CONTENTS

Chapter 14. Basic Swing

Topics in This Chapter

- Building Swing applets and applications
- Changing the GUI look and feel
- Adding custom borders to components
- Creating text and image buttons
- Using HTML in labels and buttons
- Selecting colors with JColorChooser
- Sending dialog alerts for user input
- Adding child frames to applications
- Building custom toolbars
- Implementing a Web browser in Swing

Many programmers consider the Swing components to be the single most significant improvement and change in the Java platform for development of graphical user interfaces. Swing adds the small touches (icons in dialog boxes, tool-tips, borders) that distinguish a commercial-quality GUI from an amateurish one.

The Swing components are standard in the Java 2 Platform and go far beyond the simple components found in the AWT. The benefits of Swing include:

- A significantly increased set of built-in controls including image buttons, tabbed panes, sliders, toolbars, color choosers, text areas that can display HTML or RTF, lists, trees, and tables.
- Increased customization of components, including border styles, text alignments, and basic drawing features. In addition, an image can be added to almost any control.
- A pluggable "look and feel" that can be changed at runtime. If desired, you can design your own look and feel.

- Many miscellaneous new features, for example, built-in double buffering, tool-tips, dockable toolbars, keyboard accelerators, and custom cursors.

Complete coverage of the entire Swing library is beyond the scope of this book. We discuss the basic usage most often employed in applications. For more in-depth treatment on the topic, please see *Core Java Foundation Classes* or *Core Swing: Advanced Programming*, both by Kim Topley.

14.1 Getting Started with Swing

Swing is standard in the Java 2 Platform but can be added to JDK 1.1 as a separate package. The JDK 1.1 version of Swing is downloadable from http://java.sun.com/products/jfc/download.html. You can also use Swing in an applet, but unless you are using Netscape 6, you will need to install the Java Plug-In (covered in Section 9.9).

Differences Between Swing and the AWT

The following subsections summarize the basic differences between Swing and the AWT.

Naming Convention

All Swing component names begin with a capital `J` and follow the format `JXxx`, where the `Xxx` represents a common component name, for example, `JFrame`, `JPanel`, `JApplet`, `JDialog`, `JButton`. All the AWT components have an almost equivalent Swing component. Nearly all the Swing components inherit directly from `JComponent`, which provides support for the pluggable look and feel, custom borders, and tool-tips.

Lightweight Components

Most Swing components are lightweight: formed by drawing in the underlying window using Java code rather than relying on native peer code to perform the drawing. The four Swing lightweight exceptions are `JFrame`, `JApplet`, `JWindow`, and `JDialog`. As the graphics in the off-screen buffer must be eventually drawn to the screen, these four *heavyweight* Swing components form a bridge to the corresponding AWT peers to perform the drawing.

Use of the paintComponent Method for Drawing

For Swing, custom drawing is performed in `paintComponent`, not `paint`. The default implementation of `paintComponent` is to invoke the user interface (UI) delegate to control the look and feel of the component. In addition, the UI delegate is responsible for erasing the off-screen buffer before drawing. Thus, you should always call the superclass `paintComponent` method before performing any drawing to guarantee that the off-screen buffer is cleared and that the component's look and feel is maintained. Thus, each `paintComponent` method should begin as follows:

```
public void paintComponent(Graphics g){
  super.paintComponent(g);
  // Drawing on the Swing component ...
  ...
}
```

Core Approach

In Swing, perform drawing in `paintComponent`, not `paint`. Always call `super.paintComponent` before performing custom drawing.

Use of the Content Pane for Adding Components

Instead of adding components directly to a `JFrame` or `JApplet`, you add the components to the "content pane," for example,

```
Container content = getContentPane();
content.add(new JButton("Welcome"));
content.add(new JLabel("JavaOne");
```

Adding a component *directly* to a `JFrame` or `JApplet` produces an error. The content pane is simply a `Container` with a layout manager of `BorderLayout`. Thus, both a `JFrame` and a `JApplet` in Swing have the same layout manager, unlike the case with an AWT `Frame` (`BorderLayout`) and `Applet` (`FlowLayout`). If you want to replace the content pane with a different container, call `setContentPane`.

Core Approach

Add components to the content pane of a `JFrame` or `JApplet`.

Double Buffering

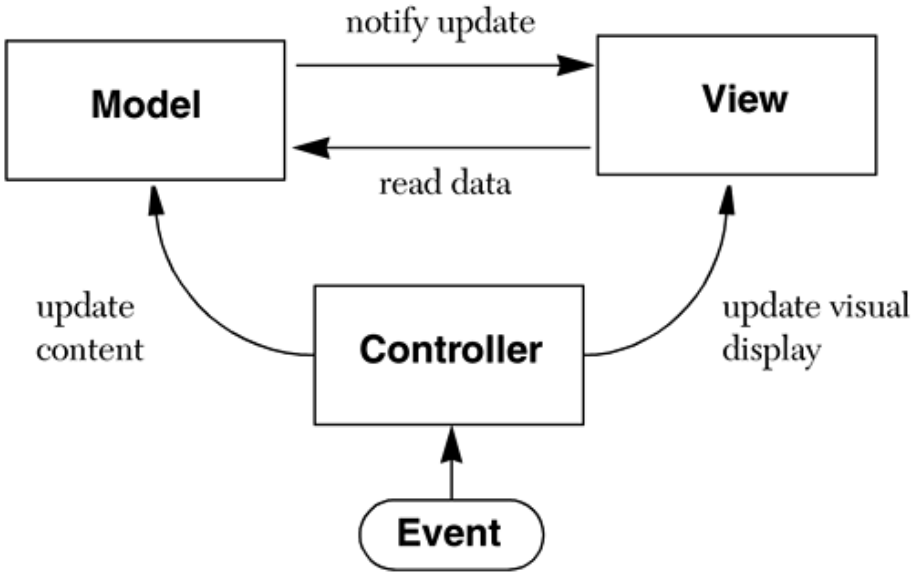
By default, a `JPanel` is double buffered. See Section 16.7 (Multithreaded Graphics and Double Buffering) for a detailed explanation of double buffering. In Swing, a single off-screen buffer, large enough to accommodate the size of the screen, is maintained by the `RepaintManager`. When the `paint` method is executed, the basic behavior is to obtain the off-screen buffer from the `RepaintManager`, call `paintComponent` for custom drawing into the buffer, and then, copy the buffer to the screen. Any drawing to a `JPanel` through `paintComponent` is drawn into the off-screen buffer; however, if you perform "out of `paint` method" drawing using `panel.getGraphics()`, then drawing is performed directly on screen and is not double buffered.

Any components placed in a double buffered container are also automatically double buffered. Since the content pane of a `JFrame` and a `JApplet` is a double buffered `Container`, all Java Swing applications are automatically double buffered.

Model-View-Controller Architecture

Lightweight Swing components are designed around a model-view-controller (MVC) architecture, shown in Figure 14-1. Conceptually, the *model* is a data structure that provides accessor methods for accessing the data. The data structure could be as simple as a couple of state variables (pressed, enabled) for a button, or as complicated as an `ArrayList` of input fields for a table. The *view* is the visual presentation of the data in the model. The *controller* is the event handler.

Figure 14-1. Schematic of a Model-View-Controller (MVC).



Each lightweight Swing component has an associated user interface (UI) delegate that inherits from `ComponentUI` and controls the look (view) and feel (listeners) of the component. When the component is instantiated, the `UIManager` returns the correct Swing UI (`LabelUI`, `TableUI`, `TreeUI`, etc.) based on the selected look and feel (Windows, Motif, Java) of the program. The UI delegate is responsible for reporting the minimum, maximum, and preferred size of the component, painting the component, and handling component events.

The beauty of the MVC architecture is that a single data model can be assigned to more than one Swing component, thereby allowing more than one view of the data. Consider a data series displayed in a table and a corresponding display of the data in a histogram chart; both views of the data reference the same data model. If the user adds a new data value to the model, both views are dynamically updated after a corresponding change event is fired from the model. The use of multiple views for complicated data is common in advanced Swing components like `JTree` and `JTable`, covered in Chapter 15 (Advanced Swing).

New Look and Feel as Default

The MVC architecture permits a pluggable look and feel of the Swing components. In Swing, the graphical user interface can have the look of a Windows, Motif, MacOS, or Java (formerly called Metal) environment. The `UIManager` class allows you to set the look and feel of the graphical user interface before any Swing components are created. When you create a component, the `UIManager` returns the *correct* UI delegate (responsible for painting the component with the selected look and feel) to the component. Thus, you should set the look and feel before creating new lightweight components. The following list summarizes additional key points about Swing look and feel:

- The default cross-platform look and feel is "Java" (or "Metal"), a custom look and feel somewhat similar to the Windows look.
- The Motif look is available on all platforms. However, Windows and Mac looks are only available on their native platforms. Technically, you can easily work around this restriction, but distributing applications that do this is currently illegal.
- The look and feel can be changed at runtime. After you set the new look and feel, calling `SwingUtilties.updateComponentTreeUI` will hand off a new UI delegate to each component. For example,

```
try {
```

```
      UIManager.setLookAndFeel(
          "javax.swing.motif.MotifLookAndFeel");
    } catch(Exception e) {
      System.out.println("LAF Error: " + e);
    }
    SwingUtilties.updateComponentTreeUI(
                  getContentPane());
```

This capability sounds cool, but it is rarely used in real life. The normal place to set the look and feel is in the constructor (or `main`) of the top-level `JFrame`, or in `init` of the `JApplet`.

- The look and feel can be set to the *native* windows look for the operating system. Call the `getSystemLookAndFeelClassName` method of `UIManager`, and pass the result to `UIManager.setLookAndFeel`. Since `setLookAndFeel` throws an exception, a direct approach is a bit inconvenient, and you might want to create a `static` method called `setNativeLookAndFeel` in a utility class.

Listing 14.1 presents a utility class, `WindowUtilities`, that provides `static` methods to set the look and feel to native, Java (Metal), and Motif, respectively. Since most users expect the native look and may be unfamiliar with the Java look (which is the default when installing the JRE), you can consider calling `setNativeLookAndFeel` at the beginning of Swing programs to obtain the native look and feel.

In addition, `WindowUtilities` provides `static` methods for displaying containers in a `JFrame`. The various `openInJFrame` methods are used throughout the chapter as a convenience to display a `JPanel` in a `JFrame`. The frames take advantage of the `ExitListener` class to exit the application when closing the frame.

Listing 14.1 `WindowUtilities.java`

```
import javax.swing.*;
import java.awt.*;   // For Color and Container classes.

/** A few utilities that simplify using windows in Swing. */

public class WindowUtilities {

  /** Tell system to use native look and feel, as in previous
   *  releases. Metal (Java) LAF is the default otherwise.
   */

  public static void setNativeLookAndFeel() {
    try {
     UIManager.setLookAndFeel(
       UIManager.getSystemLookAndFeelClassName());
    } catch(Exception e) {
      System.out.println("Error setting native LAF: " + e);
    }
  }

  public static void setJavaLookAndFeel() {
    try {
     UIManager.setLookAndFeel(
```

```
      UIManager.getCrossPlatformLookAndFeelClassName());
    } catch(Exception e) {
      System.out.println("Error setting Java LAF: " + e);
    }
  }
  public static void setMotifLookAndFeel() {
    try {
      UIManager.setLookAndFeel(
        "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
    } catch(Exception e) {
      System.out.println("Error setting Motif LAF: " + e);
    }
  }

  /** A simplified way to see a JPanel or other Container. Pops
   *  up a JFrame with specified Container as the content pane.
   */

  public static JFrame openInJFrame(Container content,
                                    int width,
                                    int height,
                                    String title,
                                    Color bgColor) {
    JFrame frame = new JFrame(title);
    frame.setBackground(bgColor);
    content.setBackground(bgColor);
    frame.setSize(width, height);
    frame.setContentPane(content);
    frame.addWindowListener(new ExitListener());
    frame.setVisible(true);
    return(frame);
  }

  /** Uses Color.white as the background color. */

  public static JFrame openInJFrame(Container content,
                                    int width,
                                    int height,
                                    String title) {
    return(openInJFrame(content, width, height,
                        title, Color.white));
  }

  /** Uses Color.white as the background color, and the
   *  name of the Container's class as the JFrame title.
   */

  public static JFrame openInJFrame(Container content,
                                    int width,
                                    int height) {
    return(openInJFrame(content, width, height,
                        content.getClass().getName(),
```

```
                        Color.white));
  }
}
```

Listing 14.2 `ExitListener.java`

```
import java.awt.*;
import java.awt.event.*;

/** A listener that you attach to the top-level JFrame of
 *  your application, so that quitting the frame exits the
 *  application.
 */
public class ExitListener extends WindowAdapter {
  public void windowClosing(WindowEvent event) {
    System.exit(0);
  }
}
```

Included in the Java 2 JDK is an excellent applet demonstrating the various components available in Swing. The SwingSet2 demonstration applet is located in the *root*`/jdk1.3/demo/jfc/SwingSet2` install directory. Figures 14-2 through 14-4 illustrate the Windows, Motif, and Java look and feel of Swing, respectively.

Figure 14-2. Windows look and feel for the Sun SwingSet2 demo.

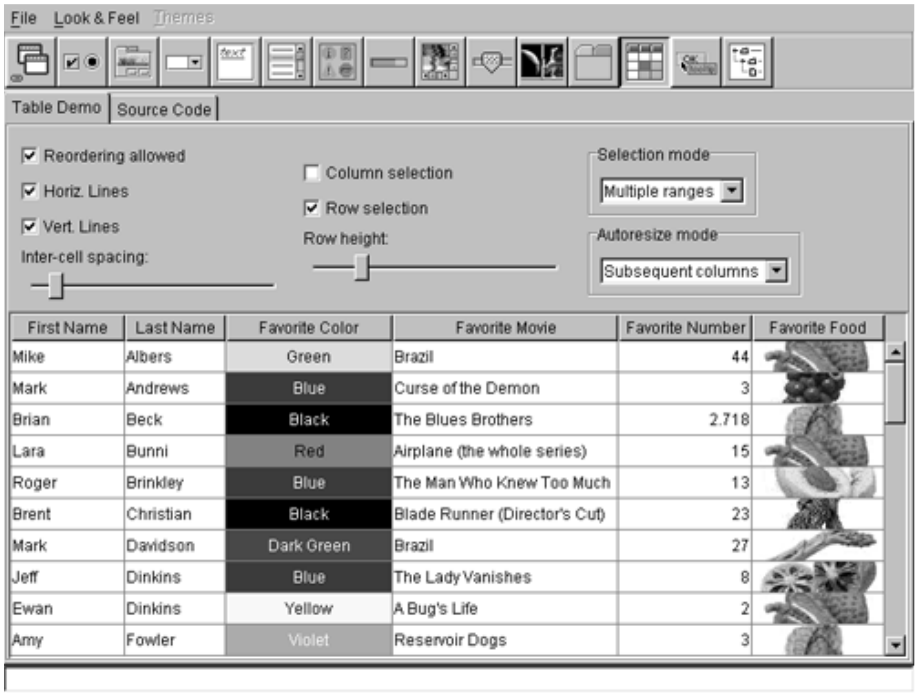
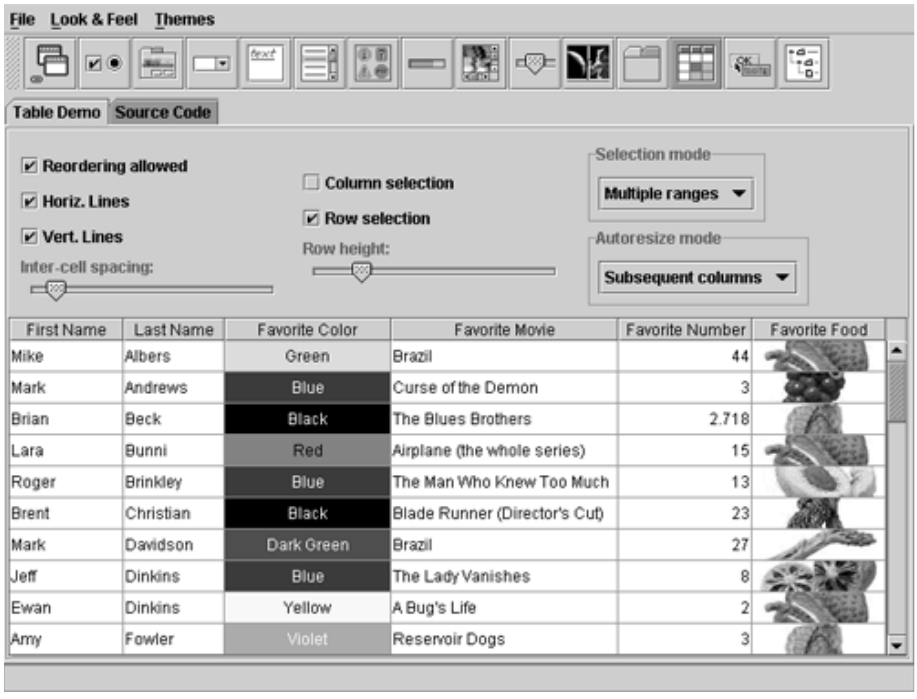


Figure 14-3. Motif look and feel for the Sun SwingSet2 demo.

Figure 14-4. Java (Metal) look and feel for the Sun SwingSet2 demo.



Don't Mix AWT and Swing Components

The z-ordering always places AWT components on top. In the Java platform, z-ordering determines the order in which components in the same container are displayed. The first component added to a container has the highest z-order; the last component added to a container has the lowest z-order. Components with a higher z-order are displayed on top of components with a lower z-order. Consequently, the windowing container always has the lowest z-order and is displayed underneath all other heavyweight components. Lightweight Swing components are always drawn in the heavyweight windowing container in which they reside, and therefore Swing components have the same z-order as their container. As a result, AWT components will always be drawn on top of any Swing components. This behavior is problematic and can catch you in many unexpected ways. For

example, consider an AWT `Button` in a Swing application that also includes a drop-down menu. Depending on the placement of the AWT `Button`, when the drop-down menu is selected, the `Button` can cover the drop-down menu list. The best advice is to always stick with AWT or move completely to Swing.

Before we present the various Swing components, you should note that the `javax.swing.SwingConstants` class defines numerous constants for positioning components. The available constants are `LEFT`, `CENTER`, `RIGHT`, `TOP`, `BOTTOM`, `NORTH`, `EAST`, `SOUTH`, `WEST`, `NORTH_EAST`, `NORTH_WEST`, `SOUTH_EAST`, `SOUTH_WEST`, `HORIZONTAL`, `VERTICAL`, `LEADING`, and `TRAILING`. The last two constants specifically identify the leading or trailing side of the component according to the locale's reading order (i.e., left-to-right and right-to-left languages).

14.2 The JApplet Component

`JApplet` is the Swing equivalent of an AWT `Applet`. The major limitation of a `JApplet` is that the only major browser that supports the Java 2 Platform is Netscape 6. However, you can add Swing to Java 1.1 in other browsers, but for the greatest Swing capabilities, we recommend the Swing version in Java 2. In order to run the Swing applet in a browser, you may need to install the Java 2 Plug-In as described in Section 9.9 (note that Netscape 6 supports JDK 1.3). As the Java Plug-In download is over 5 Mbytes, the use of Swing applets on the Internet is not common; use Swing only for an intranet environment where the plug-in can be located on a local server for downloading.

A `JApplet` is a heavyweight Swing component and inherits directly from `Applet` instead of inheriting from `JComponent` as with the lightweight Swing components. Therefore, the `JApplet` class inherits the familiar methods of `init`, `start`, `stop`, and `destroy`. As a Swing applet though, it exhibits some major differences exist. For instance,

- A `JApplet` contains a content pane in which to add components. Changing other properties like the layout manager, background color, etc., also applies to the content pane. Access the content pane through `getContentPane`.
- The default layout manager is `BorderLayout` (as with `Frame` and `JFrame`), not `FlowLayout` (as with `Applet`). `BorderLayout` is really the layout manager of the content pane.
- The default look and feel is Java (Metal), so you have to explicitly switch the look and feel if you want the native look.
- Drawing is done in `paintComponent`, not `paint`. You don't draw directly in a `JApplet`, however. Instead you add a `JPanel` to the content pane, override the `paintComponent` method of the added `JPanel`, and perform all drawing in the `JPanel`.
- Double buffering is turned on by default.

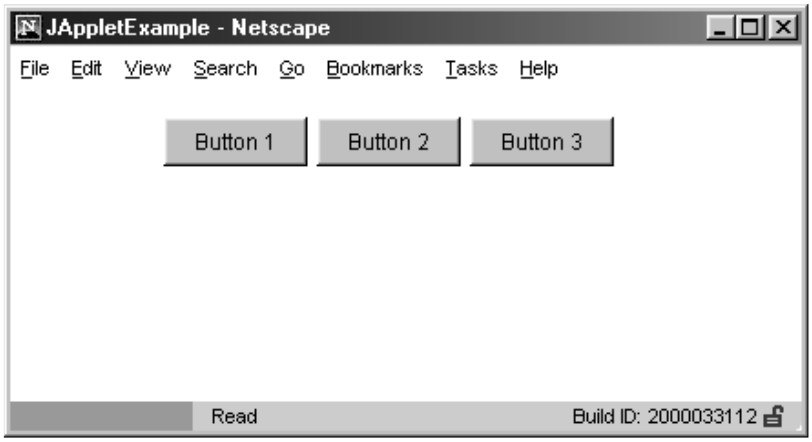
Core Note

The default layout manager of a `JApplet` is `BorderLayout`, not `FlowLayout` as in an `Applet`.

Listing 14.3 provides a simple example showing the steps required to create what you would have in the AWT if you had a simple applet whose `init` method did nothing but drop three buttons into the window. In the Swing version, to achieve the same positioning of the buttons, the

`LayoutManager` changed to `FlowLayout`. The result in Netscape 6 is shown in Figure 14-5.

Figure 14-5. Swing `JButtons` shown in Netscape 6 on Windows 98.



Listing 14.3 `JAppletExample.java`

```
import java.awt.*;
import javax.swing.*;

/** Tiny example showing the main differences in using
 *  JApplet instead of Applet: using the content pane,
 *  getting Java (Metal) look and feel by default, and
 *  having BorderLayout be the default instead of FlowLayout.
 */
public class JAppletExample extends JApplet {
  public void init() {
    WindowUtilities.setNativeLookAndFeel();
    Container content = getContentPane();
    content.setBackground(Color.white);
    content.setLayout(new FlowLayout());
    content.add(new JButton("Button 1"));
    content.add(new JButton("Button 2"));
    content.add(new JButton("Button 3"));
  }
}
```

14.3 The JFrame Component

The `JFrame` is the Swing equivalent of the AWT `Frame` and, as in the AWT, the `JFrame` is a starting point for graphical applications. Similar to `JApplet`, a `JFrame` is a *heavyweight* Swing component and does not inherit from `JComponent`. `JFrame` inherits directly from `Frame`. The main differences between a `JFrame` and an AWT `Frame` are these:

- Components are added to the content pane, not directly to the frame. Changing the layout manager, background color, etc., also applies to the content pane. Access the content pane through `getContentPane`.
- `JFrame`s close automatically when you click on the Close button (unlike AWT `Frames`). However, closing the last `JFrame` does not result in your program exiting the Java

application. So, your "main" `JFrame` still needs a `WindowListener` to call `System.exit`. Or, alternatively, if using JDK 1.3, you can call `setDefaultCloseOpertion(EXIT_ON_CLOSE)`. This latter case permits the `JFrame` to close; however, you won't be able to complete any house cleaning as you might in the `WindowListener`.

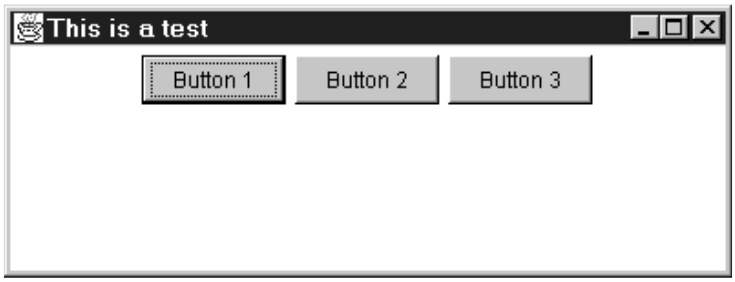
- The default look and feel is Java (Metal), so you have to explicitly switch the look and feel if you want the native look.

Core Note

Child Swing frames automatically close when you click the Close button; however, the parent frame still requires a `WindowListener` to invoke `System.exit`.

Listing 14.4 shows the steps required to imitate what you would get in the AWT if you popped up a simple `Frame`, set the layout manager to `FlowLayout`, and dropped three buttons into the frame. The complete listings for `WindowUtilities.java` and `ExitListener.java`, required by this application, are provided in Section 14.1 (Getting Started with Swing) and can be downloaded from the on-line archive at http://www.corewebprogramming.com/. The result is shown in Figure 14-6.

Figure 14-6. A Swing `JFrame` containing three buttons.



Listing 14.4 `JFrameExample.java`

```
import java.awt.*;
import javax.swing.*;

/** Tiny example showing the main difference in using
 *  JFrame instead of Frame: using the content pane
 *  and getting the Java (Metal) look and feel by default
 *  instead of the native look and feel.
 */

public class JFrameExample {
  public static void main(String[] args) {
    WindowUtilities.setNativeLookAndFeel();
    JFrame f = new JFrame("This is a test");
    f.setSize(400, 150);
    Container content = f.getContentPane();
    content.setBackground(Color.white);
    content.setLayout(new FlowLayout());
    content.add(new JButton("Button 1"));
```

```
    content.add(new JButton("Button 2"));
    content.add(new JButton("Button 3"));
    f.addWindowListener(new ExitListener());
    f.setVisible(true);
  }
}
```

14.4 The JLabel Component

In many cases, `JLabel` is used exactly like `Label`: as a way to display text. However, a `JLabel` has three major features that a `Label` does not. A `JLabel` can display an image instead of or in addition to the text, can have borders, and can use HTML content to format the label. These three major features are discussed next.

New Features: Images, Borders, and HTML Content

The first new feature of a `JLabel` is image display, which you use usually by supplying an `ImageIcon` either to the constructor or through a call to `setIcon`. The use of icons in `JLabel` is just like the use in `JButton`; see Section 14.5 (The JButton Component) for additional details and code examples. For an example, however, see the third `JLabel` in Listing 14.5. Note that even though `JLabel`s and `JButton`s have significant common functionality, they are not related through inheritance.

The second new feature for labels is label borders. The use of borders is covered in Section 14.6 (The JPanel Component). For a quick preview, however, see the example in Listing 14.5 that uses titled borders.

The third new feature, and the one that we are focusing on here, is label formatting with HTML. The idea is that if the string for the label begins with `<html>`, then the string is interpreted as HTML rather than taken as literal characters. The ability to interpret HTML allows you to create multiline labels, labels with mixed colors and fonts, and various other fancy effects. This capability to render HTML also applies to `JButton`. Although nice, this feature also has several significant limitations:

- HTML labels only work in JDK 1.2.2 or later, or in Swing 1.1.1 or later. Since the Java platform provides no programmatic way to test if this capability is supported, using HTML labels can cause significant portability problems.
- In JDK 1.2 the label string must begin with `<html>`, not `<HTML>`. Case-insensitive HTML tags are supported in JDK 1.3.
- In JDK 1.2, if you would like to include an image in the label, you must supply an `ImageIcon` to the `JLabel` constructor or use `setIcon`; the HTML label cannot have an `IMG` tag. In JDK 1.3, embedded images are supported in the HTML, but the technique required to embed an image is beyond the scope of this book.
- `JLabel` fonts are ignored if HTML is used. If you use HTML, all font control must be performed by HTML. For example, one would think that the following would result in large, bold Serif text, but the code actually results in small, bold Sans Serif text instead:

```
JLabel label =
  new JLabel("<html>Bold Text</html>");
label.setFont(new Font("Serif",Font.BOLD,36));
...
```

- You must use `<P>`, not `<BR>`, to force a line break. The `<BR>` tag is ignored, and `<P>` in a

`JLabel` or `JButton` works like `<BR>` does in "real" HTML, starting a new line but not leaving a blank line in between.

- Other HTML support is spotty. Be sure to test each HTML construct you use. Permitting the user to enter HTML text at runtime is asking for trouble.

JLabel Constructors

The `JLabel` class has six constructors. The four most common constructors are listed below:

public JLabel()

public JLabel(String label)

public JLabel(Icon image)

public JLabel(String label, Icon image, int hAlignment)

The `JLabel` constructors permit direct creation of an empty label, a text label, and an image label. The last constructor creates a label with both an image and an icon. In the last constructor, you must also specify the desired horizontal alignment of the text-icon pair relative to the label itself. Legal values are `LEFT`, `CENTER`, `RIGHT`, `LEADING`, or `TRAILING`.

Useful JLabel Methods

The following paragraphs list the more common methods of the `JLabel` class. All property value methods have corresponding `get` methods.

public void setHorizontalAlignment(int alignment)

public void setVerticalAlignment(int alignment)

These methods set the horizontal and vertical alignment of the text-icon pair, respectively, relative to the label itself. The legal alignments are defined in `javax.swing.SwingConstraints` and include `LEFT`, `CENTER`, `RIGHT`, `LEADING`, and `TRAILING` for horizontal alignment, and `TOP`, `CENTER`, and `BOTTOM` for vertical alignment.

public void setHorizontalTextPosition(int alignment)

public void setVerticalTextPosition(int alignment)

These methods set the horizontal and vertical position of the text, relative to label's image, if present. The legal values are the same as those for setting the text-icon pair alignment.

public void setIcon(Icon image) public void setDisabledIcon(Icon image)

These two methods specify the image for the `JLabel` when enabled and disabled, respectively. A `JLabel` is disabled through `setEnabled(false)`.

public void setText(String label)

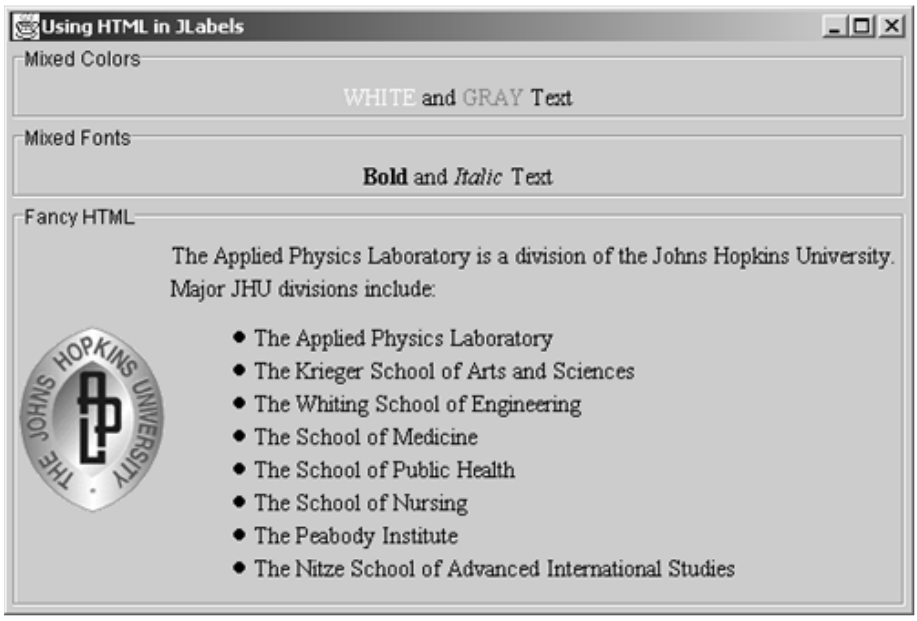
public void setFont(Font font)

The first method sets the text to display in the label. The text can include HTML tags if

enclosed in <html>...</html>. The second method sets the font style of the label.

Examples of the new JLabel features are given in Listing 14.5, with the result shown in Figure 14-7. The first and second labels use simple HTML text to control the text colors and to set the text font, respectively. The last label incorporates both an image and HTML containing an unordered list, <UL>. All three labels incorporate a titled border (see Section 14.6, "The JPanel Component," for border styles).

Figure 14-7. Swing components provide HTML text support for captions.



Listing 14.5 JLabels.java

```
import java.awt.*;
import javax.swing.*;

/** Simple example illustrating the use of JLabel, especially
 *  the ability to use HTML text (Swing 1.1.1 and Java 1.2.2 and
 *  later only!).
 */

public class JLabels extends JFrame {
  public static void main(String[] args) {
    new JLabels();
  }

  public JLabels() {
    super("Using HTML in JLabels");
    WindowUtilities.setNativeLookAndFeel();
    addWindowListener(new ExitListener());
    Container content = getContentPane();
    Font font = new Font("Serif", Font.PLAIN, 30);
    content.setFont(font);
    String labelText =
      "<html><FONT COLOR=WHITE>WHITE</FONT> and " +
      "<FONT COLOR=GRAY>GRAY</FONT> Text</html>";
```

```
    JLabel coloredLabel =
      new JLabel(labelText, JLabel.CENTER);
    coloredLabel.setBorder
      (BorderFactory.createTitledBorder("Mixed Colors"));
    content.add(coloredLabel, BorderLayout.NORTH);
    labelText =
      "<html><B>Bold</B> and <I>Italic</I> Text</html>";
    JLabel boldLabel =
      new JLabel(labelText, JLabel.CENTER);
    boldLabel.setBorder
      (BorderFactory.createTitledBorder("Mixed Fonts"));
    content.add(boldLabel, BorderLayout.CENTER);
    labelText =
      "<html>The Applied Physics Laboratory is a division " +
      "of the Johns Hopkins University." +
      "<P>" +
      "Major JHU divisions include:" +
      "<UL>" +
      "  <LI>The Applied Physics Laboratory" +
      "  <LI>The Krieger School of Arts and Sciences" +
      "  <LI>The Whiting School of Engineering" +
      "  <LI>The School of Medicine" +
      "  <LI>The School of Public Health" +
      "  <LI>The School of Nursing" +
      "  <LI>The Peabody Institute" +
      "  <LI>The Nitze School of Advanced International Studies" +
      "</UL>" +
      "</html>";
      JLabel fancyLabel =
      new JLabel(labelText,
                 new ImageIcon("images/JHUAPL.gif"),
                 JLabel.CENTER);
    fancyLabel.setBorder
      (BorderFactory.createTitledBorder("Fancy HTML"));
    content.add(fancyLabel, BorderLayout.SOUTH);
    pack();
    setVisible(true);
  }
}
```

14.5 The JButton Component

Simple uses of `JButton` are similar to those for the AWT `Button`. You create a `JButton` with a `String` as a label and then drop the button into a window. Events are normally handled just as with a `Button`: you attach an `ActionListener` through the `addActionListener` method.

New Features: Icons, Alignment, and Mnemonics

The most obvious new feature is the ability to associate images with buttons. Many Swing controls allow the inclusion of icons. Swing introduced a utility class called `ImageIcon` that lets you easily specify an image file (JPEG or GIF, including animated GIF) for the icon. The simplest way to associate an image with a `JButton` is to pass the `ImageIcon` to the constructor, either in place

of the text or in addition to the text. `JButton` actually defines seven associated images:

1. The main image (use `setIcon` to specify the main image if not supplied in the constructor).
2. The image to use when the button is pressed (`setPressedIcon`).
3. The image to use when the mouse is over the button (`setRolloverIcon`, but first you need to call `setRolloverEnabled(true)`).
4. The image to use when the button is disabled (`setDisabledIcon`).
5. The image to use when the button is selected and enabled (`setSelectedIcon`).
6. The image to use when the button is selected but disabled (`setDisabledSelectedIcon`).
7. The image to use when the mouse is over the button while selected (`setRolloverSelectedIcon`).

The images—`setSelectedIcon`, `setRolloverSelectedIcon`, and `setDisabledSelectedIcon`—for a selected button, are supported in JDK 1.3. but not in JDK 1.2 You can only select a button programmatically by calling `setSelected(true)`. Selecting a button does not trigger an `ActionEvent`. If you are not creating a custom, selectable button to maintain state, you might choose instead a standard `JToggleButton` (not covered; see on-line API for `javax.swing.JToggleButton`) that is selectable either programmatically or by the user.

You can also change the alignment of the text, icon, or text-icon pair in the button through `setHorizontalAlignment` and `setVerticalAlignment` (only valid if button is larger than the preferred size), and you can change the position of the text relative to the icon through `setHorizontalTextPosition` and `setVerticalTextPosition`). The pixel gap between the text and icon is controlled through `setIconTextGap`.

You can also easily set keyboard mnemonics through `setMnemonic`. Doing so results in the specified character being underlined on the button and also results in `ALT-char` activating the button.

HTML in Button Labels

In JDK 1.2.2 and Swing 1.1.1 (and later), Sun added the ability to use HTML to describe the text in `JButton`s and `JLabel`s. Now you can easily have multiline text in buttons, mixed fonts and colors, and other fancy features in your buttons. Examples of HTML text in labels are given in Section 14.4 (The JLabel Component).

JButton Constructors

The `JButton` class has five constructors:

public JButton()

public JButton(String label)

public JButton(Icon image)

public JButton(String label, Icon image)

public JButton(Action action)

The first four constructors create an empty `JButton`, a `JButton` with a text label, a `JButton` with an image label, and a `JButton` with both a text and an image label, respectively. The fifth constructor, new in JDK 1.3, accepts an `Action` object to share state information (icon, label, for example) with other components.

Useful JButton (AbstractButton) Methods

The `JButton` class defines very few methods itself. However, `JButton` does implement a robust number of abstract methods from the `AbstractButton` class. The more common methods are listed below. Properties with `set` methods have corresponding `get` methods but are not listed.

public void setAction(Action action)

public Action getAction()

These two methods, new in JDK 1.3, set and get the `Action` object for the button. `Action` objects can define the text and icon to use for the button, as well as define an `actionPerformed` method. `Action`s permit you to share state and event handling among multiple components that interact with the user, for example, a button in a panel and a button in a toolbar.

public void setHorizontalAlignment(int alignment)

public void setVerticalAlignment(int alignment)

These methods set the horizontal and vertical alignment of the icon-text pair, respectively, relative to the button itself. The legal alignments values are the same as defined for `JLabel`.

public void setHorizontalTextPosition(int alignment)

public void setVerticalTextPosition(int alignment)

These methods set and get the horizontal and vertical position of the text, respectively, relative to button's image. The legal values are the same as those for setting the icon-text alignment for a `JLabel`.

public void setText(String label)

public void setFont(Font font)

The first method defines the text to display on the button, and the second method defines the font style of the text on the button. The text can include HMTL tags if enclosed in `<html>` ... `</html>`.

public void setIcon(Icon image)

public void setPressedIcon(Icon image)

public void setRolloverIcon(Icon image)

public void setDisabledIcon(Icon image)

The first method sets the image on the `JButton`. The next three methods set the image displayed when the button is pressed, when the mouse goes over the button, and when

the button is disabled, respectively. A `JButton` can be disabled through `setEnabled(false)`.

public void setEnabled(boolean state)

The `setEnable` method enables (`true`) or disables (`false`) the button. By default, a button is enabled.

public void setMargin(Insets margins)

This method specifies the margins between the button content and the button boundaries. For example,

```
JButton button = new JButton("Continue");
button.setMargin(new Insets(10,5,10,5));
```

sets a 10-pixel margin above and below the button text, and a 5-pixel margin to the left and right of the text.

Listing 14.6 illustrates the basic three basic types of buttons: a text button, an image button, and a combination button comprising both text and an image. The result is shown in Figure 14-5.

Listing 14.6 `JButtons.java`

```
import java.awt.*;
import javax.swing.*;

/** Simple example illustrating the use of JButton, especially
 *  the new constructors that permit you to add an image.
 */

public class JButtons extends JFrame {
  public static void main(String[] args) {
    new JButtons();
  }

  public JButtons() {
    super("Using JButton");
    WindowUtilities.setNativeLookAndFeel();
    addWindowListener(new ExitListener());
    Container content = getContentPane();
    content.setBackground(Color.white);
    content.setLayout(new FlowLayout());
    JButton button1 = new JButton("Java");
    content.add(button1);
    ImageIcon cup = new ImageIcon("images/cup.gif");
    JButton button2 = new JButton(cup);
    content.add(button2);
    JButton button3 = new JButton("Java", cup);
    content.add(button3);
    JButton button4 = new JButton("Java", cup);
    button4.setHorizontalTextPosition(SwingConstants.LEFT);
    content.add(button4);
    pack();
```

```
    setVisible(true);
  }
}
```

Figure 14-8. Swing `JButtons` support both text and images.

14.6 The JPanel Component

In the simplest case, you use a `JPanel` exactly the same as you would a `Panel`: Allocate the panel, drop components into the panel, and then add the `JPanel` to a `Container`. However, `JPanel` also acts as a replacement for `Canvas` (there is no `JCanvas`). When using `JPanel` as a drawing area in lieu of a `Canvas`, you need to follow two additional steps. First, you should set the preferred size with `setPreferredSize` (recall that the preferred size of a `Canvas` is just its current size, whereas a `Panel` and `JPanel` determine their preferred size from the components they contain). Second, you should use `paintComponent`, not `paint`, for drawing. And since double buffering is turned on by default, the first thing you normally do in `paintComponent` is clear the off-screen bitmap through `super.paintComponent`, as in

```
public void paintComponent(Graphics g) {
  super.paintComponent(g);
  ...
}
```

Note that if you are using Swing in Java 2, you can cast the `Graphics` object to a `Graphics2D` object and do all sorts of new stuff added in the Java 2D package; see Chapter 10 (Java 2D: Graphics in Java 2).

JPanel Constructors

`JPanel` has four constructors, as listed below. As a convenience, `JPanel` lets you supply the `LayoutManager` to the constructor in addition to specifying the manager later through `setLayout`. Virtually all `JPanel` methods are inherited from `JComponent` and not discussed (see the JComponent API for available methods).

public JPanel()

public JPanel(LayoutManager manager)

public JPanel(boolean isDoubleBuffered)

public JPanel(LayoutManager manager, boolean isDoubleBuffered)

The first constructor creates a `JPanel` with a default layout manager of `FlowLayout` and double buffering turned on. The remaining three constructors allow you to explicitly specify the layout manager and the double buffering property.

New Feature: Borders

Aside from double buffering, the most obvious new feature of `JPanel` is the ability to assign

borders. Actually, borders are available for every `JComponent`, and most of the Swing components install their own border. `JPanel` is about the only case where it makes real sense to install your own border. Swing gives you seven basic border types: titled, etched, beveled (regular plus a "softer" version), line, matte, compound, and empty. You can also create your own border, of course. You assign a `Border` through the `setBorder` method, and you create the `Border` either by calling constructors directly or, more often, by using one of the convenience factory methods in `BorderFactory`. For example:

```
JPanel p = new JPanel();
p.setBorder(BorderFactory.createTitledBorder("Java"));
```

The factory methods reuse existing `Border` objects whenever possible. Thus, to save resources, you should check for the available `BorderFactory` methods before creating a new `Border` object.

Core Approach

Whenever possible use one of the `BorderFactory` `createXxxBorder` methods to create a `Border` object.

Useful BorderFactory Methods

The `BorderFactory` class provides 23 `static` methods for creating different styles of `Border`s. The `Border` class is found in the `javax.swing.border` package. Also found in the `border` package is the `AbstractBorder` class, which is a base class implementing `Border`, from which `EmptyBorder`, `TitledBorder`, `LineBorder`, `EtchedBorder`, `BevelBorder`, `SoftBevelBorder`, `MatteBorder`, and `CompoundBorder` are derived. Of the 23 available factory methods, the most common are listed below. All methods are declared `public static`.

Border createEmptyBorder(int top, int left, int bottom, int right)

This factory method creates an `EmptyBorder` object that simply adds space (margins) around the component.

Border createLineBorder(Color color)

Border createLineBorder(Color color, int thickness)

These two factory methods create a colored `LineBorder`. By default, the border is one pixel wide. In the second constructor, you can explicitly state the border thickness in pixels. The `LineBorder` class also provides two `static` methods for creating a 1-pixel-wide black- or gray-lined border: `createBlackLineBorder` and `createGrayLineBorder`.

TitledBorder createTitledBorder(String title)

TitledBorder createTitledBorder(Border border, String title)

These two factory methods create a `TitledBorder` with the title located in the default position (`TOP`) at the left of the border's top line. By default, the border is an etched line unless you explicitly provide a border style as in the second constructor. `BorderFactory` provides additional factory methods to position the title. The `TitledBorder` class provides `setTitlePosition` and

`setTitleJustification` methods also. The position of title is relative to the top and bottom border line: `ABOVE_TOP`, `TOP`, `BELOW_TOP`, and `ABOVE_BOTTOM`, `BOTTOM`, `BELOW_BOTTOM`. The title justification can be `LEFT`, `CENTER`, or `RIGHT`. You can also set the color and font of the title through `setTitleColor` and `setTitleFont` or by selecting other `BorderFactory` factory methods.

Border createEtchedBorder()

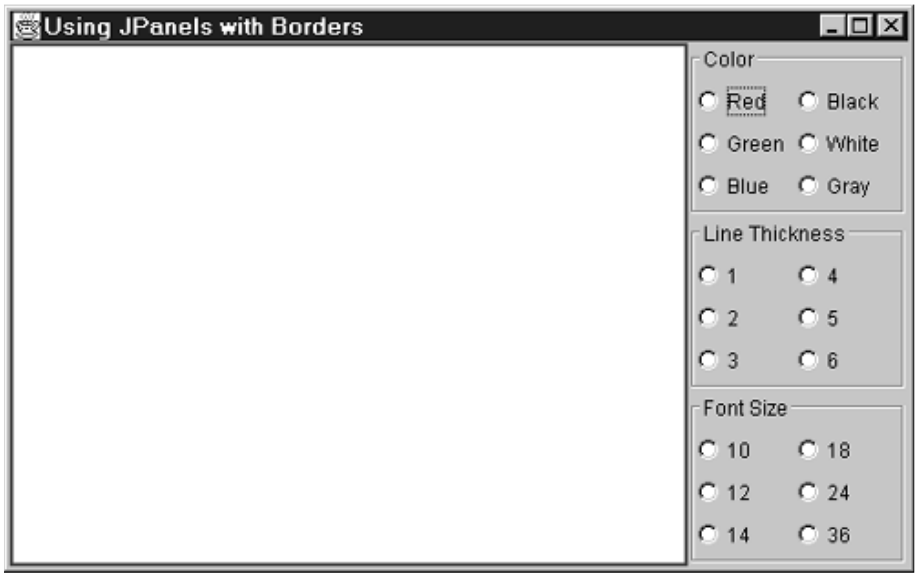
Border createEtchedBorder(Color highlight, Color shadow)

The first factory method creates a `LOWERED EtchedBorder` that has a 2-pixel-wide lowered groove. The groove is drawn in two colors to provide a three-dimensional effect. By default, the highlight color is slightly lighter than the background color and the shadow color is slightly darker than the background color. The second factory method allows you to explicitly specify the highlight and shadow colors. For a raised border, you must create an `EtchedBorder` object directly, passing in an argument of `RAISED` for the etch type.

In addition to the methods listed above, `BorderFactory` provides other static methods for creating empty, line, titled, and etched borders. Furthermore, the `BorderFactory` class provide methods for creating a beveled border (`createBevelBorder`), a raised or lowered bevel border (`createRaisedBevelBorder`, `createLoweredBevelBorder`), a matte border that paints a solid color or an `Icon` around the border (`createMatteBorder`), and a compound border made up of two borders (`createCompoundBorder`).

Listing 14.7 creates a simple `JPanel` with a blue custom `LineBorder` two pixels wide. Before setting the border, we set the `JPanel`'s preferred size to a `Dimension` of 400 pixels wide by 0 pixels high. The height of the `JPanel` is irrelevant, since the `JPanel` is placed in the `WEST` location of the `JFrame BorderLayout`. In addition, this example creates three panels with different button choices (see Listing 14.8). Each choice panel has a `TitledBorder`. The result is shown in Figure 14-9.

Figure 14-9. `JPanels` provide an added touch of custom borders.



Listing 14.7 `JPanels.java`

```
import java.awt.*;
```

```
import javax.swing.*;

/** Simple example illustrating the use of JPanels, especially
 *  the ability to add Borders.
 */

public class JPanels extends JFrame {
  public static void main(String[] args) {
    new JPanels();
  }

  public JPanels() {
    super("Using JPanels with Borders");
    WindowUtilities.setNativeLookAndFeel();
    addWindowListener(new ExitListener());
    Container content = getContentPane();
    content.setBackground(Color.lightGray);
    JPanel controlArea = new JPanel(new GridLayout(3, 1));
    String[] colors = { "Red", "Green", "Blue",
                        "Black", "White", "Gray" };
    controlArea.add(new SixChoicePanel("Color", colors));
    String[] thicknesses = { "1", "2", "3", "4", "5", "6" };
    controlArea.add(new SixChoicePanel("Line Thickness",
                                       thicknesses));
    String[] fontSizes = { "10", "12", "14", "18", "24", "36" };
    controlArea.add(new SixChoicePanel("Font Size",
                                       fontSizes));
    content.add(controlArea, BorderLayout.EAST);
    JPanel drawingArea = new JPanel();
    // Preferred height is irrelevant, since using WEST region.
    drawingArea.setPreferredSize(new Dimension(400, 0));
    drawingArea.setBorder
                (BorderFactory.createLineBorder (Color.blue, 2));
    drawingArea.setBackground(Color.white);
    content.add(drawingArea, BorderLayout.WEST);
    pack();
    setVisible(true);
  }
}
```

Listing 14.8 SixChoicePanel.java

```
import java.awt.*;
import javax.swing.*;

/** A JPanel that displays six JRadioButtons. */

public class SixChoicePanel extends JPanel {
  public SixChoicePanel(String title, String[] buttonLabels) {
    super(new GridLayout(3, 2));
    setBackground(Color.lightGray);
    setBorder(BorderFactory.createTitledBorder(title));
```

```
    ButtonGroup group = new ButtonGroup();
    JRadioButton option;
    int halfLength = buttonLabels.length/2; // Assumes even length
    for(int i=0; i<halfLength; i++) {
      option = new JRadioButton(buttonLabels[i]);
      group.add(option);
      add(option);
      option = new JRadioButton(buttonLabels[i+halfLength]);
      group.add(option);
      add(option);
    }
  }
}
```

14.7 The JSlider Component

In the AWT, the `Scrollbar` class did double duty as a control for interactively selecting numeric values and as a widget used to control scrolling. This double duty resulted in poor-looking sliders. Swing gives you a "real" slider: `JSlider`. You create a `JSlider` in a manner similar to creating a `Scrollbar`: the zero-argument constructor creates a horizontal slider with a range from 0 to 100 and an initial value of 50. You can also supply the orientation (through `JSlider.HORIZONTAL` or `JSlider.VERTICAL`), the range, and initial value to the constructor. You handle events by attaching a `ChangeListener`. The `JSlider stateChanged` method normally calls `getValue` to look up the current `JSlider` value.

New Features: Tick Marks and Labels

Swing sliders can have major and minor tick marks. Turn tick marks on through `setPaintTicks(true)`, and then specify the tick spacing with `setMajorTickSpacing` and `setMinorTickSpacing`. If you want to limit users to selecting values only at the tick marks, call `setSnapToTicks(true)`. Turn on drawing of labels at major tick marks through `setPaintLabels(true)`. You can also specify arbitrary labels (including `ImageIcons` or other components) by creating a `Dictionary` with `Integers` as keys and `Components` as values and then associating the `Dictionary` with the slider through `setLabelTable`.

Other `JSlider` capabilities include borders (as with all `JComponent`s), sliders that go from high to low instead of low to high (`setInverted(true)`), and the ability to determine when the mouse is in the middle of a drag (when `getValueIsAdjusting()` returns `true`) so that you can postpone action until the drag finishes.

JSlider Constructors

The `JSlider` class has five constructors:

public JSlider()

public JSlider(int orientation)

public JSlider(int min, int max)

public JSlider(int min, int max, int initialValue)

public JSlider(int orientation, int min, int max, int initialValue)

The first constructor creates a `JSlider` with a minimum and maximum value of 0 and 100, respectively. The initial value is 50. The second constructor creates a `JSlider` with the same values as the no-argument constructor; however, you can specify the orientation as either `HORIZONTAL` or `VERTICAL`. The remaining three constructors allow you to specify different minimum, maximum, and initial values, as well as orientation.

Useful JSlider Methods

`JSlider` defines 43 methods, of which only the 11 most popular are listed. All `set` methods have corresponding `get` methods.

public void setMinimum(int min)

public void setMaximum(int max)

public void setValue(int initialValue)

public void setOrientation(int orientation)

These methods allow you to set minimum, maximum, and initial slider value, as well as the orientation of the slider. The orientation can be `HORIZONTAL` or `VERTICAL`.

public void setPaintTicks(boolean paint)

public void setMinorTickSpacing(int stepSize)

public void setMajorTickSpacing(int stepSize)

By default, a `JSlider` does not display tick marks until you call `setPaintTicks(true)` and set the minor or major (or both) tick spacing by calling `setMinorTickSpacing` or `setMajorTickSpacing`.

Core Note

Tick marks on sliders will not be drawn unless either the major or minor tick spacing is set and `setPaintTicks` is `true`.

public void setSnapToTicks(boolean snap)

Setting `setSnapToTicks(true)` forces the slider cursor to align with a tick mark; a value of `false` (default) allows positioning of the slider cursor from the minimum to maximum values inclusive.

public void setInverted(boolean inverted)

The `setInverted` method reverses the maximum and minimum end points of the slider.

public void setPaintLabels(boolean paint)

public void setLabelTable(Dictionary labels)

The first method turns on or off the display of slider labels, which can be either `String`s or `Icon`s. However, labels are not actually drawn unless the tick spacing is also set. The

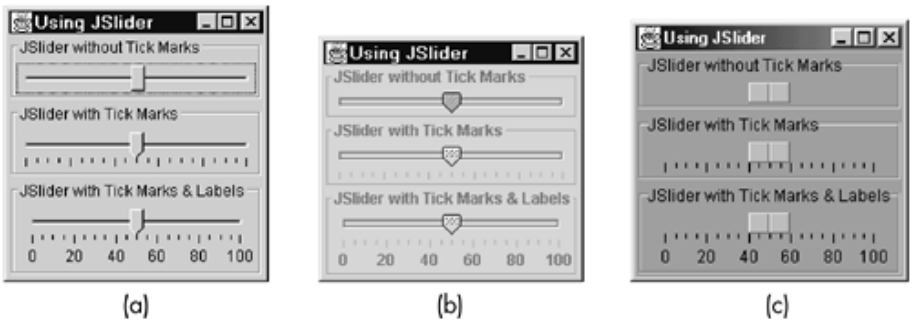
`JSlider` can automatically generate numerical labels, or you can define custom labels through `setLabelTable` by supplying a `Hashtable` where each key-data pair is represented by an `Integer` value and a `JComponent` (typically, a `JLabel`).

Core Note

Labels will not be drawn unless either the major or minor tick spacing is set and `setPaintLabels` *is* `true`*.*

Listing 14.9 provides an example of three sliders, one without tick marks, one with tick marks, and one with tick marks and labels. Figure 14-10 shows the result for a Windows, Motif, and Java look and feel.

Figure 14-10. `JSliders` provide full flexibility to add tick marks and labels: (a) look and feel for Windows, (b) Java (Metal), and (c) Motif.



Listing 14.9 `JSliders.java`

```
import java.awt.*;
import javax.swing.*;

/** Simple example illustrating the use of JSliders, especially
 *  the ability to specify tick marks and labels.
 */

public class JSliders extends JFrame {
  public static void main(String[] args) {
    new JSliders();
  }

  public JSliders() {
    super("Using JSlider");
    WindowUtilities.setNativeLookAndFeel();
    addWindowListener(new ExitListener());
    Container content = getContentPane();
    content.setBackground(Color.white);

    JSlider slider1 = new JSlider();
    slider1.setBorder(BorderFactory.createTitledBorder
                        ("JSlider without Tick Marks"));
    content.add(slider1, BorderLayout.NORTH);
```

```
    JSlider slider2 = new JSlider();
    slider2.setBorder(BorderFactory.createTitledBorder
                      ("JSlider with Tick Marks"));
    slider2.setMajorTickSpacing(20);
    slider2.setMinorTickSpacing(5);
    slider2.setPaintTicks(true);
    content.add(slider2, BorderLayout.CENTER);

    JSlider slider3 = new JSlider();
    slider3.setBorder(BorderFactory.createTitledBorder
                      ("JSlider with Tick Marks & Labels"));
    slider3.setMajorTickSpacing(20);
    slider3.setMinorTickSpacing(5);
    slider3.setPaintTicks(true);
    slider3.setPaintLabels(true);
    content.add(slider3, BorderLayout.SOUTH);

    pack();
    setVisible(true);
  }
}
```

14.8 The JColorChooser Component

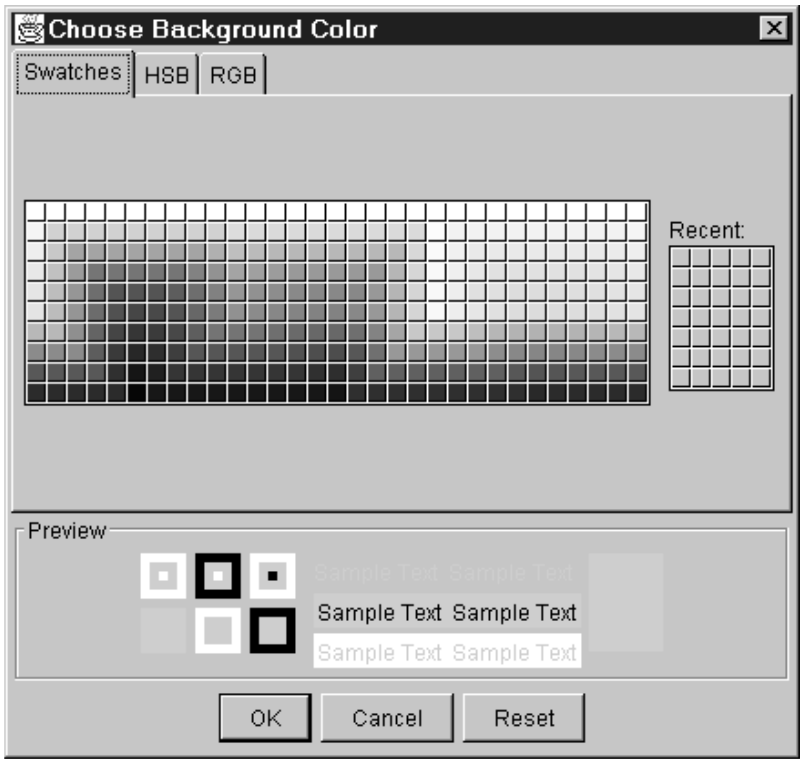
`JColorChooser` is a component that is new to Swing; the AWT has no equivalent. `JColorChooser` lets the user interactively select a `Color`. The default behavior is to present a dialog box containing a tabbed pane by which the user chooses the color through swatches, HSB (hue, saturation, brightness) values, or RGB values.

The simplest use is to call `JColorChooser.showDialog`, supplying parent `Component` as the first argument, the title as the second argument, and the initial color selection as the third argument, for example,

```
JColorChooser.showDialog
  (parent, "Select Background", getBackground());
```

The dialog presents an OK, Cancel, and Reset button as shown in Figure 14-11. If the user selects OK, the return value is the `Color` chosen. If the user cancels the dialog, either by pressing the `Esc` key, selecting the Cancel button, or closing the dialog, then the return value is `null`.

Figure 14-11. Selection of a color through the `JColorChooser` swatch tabbed pane.



You can also allocate a `JColorChooser` through a constructor. This approach is common if you want to display the color chooser somewhere other than in a pop-up dialog or if you are likely to display the color chooser many times. In the latter case, pass the `JColorChooser` instance to `JColorChooser.createDialog`, as in

```
JColorChooser chooser = new JColorChooser();
JDialog dialog = new JColorChooser.createDialog(
        this,           // parent component
        "Select Color", // title
        true,           // open as modal
        chooser,        // color chooser for dialog
        okListener,     // handle selecting color
        exitListener);  // handle cancel selection
```

The `okListener` and `exitListener` should implement `ActionListener` to capture the event of the user selecting a color and canceling the color chooser, respectively. A call to `dialog.setVisible(true)` displays the dialog containing the color chooser. Creating a `JColorChooser` and binding it to a `JDialog` produces significant performance benefits since `showDialog` creates a new instance of a `JColorChooser` each time.

Constructors

The two most common `JColorChooser` constructors are:

public JColorChooser() public JColorChooser(Color initialColor)

The first constructor creates a `JColorChooser` with an initial color selection of `Color.white`. The second constructor permits you to explicitly specify the initial selection color.

Useful JColorChooser Methods

The following paragraphs list the six most common methods of the `JColorChooser` class.

public static Color showDialog(Component parent, String title, Color initialColor)

This `static` method creates a modal dialog containing a *new* `JColorChooser` with the specified title and initial color selection. The `JColorChooser` is immediately displayed with the Swatch color tab. Either the selected `Color` is returned or, if the user selects Cancel, `null` is returned.

public static JDialog createDialog(Component parent, String title, boolean modal, JColorChooser chooser, ActionListener okListener ActionListener cancelListener)

This `static` method creates a customized `JDialog` holding the passed-in `JColorChooser`. The dialog can be modal or nonmodal. When creating the dialog, you must specify the two `ActionListener`s that will handle the user selecting OK and Cancel, respectively. After creating the dialog, you can set the color and display the dialog through `setColor` and `setVisible`, as in

```
JColorChooser chooser = new JColorChooser();
JDialog dialog = new JColorChooser.createDialog(...);
chooser.setColor(someColor);
dialog.setVisible(true);
```

In the `okListener`, determine the selected color from `getColor`, for example,

```
class okListener implements ActionListener {
  public void actionPerformed(ActionEvent e)) {
    Color color = chooser.getColor();
    ...
    repaint();
  }
}
```

public Color getColor()

public void setColor(Color color)

public void setColor(int red, int green, int blue)

public void setColor(int color)

The first method returns the selected `Color`. The remaining three methods set the selected color of the `JColorChooser`. In the second `setColor` method, the RGB `int` values must range from 0–255 inclusive. In the third `setColor` method, the `int` represents four packed color bytes. See the `Color` class on page 372 for details.

The simple example in Listing 14.10 creates a small `JFrame` with a button that pops up a `JColorChooser`. Upon selection of a color, the background of the `JFrame` content pane is set to the color selected. Figure 14-11 show the Swatch tab of the `JColorChooser`.

Listing 14.10 `JColorChooserTest.java`

```
import java.awt.*;
import java.awt.event.*;
```

```
import javax.swing.*;

/** Simple example illustrating the use of JColorChooser. */

public class JColorChooserTest extends JFrame
                               implements ActionListener {
  public static void main(String[] args) {
    new JColorChooserTest();
  }

  public JColorChooserTest() {
    super("Using JColorChooser");
    WindowUtilities.setNativeLookAndFeel();
    addWindowListener(new ExitListener());
    Container content = getContentPane();
    content.setBackground(Color.white);
    content.setLayout(new FlowLayout());
    JButton colorButton
      = new JButton("Choose Background Color");
    colorButton.addActionListener(this);
    content.add(colorButton);
    setSize(300, 100);
    setVisible(true);
  }

  public void actionPerformed(ActionEvent e) {
    // Args are parent component, title, initial color.
    Color bgColor
      = JColorChooser.showDialog(this,
                                 "Choose Background Color",
                                 getBackground());
    if (bgColor != null)
      getContentPane().setBackground(bgColor);
  }
}
```

14.9 Internal Frames

Many commercial Windows products such as Microsoft PowerPoint, Corel Draw, Borland JBuilder, and Allaire HomeSite are Multiple Document Interface (MDI) applications. This means that the program has one large "desktop" pane that holds all other windows. The other windows can be iconified (minimized) and moved around within this desktop pane, but not moved outside the pane. Furthermore, minimizing the desktop pane hides all the contained windows as well.

Swing introduced MDI support by means of two main classes. The first class, `JDesktopPane`, serves as a holder for the other windows. The second class, `JInternalFrame`, acts mostly like a `JFrame`, except that a `JInternalFrame` is constrained to stay inside the `JDesktopPane`. Using the `JInternalFrame` constructor with just a title results in an internal frame that is not resizable, closable, maximizable, or minimizable (i.e., iconifiable). However, `JInternalFrame` provides a five-argument constructor that accepts boolean values for each of these properties (in the order mentioned above). Internal frames also have two useful methods for controlling the z-order: `moveToFront` and `moveToBack`.

JInternalFrame Constructors

The `JInternalFrame` actually has six constructors to specify the initial behavior of the frame. The simplest two constructors, along with a fully specified constructor, are listed below:

public JInternalFrame()

public JInternalFrame(String title)

public JInternalFrame(String title, boolean resizable, boolean closeable, boolean maximizable, boolean iconifiable)

By default, an internal frame does not support the ability to resize, close, maximize, and iconify. Thus, the first two constructors create frames that the user can only move about on the desktop pane. The last constructor allows you to explicitly control the frame functionality.

Useful JInternalFrame Methods

The `JInternalFrame` class provides many methods similar to those found in `JFrame`. The most common ones are summarized below.

public void setCloseable(boolean closeable)

public void setIconifiable(boolean iconifiable)

public void setMaximizable(boolean maximizable)

public void setResizable(boolean resizable)

These four methods allow you to programmatically set the behavior of the internal frame. Each method has a corresponding `isXxx` method instead of a `getXxx` method to query the state behavior of the frame. Each `set` method can throw a `PropertyVetoException`.

public String getTitle()

public void setTitle(String title)

These two methods allow you to set and retrieve the title of the internal frame, respectively.

public void moveToBack()

public void moveToFront()

These two methods position the z-order of the internal frame relative to the other internal frames in the desktop pane.

public void show()

public void dispose()

The `show` method makes the internal frame visible on the screen. If the frame is already visible, then the frame is moved to the front. Before showing the frame, you must call `setBounds` (or `setLocation` and `setSize`) to position and establish the size of the frame. The `JLayeredPane` holding the internal frames has a `null` layout

manager. Thus, you are responsible for sizing and positioning the frame before displaying it. The second method, `dispose`, sets the visibility of the frame to `false` and, if not closed, fires an `INTERNAL_FRAME_CLOSED` event.

Core Note

Internal frames will not appear (`show`) on the desktop pane unless the bounds of the frame are explicitly specified.

public void setFrameIcon(Icon image)

This method sets the icon that is displayed in the upper-left corner of the internal frame.

public void addInternalFrameListener(InternalFrameListener listener)

public void removeInternalFrameListener(InternalFrameListener listener)

`JInternalFrame`s generate `InternalFrameEvent`s equivalent to `WindowEvent`s. The `InternalFrameListener` class defines seven abstract methods to handle internal frame events:

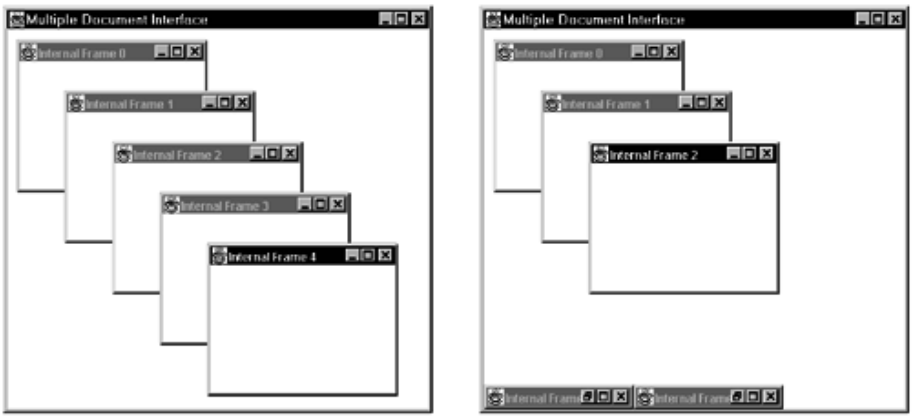
- `internalFrameActivated`
- `internalFrameClosed`
- `internalFrameClosing`
- `internalFrameDeactivated`
- `internalFrameDeiconified`
- `internalFrameIconified`
- `internalFrameOpened`

Either attach or remove a listener implementing these event methods through `addInternalFrameListener` or `removeInternalFrameListener`, respectively.

The `JDesktopPane` class provides two methods to obtain the `JInternalFrame`s contained in the desktop: `getAllFrames`, which returns an array of all visible and iconized internal frames, and `getAllFramesInLayer(layer)`, which returns all internal frames at the specified `layer` in the `JDesktopPane`.

The following example in Listing 14.11 creates five empty internal frames inside a desktop pane, which in turn resides inside a `JFrame`. Note that in JDK 1.2 the default visibility for a `JInternalFrame` is `true`. However, in JDK 1.3, the default visibility is `false` and you must call `setVisible(true)` to display the frame. The screen capture in Figure 14-12 (a) shows the result with all frames open, and Figure 14-12 (b) shows the result with two of the frames minimized.

Figure 14-12. Swing `JFrame`s support internal child frames: (a) all internal frames open, and (b) two internal frames minimized.



Core Note

In JDK 1.2, the default visibility of an internal frame is `true`*, whereas in JDK 1.3, the default visibility of an internal frame is* `false`*.*

Listing 14.11 `JInternalFrames.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/** Simple example illustrating the use of internal frames. */

public class JInternalFrames extends JFrame {
  public static void main(String[] args) {
    new JInternalFrames();
  }

  public JInternalFrames() {
    super("Multiple Document Interface");
    WindowUtilities.setNativeLookAndFeel();
    addWindowListener(new ExitListener());
    Container content = getContentPane();
    content.setBackground(Color.white);
    JDesktopPane desktop = new JDesktopPane();
    desktop.setBackground(Color.white);
    content.add(desktop, BorderLayout.CENTER);
    setSize(450, 400);
    for(int i=0; i<5; i++) {
      JInternalFrame frame
        = new JInternalFrame(("Internal Frame " + i),
                             true, true, true, true);
      frame.setLocation(i*50+10, i*50+10);
      frame.setSize(200, 150);
      frame.setBackground(Color.white);
      frame.setVisible(true);
      desktop.add(frame);
```

```
        frame.moveToFront();
      }
      setVisible(true);
    }
}
```

14.10 The JOptionPane Component

Static methods in the `JOptionPane` class let you easily create modal dialogs to show messages (`JOptionPane.showMessageDialog`), to ask for confirmation (`JOptionPane.showConfirmDialog`), to let the user enter text or to choose among predefined options (`JOptionPane.showInputDialog`), or to choose among a variety of buttons (`JOptionPane.showOptionDialog`). Each of these methods either returns an `int` specifying which button was pressed or returns a `String` specifying the option selected.

Useful JOptionPane Methods

`JOptionPane` is a robust class defining 26 constants, 7 constructors, and 59 methods. We only present five `static` methods for creating messages and input dialogs. All five methods define the same first three arguments for specifying the *parent* component of the dialog, the *message* to display in the dialog, and the *title* of the dialog window. Table 14.1 lists several helpful class constants typically used for arguments to `JOptionPane` methods.

Table 14.1. `JOptionPane` defined constants

Message Icon Type	`PLAIN_MESSAGE`
	`INFORMATION_MESSAGE`
	`QUESTION_MESSAGE (default)`
	`WARNING_MESSAGE`
	`ERROR_MESSAGE`

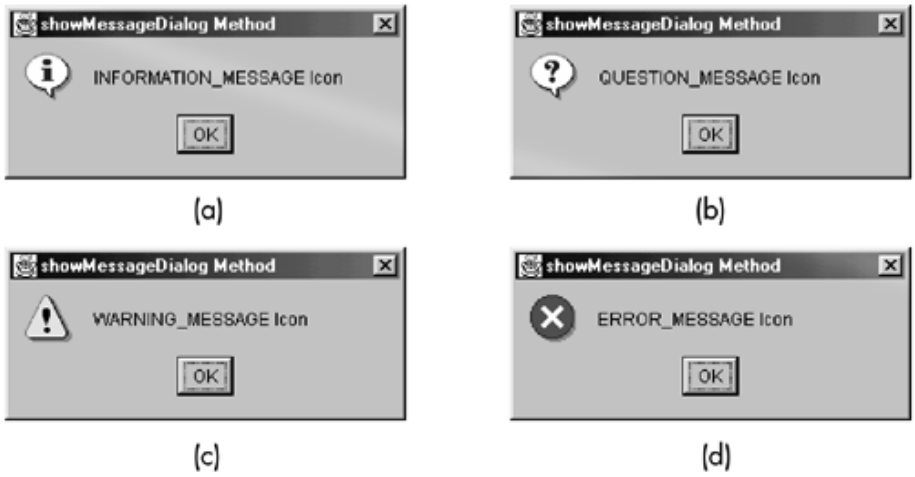
Confirm Dialog Type	`DEFAULT_OPTION`
	`OK_CANCEL_OPTION`
	`YES_NO_OPTION`
	`YES_NO_CANCEL_OPTION (default)`

Confirm Dialog Return Type	`YES_OPTION`
	`NO_OPTION`
	`CANCEL_OPTION`
	`CLOSED_OPTION`

public static void showMesssageDialog(Component parent, Object message, String title, int iconType)

This method creates a simple *modal* dialog displaying an icon, a message, and an OK button. Since the purpose of this dialog is to simply present the user a message while blocking the main thread of execution, this method returns no response value. Alongside the text message, the dialog can display one of four message icons based on the `iconType` value defined in Table 14.1. Example message dialogs are shown in Figure 14-13.

Figure 14-13. Various message dialogs (Windows look and feel) with selected message icons: (a) information, (b) question, (c) warning, and (d) error.

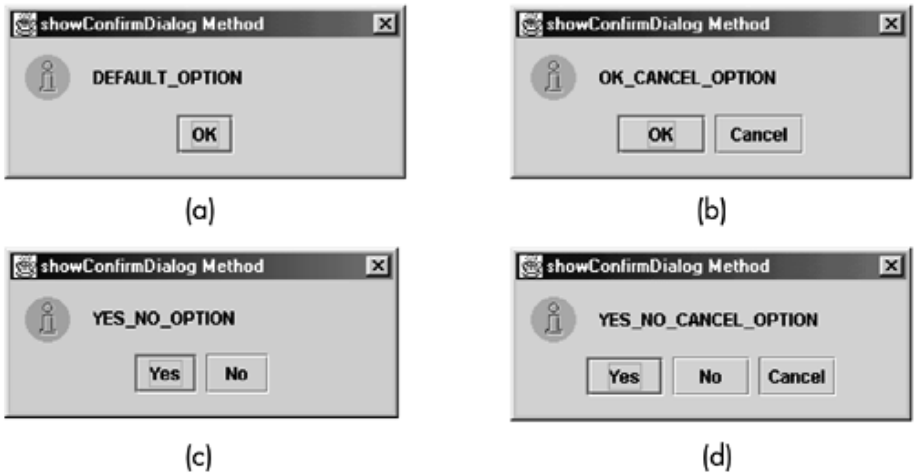


Specifying `PLAIN_MESSAGE` results in no displayed icon. Typically, the `message` argument is a `String`. However, the message can be any `Object`; a `JComponent` message is displayed as the component itself, an `Icon` message is displayed wrapped in a `JLabel`, and any other `Object` type is converted to a `String` through `toString` before displaying.

public static int showConfirmDialog(Component parent, Object message, String title, int optionType, int iconType)

The `showConfirmDialog` method creates a modal dialog where the user response is returned as an `int` mapping to one of the confirm dialog return types defined in Table 14.1 (`YES_OPTION`, `NO_OPTION`, `CANCEL_OPTION`, and `CLOSE_OPTION`). The `optionType` (legal option types are summarized in Table 14.1) determines which buttons are displayed, as shown in Figure 14-14.

Figure 14-14. Various confirmation dialogs (Java look and feel): (a) default, (b) OK/Cancel, (c) Yes/No, and (d) Yes/No/Cancel.



The following example code creates a confirmation dialog and processes the user response.

```
int response = JOptionPane.showConfirmDialog
                  (parentComponent,
                   "Do you like Java?",
```

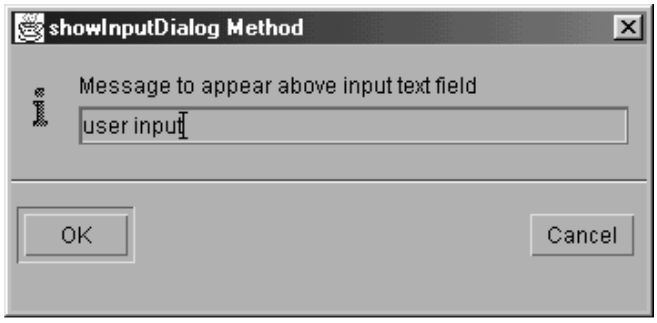
```
                    "Confirm Dialog Example",
                    JOptionPane.YES_NO_OPTION,
                    JOptionPane.QUESTION_MESSAGE);
if (response == JOptionPane.YES_OPTION) {
  // Do something for Yes option.
} else if (response == JOptionPane.NO_OPTION) {
  // Do something for No options.
} else {
  // Do something else.
}
```

The `iconType` defines the icon displayed along with the message. See Table 14.1 for legal values.

public static String showInputDialog(Component parent, Object message, String title, int iconType)

The `showInputDialog` method creates a modal dialog with an icon, textfield for user input, OK button, and Cancel button, as shown in Figure 14-15. Input dialogs always present an OK and Cancel button. Selecting OK returns the `String` in the textfield; otherwise, `null` is returned when Cancel is selected. The icon displayed alongside the message is determined by `iconType`. See Table 14.1 for legal `iconType` values.

Figure 14-15. `JOptionPane` input dialog (Motif look and feel).



public static Object showInputDialog(Component parent, Object message, String title, int iconType, Icon icon, Object[] selections Object initialSelection)

This `showInputDialog` method provides additional flexibility by allowing you to specify an array of objects (`selections`) to present to the user for selection. The selection values are displayed as `String`s, so the corresponding objects must provide a `toString` method. If the array contains fewer than 20 items, a combo box is presented (as shown in Figure 14-16); otherwise, a list is presented to the user. The `initialSelection` for the array is required. Selecting OK returns the `toString` value of the selected object; otherwise, a `null` string is returned when Cancel is selected. The icon displayed alongside the message is determined by the `iconType` parameter as defined in Table 14.1. Alternatively, you can specify a custom icon to display next to the dialog message (instead of the standard message icons) by supplying the `icon` argument.

Figure 14-16. `JOptionPane` input dialog where input is a combo box (Java look and feel).

public static Object showOptionDialog(Component parent, Object message, String title, int iconType, Icon image, Object[] selections Object initialSelection)

The `showOptionDialog` allows you to customize the dialog buttons (as shown in Figure 14-17) instead of going with the traditional OK and Cancel buttons. Typically, the `selections` array defines an array of `JButton`s, and the `initialSelection` specifies which `JButton` should receive initial focus. Any `String` object specified in the selection array is internally converted to a `JButton`. For example, the following is valid for defining the selection array:

Figure 14-17. `JOptionPane` option dialog where the options are an array of buttons (Windows look and feel).


```
Object[] selections = { "OK,
                        "Cancel";
                        new JButton("End Program") };
```

Note that any `JComponent` is also allowed in the selection array. However, the traditional use is to simply display a button. The icon displayed alongside the message is determined by the `iconType`. See Table 14.1 for legal `iconType` values. If you prefer, you can specify a custom icon to display next to the dialog message by supplying an `icon` argument.

Don't underestimate the flexibility and power of `showOptionDialog` and `showConfirmDialog` for capturing user input. Both of these methods accept an `Object` for the message, not just a `String`. You could easily populate a `JPanel` with check boxes or radio buttons and pass in the `JPanel` as the message argument for the dialog window to display.

All the dialog figures in this section were created from `JOptionPaneExamples.java`. The source code for this program can be downloaded from the on-line archive at http://www.corewebprogramming.com/.

14.11 The JToolBar Component

Swing provides a nice new component not available in the AWT: `JToolBar`. In the most basic use, `JToolBar` is little more than a `JPanel` acting as a container to hold small buttons. However, the main distinction is that `JToolBar` is dockable (or floatable), meaning that a `JToolBar` can be dragged out of the original window and kept as a stand-alone window. A `JToolBar` can also be

dragged back into the window or, if originally located at the top, can even be moved to the side of the window.

To build a `JToolBar`, you simply call the empty constructor (for a horizontal toolbar) or pass in `JToolBar.VERTICAL`. You typically place a horizontal toolbar in the `NORTH` or `SOUTH` region of a container that uses `BorderLayout`, and a vertical toolbar in the `EAST` or `WEST` region.

Core Approach

Place a horizontal toolbar in the `NORTH` or `SOUTH` location of a container by using `BorderLayout`. Similarly, place a vertical toolbar in the `EAST` or `WEST` location of a container by using `BorderLayout`.

The only complexity arising from `JToolBar` comes from the buttons placed inside. You could just drop normal `JButton`s into the toolbar or call `add` on an `Action` (a special subclass of `ActionListener` that includes information on labels and icons), which automatically creates a `JButton`. But the problem in both cases is that a graphical button for a toolbar differs from a normal `JButton` in two ways:

1. A toolbar button should be very small, whereas `JButton` maintains relatively large margins. Solution: call `setMargin` with an `Insets` object with all margin values zero (or at least small).

2. A `JButton` puts text labels to the right of the icon, but in toolbars we usually place the label below the icon. Solution: call `setVerticalTextPosition(BOTTOM)` and `setHorizontalTextPosition(CENTER)`.

An example of a floating `JToolBar`, following this technique, is shown in Figure 14-18.

Figure 14-18. A floating `JToolBar` built with `JButton`s having zero `Insets`.

Since setting the insets margins to zero and placing the button text at the bottom of the icon is a common task for virtually all buttons, Listing 14.12 provides a small class that extends `JButton` to implement these modifications.

Listing 14.12 `ToolBarButton.java`

```
import java.awt.*;
import javax.swing.*;

/** Part of a small example showing basic use of JToolBar.
 *  The point here is that dropping a regular JButton in a
 *  JToolBar (or adding an Action) in JDK 1.2 doesn't give
 *  you what you want -- namely, a small button just enclosing
 *  the icon, and with text labels (if any) below the icon,
 *  not to the right of it. In JDK 1.3, if you add an Action
 *  to the toolbar, the Action label is no longer displayed.
 */
```

```
public class ToolBarButton extends JButton {
  private static final Insets margins =
    new Insets(0, 0, 0, 0);

  public ToolBarButton(Icon icon) {
    super(icon);
    setMargin(margins);
    setVerticalTextPosition(BOTTOM);
    setHorizontalTextPosition(CENTER);
  }

  public ToolBarButton(String imageFile) {
    this(new ImageIcon(imageFile));
  }

  public ToolBarButton(String imageFile, String text) {
    this(new ImageIcon(imageFile));
    setText(text);
  }
}
```

JToolBar Constructors

`JToolBar` only has two constructors:

public JToolBar()

public JToolBar(int orientation)

The first constructor creates a `JToolBar` with a default horizontal orientation. The second constructor lets you specify an orientation of `HORIZONTAL` or `VERTICAL`.

Useful JToolBar Methods

`JToolBar` defines 23 methods. Only the five most common methods are covered below.

public JButton add(Action action)

This method creates a button capable of handling user interaction and then adds the button to the toolbar. The `Action` class is simply an interface extending `ActionListener` (which defines an `actionPerformed` method for handling an `ActionEvent`). The typical approach is to pass in an object that inherits from the concrete `AbstractAction` class (which implements the methods in the `Action` interface). `AbstractAction` allocates `Icon` and `String` instance variables for storing the button image and text. This concept is best illustrated in an example.

```
public class MyApplet extends JApplet{
  public void init{
    JToolBar toolbar = new JToolBar();
    toolbar.add(new PrintAction());
    getContentPane().add(toolbar,
                         BorderLayout.WEST);
    ...
```

```
    }
  ...
  }
  // "Button" for toolbar to handle event
  class PrintAction extends AbstractAction {
    public PrintAction() {
      super("Print",new ImageIcon("print.gif"));
    }
    public void actionPerformed(ActionEvent evt) {
     System.out.println("Print button selected.");
      ...
    }
  }
```

Alternatively, you could attach an `ActionListener` to a `JButton` and then add the button to the toolbar by calling `add(someButton)`, remembering that `container.add(...)` is in the `JToolBar` inheritance hierarchy.

In JDK 1.2, when an `Action` is added to a toolbar, the label is shown along with the icon. In JDK 1.3, only the `Action` icon is shown—the `Action` label is not displayed. This is the preferred behavior, as toolbars usually have tool-tips, not labels.

Core Note

In JDK 1.2, both the label and icon are displayed for an `Action` added to a toolbar. In JDK 1.3, only the icon is shown for an `Action` added to the toolbar.

Also, note that if you consult the JDK 1.3 API, the recommendation is to no longer add an `Action` to a toolbar, but to bind the `Action` to a button and then add the button to the toolbar, as in

```
JToolBar toolbar = new JToolBar();
JButton button = new JButton();
button.setAction(new PrintAction());
toolbar.add(button);
```

However, both the button label and icon are displayed in the toolbar for this case.

public void addSeparator()

public void addSeparator(Dimension size)

These two methods add a separator (blank space) between toolbar buttons. The default separator size is determined by the current look and feel. Alternatively, you can specify the width and height of the spacer by passing in a `Dimension` object.

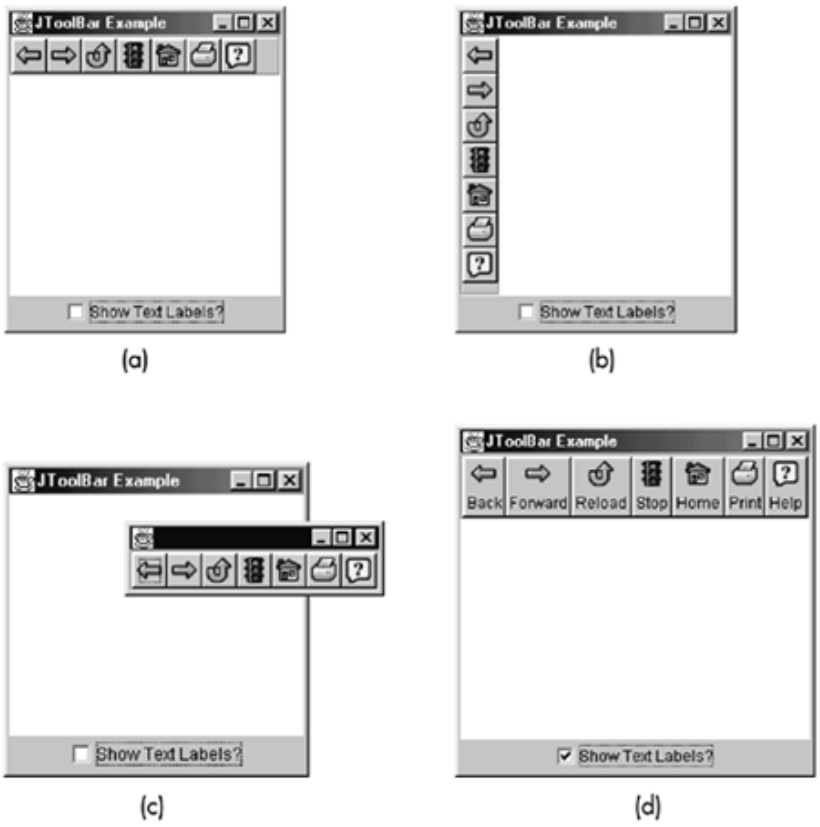
public void setFloatable(boolean float)

public void setOrientation(int orientation)

These two methods set the floating and orientation (`JToolBar.HORIZONTAL`, `JToolBar.VERTICAL`) property of the toolbar. Both methods have corresponding `getXxx` accessors for the retrieving these property values.

The example in Listing 14.13 and Listing 14.14 creates a simple `JFrame` with a `JToolBar` at the top. The toolbar is intended to look somewhat like a toolbar that might come with a simple Web browser. Text labels are initially turned off but can be toggled on or off by the use of a `JCheckBox`. Each `JButton` has an added tool-tip. Figure 14-19 illustrates the toolbar docked in the horizontal position, vertical position, floating independently, and with labels. Closing a floating toolbar returns the toolbar to the initial docked position.

Figure 14-19. A `JToolBar` (a) horizontally docked, (b) vertically docked, (c) floating, and (d) labeled.



Listing 14.13 `ToolBarExample.java`

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

/** Small example showing basic use of JToolBar. */

public class JToolBarExample extends JFrame
                             implements ItemListener {
  private BrowserToolBar toolbar;
  private JCheckBox labelBox;

  public static void main(String[] args) {
    new JToolBarExample();
  }

  public JToolBarExample() {
    super("JToolBar Example");
```

```
    WindowUtilities.setNativeLookAndFeel();
    addWindowListener(new ExitListener());
    Container content = getContentPane();
    content.setBackground(Color.white);

    JPanel panel = new JPanel(new BorderLayout());
    labelBox = new JCheckBox("Show Text Labels?");
    labelBox.setHorizontalAlignment(SwingConstants.CENTER);
    labelBox.addItemListener(this);
    panel.add(new JTextArea(10,30), BorderLayout.CENTER);
    panel.add(labelBox, BorderLayout.SOUTH);

    toolbar = new BrowserToolBar();
    content.add(toolbar, BorderLayout.NORTH);
    content.add(panel, BorderLayout.CENTER);

    pack();
    setVisible(true);
  }

  public void itemStateChanged(ItemEvent event) {
    toolbar.setTextLabels(labelBox.isSelected());
    pack();
  }
}
```

Listing 14.14 `BrowserToolBar.java`

```
import java.awt.*;
import javax.swing.*;

/** Part of a small example showing basic use of JToolBar.
 *  Creates a small dockable toolbar that is supposed to look
 *  vaguely like one that might come with a Web browser.
 *  Makes use of ToolBarButton, a small extension of JButton
 *  that shrinks the margins around the icon and puts text
 *  label, if any, below the icon.
 */

public class BrowserToolBar extends JToolBar {
  public BrowserToolBar() {
    String[] imageFiles =
      { "Left.gif", "Right.gif", "RotCCUp.gif",
        "TrafficRed.gif", "Home.gif", "Print.gif", "Help.gif" };
    String[] toolbarLabels =
      { "Back", "Forward", "Reload", "Stop",
        "Home", "Print", "Help" };
    Insets margins = new Insets(0, 0, 0, 0);
    for(int i=0; i<toolbarLabels.length; i++) {
      ToolBarButton button =
        new ToolBarButton("images/" + imageFiles[i]);
      button.setToolTipText(toolbarLabels[i]);
```

```
      button.setMargin(margins);
      add(button);
    }
  }

  public void setTextLabels(boolean labelsAreEnabled) {
    Component c;
    int i = 0;
    while((c = getComponentAtIndex(i++)) != null) {
      ToolBarButton button = (ToolBarButton)c;
      if (labelsAreEnabled) {
        button.setText(button.getToolTipText());
      } else {
        button.setText(null);
      }
    }
  }
}
```

14.12 The JEditorPane Component

`JEditorPane` is sort of a fancy text area that can display text derived from different file formats. The built-in version supports HTML and RTF (Rich Text Format) only, but you can build "editor kits" to handle special-purpose applications. In principle, you choose the type of document you want to display by calling `setContentType`, and you specify a custom editor kit through `setEditorKit`. Note that unless you extend the `JEditorPane` class, the only legal choices are `text/html`, `text/plain`, (which is also what you get if you supply an unknown type), and `text/rtf`.

In practice, however, `JEditorPane` is almost always used for displaying HTML. If you have plain text, you might as well use `JTextField`. RTF support is currently somewhat primitive. You put content into the `JEditorPane` in one of four ways.

1. The most common way to build a `JEditorPane` is through the constructor, where you supply either a `URL` object or a `String` corresponding to a URL (which, in applications, could be a file `URL` to read off the local disk). Note that the constructor accepting a `URL` throws an `IOException`, so the call needs to be placed inside a `try/catch` block.

2. You can use `setPage` on a `JEditorPane` instance that was created through the empty constructor. The `setPage` method also takes either a `URL` object or a `String` and also throws an `IOException`. This approach is generally used when the content is determined at runtime by some user action.

3. You can use `setText` on a `JEditorPane` instance, supplying a `String` that is the actual content.

4. On occasion, you can use `read`, supplying an `InputStream` and an `HTMLDocument` object.

In principle, a `JEditorPane` can be editable, but in practice, editing tends to look pretty poor, so a `JEditorPane` is most often used simply to display HTML. Beforehand, you would call `setEditable(false)` on the editor pane. In addition, as with all Swing components, you enable scrolling by dropping the editor pane in a `JScrollPane`. The following illustrates the most

common way to use JEditorPane:

```
String url = "http://host/path/file.html";
try {
  JEditorPane htmlPane = new JEditorPane(url);
  htmlPane.setEditable(false);
  someWindow.add(new JScrollPane(htmlPane);
} catch(IOException ioe) {
  System.err.println("Error displaying " + url);
}
```

Following Hypertext Links

In most cases, you use a noneditable JEditorPane to display HTML text. In such a case, you can detect when the user selects a link, you can determine which link was selected, and you can replace the contents of the JEditorPane with the document at the specified URL (by using setPage). To follow hyperlinks, attach a HyperlinkListener (notice that the method name is HyperlinkListener, not HyperLinkListener) through addHyperlinkListener, and implement the hyperlinkUpdate method to catch the events. Once you receive the event, look up the specific event type through getEventType and compare the event to HyperlinkEvent.EventType.ACTIVATED. This last point was not needed in early releases of Swing, but as of Java 1.2, if you neglect the step, then the link will be followed whenever the mouse simply moves over the link. Here is an example.

```
public class SomeWindow extends JFrame
                        implements HyperlinkListener {
  private JEditorPane htmlPane;

  ...

  public void hyperlinkUpdate(HyperlinkEvent event) {
    if (event.getEventType() ==
                HyperlinkEvent.EventType.ACTIVATED) {
      try {
        htmlPane.setPage(event.getURL());
      } catch(IOException ioe) {
        // Some warning to user
      }
    }
  }
}
```

JEditorPane Constructors

JEditorPane provides four constructors:

public JEditorPane()

public JEditorPane(String url)

public JEditorPane(URL url)

public JEditorPane(String mimeType, String document)

By default, the `JEditorPane` treats the document as plain text. By providing a URL, either as a `String` or `URL` object, you install the HTML editor kit and load the page associated with the URL. Both the second and third constructor can throw an `IOException`. The last constructor allows you to specify the MIME type (`text/plain`, `text/html`, or `text/rtf`) and the document.

Useful JEditorPane Methods

The following paragraphs list the more common `JEditorPane` methods. All property values have corresponding `get` and `set` methods.

public String getContentType()

public void setContentType(String mimeType)

These two methods get and set the MIME type, respectively. The supported MIME types are `text/plain`, `text/html`, and `text/rtf`.

public String getText()

public void setText(String document)

These two methods either retrieve or set the document in the `JEditorPane`. The text is interpreted according to the editor kit for the MIME type.

public void replaceSelection(String newText)

This thread-safe (blocking) method replaces the selected document content with the new text. Passing in `null` removes the selected text. If no text is highlighted, then the text is inserted at the cursor caret point. Replacing text is only allowed if the document is editable, `setEditable(true)`.

public URL getPage()

public void setPage(String page)

public void setPage(URL url)

The first method, `getPage`, returns the URL of the displayed document. The next two methods define and retrieve the document page for the `JEditorPane`. The content type is inferred from the URL and, further, allows registration of the appropriate editor. An absolute URL is required. Both methods can throw an `IOException` and require a `try/catch` block.

public synchronized void addHyperlinkListener(HyperlinkListener listener)

public synchronized void removeHyperlinkListener(HyperlinkListener listener)

These two methods add and remove the `HyperlinkListener` that is invoked when a hyperlink is selected, respectively. If `setEditable` is `true`, then hyperlinks are not followed.

public void read(InputStream in, Object description)

This method reads the input stream into the `JEditorPane` document. If the description of the document is of type `HTMLDocument` and the `HTMLEditorKit` is registered,

then the stream is read by the `HTMLEditorKit`; otherwise, the stream is read by the superclass (`JTextComponent`) as plain text.

Implementing a Simple Web Browser

By adding a simple URL text field to a `JEditorPane` that can display HTML and follow links, you have a simple but functioning Web browser. Listing 14.15 and Listing 14.16 provide the code to create a simple Web browser capable of following hyperlinks. The initial page loaded is http://www.corewebprogramming.com/. Figure 14-20 shows the result after we loaded the Milton S. Eisenhower Research and Technology Development Center Web page at http://www.jhuapl.edu/rc/.

Figure 14-20. A simple browser created with Swing components. `JEditorPane` provides support for displaying HTML.

Listing 14.15 `Browser.java`

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;

/** Very simplistic "Web browser" using Swing. Supply a URL on
 *  the command line to see it initially and to set the
 *  destination of the "home" button.
 */

public class Browser extends JFrame implements HyperlinkListener,
                                                ActionListener {
```

```
  public static void main(String[] args) {
    if (args.length == 0)
      new Browser("http://www.corewebprogramming.com/");
    else
      new Browser(args[0]);
  }

  private JIconButton homeButton;
  private JTextField urlField;
  private JEditorPane htmlPane;
  private String initialURL;

  public Browser(String initialURL) {
    super("Simple Swing Browser");
    this.initialURL = initialURL;
    addWindowListener(new ExitListener());
    WindowUtilities.setNativeLookAndFeel();
    JPanel topPanel = new JPanel();
    topPanel.setBackground(Color.lightGray);
    homeButton = new JIconButton("home.gif");
    homeButton.addActionListener(this);
    JLabel urlLabel = new JLabel("URL:");
    urlField = new JTextField(30);
    urlField.setText(initialURL);
    urlField.addActionListener(this);
    topPanel.add(homeButton);
    topPanel.add(urlLabel);
    topPanel.add(urlField);
    getContentPane().add(topPanel, BorderLayout.NORTH);

    try {
        htmlPane = new JEditorPane(initialURL);
        htmlPane.setEditable(false);
        htmlPane.addHyperlinkListener(this);
        JScrollPane scrollPane = new JScrollPane(htmlPane);
        getContentPane().add(scrollPane, BorderLayout.CENTER);
    } catch(IOException ioe) {
       warnUser("Can't build HTML pane for " + initialURL
                + ": " + ioe);
    }

    Dimension screenSize = getToolkit().getScreenSize();
    int width = screenSize.width * 8 / 10;
    int height = screenSize.height * 8 / 10;
    setBounds(width/8, height/8, width, height);
    setVisible(true);
  }

  public void actionPerformed(ActionEvent event) {
    String url;
    if (event.getSource() == urlField) {
      url = urlField.getText();
```

```
    } else { // Clicked "home" button instead of entering URL.
      url = initialURL;
    }
    try {
      htmlPane.setPage(new URL(url));
      urlField.setText(url);
    } catch(IOException ioe) {
      warnUser("Can't follow link to " + url + ": " + ioe);
    }
  }

  public void hyperlinkUpdate(HyperlinkEvent event) {
    if (event.getEventType() ==
        HyperlinkEvent.EventType.ACTIVATED) {
      try {
        htmlPane.setPage(event.getURL());
        urlField.setText(event.getURL().toExternalForm());
      } catch(IOException ioe) {
        warnUser("Can't follow link to "
                 + event.getURL().toExternalForm() + ": " + ioe);
      }
    }
  }

  private void warnUser(String message) {
    JOptionPane.showMessageDialog(this, message, "Error",
                                  JOptionPane.ERROR_MESSAGE);
  }
}
```

Listing 14.16 `JIconButton.java`

```
import javax.swing.*;

/** A regular JButton created with an ImageIcon and with borders
 *  and content areas turned off.
 */

public class JIconButton extends JButton {
  public JIconButton(String file) {
    super(new ImageIcon(file));
    setContentAreaFilled(false);
    setBorderPainted(false);
    setFocusPainted(false);
  }
}
```

HTML Support and JavaHelp

Although HTML support is getting better, `JEditorPane` still only supports a subset of the HTML 4.0 standard. Many constructs cannot be displayed properly, and worse yet, many standard constructs crash the `JEditorPane`, generating long and ugly error messages. So, although writing a simple browser is fun, it is risky in a real application to accept HTML input that you haven't

tested previously. So perhaps the single best use of `JEditorPane` is to display noneditable HTML that you have written (and tested!) for on-line help to your application. The use of `JEditorPane` to display on-line help is such a good idea that Sun has provided a small package called `JavaHelp` that assists by creating an outline (displayed in a `JTree`), generating an index, and so forth. See the http://java.sun.com/projects/javahelp/ home page for more details.

14.13 Other Simple Swing Components

Finally, we'll very briefly mention a couple of other simple Swing components that map to AWT equivalents: `JCheckBox`, `JRadioButton`, `JTextField`, and `JTextArea`. In addition, we'll briefly cover `JFileChooser` for opening a dialog to select files. For additional information on these topics, please see *Core Java Foundation Classes* by Kim Topley.

The JCheckBox Component

`JCheckBox` is similar to `Checkbox` (but note the capital B in `JCheckBox`). You can attach either an `ActionListener` or an `ItemListener` to monitor events. If you use an `ActionListener`, you'll want to call `isSelected` to distinguish a selection from a deselection. If you use an `ItemListener`, the `ItemEvent` itself has information regarding the selection state: call `getStateChange` and compare the result to `ItemEvent.SELECTED` or `ItemEvent.DESELECTED`. Unless you are sure that the background color of the `JCheckBox` matches the background color of the `Container`, you should call `setContentAreaFilled(false)`. In addition, you can supply an icon to replace the normal checkable square (through `setIcon`), but if you do, be sure to also supply an icon to display when the check box is selected (`setSelectedIcon`). Listing 14.17 demonstrates creating check boxes and processing `ItemListener` and `ActionListener` events to obtain the state of the check box. The result of `JCheckBoxTest.java` is shown in Figure 14-21.

Figure 14-21. A standard `JCheckBox`.

Listing 14.17 `JCheckBoxTest.java`

```
import javax.swing.*;
import java.awt.event.*;

public class JCheckBoxTest extends JPanel
                           implements ItemListener,
                                      ActionListener{
  JCheckBox checkBox1, checkBox2;

  public JCheckBoxTest() {
    checkBox1 = new JCheckBox("Java Servlets");
    checkBox2 = new JCheckBox("JavaServer Pages");
    checkBox1.setContentAreaFilled(false);
    checkBox2.setContentAreaFilled(false);
    checkBox1.addItemListener(this);
    checkBox2.addActionListener(this);
```

```
    add(checkBox1);
    add(checkBox2);
  }

  public void actionPerformed(ActionEvent event) {
    System.out.println("JavaServer Pages selected: " +
                       checkBox2.isSelected());
  }

  public void itemStateChanged(ItemEvent event) {
    JCheckBox checkbox = (JCheckBox)event.getItem();

    if (event.getStateChange() == ItemEvent.SELECTED) {
      System.out.println(checkbox.getText() + " selected.");
    } else {
      System.out.println(checkbox.getText() + " deselected.");
    }
  }

  public static void main(String[] args) {
    JPanel panel = new JCheckBoxTest();
    WindowUtilities.setNativeLookAndFeel();
    WindowUtilities.openInJFrame(panel, 300, 75);
  }
}
```

The JRadioButton Component

A `JRadioButton` is somewhat similar to a `Checkbox` inside a `CheckboxGroup` (see Section 13.16, "Check Box Groups (Radio Buttons)"). The approach is to create several `JRadioButton`s and add them to a `ButtonGroup`. All radio buttons added to the `ButtonGroup` are logically grouped together. As with `JCheckBox`, you can attach either an `ActionListener` or an `ItemListener` to process events. However, only the radio button that is clicked will receive an `ActionEvent`, and both the one clicked and the one that becomes deselected receive an `ItemEvent`. Unless you are sure that the background color of the `JRadioButton` matches the background color of the `Container`, you should call `setContentAreaFilled(false)`. As with `JCheckBox`, you can supply an icon to replace the normal button look (through `setIcon`), but if you do, also supply an icon to be displayed when the radio button is selected (`setSelectedIcon`). An example of three radio buttons is provided in Listing 14.18, with the results shown in Figure 14-22.

Figure 14-22. A logical group of `JRadioButton`s.

Listing 14.18 `JRadioButtonTest.java`

```
import javax.swing.JRadioButton;
```

```
import javax.swing.ButtonGroup;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JRadioButtonTest extends JPanel
                              implements ItemListener {

  public JRadioButtonTest() {

    String[] labels = {"Java Swing","Java Servlets",
                       "JavaServer Pages"};
    JRadioButton[] buttons = new JRadioButton[3];
    ButtonGroup group = new  ButtonGroup();

    for(int i=0; i<buttons.length; i++) {
      buttons[i] = new JRadioButton(labels[i]);
      buttons[i].setContentAreaFilled(false);
      buttons[i].addItemListener(this);
      group.add(buttons[i]);
      add(buttons[i]);
    }
  }

  public void itemStateChanged(ItemEvent event) {
    JRadioButton radiobutton = (JRadioButton)event.getItem();

    if (event.getStateChange() == ItemEvent.SELECTED) {
      System.out.println(radiobutton.getText() + " selected.");
    } else {
      System.out.println(radiobutton.getText() + " deselected.");
    }
  }

  public static void main(String[] args) {
    JPanel panel = new JRadioButtonTest();
    WindowUtilities.setNativeLookAndFeel();
    WindowUtilities.openInJFrame(panel, 400, 75);
  }
}
```

The JTextField Component

The basic use of the `JTextField` parallels almost exactly the AWT `TextField`: specify the initial text, width, and alignment in the constructor. Alternatively, after creating a `JTextField` with the no argument constructor, you can use `setText`, `setColumns`, and `setHorizontalAlignment`. Legal alignment values are `JTextField.LEFT`, `JTextField.CENTER`, and `JTextField.RIGHT`. To retrieve the entered text, call `getText`. Capture `ActionEvent`s on Enter and `DocumentEvent`s on regular keys. Note that `JTextField` does not do double duty as a password field; use `JPasswordField` instead.

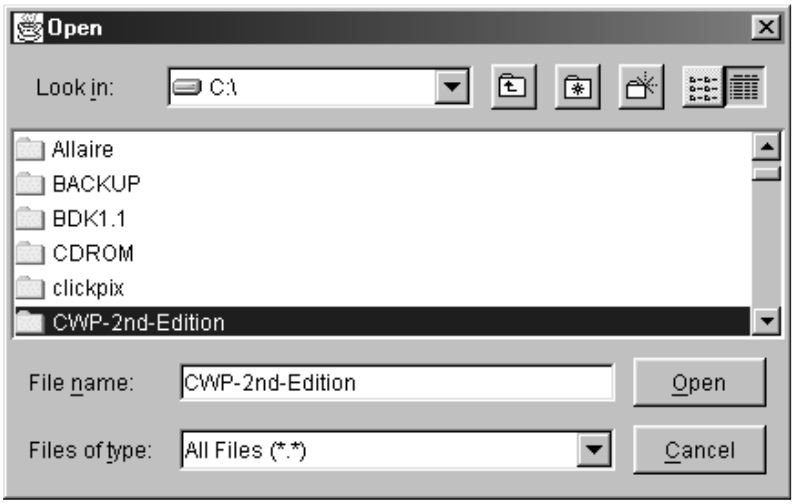
The JTextArea Component

`JTextArea` is similar to the AWT `TextArea`, but two things are different. First, unlike the case in `TextArea`, scrolling is not a built-in behavior. Instead, as with other Swing components, you control the scrolling behavior by wrapping the component in a `JScrollPane`. Second, `JTextArea` is only for simple text, but Swing also provides `JTextPane` and `JEditorPane`, which support much more complex options. In particular, `JEditorPane`, discussed in Section 14.12 (The JEditorPane Component), can display HTML and RTF text.

The JFileChooser Component

The `JFileChooser` control, shown in Figure 14-23, lets users interactively select a file by browsing directories. Normal use involves allocating a `JFileChooser` by passing in a `String` for the directory to the constructor (pass in "." to indicate the current directory). You can optionally specify a default file (`setSelectedFile`) and limit the file types displayed by writing and attaching a `FileFilter`. You can set the dialog title through `setDialogTitle`, and you can display the file chooser by calling `showOpenDialog` or `showSaveDialog`, (passing in the parent `Frame`). `FileChooser` returns an `int` value. If the `int` matches `JFileChooser.APPROVE_OPTION`, then the user selected a file and did not choose Cancel. Call `getSelectedFile` to retrieve the selected file. For example,

Figure 14-23. `JFileChooser` interactively permits the user to select one or multiple files.




```
JFileChooser chooser = new JFileChooser(".");
int result = chooser.showOpenDialog(parent);
File file = chooser.getSelectedFile();
if (file != null &&
        result == JFileChooser.APPROVE_OPTION) {
  // Do something
}
```

To permit the user to select multiple files, call `setMultiSelectionEnabled(true)` before opening the dialog, and use `getSelectedFiles` to retrieve the response and store it in a `File` array.

Lastly, note that all `JComponent`s can have a custom `Border` by calling `setBorder`, as covered in Section 14.6 (The JPanel Component), and can display custom tool-tips when the mouse pauses over the component (see `setToolTipText`).

14.14 Summary

This chapter introduced many of the standard Swing components that map to the familiar AWT components, including `JLabel`, `JButton`, `JSlider`, `JPanel`, `JApplet`, and `JFrame`, plus some new components like `JColorChooser`, `JOptionPane`, and `JInternalFrame` not seen in the AWT. Except for the windowing components, Swing components are all lightweight and drawn completely to memory through Java code. Instead of adding components to windows directly, you add components to the content pane. The content pane in `JFrame` and `JApplet` is simply a `JPanel` that by default implements double buffering. `JFrame` and `JApplet` now default to the same layout manager, `BorderLayout`.

As seen, Swing components are very rich and exciting, adding the capability for a selectable look and feel, custom borders, and tool-tips. Buttons can now include images instead of just text and can even display HTML. Sliders are reversible and can have image labels for the tick marks. The modal `JOptionPane` can present nearly any `JComponent` for the user to select a choice or enter a response, from text input, to a combo list, to check boxes, to custom buttons. Selecting a color is as simple as displaying a `JColorChooser`.

In this chapter, we've only scratched the surface of the Swing API. The next chapter examines more advanced components, for instance, `JList`, `JTree`, and `JTable`. These advanced components are slightly more complicated because they are built around data models and custom cell renderers. Once mastered, though, these advanced Swing components will allow you to create powerful and professional user interfaces for your Java programs.

 CONTENTS