition and audio output. In a broader sense, however, assistive technologies can be robust application suites that can assist just about anybody with anything.

It helps to think of accessibility as a two-sided conversation. This conversation takes place between an assistive technology and the "accessibility-friendly" components of applications. Assistive technologies are quite powerful: Java grants them access to all the components in the virtual machine that they can interface with, as well as the ability to interface with the windowing event queue. The latter gives an assistive technology the ability to monitor graphical events. The former means that an assistive technology can directly interact with the GUI widgets of one or more applications quickly and easily, without disrupting the application beneath it.

In order for components to interface with assistive technologies, they must implement special "accessible" interfaces. (The Swing components already implement these interfaces.) There are six unique interfaces for exporting accessible functionality: one each for actions, component properties, selections, text properties, hyperlinks, and bounded-range values. For each type of accessibility exported, one or more objects inside that component can be directly manipulated through that interface. For example, the JTextField Swing component supports text accessibility. Therefore, a voice-recognition program can insert letters in the field by obtaining the text-accessible interface and copying out the words that the user has spoken.

As there are two sides to accessibility, there are also two programming packages to deal with: the Accessibility API and the Accessibility Utility API. The first API defines the interfaces and classes that the programmer must implement on the application side in order for the components to be accessibility-friendly. The Accessibility Utility API, on the other hand, is a package of utilities that assistive technologies can incorporate into their own classes in order to "hook" into the JVM and interface with the application. Sun bundles the former with the Swing libraries. The latter is distributed independently, typically by the assistive technology vendor, similar to peripheral vendors providing drivers for their equipment.

Another way of creating an assistive interface in JDK 1.2 or higher is to take advantage of the MultiLookandFeel with one or more Swing components. For example, in addition to a graphical L&F, you might also redirect certain elements of the UI delegates to an audio output or a braille display. While a new L&F can technically be considered an assistive technology, this approach is better explained in [Chapter 26](#).

IBM has developed a great site on developing accessible applications in Java. You can check out this resource at <http://www-3.ibm.com/able/accessjava.html>.

25.1 How Accessibility Works

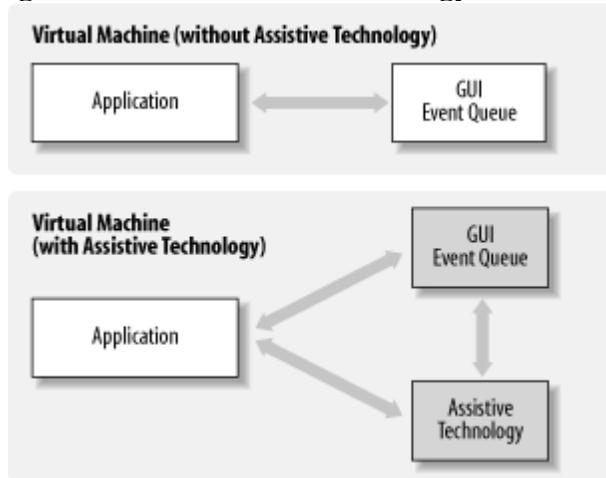
Assistive technologies are a feature of Java. These technologies typically manifest themselves as a class archive, such as a JAR file, that resides separately on the user's local machine. As we mentioned earlier, however, Java offers assistive technologies a wide latitude of control over an application for two reasons:

- Java loads assistive technologies into the same virtual machine as the application (or applications) with which they intend to interface.
- Assistive technologies can use or even replace the virtual machine's sole windowing event queue (`java.awt.EventQueue`).

Let's take a look at an accessibility-enabled Java virtual machine (JVM). When a JVM is started, it searches a configuration file for a list of specialized classes (we'll provide specifics about this later in this chapter). If it finds them, it loads the classes into the same namespace as the application that it is about to execute. Thus, when the Java application starts, the assistive technologies start with it.

The second important step is replacing the GUI event queue with an appropriate queue for the assistive technology. Hence, the application, GUI event queue, and assistive technology work together to make accessibility possible. [Figure 25-1](#) shows how an assistive technology interfaces with the JVM.

Figure 25-1. An assistive technology in the JVM



25.1.1 Evolving Accessibility Support

Before we get started, it's important to say that the accessibility features and even the approach to accessibility varies a bit depending on the version of Java you're using. The next few sections briefly describe the main differences you should be aware of; other differences are noted throughout the chapter.

25.1.1.1 Version 1.1 accessibility

A 1.1 JVM searches the `awt.properties` configuration file for a list of specialized accessibility classes and, if they are there, loads them before starting up.

Here's a segment of the `awt.properties` file with the appropriate additions for JDK 1.1 accessibility highlighted in bold:

`AWT.alt=Alt`

25.2 The Accessibility Package

Now let's discuss the issues an assistive technology encounters when hooking into an accessibility-friendly application.

25.2.1 The Path to Determining Accessibility

Almost all Swing objects support one or more forms of accessibility, which means that they implement the Accessible interface. However, for an assistive technology to find out which types of accessibility an application supports, the technology needs to do some investigating. The typical course of action runs like this:

1.

The assistive technology locates a desired component in the target application with the help of the Accessibility Utility APIs. Once found, it invokes the `getAccessibleContext()` method of the component object, which is the sole method of the Accessible interface. This method returns a customized AccessibleContext object, often an inner class of the component.

2.

The assistive technology can then use the AccessibleContext object to retrieve the name, description, role, state, parent, and children components of the accessible component in question.

3.

The assistive technology can register for any property change events in the component it's interested in.

4.

The assistive technology can call upon several standardized methods to determine whether those types of accessibility are supported. All AccessibleContext objects have these interface methods. If any of these methods return null, then the component does not support the specific accessibility type.

5.

If they are not null, the objects returned by each of the various methods can then be used to access the functionality on the component.

Different components implement different types of accessibility. For example, a tree might export accessibility action, selection, and component properties, while a button might simply export component properties. Components do not have to support all five steps. Hence, assistive technologies need to determine exactly which accessible functionality each component supports by querying that component for information.

25.2.2 The Accessible Interface

All components that need to export functionality to outside objects must implement the Accessible interface. This interface consists of only one method: `getAccessibleContext()`. Assistive technologies must always call this method first to retrieve an accessible-context object from the component. This object can then be used to query the component about its accessible capabilities.

25.2.2.1 Method

`public abstract AccessibleContext getAccessibleContext()`

Return the AccessibleContext object of the current component, or null if the component does not support accessibility.

25.2.3 The AccessibleContext Class

25.3 Other Accessible Objects

Before going further, there are several simple objects in the accessibility package used by AccessibleContext that we should discuss in more detail.

25.3.1 The AccessibleState Class

Each accessible component can have one or more states associated with it. An assistive technology can query these states at any time to determine how best to deal with the component. The accessible states can only be retrieved, however, and not set. There are two classes that the Accessible package uses to handle states: AccessibleState and AccessibleStateSet.

The AccessibleState class contains an enumeration of static objects that define states that any accessible component can have. Note that a component can be in more than one state at any time. A list of the possible states that an accessible object can be in, along with a brief description of each, is shown in [Table 25-3](#).

Table 25-3. AccessibleState constants

State	Meaning
ACTIVE	The window, dialog, or frame is the active one.
ARMED	The object, such as a button, has been pressed but not released, and the mouse cursor is still over the button.
BUSY	The object is busy processing and should not be interrupted.
CHECKED	The object is checked.
COLLAPSED	The object, such as a node in a tree, is collapsed.
EDITABLE	The object supports any form of editing.
ENABLED	The object is enabled.
EXPANDABLE	The object, such as a node in a tree, can report its children.
EXPANDED	The object, such as a node in a tree, is expanded.
FOCUSABLE	The object can accept the focus.

25.4 Types of Accessibility

Accessible components can export several types of assistive functionalities—for example: actions, text properties, component properties, icon properties, selections, table properties, hypertext, and bounded-range value properties. Most of these functions are already present in the Swing components, so if you stick closely to Swing, you probably won't need to implement these interfaces in your components. In an effort to explain how one might implement these interfaces, we have provided a simple example showing how to add AccessibleAction support to an AWT-based component.

25.4.1 The AccessibleAction Interface

The AccessibleAction interface outlines the methods that an accessible object or component must have to export its actions. The idea is that an assistive technology can determine the correct action by obtaining the total number of actions that the component exports, then reviewing each of their descriptions to resolve the correct one. Once this has occurred, the doAccessibleAction() method can be called with the correct index to invoke the required method.

25.4.1.1 Properties

The properties in [Table 25-5](#) must be readable through the AccessibleAction interface. accessibleActionCount stores the number of accessible actions that the component implements. The indexed property accessibleActionDescription provides a string describing the action associated with the given index. The action with index 0 is the component's default action.

Table 25-5. AccessibleAction properties

Property	Data type	get	is	set	Default value
accessibleActionCount	int				
accessibleActionDescription	Action				
indexed					

25.4.1.2 Method

public abstract boolean doAccessibleAction(int index)

Invokes an action based on the given index. The method returns false if an action with that index does not exist. It returns true if successful.

25.4.2 The AccessibleComponent Interface

The AccessibleComponent interface should be supported by any component that is drawn to a graphical context on the screen. Assistive technologies can use this interface to change how the component is drawn. Almost all of the methods in this interface call equivalent methods in the java.awt.Component class, so you will find this accessibility

25.5 Classes Added in SDK 1.3 and 1.4

The 1.3 and 1.4 SDKs both augmented the Accessibility package. Three interfaces in particular stand out: AccessibleIcon, AccessibleEditableText, and AccessibleTable.

25.5.1 The AccessibleIcon Interface

Added in the 1.3 release, the AccessibleIcon interface allows assistive technologies to get information about any icons on a component (such as a button or a label). Notice that the `getAccessibleIcon()` method from the `AccessibleContext` class returns an array of icons. While Swing labels and buttons do not support multiple icons, custom components can freely do so and still provide useful information to assistive technologies.

25.5.1.1 Properties

[Table 25-12](#) lists the three descriptive properties for AccessibleIcon. `accessibleIconDescription` is the most useful property and can be set through the accessible context as it is on other components.

Table 25-12. AccessibleIcon properties

Property	Data type	get	is	set	Default value
<code>accessibleIconDescription</code>	String				
<code>accessibleIconHeight</code>	int				
<code>accessibleIconWidth</code>	int				

25.5.2 The AccessibleEditableText Interface

While the `AccessibleText` interface has always been part of Swing, an editable representation for text did not exist. With the release of the 1.4 SDK, `AccessibleEditableText` fills that gap.

25.5.2.1 Property

As an extension of the `AccessibleText` interface, most of the properties for `AccessibleEditableText` are inherited. One writable property to alter the contents of the text component, `textContents`, was added. It is shown in [Table 25-13](#).

Table 25-13. AccessibleEditableText property

Property	Data type	get	is	set	Default value

25.6 The Accessibility Utility Classes

So far, we've seen how the Accessibility APIs help make Swing and AWT components easier to interface with assistive technologies. However, we haven't seen what's available on the other side of the contract. In reality, there are several classes that help assistive technologies interface with the JVM on startup, communicate with accessibility-friendly components, and capture and interpret various system events. These classes are called the *accessibility utility* classes. They are not part of Swing; instead, they exist as a separate package, com.sun.java.accessibility.util, which is distributed by Sun. (You can download this package from <http://java.sun.com/products/jfc/>.) The utility classes are crucial to assistive technology developers who wish to create specialized solutions that can communicate with any accessibility-friendly application.

Specifically, the accessibility utility classes can provide assistive technologies with:

- A list of the top-level windows of all Java applications currently executing under that virtual machine
- Support for locating the window that has input focus
- Support for locating the current mouse position
- Registration for listening for when top-level windows appear and disappear
- The ability to register listeners for and insert events into the windowing event queue

For the purposes of this chapter, we will discuss only the central classes in the Accessibility Utilities API. We begin with the class that allows assistive technologies to bridge the gap in the application's component: EventQueueMonitor.

25.6.1 The EventQueueMonitor Class

The EventQueueMonitor class is, in effect, a gateway to the system event queue. This class provides the central functionality for any assistive technology to capture and interpret system events, monitor the mouse position, and locate components related to screen position. EventQueueMonitor is a subclass of the AWT EventQueue class. This allows it to masquerade as the system event queue for the current JVM. Recall that the system event queue has its own thread, inserting and removing windowing events as necessary. With the EventQueueMonitor class, an application can register listeners for and post events to the system event queue thread—a critical task for assistive technologies.

To do this in JDK 1.1, [1] the EventQueueMonitor must replace the traditional system event queue at startup.

[1] In JDK 1.2 and higher, the loading of the event queue monitor is not necessary.

Currently, Java allows this to occur only if the following property is set in the *awt.properties* file:

AWT.EventQueueClass=com.sun.java.accessibility.util.EventQueueMonitor

25.7 Interfacing with Accessibility

The following code shows how to create a simple assistive technology that can monitor events on the system event queue and interface with accessible components. The example consists of one class, `AssistiveExample`. This class creates a small window containing two labels and five checkboxes, which are repeatedly updated when the mouse comes to rest over an accessible component for longer than half a second.

Note that while using 1.2 or higher accessibility, we have to check to see if the GUI is ready for us to start firing accessibility-related commands. We do this by checking the `EventQueueMonitor.isGUIInitialized()` method. This method returns a boolean indicating whether the GUI will accept accessibility commands. If it does, then we're fine. If it doesn't, then we must register ourselves to be notified when the GUI becomes available. This uses the `GUIMonitor` interface, which we explained earlier.

To use the `AssistiveExample` class, simply create a new `AssistiveExample` object from an existing application. The constructor creates a frame and makes it visible. For an example, check the source of the `BigExample` class in the online code files for this chapter.

Finally, note that we have a single button in our assistive example that performs the first action reported by the accessible context. You can use the Tab key to bring this button into focus while pointing with the mouse, then press the space bar to fire the action.

```
// AssistiveExample.java
```


Chapter 26. Look and Feel

In [Chapter 1](#), we introduced the concept of Swing's Pluggable L&F (PLAF) architecture. In this chapter, we'll get into a variety of topics related to PLAF. The chapter includes:

- A discussion of the default L&F presented to users on different platforms, and the trade-offs involved in changing it for your application
- An overview of how the Swing component classes work together with the UI-delegate classes
- A detailed explanation of the various PLAF-related classes in the javax.swing package, as well as some of the important classes and interfaces from javax.swing.plaf
- An explanation of how PLAF fits into JFC's accessibility framework using MultiLookAndFeel.
- Detailed discussions of strategies for customizing the L&F of your applications using the following techniques:
 - Modification of specific component properties
 - Modification of resource defaults
 - Use of themes in the Metal L&F
 - Use of customized client properties in the Metal L&F
 - Replacement of specific UI delegates
 - Creation of a new L&F from scratch

This chapter contains a lot of technical detail. You can do a lot with Swing's PLAF architecture without understanding everything we cover here. If you're interested in customizing the L&F of your applications, but don't mind if there are a few things that don't quite make sense, you can skim the next few sections and jump right into [Section 26.6](#). If you want to understand exactly how everything works, read on.

26.1 Mac OS X and the Default Look-and-Feel

On most platforms, unless the user or application has explicitly specified otherwise, Swing uses the Metal cross-platform L&F, and developers have likely become accustomed to this. With the advent of Mac OS X (which includes a tightly integrated Java environment that tracks Sun's latest releases), things change slightly. Mac users have strong expectations about the appearance and behavior of their applications, and Apple intends Java to be a first-class development environment on its OS. Because of this, the Mac L&F is the default under Mac OS X. This L&F allows Java applications to appear and behave like other Macintosh applications. This makes them more likely to be familiar to and adopted by Mac users.

As long as developers have made good use of Java's layout managers to account for differences in rendering between different machines, their applications translate to this new environment well. Unfortunately, not all applications have been designed in this way, and it is sometimes felt that the burden of testing under multiple different L&Fs is too difficult. Although you can use the mechanisms described later in this chapter to force your application to use a particular L&F (like the always available Metal), you should avoid taking this step lightly; it takes choices away from your users and definitely reduces the likelihood of the application's acceptance on highly consistent platforms like the Macintosh. If, despite this, you feel such a restriction is necessary, at least provide an easily found configuration option by which the users can choose to overrule you and use their preferred L&F at their own risk, so to speak.

If you do want to work with the Mac L&F, but lack a Mac on which you can test your application's layout, you can download the MacMetrics Metal theme from this book's web site: <http://www.oreilly.com/catalog/jswing2/>. This theme modifies Metal so that its components closely approximate the dimensions of the Mac L&F. It was graciously shared with us by Lee Ann Rucker who developed it while creating the Mac L&F itself. [Figure 26-1](#) shows an example of how the theme alters a Metal interface. To use it, simply be sure *MacMetrics.jar* is in your application's classpath, and arrange for the following lines of code to be executed:

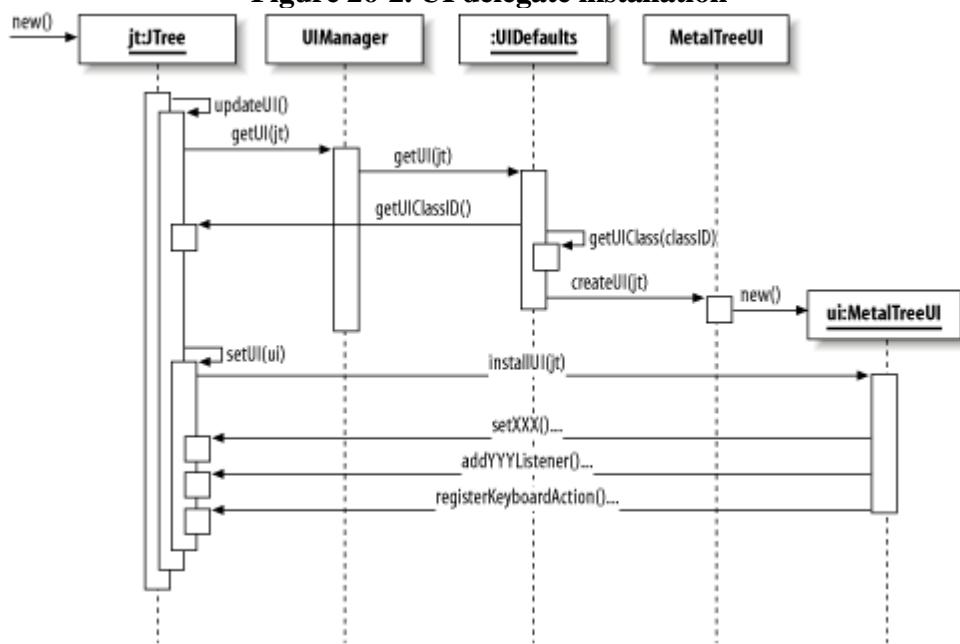
```
import javax.swing.plaf.metal.MetalLookAndFeel;
```

26.2 How Does It Work?

As you probably already know, each instance of a given Swing component uses a UI delegate to render the component using the style of the currently installed L&F. To really understand how things work, it helps to peek under the hood for a moment to see which methods are called at a few key points. The first point of interest is component creation time. When a new Swing component is instantiated, it must associate itself with a UI delegate object. [Figure 26-2](#) shows the important steps in this process. [1]

[1] We do not show every method call. The illustration provides a high-level overview of the process.

Figure 26-2. UI delegate installation



In this figure, we show what happens when a new JTree component is created. The process is the same for any Swing component:

1.

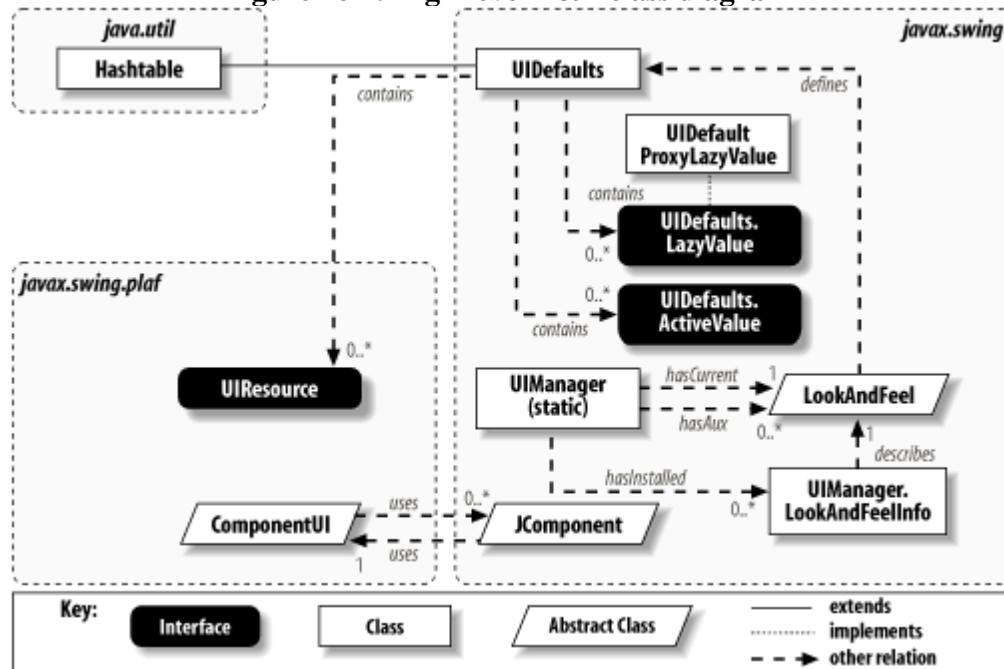
First, the constructor calls updateUI(). Each Swing component class provides an updateUI() method that looks something like this:

```
public void updateUI( ) {
```


26.3 Key Look-and-Feel Classes and Interfaces

In this section, we'll take an in-depth look at several key classes and interfaces that make up the Swing PLAF design. [Figure 26-4](#) shows the relationships between the classes (and interfaces) we will examine in this section.

Figure 26-4. High-level L&F class diagram



Before we look at the details of each of these classes, we'll quickly describe the role each one plays:

LookAndFeel

The abstract base class from which all the different L&Fs extend. It defines a number of static convenience methods, as well as some abstract methods required by every L&F.

UIDefaults

An L&F is responsible for defining a set of default properties. UIDefaults is a Hashtable subclass that holds these properties. The properties include UIClassID to ComponentUI subclass mappings (e.g., "TreeUI" to MetalTreeUI) as well as lower-level defaults, such as colors and fonts.

UIDefaults.ActiveValue and UIDefaults.LazyValue

These inner interfaces of UIDefaults enable some optimizations for resource values.

UIResource

This is an empty interface (like Serializable or Cloneable) used to tag property values. It allows values defined by the L&F to be distinguished from values set by the user, as described in [Section 26.3.5](#) later in this chapter.

UIManager

If you've ever changed the L&F of a Swing program at runtime, you're probably already familiar with this class. UIManager is responsible for tracking a global view of the L&Fs available in an application. It keeps track of the currently installed L&F and provides a mechanism to change the L&F. All of its methods are static, but it does provide a mechanism that allows multiple applets within a single virtual machine to use different L&Fs.

UIManager.LookAndFeelInfo

This inner class is used to describe available L&Fs without actually having to load the L&F classes. UIManager uses this class to provide a list of available L&Fs.

ComponentUI

This is the base class common to all UI delegates. It defines all of the methods related to painting and sizing that the

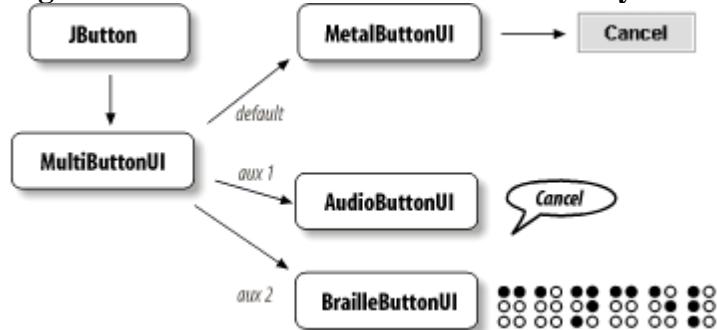
26.4 The MultiLookAndFeel

Before we get into creating our own L&F, we'll take a quick detour to explore MultiLookAndFeel. This is the L&F that allows accessible interfaces to be incorporated into Swing applications. It can also be used to add sound effects, support for automated testing, and more.

By this point, you're probably at least aware of the concept of accessibility as it applies to JFC and Swing. If you read [Chapter 25](#), you're aware of more than just the concept. The last piece of the accessibility puzzle is Swing's multiple L&F support.

The idea behind MultiLookAndFeel is to allow multiple L&Fs to be associated with each component in a program's GUI without the components having to do anything special to support them. By allowing multiple L&Fs, Swing makes it easy to augment a traditional L&F with auxiliary L&Fs, such as speech synthesizers or braille generators. [Figure 26-10](#) gives a high-level view of how this might work.

Figure 26-10. MultiLookAndFeel and Auxiliary L&F



In this diagram, we show a JButton in a multiplexing UI environment. The button's UI delegate is actually a MultiButtonUI, which is contained in the javax.swing.plaf.multi package. This is a special delegate that can support any number of additional ButtonUI objects. Here, we show the default UI delegate (MetalButtonUI) and two (hypothetical) auxiliary delegates, AudioButtonUI and BrailleButtonUI.

The MultiButtonUI (and all the other multiplexing delegates) keeps track of a vector of "real" UI objects. The first element in the vector is guaranteed to be the default, screen-based L&F. This default receives special treatment. When the JButton sends a query to its UI delegate, the MultiButtonUI forwards the query first to its default L&F, and then to each of the auxiliary L&Fs. If the method requires a return value, only the value returned from the default L&F is used; the others are ignored.

26.4.1 Creating an Auxilliary Look-and-Feel

Creating a fully implemented auxiliary L&F is beyond the scope of this book. However, we will throw something together to give you a feel for what it would take to build one.

In this example, we define an L&F called StdOutLookAndFeel. This simple-minded L&F prints messages to the screen to describe which component has the focus. To make the example reasonably concise, we show only the implementation of the UI delegate for JButtons, which does nothing more than print messages when a button gains or loses focus.

First, let's take a look at our StdOutLookAndFeel class:

```
// StdOutLookAndFeel.java
```


26.5 Auditory Cues

SDK 1.4 introduced a framework for providing auditory cues in response to user interface actions without the complexity of adding an auxiliary L&F. Core support is provided by BasicLookAndFeel through an audio action map (see [Table B-45](#) in [Appendix B](#)) that is available for use by subclasses.

It should be noted that, although the framework is in place, none of the standard L&Fs activate any sounds, at least as of SDK 1.4.1. Comments in the source code indicate that the activation has been temporarily disabled due to bugs in sound playback. So although this is an interesting area to watch, and you should feel free to experiment with customizations to activate various sounds, until Sun decides auditory cues are ready to be unleashed, it's probably best not to rely on them.

With that caveat, here's how the mechanism is set up. When `getAudioActionMap()` is called for the first time in `BasicLookAndFeel`, a series of initialization steps occur, with many opportunities for customization of the sort described in the next section:

1.

The UI defaults are consulted to look up the list of sounds to be loaded, under the key "auditoryCues.cueList". This is how [Table B-45](#) gets its values.

2.

For each cue, `createAudioAction()` is called to create the actual Action object that plays the sound. `BasicLookAndFeel` uses Java Sound to load and play named audio files, and the Metal L&F takes advantage of this mechanism, as shown in [Table 26-5](#) (the files are in the `javax.swing.plaf.metal.sounds` package within the runtime JAR). The Windows L&F takes a different approach and maps the cues in the list to appropriate Windows desktop properties so they reflect the choices made in the user's native desktop themes. This mapping is shown in [Table 26-6](#).

Table 26-5. Auditory cues mapped to sound files in MetalLookAndFeel

Auditory cue	File
<code>CheckBoxMenuItem.commandSound</code>	<i>MenuItemCommand.wav</i>
<code>InternalFrame.closeSound</code>	<i>FrameClose.wav</i>
<code>InternalFrame.maximizeSound</code>	<i>FrameMaximize.wav</i>
<code>InternalFrame.minimizeSound</code>	<i>FrameMinimize.wav</i>
<code>InternalFrame.restoreDownSound</code>	<i>FrameRestoreDown.wav</i>
<code>InternalFrame.restoreUpSound</code>	<i>FrameRestoreUp.wav</i>
<code>MenuItem.commandSound</code>	<i>MenuItemCommand.wav</i>

26.6 Look-and-Feel Customization

In this section, we'll look at different ways that you can change the way components in your application appear. We'll start with the simplest approach—making property changes on a per-component basis—and work our way through several increasingly powerful (and complicated) strategies. In the last section, we'll show you how to build your own L&F from the ground up.

26.6.1 Modification of Component Properties

This is the most obvious way to change the look of a component, and it's certainly not new to Swing. At the very top of the Java component hierarchy, `java.awt.Component` defines a number of fundamental properties, including foreground, background, and font. If you want to change the way a specific component looks, you can always just change the value of these properties, or any of the others defined by the specific components you are using. As we said, this is nothing new to the Swing PLAF architecture, but we don't want to lose sight of the fact that you can still make many changes in this way.

Bear in mind that platform-specific L&F constraints on the rendering of certain components may prevent some property changes from being honored. For example, Mac buttons are always the same color.

26.6.2 Modification of the UI Defaults

Modifying component properties lets you customize individual components. But what if you want to make more global changes? What if you want to change things that aren't exposed as component properties?

This is where `UIResources` come into play. There are more than 300 different resources defined by the Swing L&Fs that you can tweak to change the way the components are displayed. These resources include icons, borders, colors, fonts, and more. ([Appendix A](#) shows a complete list of the properties defined by the `BasicLookAndFeel`, the base class for all of the Swing-provided L&Fs.) Unfortunately, not all of the Swing L&Fs adhere strictly to these resource names. In the following example, several of the resource names we've used are specific to the Metal L&F.

26.6.2.1 Making global changes with defaults

In this example, we change the defaults for a variety of resources to give you an idea of the types of things you can affect. Specifically, we change the border used by buttons, the title font and icons used by internal frames, and the width used by scrollbars.

Because we're making these changes globally, any components that use these properties are affected by what we do.

```
// ResourceModExample.java
```


26.7 Creation of a Custom Look-and-Feel

Everything we've covered in this chapter up to this point has been useful background information for the ultimate application customization strategy: creating your own L&F. As you might guess, this is not something you'll do in an afternoon, nor is it usually something you should consider doing at all. However, thanks to the L&F framework, it's not as difficult as you might think. In a few instances, it actually makes sense, such as when you're developing a game. You'll likely find that the most difficult part is coming up with a graphical design for each component.

There are basically three different strategies for creating a new L&F:

- Start from scratch by extending `LookAndFeel` and extending each of the UI delegates defined in `javax.swing.plaf`.
- Extend the `BasicLookAndFeel` and each of the abstract UI delegates defined in `javax.swing.plaf.basic`.
- Extend an existing L&F, like `MetalLookAndFeel`, and change only selected components.

The first option gives you complete control over how everything works. It also requires a lot of effort. Unless you are implementing an L&F that is fundamentally different from the traditional desktop L&Fs, or you have some strong desire to implement your own L&F framework from scratch, we strongly recommend that you do not use this approach.

The next option is the most logical if you want to create a completely new L&F. This is the approach we'll focus on in this section. The `BasicLookAndFeel` has been designed as an abstract framework for creating new L&Fs. Each of the Swing L&Fs extends `Basic`. The beauty of using this approach is that the majority of the programming logic is handled by the framework—all you really have to worry about is how the different components should look.

The third option makes sense if you want to use an existing L&F, but just want to make a few tweaks to certain components. If you go with this approach, you need to be careful not to do things that confuse your users. Remember, people expect existing L&Fs to behave in certain familiar ways.

26.7.1 The PlainLookAndFeel

We'll discuss the process of creating a custom L&F by way of example. In this section, we'll define bits and pieces of an L&F called `PlainLookAndFeel`. The goal of this L&F is to be as simple as possible. We won't be doing anything fancy with colors, shading, or painting—this book is long enough without filling pages with fancy `paint()` implementations.

Instead, we'll focus on how to create an L&F. All of our painting is done in black, white, and gray, and we use simple, single-width lines. It won't be pretty, but we hope it is educational.

26.7.2 Creating the LookAndFeel Class

The logical first step in the implementation of a custom L&F is the creation of the `LookAndFeel` class itself. As we've said, the `BasicLookAndFeel` serves as a nice starting point. At a minimum, you'll need to implement the five abstract methods defined in the `LookAndFeel` base class (none of which is implemented in `BasicLookAndFeel`). Here's a look at the beginnings of our custom L&F class:

```
// PlainLookAndFeel.java
```


Chapter 27. Swing Utilities

There are many tasks you run into that are common and not terribly difficult. Hence, they get rewritten several times in several small variations. Ideally, you would code the task up into a method or class, and keep it around for reuse later. There are several small classes and methods strewn about the javax.swing package that fall into this category. This chapter presents those bits and pieces and points out where you might be able to put them to use in your own code. They may not be as fundamental as the wheel, but anytime you don't have to rewrite something, you're doing your part to create better OO programs!

We've called this chapter "Swing Utilities," but in fact it covers a range of subjects. In order of presentation, we'll look at:

Swing utilities

The SwingUtilities class and SwingConstants interface

Timers

The Timer class

Tooltips

The TooltipManager and JToolTip classes

Rendering

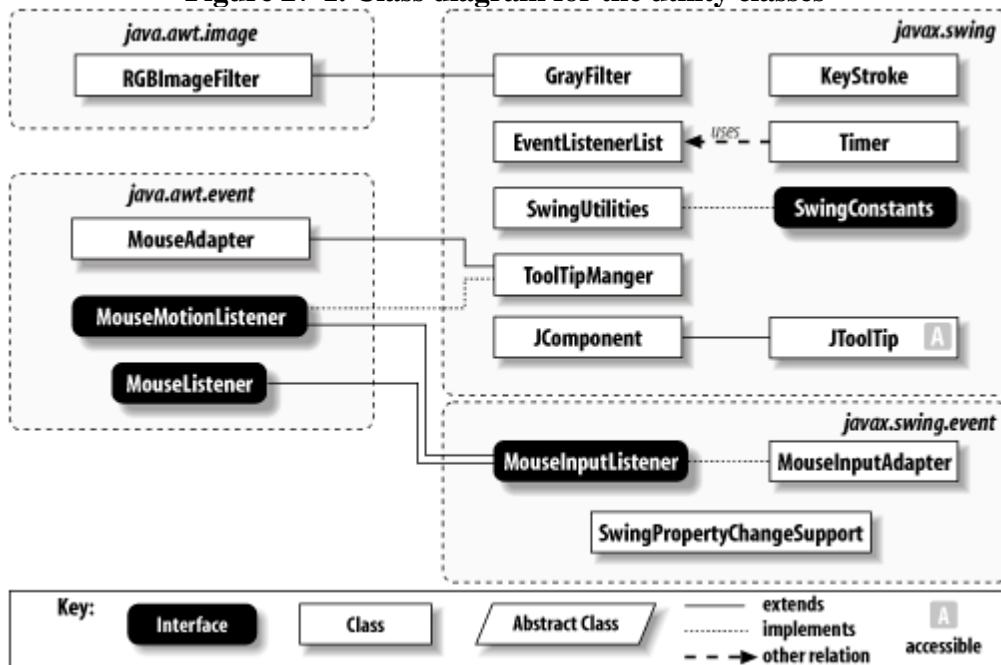
The CellRendererPane, Renderer, and GrayFilter classes

Events

The EventListenerList, KeyStroke, MouseInputAdapter, and SwingPropertyChangeSupport classes

[Figure 27-1](#) shows the classes covered in this chapter.

Figure 27-1. Class diagram for the utility classes



27.1 Utility Classes

The utilities presented here are meant to be used with any part of your application. The constants from SwingConstants and static methods of the SwingUtilities class are used throughout the Swing source code and will probably be useful to you as well. You'll find a lot of these utilities fairly straightforward, and maybe even easy to reproduce with your own code. But try to familiarize yourself with these APIs; they're meant to keep you from reinventing those wheels with each new application you write.

27.1.1 The SwingUtilities Class

This class serves as a collection point for several methods common in more advanced GUI development projects. You probably won't use all of the methods in any one application, but some of the methods will doubtless come in handy from time to time. While the purpose of many of these methods is obvious from their signatures, here's a brief description of the utility calls at your disposal. For a more detailed discussion of the invokeLater() and invokeAndWait() methods, check out [Chapter 28](#).

27.1.1.1 Constructor

public SwingUtilities()

The constructor for SwingUtilities is public, but all of the public methods are static, so you do not need to create an instance.

27.1.1.2 Class methods

public static Rectangle calculateInnerArea (JComponent c, Rectangle r)

Calculate the position and size of the inner area (excluding the border) of c. The bounds are placed in r and r is returned. This method was introduced in SDK 1.4.

public static Rectangle[] computeDifference(Rectangle rectA, Rectangle rectB)

Return the regions in rectA that do not overlap with rectB. If rectA and rectB do not overlap at all, an empty array is returned. The number of rectangles returned depends on the nature of the intersection.

public static Rectangle computeIntersection(int x, int y, int width, int height, Rectangle dest)

Return the intersection of two rectangles (the first represented by (x, y, width, height)), without allocating a new rectangle. Instead, dest is modified to contain the intersection and then returned. This can provide a significant performance improvement over the similar methods available directly through the Rectangle class, if you need to do several such intersections.

public static int computeStringWidth(FontMetrics fm, String str)

Given a particular font's metrics, this method returns the pixel length of the string str.

public static Rectangle computeUnion(int x, int y, int width, int height, Rectangle dest)

Return the union of the rectangle represented by (x, y, width, height) and dest. As with computeIntersection(), dest is modified and returned; no new Rectangle object is allocated.

public static MouseEvent convertMouseEvent(Component source, MouseEvent sourceEvent, Component destination)

Return a new MouseEvent, based on sourceEvent with the (x, y) coordinates translated to destination's coordinate system and the source of the event set as destination, provided destination is not null. If it is, source is set as the source for the new event. The actual translation of (x, y) is done with convertPoint().

public static Point convertPoint(Component source, Point aPoint, Component destination) public static Point convertPoint(Component source, int x, int y, Component destination)

27.2 The Timer Class

The Timer class provides a mechanism to generate timed events. It has properties and events, and thus can be used in application builders that understand JavaBeans. It fires an ActionEvent at a given time. The timer can be set to repeat, and an optional initial delay can be set before the repeating event starts.

27.2.1 Properties

The Timer class properties give you access to the timer delays and nature of the event firing loops. They are listed in [Table 27-2](#). The delay property dictates the length between repeated timer events (if repeats is true) and initialDelay determines how long to wait before starting the regular, repeating events. Both properties expect values in milliseconds. If your timer is not repeating, then the value of initialDelay determines when the timer fires its event. You can check to see if the timer is running with the running property. The coalesce property dictates whether or not the timer combines pending events into one single event (to help listeners keep up). For example, if the timer fires a tick every 10 milliseconds, but the application is busy and has not handled events for 100 milliseconds, 10 action events are queued up for delivery. If coalesce is false, all 10 of these are delivered in rapid succession. If coalesce is true (the default), only one event is fired. The logTimers property can be turned on to generate simple debugging information to the standard output stream each time an event is processed.

Table 27-2. Timer properties

Property	Data type	get	is	set	Default value
actionListeners1.4	ActionListener[]				Empty array
delay	int				From constructor
coalesce	boolean				true
initialDelay	int				this.delay
logTimers	boolean				false
repeats	boolean				true
running	boolean				false
1.4since 1.4					

27.2.2 Events

27.3 Tooltips

You have probably already seen several examples of using basic tooltips with components such as JButton and JLabel. The following classes give you access to much more of the tooltip system in case you need to develop something beyond simple text tips.

27.3.1 The ToolTipManager Class

This class manages the tooltips for an application. Following the singleton pattern, any given virtual machine will have, at most, one ToolTipManager at any time—a new instance is created when the class is loaded. (Of course, you're already familiar with Design Patterns by Gamma et al., right?) You can retrieve the current manager using the ToolTipManager.sharedInstance() method.

27.3.1.1 Properties

The ToolTipManager properties shown in [Table 27-3](#) give you control over the delays (in milliseconds) involved in showing tooltips and determines whether or not tooltips are even active. The enabled property determines whether or not tooltips are active. The dismissDelay property determines how long a tooltip remains on the screen if you don't do something to dismiss it manually (such as move the mouse outside the component's borders). The initialDelay property determines how long the mouse must rest inside a component before the tooltip pops up, and reshownDelay determines how long you must wait after leaving a component before the same tooltip shows up again when you reenter the component. (These properties are used by a timer whose delays are triggered by mouse events.) If the lightWeightPopupEnabled property is true, all-Java tooltips are used. A false value indicates native tooltips should be used.

Table 27-3. ToolTipManager properties

Property	Data type	get	is	set	Default value
dismissDelay	int				4000
enabled	boolean				true
initialDelay	int				750
lightWeightPopupEnabled	boolean				true
reshowDelay	int				500

27.3.1.2 Miscellaneous methods

If need be, you can manually register or unregister components with the manager. Normally this is done using the IComponent.setToolTipText() method for the component itself. If you pass in a non-null tip string, the component is

27.4 Rendering Odds and Ends

Both the JTree and JTable classes make use of cell renderers to display cells and cell editors to modify cell values. The following classes round out the utilities available for rendering information.

27.4.1 The CellRendererPane Class

This utility class was built to keep renderers from propagating repaint() and validate() calls to the components using renderer components such as JTree and JList. If you played around with creating your own renderers for any of the Swing components that use them, you'll recall that you did not use this class yourself. Normally this pane is wrapped around the renderer and the various paintComponent() methods below are used to do the actual drawing. Developers do not normally need to worry about this class.

27.4.2 The Renderer Interface

The Swing package includes a Renderer interface (which does not appear to be used within Swing itself) with the following methods:

public Component getComponent()

Return a Component you can use with something like the SwingUtilities.paintComponent() method to draw the value on the screen.

public void setValue(Object aValue, boolean isSelected)

This method can initialize the rendering component to reflect the state of the object aValue.

This interface could be useful if you were to create a library of renderers for use with your own applications; however, it is not implemented anywhere in the Swing package as of SDK 1.4.1.

27.4.3 The GrayFilter Class

The GrayFilter class is an extension of the java.awt.image.RGBImageFilter class. This class contains a static method that returns a "disabled" version of an image passed in. The image is converted to a grayscale version, and some lighter parts of the image are amplified to ensure the image is recognizable. All of the components that can display images use this class to present a default disabled version of the image if an explicit disabled image is not provided.

If you want the gory details on image manipulation with Java 2, check out Java 2D Graphics by Jonathan Knudsen (O'Reilly & Associates).

27.4.3.1 Constructor

public GrayFilter(boolean brighter, int percent)

Create an instance of the GrayFilter class that you can use to do your own filtering. (Normally you don't call this, but use createDisabledImage() instead.) Both the brighter and percent arguments are used to convert color pixels to appropriately shaded gray pixels.

27.4.3.2 Image methods

public static Image createDisabledImage(Image i)

Use this method to retrieve a grayed-out version of the image i. This method creates an instance of the GrayFilter class with brighter turned on and a gray percent of 50.

public static Image createImage(int width, int height, int depth, ImageObserver observer)

27.5 Event Utilities

If you extend one of the Swing components to add functionality, or indeed, build your own component from scratch, you need to handle event listeners for any events you might generate. The `EventListenerList` class (from the `javax.swing.event` package) is designed to aid in that task. This class is similar in many ways to the `AWTEventMulticaster`; however, it supports any type of listener and assumes you'll use only the appropriate listeners for a given event type.

The `KeyStroke` class can also help handle keyboard events. Rather than listening to every key that gets pressed and throwing out the things you don't care about, you can use the `KeyStroke` class to register specific actions with specific keys. The `MouseInputAdapter` can help deal with the other common low-level event generator: the mouse. And last but not least, this section also covers the `SwingPropertyChangeSupport` class to show you a fast way of generating property change events.

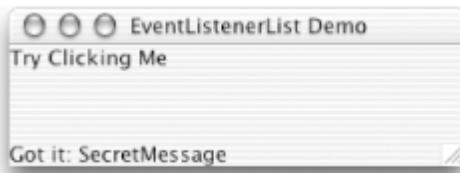
27.5.1 The `EventListenerList` Class

If your component generates events, it must contain methods to add and remove interested listeners. Following the JavaBeans design patterns, these are the `addTypeListener()` and `removeTypeListener()` methods. Typically you store the listeners in a collection, and then use the vector as a rollcall for who to send events to when the time comes. This is a very common task for components that generate events, and the `EventListenerList` can help lift some (but certainly not all) of the burden of coding the event firing.

The `EventListenerList` stores listeners as pairs of objects: one object to hold the listener's type and one to hold the listener itself. At any time, you can retrieve all of the current listeners as an array of `Objects` and use that array to fire off any events you need.

Here is a `SecretLabel` class that extends `JLabel` and fires `ActionEvent` messages when clicked. The label does not give any indication it has been clicked; that's why it's called secret. The code for this label demonstrates how an `EventListenerList` is typically used. [Figure 27-3](#) shows the `SecretLabel` up and running.

Figure 27-3. A `JLabel` that uses an `EventListenerList` to facilitate dispatching events



We set up the standard `addActionListener()` and `removeActionListener()` methods, which delegate to the listener list, and pass in the type of listener we're attaching. (Recall that `EventListenerList` can store any type of listener.) When we actually fire an event, we search the listener list array, checking the even-numbered indices for a particular type of listener. If we find the right type (`ActionListener`, in this case) we use the next entry in the list as an event recipient. You can find other such examples throughout the Swing package in such models as `DefaultTreeModel`, `DefaultTableModel`, and `DefaultButtonModel`.

The `EventListenerList` provides a generic dispatcher that works in just about every situation. However, it is not the only way to dispatch events. For a particular application you may find a more efficient means.

Chapter 28. Swing Under the Hood

While writing the first edition of this book, we sent mail to several Java newsgroups asking what topics developers would like to see explained in more detail. We received a tidal wave of responses, encompassing some rather arcane parts of Swing. Of those that were not covered elsewhere in the book, most of the replies concerned the same five areas:

- Understanding focus in Swing and altering the flow of focus over components
- Multithreading issues in Swing
- Mixing lightweight and heavyweight (Swing and AWT) components in Swing
- The Swing RepaintManager
- Creating your own Swing component

In response to your requests, we offer this collection of tips, tricks, and things that made us cry "Eureka!" at about two in the morning. Since many of these topics had little or no documentation when Swing was created, the authors had to dig through the Swing source code much more than normal to answer these questions. Hence, we called this chapter "Swing Under the Hood." As Swing has evolved, we've updated it to reflect today's realities and to incorporate new content.

28.1 Working with Focus

Prior to SDK 1.4, focus issues in Java were complicated and not adequately implemented. The problems were deeper than bugs in the code—of which there were certainly plenty, many due to inconsistencies between platforms. The design itself was fundamentally flawed.[\[1\]](#)

[1] For more details about the deficiencies and how they have been remedied, see Sun's article, "The AWT Focus Subsystem for Merlin" at <http://java.sun.com/products/jfc/tsc/articles/merlin/focus/>.

Because of these deep changes, the "right way" to do things now differs substantially from previous versions of Java. To avoid confusion, we discuss only the focus model introduced in 1.4. Developers who work with earlier releases should refer to the book's web site (<http://www.oreilly.com/catalog/jswing2>) for the first edition's version of this section.

Here are some highlights of changes in the focus system in 1.4:

- FocusManager has been deprecated and replaced by KeyboardFocusManager.
- It's now possible to learn which component currently has the focus.
- Lightweight children of Window (not just Frame or Dialog) are able to receive keyboard input.
- When focus changes, the component gaining focus can find out which one lost it, and vice versa.
- There is far less platform-dependent code; it has been replaced by a very extensible public API in AWT, with many levels where custom logic can be plugged in. Heavyweight and lightweight focus is much better integrated.
- Components can lose focus temporarily (e.g., to a scrollbar).
- It's a lot easier to work with.

The net result of all these changes is that the system tends to work the way you want it to, and even when you want to add fancy new features, doing so requires less effort and simpler code.

28.1.1 Overview

Focus specifies the component that receives keyboard events when the user presses keys on the keyboard. In managing focus, it is also important to define how and when focus transfers from one component to another, and the order in which components are traversed when focus moves forward or backward. These concepts are strongly linked: if a component can receive focus at all, then it participates in a focus traversal order, and vice versa.

In Swing, the KeyboardFocusManager keeps track of these relationships and interacts appropriately with

28.2 Multithreading Issues in Swing

As we mentioned at the beginning of the book, threading considerations play a very important role in Swing. This isn't too surprising if you think about the fact that Java is a highly multithreaded environment, but there is only a single display on which all components must render themselves in an organized, cooperative way. Some of the low-level code that interacts with peers in the native windowing system is not reentrant, which means that it can't be invoked safely while it's in the middle of doing something. Of course, that's exactly what might happen if different threads interact with it! Because of this fundamental limitation, there would have been no real benefit to undertaking the major effort of making the Java methods in the Swing components themselves thread-safe—and indeed, with few exceptions, they are not.

In order to address this issue, Swing requires that all activity that interacts with components take place on a single thread, known as the event dispatch thread. This also allows Swing to consolidate repaints and otherwise improve the efficiency of the drawing process.

This restriction actually takes effect only after a component is realized, meaning that it's been painted on-screen or is ready to be painted. A top-level component is realized when it is made visible or has its pack() method called. At that point, any components it contains are also realized, and any components added to it later are immediately realized. This does mean, though, that it's safe to set up your application's interface components in your main application thread, as long as the last thing your code does to them is realize them. This is the way our code examples are structured. It is similarly safe to set up an applet's interface objects in its init() method.

However, this does place a burden on you as a developer to be aware of the threads that might invoke your methods. If it's possible that an arbitrary application thread can call a method that would interact with Swing components, that activity needs to be separated and deferred to a method that runs on the event dispatch thread.

By confining updates of component visual state to the event dispatch thread, you keep changes in sync with the repainting requests of the RepaintManager and avoid potential race conditions.

It is always safe to call the repaint() and revalidate() methods defined in JComponent. These methods arrange for work to be performed later in the event dispatch thread, using the mechanisms described below. It's also always safe to add or remove event listeners, as this has no effect on an ongoing event dispatch.

28.2.1 When Is Thread Safety an Issue?

It's important to understand when you need to worry about this and when you do not. In many situations, you'll update the state of your user interface only in response to various user events (such as a mouse click). In this case, you can freely update your components since your event-handling methods (in your listeners) are automatically invoked by the event dispatch thread.

The only time you have to worry about updating the state of a Swing component is when the request for the update comes from some other thread. The simplest (and very common) example is when you want to display an asynchronous progress bar. Or suppose you want a display driven by some outside process responsible for notifying your application when the world changes, such as a sports score ticker that shows the scores of a collection of games as they are being played. A separate server process might push new scores to your client program. This program would have some sort of socket (or higher-level protocol) listener thread to handle the input from the server. This new data needs to be reflected in the user interface. At this point, you'd need to ensure that the updates were made in the event dispatch thread.

Another common scenario involves responding to a user request that may take a long time to be processed. Such requests might need to access a database, invoke a remote method on an RMI or CORBA object, load new Java classes, etc. In situations like these, the event-handling thread must not be held up while lengthy processing takes place. If it is, the user interface would become completely unresponsive until the call completes. The correct

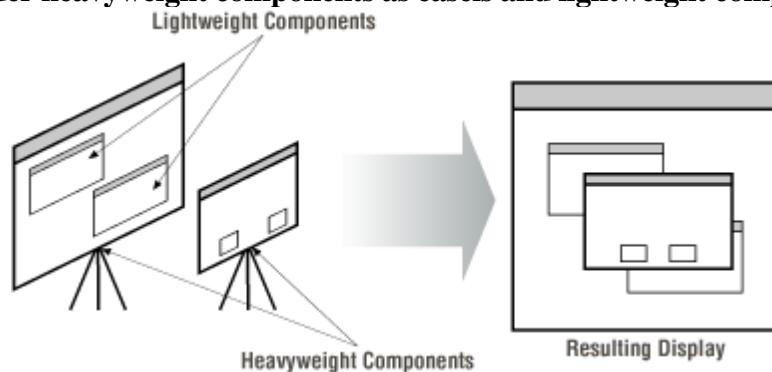
28.3 Lightweight Versus Heavyweight Components

The issue of lightweight versus heavyweight components has complicated Swing since its inception. The concept of lightweight components managed entirely by Java code is, of course, one of the major benefits introduced by Swing. What has confused the majority of programmers from the start is the issue of z-order, or layering, between Swing lightweight components and AWT heavyweight components.

28.3.1 Understanding the Z-Order

In Swing, it might help to think of a heavyweight component as an artist's easel. Top-level components in Swing (JWindow, JDialog, JApplet, and JFrame) are heavyweight, while everything else isn't. Lightweight components added to those top-level heavyweights can be thought of as drawings on the canvas of each easel. Therefore, if another heavyweight component (another easel) is moved in front of the original easel, or even attached to it, the lightweight paintings on the original easel are obscured. [Figure 28-6](#) demonstrates this analogy.

Figure 28-6. Consider heavyweight components as easels and lightweight components as drawings



The same is true for how Swing interprets the z-order of lightweight and heavyweight components, even in a container. If a heavyweight component is added to a container that has lightweight components, the heavyweight is always on top; the lightweight components must share the same z-order as the parent container. In addition, lightweight components cannot draw themselves outside the container (easel) they reside in, or they are clipped.

28.3.2 Mixing Swing and AWT

Our first bit of advice is: don't do it. If you can get around mixing the two (by solely using lightweight components), then you'll save yourself a mountain of testing grief. That being said, let's discuss some of the common problems that you're likely to run into if you decide, or are forced, to make the attempt.

28.3.2.1 Overlapping heavyweight and lightweight components

As we mentioned above, the heavyweight component always displays itself on top despite the intended z-order. The basic strategy is to ensure that lightweight components and heavyweight components in the same container do not overlap. On that note, [Table 28-2](#) shows a list of layout managers and panes that can and cannot be used to mix lightweight and heavyweight components.

Table 28-2. Heavyweight-friendly Swing layout managers and panes

Layout manager	Can be used to mix heavyweight and lightweight?
Border Layout	X

28.4 Painting and Repainting

Repainting is a fundamental task for a graphical application, but one that is rarely explained in the detail you'd expect for something so central. This section is intended to give you a better feel for how painting and repainting via the repaint manager and JComponent work. You typically do not need to get involved with the RepaintManager class, and only the extremely brave override it. However, there are some instances in which a firm understanding of the repaint manager can help avoid confusion, and, starting with SDK 1.4, Swing provides a way for savvy code to take advantage of accelerated graphics hardware when it is available.

28.4.1 Swing Responsibilities

Recall that Swing uses lightweight components, which are drawn inside heavyweight top-level containers. Since the operating system has no knowledge of lightweight components (that's what makes them lightweight), it can't help coordinate their repainting. To continue the analogy first presented in the earlier lightweight and heavyweight discussion, Swing is responsible for painting and repainting everything inside its own easels. Swing delegates this duty to a RepaintManager class, which organizes and schedules repainting when told to do so.

28.4.2 The RepaintManager Class

The RepaintManager class is responsible for keeping track of the components (and the components' parts) that have become *dirty*, which means that they need to be repainted. Note that the "dirty region" does not necessarily include the entire region of affected components, but often only portions of them. The RepaintManager is also charged with the second responsibility of revalidating components that have been marked invalid. Both responsibilities ultimately result in the same thing: redrawing the component.

There is only one RepaintManager per thread group. Like the FocusManager class, you don't instantiate one directly. Instead, the static methods `currentManager()` and `setCurrentManager()` retrieve and set the current repaint manager, respectively. (Note that there is no `RepaintManager.getCurrentManager()` method.) Once a repaint manager is activated for a thread group, it remains active until it is replaced or that thread group is shut down.

You typically access the current manager as follows:

```
RepaintManager rm = RepaintManager.currentManager();
```

At the heart of the RepaintManager are two data structures: a Hashtable of component references and their rectangular regions that need to be repainted, and a Vector of invalidated components. You can add component regions to the hashtable of dirty regions with the `addDirtyRegion()` method. Likewise, you can add a component to the invalidation vector with a call to `addInvalidComponent()`. If you wish to remove a component from the invalidation vector, you can remove it with a call to `removeInvalidComponent()`.

Here are some important rules for working with the repaint manager:

- If a component has a dirty region on the repaint queue, and another region from the same component is added, the repaint manager takes the rectangular union of the two sections. As a result, there is never more than one dirty region per component on the queue at any time.
- If a component has been invalidated with the `addInvalidComponent()` method, the RepaintManager invalidates the first ancestor of this component to return true for the `isValidateRoot()` method (typically a container). This has the desirable side effect of invalidating all the components below it.

28.5 Creating Your Own Component

So you've been bitten by the bug. There isn't a component anywhere in the Swing library that fits your needs, and you've decided that it's time to write your own. Unfortunately, you're dreading the prospect of creating one. Maybe you've heard somewhere that it is a complex task, or your jaw is still bouncing on the floor after browsing through some of the Swing component source code.

This section should help dispel those fears. Creating your own component isn't hard—just extend the `JComponent` class with one of your own and away you go! On the other hand, getting it to behave or even display itself correctly can take a bit of patience and fine-tuning. So here is a step-by-step guide to steer you clear of the hidden "gotchas" that lurk in the task of creating a component.

When creating Swing components, it's always a good idea to adhere to the JavaBeans standards. Not only can such components be used programmatically, but they can also be plugged into one of the growing number of GUI-builder tools. Therefore, whenever possible, we try to highlight areas that you can work on to make your components more JavaBeans-friendly.

28.5.1 Getting Started

First things first. If you haven't already, you should read through the `JComponent` section of [Chapter 3](#). This will help you get a feel for the kinds of features you can expect in a Swing component and which ones you might want to use (or even disable) in your own component. If you are creating a component that is intended as a container,^[2] be sure to glance at the overview sections on focus policies and layout managers as well. Remember that you can use any layout manager with a Swing component.

[2] This is sort of confusing. Because of the class hierarchy of `JComponent`, all classes that extend it are capable of acting as containers. For example, it is legal to add a `JProgressBar` to a `JSlider`. Clearly, the slider is not meant to act as a container, but Swing will allow it nevertheless . . . with undefined results.

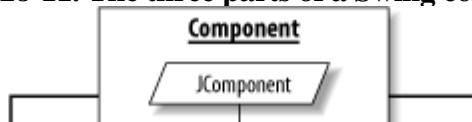
After you've done that, you're ready to start. Let's go through some steps that will help you flesh out that component idea into working Swing code.

28.5.1.1 You should have a model and a UI delegate

If you really want to develop your idea into a true Swing component, you should adhere to the MVC-based architecture of Swing. This means defining models and UI delegates for each component. Recall that the model is in charge of storing the state information for the component. Models typically implement their own model interface, which outlines the accessors and methods that the model must support. The UI delegate is responsible for painting the component and handling any input events that are generated. The UI-delegate object always extends the `ComponentUI` class, which is the base class for all UI-delegate objects. Finally, the component class itself extends the abstract `JComponent` and ties together the model and the delegate.

[Figure 28-11](#) shows the key classes and interfaces involved in creating a Swing component. The shaded boxes indicate items that the programmer must provide. This includes the model, the basic UI delegate and its type class, and an implementation of the component to bundle the model and UI delegate pieces together. Finally, you may need to create your own model interface if a suitable one does not exist.

Figure 28-11. The three parts of a Swing component



Appendix A. Look-and-Feel Resources

Table A-1 shows a complete list of the component UI resources (with the resource name and its expected value type) defined by BasicLookAndFeel. Application-wide changes can be made to these properties using UIManager.put(). For example, the following line would cause all JButtons instantiated after this call to be created with a background color of black:

```
UIManager.put("Button.background", Color.black);
```

Alternately, a custom L&F typically defines values for many of these properties in its initComponentDefaults() method. In this case, most resource values should be tagged as UIResources. For more information, see [Chapter 26](#).

Table A-1. Swing PLAF resources

Resource name	Type
AuditoryCues.cuelist	Array of cue names
AuditoryCues.allAuditoryCues	Array of cue names
AuditoryCues.noAuditoryCues	Array containing "mute"
AutitoryCues.playlist	null (override to give cues)
Button.background	Color
Button.border	Border
Button.darkShadow	Color
Button.font	Font
Button.focusInputMap	InputMap
Button.foreground	Color
Button.highlight	Color
Button.light	Color

Appendix B. Component Actions

This appendix lists the various JComponent descendants and the Actions you can pull from their respective ActionMaps. If the action is bound to a keystroke via the component's InputMaps, those associations are listed for the Metal and Mac L&Fs. The Windows and Motif L&Fs are quite similar to the Metal L&F, but if you want the definitive list, you should also check out the API document on this topic (
<yourJavaDocDir>/api/javax/swing/doc-files/Key-Index.html or the online copy of the docs (
<http://java.sun.com/j2se/1.4/docs/>).

It's important to remember that there are three input maps associated with a component—one for each of three focus states. If you recall from [Chapter 3](#), the possibilities are WHEN_FOCUSED, WHEN_ANCESTOR_OF_FOCUSED_COMPONENT, and WHEN_IN_FOCUSED_WINDOW. The titles of the tables indicate which state the table describes.

The information in this appendix is largely based on SDK 1.4, but the Mac L&F reflects the 1.3.1 release of the SDK on OS X. Some components did not exist until 1.4. Some actions are bound in 1.4 that are not bound in 1.3 (rarely vice versa, although some unbound actions did exist in 1.3 that are no longer part of 1.4). Because of this variability, we are including our Mapper application (*Mapper.java* in the utilities available on the book's web site). You can run the application, type in a class, choose the input map condition, and grab a current set of bound actions. You can also see the InputMap or the ActionMap without the associated binding if you prefer.

In Tables [Table B-1](#) through [Table B-45](#), you'll find key names with a -P or -R suffix, which stands for "pressed" or "released," respectively. Where you see the arrow keys (up, down, left, right), be aware that some keyboards have two sets of these keys. Either set should work with the bindings listed.

Finally, at the end of this appendix, we list the actions that were introduced as part of the auditory feedback mechanism introduced with SDK 1.4.

B.1 JButton

Table B-1. Bound JButton actions (when focused)

ActionMap entry	Metal L&F	Mac L&F
pressed	Space-P	Space-P
released	Space-R	Space-R

B.2 JCheckBox

Table B-2. Bound JCheckBox actions (when focused)

ActionMap entry	Metal L&F	Mac L&F
pressed	Space-P	Space-P
released	Space-R	Space-R

B.3 JCheckBoxMenuItem

Table B-3. Unbound JCheckBoxMenuItem action

doClick	
---------	--

B.4 JComboBox

Table B-4. Bound JComboBox actions (when ancestor of focused component)

ActionMap entry	Metal L&F	Mac L&F
endPassThrough	End-P	Not bound
enterPressed	Enter-P	Enter-P
hidePopup	Escape-P	Not bound
homePassThrough	Home-P	Not bound
pageDownPassThrough	Page Down-P	Not bound
pageUpPassThrough	Page Up-P	Not bound
selectNext	Down-P	Not bound
selectPrevious	Up-P	Not bound
spacePopup	Space-P	Not bound
togglePopup	Alt Down-P Alt Up-P	Not bound

B.5 JDesktopPane

Table B-5. Bound JDesktopPane actions (when ancestor of focused component)

ActionMap entry	Metal L&F	Mac L&F
close	Ctrl F4-P	Not bound
down	Down-P	Not bound
escape	Escape-P	Not bound
left	Left-P	Not bound
maximize	Ctrl F10-P	Not bound
minimize	Ctrl F9-P	Not bound
move	Ctrl F7-P	Not bound
navigateNext	Ctrl F12-P	Not bound
navigatePrevious	Ctrl+Shift F12-P	Not bound
resize	Ctrl F8-P	Not bound
restore	Ctrl F5-P	Not bound
right	Right-P	Not bound
selectNextFrame	Ctrl F6-P Ctrl+Alt F6-P Ctrl Tab-P	Not bound
selectPreviousFrame	Ctrl+Alt+Shift F6-P	Not bound
shrinkDown	Shift Down-P	Not bound

B.6 JEditorPane

Table B-6. Bound JEditorPane actions (when focused)

ActionMap entry	Metal L&F	Mac L&F
caret-backward	Left-P	Left-P
caret-begin	Ctrl Home-P	Home-PCommand Up-P
caret-begin-line	Home-P	Command Left-P
caret-begin-paragraph	Not bound	Option Up-P
caret-down	Down-P	Down-P
caret-end	Ctrl End-P	End-PCommand Down-P
caret-end-line	End-P	Command Right-P
caret-end-paragraph	Not bound	Option Down-P
caret-forward	Right-P	Right-P
caret-next-word	Ctrl Right-P	Option Right-P
caret-previous-word	Ctrl Left-P	Option Left-P
caret-up	Up-P	Up-P
copy-to-clipboard	Ctrl C-PCopy-P	Command C-P
cut-to-clipboard	Ctrl X-PCut-P	Command X-P
delete-next	Delete-P	Delete-PShift Delete-P

B.7 JFormattedTextField

Table B-9. Bound JFormattedTextField actions (when focused)

ActionMap entry	Metal L&F	Mac L&F
caret-forward	Right-P	This component does not exist in the 1.3 release of the Java 2 SDK.
selection-begin-line	Shift Home-P	
cut-to-clipboard	Ctrl X-PCut-P	
caret-backward	Left-P	
toggle-componentOrientation	Ctrl+Shift O-P	
caret-next-word	Ctrl Right-P	
delete-previous	Backspace*	
caret-previous-word	Ctrl Left-P	
unselect	Ctrl \ -P	
delete-next	Delete-P	
select-all	Ctrl A-P	
selection-end-line	Shift End-P	
notify-field-accept	Enter-P	
reset-field-edit	Escape-P	
caret-end-line	End-P	

B.8 JInternalFrame

Table B-12. Unbound JInternalFrame actions

CheckBoxMenuItem.commandSound	InternalFrame.restoreDownSound
showSystemMenu	OptionPane.questionSound
OptionPane.errorSound	PopupMenu.popupSound
InternalFrame.maximizeSound	InternalFrame.closeSound
OptionPane.informationSound	OptionPane.warningSound
InternalFrame.restoreUpSound	RadioButtonMenuItem.commandSound
InternalFrame.minimizeSound	MenuItem.commandSound

B.9 JLabel

Nothing defined.

B.10 JList

Table B-13. Bound JList actions (when focused)

ActionMap entry	Metal L&F	Mac L&F
clearSelection	Ctrl \-P	Not bound
copy	Copy-PCtrl C-P	Not bound
cut	Ctrl X-PCut-P	Not bound
paste	Ctrl V-P Paste-P	Not bound
scrollDown	Page Down-P	Page Down-P
scrollDownExtendSelection	Shift Page Down-P	Shift Page Down-P
scrollUp	Page Up-P	Page Up-P
scrollUpExtendSelection	Shift Page Up-P	Shift Page Up-P
selectAll	Ctrl /-PCtrl A-P	Command A-P
selectFirstRow	Home-P	Home-P
selectFirstRowExtendSelection	Shift Home-P	Shift Home-P
selectLastRow	End-P	End-P
selectLastRowExtendSelection	Shift End-P	Shift End-P
selectNextColumn	Right-P	Not bound

B.11 JMenu

Table B-14. Unbound JMenu actions

doClick	selectMenu
---------	------------

B.12 JMenuBar

Table B-15. Bound JMenuBar actions (when in focused window)

ActionMap entry	Metal L&F	Mac L&F
takeFocus	F10-P	Not bound

B.13 JMenuItem

Table B-16. Unbound JMenuItem action

doClick	
---------	--

B.14 JOptionPane

Table B-17. Bound JOptionPane actions (when in focused window)

ActionMap entry	Metal L&F	Mac L&F
close	Escape-P	Escape-P

B.15 JPasswordField

Table B-18. Bound JPasswordField actions (when focused)

ActionMap entry	Metal L&F	Mac L&F
caret-backward	Left-P	Left-P
caret-begin-line	Home-P	Up-PCommand Left-P
caret-end-line	End-P	Down-PCommand Right-P
caret-forward	Right-P	Right-P
caret-next-word	Ctrl Right-P	Option Right-P
caret-previous-word	Ctrl Left-P	Option Left-P
copy-to-clipboard	Ctrl C-PCopy-P	Command C-P
cut-to-clipboard	Cut-PCtrl X-P	Command X-P
delete-next	Delete-P	Delete-PShift Delete-P
delete-previous	Backspace*	Shift Backspace-PBackspace-P
notify-field-accept	Enter-P	Enter-P
paste-from-clipboard	Ctrl V-PPaste-P	Command V-P
select-all	Ctrl A-P	Command A-P
selection-backward	Shift Left-P	Shift Left-P
selection-begin-line	Shift Home-P	Shift Up-P

B.16 JPopupMenu

Nothing defined.

B.17 JProgressBar

Nothing defined.

B.18 JRadioButton

Table B-20. Bound JRadioButton actions (when focused)

ActionMap entry	Metal L&F	Mac L&F
pressed	Space-P	Space-P
released	Space-R	Space-R

B.19 JRadioButtonMenuItem

Table B-21. Unbound JRadioButtonMenuItem action

doClick	
---------	--

B.20 JRootPane

Table B-22. Unbound JRootPane actions

press	release
-------	---------



B.21 JScrollBar

Table B-23. Bound JScrollBar actions (when focused)

ActionMap entry	Metal L&F	Mac L&F
positiveUnitIncrement	Down-PRight-P	Not bound
positiveBlockIncrement	Page Down-P	Not bound
negativeUnitIncrement	Left-PUp-P	Not bound
negativeBlockIncrement	Page Up-P	Not bound
minScroll	Home-P	Not bound
maxScroll	End-P	Not bound

B.22 JScrollPane

Table B-24. Bound JScrollPane actions (when ancestor of focused component)

ActionMap entry	Metal L&F	Mac L&F
scrollDown	Page Down-P	Page Down-P
scrollEnd	Ctrl End-P	End-P
scrollHome	Ctrl Home-P	Home-P
scrollLeft	Ctrl Page Up-P	Command Left-P
scrollRight	Ctrl Page Down-P	Command Right-P
scrollUp	Page Up-P	Page Up-P
unitScrollDown	Down-P	Down-P
unitScrollLeft	Left-P	Left-P
unitScrollRight	Right-P	Right-P
unitScrollUp	Up-P	Up-P

B.23 JSlider

Table B-25. Bound JSlider actions (when focused)

ActionMap entry	Metal L&F	Mac L&F
maxScroll	End-P	End-P
minScroll	Home-P	Home-P
negativeBlockIncrement	Ctrl Page Down-PPage Down-P	Page Down-P
negativeUnitIncrement	Down-PLeft-P	Down-PLeft-P
positiveBlockIncrement	Page Up-PCtrl Page Up-P	Page Up-P
positiveUnitIncrement	Right-PUp-P	Right-PUp-P

B.24 JSpinner

Table B-26. Bound JSpinner actions (when ancestor of focused component)

ActionMap entry	Metal L&F	Mac L&F
decrement	Down-P	This component does not exist in the 1.3 release of the Java 2 SDK.
increment	Up-P	

B.25 JSplitPane

Table B-27. Bound JSplitPane actions (when ancestor of focused component)

ActionMap entry	Metal L&F	Mac L&F
negativeIncrement	Up-PLeft-P	Up-PLeft-P
positiveIncrement	Down-PRight-P	Down-PRight-P
selectMax	End-P	End-P
selectMin	Home-P	Home-P
startResize	F8-P	F8-P
toggleFocus	F6-P	F6-P

B.26 JTabbedPane

Table B-28. Bound JTabbedPane actions (when focused)

ActionMap entry	Metal L&F	Mac L&F
navigateDown	Down-P	Down-P
navigateLeft	Left-P	Left-P
navigateRight	Right-P	Right-P
navigateUp	Up-P	Up-P
requestFocusForVisibleComponent	Ctrl Down-P	Command Down-P

Table B-29. Bound JTabbedPane actions (when ancestor of focused component)

ActionMap entry	Metal L&F	Mac L&F
navigatePageDown	Ctrl Page Down-P	Command Page Down-P
navigatePageUp	Ctrl Page Up-P	Command Page Up-P
requestFocus	Ctrl Up-P	Command Up-P

Table B-30. Unbound JTabbedPane actions

navigateNext	scrollTabsForwardAction
navigatePrevious	setSelectedIndex
scrollTabsBackwardAction	

B.27 JTable

Table B-31. Bound JTable actions (when ancestor of focused component)

ActionMap entry	Metal L&F	Mac L&F
cancel	Escape-P	Escape-P
copy	Copy-PCtrl C-P	Not bound
cut	Ctrl X-PCut-P	Not bound
paste	Ctrl V-PPaste-P	Not bound
scrollDownChangeSelection	Page Down-P	Page Down-P
scrollDownExtendSelection	Shift Page Down-P	Shift Page Down-P
scrollLeftChangeSelection	Ctrl Page Up-P	Command Left-P
scrollLeftExtendSelection	Ctrl+Shift Page Down-P	Command+Shift Right-P
scrollRightChangeSelection	Ctrl Page Down-P	Command Right-P
scrollRightExtendSelection	Ctrl+Shift Page Up-P	Command+Shift Left-P
scrollUpChangeSelection	Page Up-P	Page Up-P
scrollUpExtendSelection	Shift Page Up-P	Shift Page Up-P
selectAll	Ctrl A-P	Command A-P
selectFirstColumn	Home-P	Home-P
selectFirstColumnExtendSelection	Shift Home-P	Not bound

B.28 JTextArea

Table B-32. Bound JTextArea actions (when focused)

ActionMap entry	Metal L&F	Mac L&F
caret-backward	Left-P	Left-P
caret-begin	Ctrl Home-P	Home-PCommand Up-P
caret-begin-line	Home-P	Command Left-P
caret-begin-paragraph	Not bound	Option Up-P
caret-down	Down-P	Down-P
caret-end	Ctrl End-P	End-PCommand Down-P
caret-end-line	End-P	Command Right-P
caret-end-paragraph	Not bound	Option Down-P
caret-forward	Right-P	Right-P
caret-next-word	Ctrl Right-P	Option Right-P
caret-previous-word	Ctrl Left-P	Option Left-P
caret-up	Up-P	Up-P
copy-to-clipboard	Ctrl C-PCopy-P	Command C-P
cut-to-clipboard	Ctrl X-PCut-P	Command X-P
delete-next	Delete-P	Delete-PShift Delete-P

B.29 JTextField

Table B-35. Bound JTextField actions (when focused)

ActionMap entry	Metal L&F	Mac L&F
caret-backward	Left-P	Left-P
caret-begin-line	Home-P	Up-PCommand Left-P
caret-end-line	End-P	Down-PCommand Right-P
caret-forward	Right-P	Right-P
caret-next-word	Ctrl Right-P	Option Right-P
caret-previous-word	Ctrl Left-P	Option Left-P
copy-to-clipboard	Ctrl C-PCopy-P	Command C-P
cut-to-clipboard	Cut-PCtrl X-P	Command X-P
delete-next	Delete-P	Delete-PShift Delete-P
delete-previous	Backspace*	Shift Backspace-PBackspace-P
notify-field-accept	Enter-P	Enter-P
paste-from-clipboard	Ctrl V-PPaste-P	Command V-P
select-all	Ctrl A-P	Command A-P
selection-backward	Shift Left-P	Shift Left-P
selection-begin	Not bound	Shift Up-P

B.30 JTextPane

Table B-37. Bound JTextPane actions (when focused)

ActionMap entry	Metal L&F	Mac L&F
caret-backward	Left-P	Left-P
caret-begin	Ctrl Home-P	Home-PCommand Up-P
caret-begin-line	Home-P	Command Left-P
caret-begin-paragraph	Not bound	Option Up-P
caret-down	Down-P	Down-P
caret-end	Ctrl End-P	End-PCommand Down-P
caret-end-line	End-P	Command Right-P
caret-end-paragraph	Not bound	Option Down-P
caret-forward	Right-P	Right-P
caret-next-word	Ctrl Right-P	Option Right-P
caret-previous-word	Ctrl Left-P	Option Left-P
caret-up	Up-P	Up-PUp-P
copy-to-clipboard	Ctrl C-PCopy-P	Command C-P
cut-to-clipboard	Ctrl X-PCut-P	Command X-P
delete-next	Delete-P	Delete-PShift Delete-P

B.31 JToggleButton

Table B-40. Bound JToggleButton actions (when focused)

ActionMap entry	Metal L&F	Mac L&F
pressed	Space-P	Space-P
released	Space-R	Space-R

B.32 JToolBar

Table B-41. Bound JToolBar actions (when ancestor of focused component)

ActionMap entry	Metal L&F	Mac L&F
navigateDown	Down-P	Down-P
navigateLeft	Left-P	Left-P
navigateRight	Right-P	Right-P
navigateUp	Up-P	Up-P

B.33 JToolTip

Nothing defined.

B.34 JTree

Table B-42. Bound JTree actions (when focused)

ActionMap entry	Metal L&F	Mac L&F
clearSelection	Ctrl \-P	Command \-P
copy	Copy-PCtrl C-P	Not bound
cut	Ctrl X-PCut-P	Not bound
extendSelection	Shift Space-P	Shift Space-P
paste	Ctrl V-PPaste-P	Not bound
scrollDownChangeLead	Ctrl Page Down-P	Command Page Down-P
scrollDownChangeSelection	Page Down-P	Page Down-P
scrollDownExtendSelection	Shift Page Down-PCtrl+Shift Page Down-P	Shift Page Down-PCommand+Shift Page Down-P
scrollLeft	Ctrl Left-P	Command Left-P
scrollRight	Ctrl Right-P	Command Right-P
scrollUpChangeLead	Ctrl Page Up-P	Command Page Up-P
scrollUpChangeSelection	Page Up-P	Page Up-P
scrollUpExtendSelection	Ctrl+Shift Page Up-PShift Page Up-P	Command+Shift Page Up-PShift Page Up-P
selectAll	Ctrl /-PCtrl A-P	Command A-P

I1@ve RuBoard

NEXT ►



B.35 JVViewport

Nothing defined.

I1@ve RuBoard

◀ PREVIOUS

NEXT ►

B.36 Non-JComponent Containers

The top-level containers such as `JDialog`, `JFrame`, and `JWindow`—and `JApplet`, technically—are not descendants of `JComponent`. As such, they don't have their own input or action maps. Certainly, their contained components have those maps. They also all have an active `JRootPane`, which has a few unbound actions (see [Table B-22](#)).

This is not to say that there aren't keystrokes that affect these components—they are simply implemented using other mechanisms.

B.37 Auditory Feedback Actions

A general framework for providing auditory feedback in response to user interface actions was introduced in Version 1.4. BasicLookAndFeel (as described in [Chapter 26](#)) provides most of the support, and other L&Fs extend it in order to actually play appropriate sounds. As of SDK 1.4.1, none of the standard L&Fs turn on any sounds by default. Comments in the source code indicate that this has been delayed due to sound bugs.

The list of possible actions to which a sound can be assigned is contained in a special ActionMap managed by BasicLookAndFeel. It is protected since only subclasses are likely to need access to it. They can obtain (or create, if necessary) the map by calling `getAudioActionMap()`. The keys in this map are shown in [Table B-45](#). The values are Actions that produce appropriate sounds when they are performed. The standard L&Fs define inner AudioAction classes for this purpose.

Table B-45. Audio actions defined by BasicLookAndFeel

CheckBoxMenuItem.commandSound	InternalFrame.closeSound
InternalFrame.maximizeSound	InternalFrame.minimizeSound
InternalFrame.restoreDownSound	InternalFrame.restoreUpSound
MenuItem.commandSound	OptionPane.errorSound
OptionPane.informationSound	OptionPane.questionSound
OptionPane.warningSound	PopupMenu.popupSound
RadioButtonMenuItem.commandSound	

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of Java Swing, Second Edition, is a spider monkey (*Ateles geoffroyi*). Most spider monkeys can be found in the forests of Central America from Southern Mexico to Panama. Almost all varieties of spider monkeys live exclusively in trees and maintain a diet of fruit and nuts.

What gives the spider monkey its name is its long limbs and tail (it sometimes resembles a spider as it moves). *A. geoffroyi*'s fur is black, brown, golden, or reddish.

Spider monkeys are social and can form groups of approximately 30 animals. They live in treetops and forage diurnally in troops often led by females, which have a more active role than males in the food-gathering process. Spider monkeys are often seen hanging by one branch or by their unusually long tails, which basically function as a fifth limb. They can even grasp objects with their tails.

When approached or threatened, spider monkeys will bark and flail wildly, which usually scares off intruders. If this tactic is unsuccessful, they will break away from their groups and retreat.

Matt Hutchinson was the production editor and copyeditor for Java Swing, Second Edition . Matt Hutchinson and Mary Brady proofread the book. Tatiana Apandi Diaz and Sarah Sherman provided quality control. Genevieve d'Entremont and Andrew Savikas provided production assistance. Brenda Miller updated the index from the first edition.

Hanna Dyer designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from the Dover Pictorial Archive. Emma Colby produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

David Futato designed the interior layout. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. The tip and warning icons were drawn by Christopher Bing. This colophon was written by Matt Hutchinson.

The online edition of this book was created by the Safari production group (John Chodacki, Becki Maisch, and Madeleine Newell) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, and Jeff Liggett.

I1@ve RuBoard

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]
]

I1@ve RuBoard

I1@ve RuBoard

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]
]

"Easter eggs"

<applet> tag (HTML) 2nd

2D API

I1@ve RuBoard

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]
]

[AbstractAction class](#)

properties table for

[AbstractBorder class](#) 2nd

custom borders and

properties table for

[AbstractButton class](#)

facilitating manager for

menus and 2nd

[AbstractColorChooserPanel class](#)

properties table for

[AbstractDocument class](#)

Content interface

[GapContent class](#)

inner classes/interfaces of

[PlainDocument class](#)

properties table for

[StringContent class](#)

[AbstractLayoutCache class](#)

[AbstractListModel class](#) 2nd

[AbstractSpinnerModel class](#)

[AbstractTableModel class](#) 2nd

[PagingModel.java \(example\)](#)

properties table for

[AbstractUndoableEdit class](#)

examples of 2nd

properties table for

[AbstractUndoableEdit interface](#)

[AbstractWriter class](#)

properties table for

accelerator keys, labels and

[accelerator property \(JMenuItem \)](#)

accelerators [See keyboard accelerators]

accept()

[FileFilter class](#)

[JFileChooser class](#)

[acceptAllFileFilter property \(JFileChooser \)](#)

[acceptAllFileFilterUsed property \(JFileChooser \)](#)

acceptDrag()

copy/move actions and

[DropTargetDragEvent class](#)

[acceptDrop\(\) \(DropTargetDropEvent\)](#)

accessibility 2nd

examples of 2nd

hierarchy of

how it works

interfacing with

Java Accessibility Helper for

javax.accessibility package for

multiple L&F support for

types of

utility classes for

[Accessibility API](#)

[Accessibility package](#)

accessibility roles

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)])

[background images](#)

[background property](#)

[AccessibleComponent interface](#)

[DefaultTableCellRenderer class](#)

[JComponent class](#)

[JInternalFrame class](#)

[JRootPane class](#)

[backingStoreEnabled property \(JViewport\)](#)

[BadLocationException class](#)

[base property](#)

[HTMLDocument class](#)

[StyleSheet class](#)

[baseFontSize property \(StyleSheet\)](#)

[BasicGraphicsUtils class](#)

[BasicJogShuttleUI.java \(example\)](#)

[BasicLookAndFeel class](#)

[component UI resources defined by \(list\)](#)

[BeepAction class](#)

[beginDraggingFrame\(\)](#)

[DefaultDesktopManager class](#)

[DesktopManager interface](#)

[beginIndex property \(Segment\)](#)

[beginResizingFrame\(\)](#)

[DefaultDesktopManager class](#)

[DesktopManager interface](#)

[beginUpdate\(\) \(UndoableEditSupport\)](#)

[bevel borders 2nd 3rd](#)

[BevelBorderUIResource class](#)

[examples of](#)

[BevelBorder class](#)

[properties table for](#)

[BevelBorderUIResource class](#)

[BevelExample.java \(example\)](#)

[bevels](#)

[bevelType property \(BevelBorder, SoftBevelBorder\)](#)

[bias](#)

[bidiRootElement property \(AbstractDocument\)](#)

[BigRenderer.java \(example\)](#)

[binding mechanism](#)

[blinkRate property \(Caret\)](#)

[blockIncrement property \(JScrollbar\)](#)

[BoldAction class](#)

[boolean property \(JProgress\)](#)

[Border interface 2nd 3rd](#)

[border property](#)

[JComponent class 2nd 3rd](#)

[TitledBorder class](#)

[BorderFactory class](#)

[borderOpaque property](#)

[AbstractBorder class](#)

[EtchedBorder class](#)

[LineBorder class](#)

[MatteBorder class](#)

[\[Soft\]BevelBorder classes](#)

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]
]

[calculateAlignedPositions\(\) \(SizeRequirements\)](#)

[calculateInnerArea\(\) \(SwingUtilities\)](#)

[calculateTiledPositions\(\) \(SizeRequirements\)](#)

[calendarField property \(SpinnerDateModel\)](#)

[Cancel button for halting long operations](#)

[cancelCellEditing\(\) \(CellEditor\)](#)

[canceled property \(ProgressMonitor\)](#)

[cancelSelection\(\) \(JFileChooser\)](#)

[canImport\(\) \(TransferHandler\)](#)

[CannotRedoException class](#) 2nd

[CannotUndoException class](#) 2nd

[canRedo\(\)](#)

[AbstractUndoableEdit class](#)

[CompoundEdit class](#)

[UndoableEdit interface](#)

[UndoManager class](#)

[canUndo\(\)](#)

[AbstractUndoableEdit class](#)

[CompoundEdit class](#)

[UndoableEdit interface](#)

[UndoManager class](#)

[canUndoOrRedo\(\) \(UndoManager\)](#)

[capacity\(\) \(DefaultListModel\)](#)

[CardLayout class \(AWT\)](#)

[Caret interface](#)

[properties table for](#)

[caret property \(JTextComponent\)](#)

[caretColor property \(JTextComponent\)](#)

[CaretEvent class](#)

[properties table for](#)

[CaretListener interface](#)

[caretPosition property \(JTextComponent\)](#)

[carets](#)

[constants for, drag and drop functionality](#)

[custom](#)

[dynamic, Drag and Drop support for](#)

[examples of](#) 2nd

[positioning and](#)

[caretUpdate\(\) \(CaretListener\)](#)

[cascading stylesheets \(CSS\)](#)

[case](#)

[CCPHandler.java \(example\)](#)

[cell renderers](#) 2nd 3rd

[example of](#)

[CellEditor interface](#)

[methods for](#)

[cellEditor property](#)

[JTable class](#)

[JTree class](#)

[TableColumn class](#)

[CellEditorListener interface](#)

[cellEditorValue property](#)

[DefaultCellEditor class](#)

[DefaultTreeCellEditor class](#)

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]
]

damage() (DefaultCaret)

data in tables [See tables]

data input area, in dialogs

data lists

data transfer, example of

data types, property tables and

database records in tables

advanced examples of

DatabaseTest.java (example)

date and time, formatting

date property (SpinnerDateModel)

date spinners

DateEditor class (JSpinner)

properties table for

DateFormatter class

deactivateFrame()

DefaultDesktopManager class

DesktopManager interface

DebugGraphics class

debugGraphicsOption property (JComponent)

DebugHTMLEditorKit.java (example)

decrIndent() (AbstractWriter)

default values, property tables and

defaultAction property (Keymap) 2nd

DefaultBoundedRangeModel class

defaultButton property

JButton class

JRootPane class

DefaultButtonExample.java (example)

DefaultButtonModel class

defaultCapable property (JButton)

DefaultCaret class

CornerCaret.java (example)

DefaultCellEditor class 2nd

properties table for

defaultCloseOperation property

JDialog class

JFrame class

JInternalFrame class

DefaultColorSelectionModel class

DefaultComboBoxModel class

defaultCursor property (HTMLEditorKit)

DefaultDesktopManager class

DefaultDocumentEvent class (AbstractDocument)

defaultDragSource property (DragSource)

DefaultEditor class (JSpinner)

properties table for

DefaultEditorKit class

CopyAction class

properties table for

defaultFocusTraversalKeys property (KeyboardFocusManager)

defaultFocusTraversalPolicy property (KeyboardFocusManager)

DefaultFormatter class

properties table for

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]
]

echoed borders

echoChar property (JPasswordField)

echoCharIsSet() (JPasswordField)

edges [See borders]

edit property (UndoableEditEvent)

editable property

JComboBox class 2nd

JTextComponent class 2nd

JTree class

editable state

EditableComboBox.java (example)

editCellAt() (JTable)

editing [See also editor kits]

DocumentFilter class

edit-merging methods for

HTML

killed edits and

nested edits and

table data 2nd

toggle edit functionality

trees

editing methods

for tree cells

editingCanceled() (CellEditorListener interface)

editingStopped() (CellEditorListener interface)

editor kit methods 2nd

editor kits

building custom

strategies for reading/writing documents and

DefaultEditorKit class

CopyAction class

EditorKit class 2nd

extending

IOStyledEditor.java (example)

JEditorPane class

registering with JEditorPane class

SimpleEditor.java (example)

StyledEditor.java (example)

StyledEditorKit class

TextAction class

editor panes 2nd 3rd 4th [See also editor kits]

editorComponent property (ComboBoxEditor)

EditorKit class 2nd

properties table for

editorKit property

JEditorPane class

JTextPane class

editToBeRedone() (UndoManager)

editToBeUndone() (UndoManager)

editValid property (JFormattedTextField)

EEL (Every Event Listener) 2nd

Element interface

properties table for

element lookup method

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)])

factory method for UI delegates

[FactoryDemo.java](#) (example)

[FancyButton.java](#) (example)

FancyCaret class

[FancyCaret.java](#) (example)

FieldBorder class

fields for component size

FieldView class

file chooser dialogs

accessories for

examples of [2nd](#) [3rd](#)

filters

thumbnails for file icons

viewer

[JFileChooser](#) class

file filter methods

file methods

filechooser package [2nd](#)

[FileDialog](#) class (AWT)

FileFilter class

[fileFilter](#) property ([JFileChooser](#))

[fileHidingEnabled](#) property ([JFileChooser](#))

[FileModel.java](#) (example)

files

chooser dialogs for

classes for displaying/filtering

dropping into a tree (example)

loading, monitoring progress of

[fileSelectionEnabled](#) property ([JFileChooser](#))

[fileSelectionMode](#) property ([JFileChooser](#))

FileSystemView class

[fileSystemView](#) property ([JFileChooser](#))

[FileTable.java](#) (example)

FileView class

[fileView](#) property ([JFileChooser](#))

Filler class [See Box.Filler class]

filter bypass methods

[filterRGB\(\)](#) ([GrayFilter](#))

filters

creating for [JFileChooser](#) dialogs

for file selection

[findColumn\(\)](#) ([AbstractTableModel](#))

[findFocusOwner\(\)](#) ([SwingUtilities](#))

fireActionPerformed()

[JFileChooser](#) class

[JTextField](#) class

Timer class

[fireAdjustmentValueChanged\(\)](#) ([JScrollBar](#))

[fireCaretUpdate\(\)](#) ([JTextComponent](#))

[fireChangedUpdate\(\)](#) ([AbstractDocument](#))

[fireColumn...\(\)](#) ([DefaultTableModel](#))

[fireDragGestureRecognized\(\)](#) ([DragGestureRecognizer](#))

[fireHyperlinkUpdate\(\)](#) ([JEditorPane](#))

[fireInsertUpdate\(\)](#) ([AbstractDocument](#))

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]
]

[GapContent class](#)

[gestureModifiers property \(DragSourceDragEvent\)](#)

[GestureTest.java \(example\)](#)

[get methods 2nd](#)

[property tables and](#)

[glass pane 2nd](#)

[visibility, JInternalFrame class and](#)

[GlassExample.java \(example\)](#)

[glassPane property](#)

[JApplet class](#)

[JFrame class](#)

[JInternalFrame class](#)

[JRootPane class](#)

[JWindow class](#)

[glue components 2nd](#)

[GlyphView class](#)

[Gosling, James](#)

[grabFocus\(\) \(JComponent\)](#)

[graphical user interfaces \(GUIs\), events and](#)

[graphics \[See images\]](#)

[Graphics class](#)

[graphics methods](#)

[graphics property](#)

[JComponent class](#)

[View class](#)

[GraphicsConfiguration class](#)

[GraphicsDevice class](#)

[GraphicsEnvironment class](#)

[GrayFilter class](#)

[GrayScalePanel.java \(example\)](#)

[GridLayout class, replacement for](#)

[gridColor property \(JTable\)](#)

[GridLayout class vs. BoxLayout class](#)

[group property \(ButtonModel\)](#)

[guiInitialized\(\) \(GUINitializedListener\)](#)

[GUINitializedListener interface](#)

[GUIs \(graphical user interfaces\), events and](#)

[GuiScreens.java \(example\)](#)

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]
]

half()

hasFocus() (JComponent)

hasListeners() (SwingPropertyChangeSupport)

HBox.java (example)

HBoxWithGlue.java (example)

HBoxWithStrut.java (example)

headerRenderer property (TableColumn)

headers

for scrollpanes 2nd

for table columns

for table rows

headerValue property (TableColumn)

heavyweight components 2nd 3rd

overlapping with lightweight

popup menu as

height property (JComponent)

helpMenu property (JMenuBar class)

hidePopup() (JComboBox)

hierarchy relationships, trees for

highestLayer() (JLayeredPane)

highlight color for borders

Highlight interface

highlightColor property (EtchedBorder)

Highlighter interface 2nd

properties table for

highlighter property (JTextComponent)

highlighters 2nd

adding multiple highlights and

example of

highlightInner property (BevelBorder, SoftBevelBorder)

highlightOuter property (BevelBorder, SoftBevelBorder)

HighlightPainter interface 2nd

highlights property

DefaultHighlighter class

Highlighter interface

horizontal scrollbars

horizontalAlignment property

AbstractButton class

JLabel class

JMenuItem class

JTextField class

horizontalScrollBarPolicy property (JScrollPane)

horizontalTextPosition property

JLabel class 2nd

JMenuItem class

horizontalVisibility property (JTextField)

hotspots

HTML 2nd

<applet> tag and

AccessibleHyperlink class

AccessibleHypertext interface 2nd

document methods for

HyperlinkEvent class 2nd 3rd

HyperlinkListener interface

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]
]

icon factory, defining

Icon interface 2nd

properties table for

icon parameter (JOptionPane)

icon property

AbstractButton class

JInternalFrame class

JLabel class

IconAndTipRenderer.java (example)

IconEditor.java (example)

iconifiable property (JInternalFrame)

iconifyFrame()

DefaultDesktopManager class

DesktopManager interface

IconPolice.java (example)

icons 2nd [See also images]

adding to buttons

for checkboxes

custom, example of in message dialog

defaults, defining for L&F

on desktop, managing

in dialogs 2nd

displayed on buttons 2nd

dynamic

for files/folders, FileView class for

inside frame titlebar

IconPolice.java (example)

IconView class

JDesktopIcon class

matte borders and, caution with

menu items and

for radio buttons

spinner editor for displaying

for trees 2nd

iconTextGap property (JLabel) 2nd 3rd

IconView class

ID property (LookAndFeel)

id property (MenuDragMouseEvent)

identifier property (TableColumn)

image buttons [See icons]

image magnifier pane

image methods

image property (ImageIcon)

ImageAction class

imageHeight property (ImageIcon)

ImageIcon class

properties table for

ImageLabelExample.java (example)

imageLoadStatus property (ImageIcon)

imageObserver property (ImageIcon)

images [See also icons]

background

on buttons, adding

debugging

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]
]

JApplet class

properties table for

Java Accessibility Helper (Sun)

Java Development Kit [See JDK SDK]

Java Foundation Classes (JFC)

Java Plug-in

Java Swing [See Swing]

Java virtual machine [See JVM]

java.awt.datatransfer package

java.awt.dnd package

Javadoc, static methods and

javax.swing package

preparing for use

javax.swing.text package

javax.swing.text.html package, View classes in

javax.swing.undo package

JButton class

actions and

icons, adding to JButtons buttons

properties table for

replacing AWT Buttons with JButtons

JCheckBox class

actions and

properties table for

JCheckBoxMenuItem class 2nd

actions and

properties table for

JColorChooser class 2nd

AbstractColorChooserPanel class

DefaultColorSelectionModel class

FontPicker.java (example)

javax.swing.colorchooser package for 2nd

JComboBox class

actions and

internal methods for

properties table for

JComponent class

building your own components

component actions and (list)

look-and-feel and

properties table for 2nd

JComponent createEditor() (JSpinner)

JDesktopIcon class

JDesktopPane class 2nd

actions and

properties table for

sample desktop and

using internal frame dialogs with

JDialog class

properties table for

JDK (Java Development Kit) 2nd [See also SDK] 3rd

Release 1.1

vs. SDK

JEditorPane class

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]

[key bindings](#)

[key selection manager](#)

[keyboard](#)

[DefaultKeyTypedAction class](#)

[functionality for, adding to components](#)

[keyboard accelerators \(shortcuts\)](#) [See also [keystrokes](#)] [2nd](#)

[JMenu objects and](#)

[menu items and](#) [2nd](#) [3rd](#)

[keyboard actions](#) [2nd](#)

[example of](#)

[online material for](#)

[keyboard events](#) [2nd](#) [3rd](#)

[focus and](#) [2nd](#)

[MenuKeyEvent class](#)

[MenuKeyListener interface](#)

[keyboard-driven selection](#)

[KeyboardFocusManager class](#)

[keyChar property](#)

[KeyStroke class](#)

[MenuKeyEvent class](#)

[keyCode property](#)

[KeyStroke class](#)

[MenuKeyEvent class](#)

[KeyEventDispatcher list](#)

[KeyEventPostprocessor list](#)

[Keymap interface](#) [2nd](#)

[properties table for](#)

[keymap property \(JTextComponent\)](#)

[KeymapExample.java \(example\)](#)

[keymaps](#) [2nd](#)

[implementation for](#)

[keys\(\)](#)

[ActionMap class](#)

[InputMap class](#)

[KeySelectionManager interface](#)

[KeyStroke class](#) [2nd](#)

[properties table for](#)

[keystrokes](#) [2nd](#) [See also [keyboard accelerators](#)] [3rd](#)

[mapping](#)

[to item selections in combo boxes](#)

[to logical action names](#)

[killed edits](#)

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)])

[L&F](#) [See look-and-feel]

[label](#) property ([JPopupMenu](#))

[labelFor](#) property ([JLabel](#)) 2nd

[labels](#)

[border around](#)

[ClockLabel.java](#) (example)

[content alignment and](#) 2nd

[content position and](#)

[examples of](#) 2nd 3rd

[images in](#)

[JLabel class](#)

[layered](#)

[SimpleJLabelExample.java](#) (example)

[for sliders](#) 2nd

[labelTable](#) property ([JSlider](#))

[LabelView class](#)

[LAFState class](#)

[largeModel](#) property ([JTree](#))

[last\(\)](#) ([Segment](#))

[lastDividerLocation](#) property ([JSplitPane](#))

[lastEdit\(\)](#) ([CompoundEdit](#))

[lastElement\(\)](#) ([DefaultListModel](#))

[lastIndexOf\(\)](#) ([DefaultListModel](#))

[lastRow](#) property ([TableModelEvent](#))

[lastVisibleIndex](#) property ([JList](#))

[layer](#) property ([JInternalFrame](#))

[layered labels](#), [OverlayLayout](#) class for creating

[layered pane](#) 2nd

[JLayeredPane class](#)

[methods for](#)

[LayeredHighlighter class](#)

[LayeredHighlighter.LayerPainter class](#)

[layeredPane](#) property

[JApplet class](#)

[JFrame class](#)

[JInternalFrame class](#)

[JWindow class](#)

[layers](#) [See also panes]

[adding components to](#)

[avoiding unnecessary](#)

[current, for frame](#)

[management methods for](#)

[layout managers](#) 2nd

[classes for](#)

[overlapping heavyweight/lightweight components](#)

[overrides by](#) 2nd

[read/write alignment properties and](#)

[ScrollPaneLayout class](#)

[size calculations for](#)

[ViewportLayout class](#)

[layout methods](#)

[layout property](#)

[JApplet class](#)

[JComponent class](#)

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]
]

Mac OS X

ColorPicker3 and
L&F for
macmetrics.jar utility for
root directory and
[magicCaretPosition property \(Caret\)](#)
[magnifier pane for images](#)
[major tick marks \(sliders\)](#)
[majorTickSpacing property \(JSlider\) 2nd](#)
[makeIcon\(\) \(LookAndFeel\)](#)
[makeVisible\(\) \(JTree\)](#)
manager property
 [MenuDragMouseEvent class](#)
 [MenuKeyEvent class](#)
[managingFocus property \(JComponent\)](#)
[manipulation methods](#)
[Mapper.java utility](#)
margin property
 [AbstractButton property](#)
 [JMenuBar class](#)
 [JPopupMenu class](#)
 [JTextComponent class](#)
 [JToolBar class](#)
[MarginBorder class](#)
mark property
 [Caret interface](#)
 [CaretEvent class](#)
[markCompletelyClean\(\) \(RepaintManager\)](#)
[markCompletelyDirty\(\) \(RepaintManager\)](#)
[MarketDataModel.java \(example\)](#)
[MarketTable.java \(example\)](#)
mask property (MaskFormatter)
MaskFormatter class
 [properties table for](#)
masks
matte borders
MatteBorder class
 [properties table for](#)
[MatteBorderUIResource class](#)
[MatteExample.java \(example\)](#)
[maxCharactersPerLine property \(JOptionPane\)](#)
maxima of bounded ranges
maximizable property (JInternalFrame)
maximizeFrame()
 [DefaultDesktopManager class](#)
 [DesktopManager interface](#)
maximum property
 [BoundedRangeModel interface](#)
 [InternationalFormatter class](#)
JInternalFrame class
JProgressBar class
JScrollBar class
JSlider class
ProgressMonitor class

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]

name property

[Element interface](#)

[JComponent class](#)

[Keymap interface](#)

[LookAndFeel class](#)

[LookAndFeelInfo class](#)

[Style interface](#)

named style methods

named styles

[nativeLookAndFeel property \(LookAndFeel\)](#)

[NavigationFilter class](#)

[navigationFilter property \(JTextComponent\)](#)

nested edit support

[newDataAvailable\(\) \(DefaultTableModel\)](#)

[newRowsAdded\(\) \(DefaultTableModel\)](#)

next()

[ElementIterator class](#)

[Segment class](#)

[nextFocusableComponent property \(JComponent\)](#) [2nd](#)

[nextTabStop\(\) \(TabExpander\)](#)

[nodeChanged\(\) \(DefaultTreeModel\)](#)

nodes [2nd](#) [See also trees, nodes and paths] [3rd](#)

adding/removing

[Drag and Drop for](#)

editing

rendering

[nodesChanged\(\) \(DefaultTreeModel\)](#)

[nodeStructureChanged\(\) \(DefaultTreeModel\)](#)

[nodesWereInserted\(\) \(DefaultTreeModel\)](#)

[nodesWereRemoved\(\) \(DefaultTreeModel\)](#)

noniconified frames

nonliterals

non-modal dialogs

[note property \(PropertyMonitor\)](#)

[number property \(SpinnerNumberModel\)](#)

[NumberEditor class \(JSpinner\)](#)

properties table for

[NumberFormatter class](#)

numbers

[formatters for editing combinations](#)

formatting

[JFormattedTextField class and](#)

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)])

[object graphs](#)

[object reuse, LineBorder class and](#)

[object serialization](#)

[ObjectTree.java \(example\)](#)

[offscreen buffers](#) [See double buffering]

offset property

[DocumentEvent interface](#)

[Position interface](#)

[offsetRequested\(\) \(BadLocationException\)](#)

[OffsetTest.java \(example\)](#)

[old\(\) \(DefaultListModel\)](#)

[oneTouchExpandable property \(JSplitPane\)](#)

[onKeyRelease property \(KeyStroke\)](#)

opaque property

[Container class](#)

[DefaultTableCellRenderer class](#)

[JComboBox class](#)

[JComponent class](#)

[JDesktopPane class](#)

[JList class](#)

[JPanel class](#)

[OpaqueExample.java \(example\) 2nd](#)

[opaqueness](#) [See transparency]

openFrame()

[DefaultDesktopManager class](#)

[DesktopManager interface](#)

optimizedDrawingEnabled property

[JComponent class 2nd](#)

[JLayeredPane class](#)

[option dialogs 2nd](#)

[return values for](#)

[options for dialogs](#)

[options parameter \(JOptionPane \)](#)

[optionType parameter \(JOptionPane \)](#)

[OptPaneComparison.java \(example\)](#)

[oraswing.jar](#)

[organization charts](#)

orientation property

[JProgressBar class](#)

[JScrollBar class](#)

[JSlider class](#)

[JToolBar class](#)

[output-generating methods](#)

[outsideBorder property \(CompoundBorder \)](#)

[OvalIcon.java \(example\)](#)

[OverlayLayout class 2nd](#)

[overridden properties](#)

overwriteMode property

[DefaultFormatter class](#)

[InternationalFormatter class](#)

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]
[]

pack()

JComponent class

JInternalFrame class

packages in Swing libraries (list)

page frames

creating

page property (JEditorPane)

PageFrame.java (example)

paging large tables

PagingModel.java (example)

PagingTester.java (example)

paint()

Caret interface

cell renderers and

ComponentUI class

DefaultCaret class

DefaultHighlighter class

DefaultHighlightPainter class

Highlighter interface

HighlightPainter interface

JComponent class 2nd 3rd

View class

paintBorder()

AbstractBorder class

Border interface

CompoundBorder class

EmptyBorder class

EtchedBorder class

JComponent class 2nd

LineBorder class

MatteBorder class

[Soft]BevelBorder classes

TitledBorder class

paintChildren() (JComponent) 2nd

paintComponent()

JComponent class

SwingUtilities class

paintComponent() (JComponent)

paintDirtyRegions() (RepaintManager)

painter property (Highlight)

paintFocus() (BasicSliderUI)

paintIcon()

Icon interface

ImageIcon class

paintImmediately() (JComponent) 2nd

painting components 2nd

debugging

examples of 2nd 3rd

paintingTile property (JComponent) 2nd

paintLabels property (JSlider) 2nd

paintLabels() (BasicSliderUI)

paintLayer() (DefaultHighlightPainter)

paintLayeredHighlights() (LayeredHighlighter) 2nd

paintMajorTickForHorizSlider() (BasicSliderUI)

I1@ve RuBoard

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]
]

[query methods](#) [2nd](#)

[QueryTableModel.java](#) (example)

[queueWindowEvent\(\)](#) ([EventQueueMonitor](#))

I1@ve RuBoard

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]
]

radio buttons 2nd

RadioButtonBorder class

RadioButtonMenuItemExample.java (example)

ranges [See bounded ranges]

read()

DefaultEditorKit class

EditorKit class

HTMLEditorKit class

JTextComponent class

ProgressMonitorInputStream class

readAttributes() (StyleContext)

readAttributeSet() (StyleContext)

readLock() (AbstractDocument)

readObject()

ImageIcon class

ObjectInputStream class

readUnlock() (AbstractDocument)

reclaim() (StyleContext)

Record.java (example)

Rectangle class, vs. SpringLayout.Constraints class properties

Rectangle interface

rectangles, viewable

redo [See undo]

redo()

AbstractUndoableEdit class

CompoundEdit class

StateEdit class

UndoableEdit interface

UndoManager class

redoPresentationName property

AbstractUndoableEdit class

CompoundEdit class

UndoableEdit interface

UndoManager class

redoTo() (UndoManager)

RedTheme.java (example) 2nd

RegexPatternFormatter.java (example)

registerComponent() (ToolTipManager)

registerEditorKitForContentType() (JEditorPane)

registeredKeyStrokes property (JComponent)

registerListeners()

DragGestureRecognizer class

MouseDragGestureRecognizer class

registerStaticAttribute() (StyleContext)

registerStaticAttributeKey() (StyleContext) 2nd

regular expressions, formatters for

rejectDrag() (DropTargetDragEvent)

rejectDrop() (DropTargetDropEvent)

relation sets

RelativeLayout

reload() (DefaultTreeModel)

remove methods

remove()

AbstractDocument class

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]
]

sample code
accessibility 2nd
actions
alignment
applets 2nd 3rd
autoscrolling
borders
 custom 2nd
 matte
 titled
bounded ranges
boxes
buttons 2nd
 adding images to
 fancy
 grouped
 toggle
carets 2nd
census form 2nd
checkboxes
color chooser dialogs 2nd
combo boxes 2nd
containers, validating/revalidating
database data 2nd
desktop environment simulation
dialogs
document content, restricting length of
drag and drop 2nd 3rd
editor kits
editors
event handling 2nd
event listeners, for buttons
Every Event Listener (EEL)
file chooser dialogs 2nd
filters
focus 2nd
formatters 2nd
frames, closing/exiting program
glass pane blocking mouse events
glue components
highlighters 2nd
HTML browsers 2nd
icon factory
icons 2nd
 for trees
input stream progress monitoring
internal frames
 dialogs for
 iconifying/deiconifying
jog shuttle component 2nd
keymaps
L&F 2nd 3rd
 Metal L&F
labels 2nd 3rd 4th

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)])

[tab](#) property ([TabSet](#))

[TabableView](#) interface

[tabbed panes](#) 2nd

[tabs in](#)

[tabCount](#) property

[JTabbedPane](#) class

[TabSet](#) class

[TabExample.java](#) (example)

[TabExpander](#) interface

[tabLayoutPolicy](#) property ([JTabbedPane](#))

[table cells](#) [See [cells](#) [tables](#), [data](#) in]

[table](#) property ([JTableHeader](#))

[TableCellEditor](#) class

[TableCellEditor](#) interface

[TableCellRenderer](#) interface 2nd

[tableChanged\(\)](#) ([TableModelListener](#))

[TableChart.java](#) (example)

[TableChartPopup.java](#) (example)

[TableColumn](#) class

[properties](#) [table](#) for

[TableColumnModel](#) interface 2nd 3rd

[properties](#) [table](#) for

[TableColumnModelEvent](#) class 2nd

[TableColumnModelListener](#) interface

[TableFeature.java](#)

[tableHeader](#) property ([JTable](#))

[TableModel](#) interface

[charting](#) [data](#) with

[properties](#) [table](#) for

[scrollable](#) [tables](#) and 2nd

[TableModelEvent](#) class

[TableModelListener](#) interface

[tables](#)

[accessibility](#) and

[classes](#)/[interfaces](#) for

[columns](#)

[ColumnExample.java](#) (example)

[examples](#) of 2nd

[implementing](#) [model](#) for

[data](#) in

[charting](#)

[database](#) [records](#)

[dynamic](#)

[editing](#)/[rendering](#) 2nd

[examples](#) of 2nd 3rd 4th 5th

[selecting](#)

[examples](#) of 2nd

[advanced](#) [examples](#)

[javax.swing.table](#) package for

[JTable](#) class 2nd

[JTableHeader](#) class

[large](#), [paging](#) for

[scrollable](#)

[TableView](#) class

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)])

[UberHandler.java \(example\)](#)

UI delegates 2nd 3rd 4th

examples of 2nd

individual

creating

replacing

L&F, changing for

UI property

[JInternalFrame](#)

[JOptionPane class](#)

UIClassID constant

[JComponent class](#)

[JContainer class](#)

UIClassID property

[JDesktopPane class](#)

[JEditorPane class](#)

[JOptionPane class](#)

[JTextPane class](#)

[UIDefaults class](#) 2nd 3rd

[ActiveValue interface](#) 2nd

[LazyValue interface](#) 2nd

[ResourceModExample.java \(example\)](#)

[UIManager class](#) 2nd

[LookAndFeelInfo class](#) 2nd

[UIManager properties, ProgressMonitor class and](#)

[UIManagerDefaults.java \(example\)](#)

[UI managers, largeModel property and](#)

[UIResource interface](#) 2nd

[UnderlineAction class](#)

undo

[AbstractUndoableEdit class](#)

[compound edits and](#)

[edit-merging methods and](#)

[examples of](#) 2nd

for sequential edits[undo

[sequential edits}](#)

[javax.swing.undo package for](#) 2nd

[killed edits and](#)

[nested edits, support for](#)

[state and](#) 2nd

undo()

[AbstractUndoableEdit class](#)

[CompoundEdit class](#)

[StateEdit class](#)

[UndoableEdit interface](#)

[UndoManager class](#)

[UndoableEdit interface](#) 2nd

[properties table for](#)

[UndoableEditEvent class](#) 2nd

[undoableEditHappened\(\)](#)

[UndoableEditListener interface](#)

[UndoManager class](#)

[UndoableEditListener interface](#) 2nd

[undoableEditListeners property \(AbstractDocument\)](#)

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]
]

valid property

[AccessibilityHyperlink class](#)

[JComponent class](#)

[validate\(\) \(JComponent\)](#)

[validateInvalidComponents\(\) \(RepaintManager\)](#)

validateRoot property

[JComponent class](#) [2nd](#)

[JRootPane class](#)

[JScrollPane class](#)

[JTextField class](#)

[validating/revalidating containers](#)

[validCharacters property \(MaskFormatter\)](#)

value property

[BoundedRangeModel interface](#)

[JFormattedTextField class](#)

[JProgressBar class](#)

[JScrollBar class](#)

[JSlider class](#)

[JSpinner class](#)

valueChanged()

[DefaultTableColumnModel class](#)

[ListSelectionListener interface](#)

[SiteFrame.java \(example\)](#)

[TreeSelectionListener interface](#)

[valueClass property \(DefaultFormatter\)](#)

[valueContainsLiteralCharacters property \(MaskFormatter\)](#)

valueForPathChanged()

[DefaultTreeModel class](#)

[TreeModel interface](#)

valueIsAdjusting property

[BoundedRangeModel class](#)

[DefaultListSelectionModel class](#)

[JList class](#)

[JScrollBar class](#)

[JSlider class](#)

[ListSelectionModel interface](#)

valueToString()

[InternationalFormatter class](#)

[MaskFormatter class](#)

valueToValue()

[DefaultFormatter class](#)

[JFormattedTextField.AbstractFormatter class](#)

[VariableHeightLayoutCache class](#)

[Vector class](#)

[verify\(\) \(InputVerifier\)](#)

[verifyInputWhenFocusTarget property \(JComponent\)](#)

[vertical scrollbars](#)

verticalAlignment property

[AbstractButton class](#)

[JLabel class](#)

[verticalScrollBarPolicy property \(JScrollPane\)](#)

[verticalTextPosition property \(JLabel\)](#) [2nd](#)

vetoable changes, JInternalFrame class and

[VetoableChangeListener interface](#) [2nd](#)

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]
]

[wallpaper](#)

[wantsInput](#) property ([JOptionPane](#))

warning messages, applets and

[warningString](#) property ([JInternalFrame](#))

[wasIcon\(\)](#) ([DefaultDesktopManager](#))

web browsers, [JApplet](#) class and

[web sites](#) [See URLs]

[web-site manager](#) sample application

[wheelScrollingEnabled](#) property ([JScrollPane](#))

[when](#) property

[MenuDragMouseEvent](#) class

[MenuKeyEvent](#) class

[width](#) property ([JComponent](#))

[windowClosing\(\)](#) events

[WindowConstants](#) interface

[windowDecorationStyle](#) property ([JRootPane](#))

[windowForComponent\(\)](#) ([SwingUtilities](#))

[windows](#)

[borderless frame](#) and

[closing](#), [JDialog](#) class for

[desktop environment simulation](#) and

[JWindow](#) class

[Splash.java](#) (example)

[TopLevelWindowListener](#) interface

[WindowConstants](#) interface

[WrappedPlainView](#) class

[wrapStyleWord](#) property ([JTextArea](#))

[write\(\)](#)

[AbstractWriter](#) class

[DefaultEditorKit](#) class

[EditorKit](#) class

[HTMLEditorKit](#) class

[HTMLWriter](#) class

[JTextComponent](#) class

[writeAttributes\(\)](#)

[AbstractWriter](#) class

[StyleContext](#) class

[writeAttributeSet\(\)](#) ([StyleContext](#))

[writeLock\(\)](#) ([AbstractDocument](#))

[writeObject\(\)](#)

[ImageIcon](#) class

[ObjectOutputStream](#) class

[writeUnlock\(\)](#) ([AbstractDocument](#))

[writing methods](#) 2nd

I1@ve RuBoard

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]
]

x property

[JComponent class](#)

[MenuDragMouseEvent class](#)

[XML documents, tree displaying](#)

[XML, custom tags and](#)

I1@ve RuBoard

I1@ve RuBoard

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]
]

y property

[JComponent class](#)

[MenuDragMouseEvent class](#)

I1@ve RuBoard

I1@ve RuBoard

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]
]

z-order 2nd
ZoneView class

I1@ve RuBoard