CHAPTER 4

# Core Swing Components

IN CHAPTER 3, WE EXPLORED THE Model-View-Controller pattern used by the components of the JFC/Swing project. In this chapter, we'll begin to explore how to use the key parts of the many available components.

All Swing components start with the `JComponent` class. Although some parts of the Swing libraries aren't rooted with the `JComponent` class, all the components share `JComponent` as the common parent class at some level of their ancestry. It's with this `JComponent` class that common behavior and properties are defined. In this chapter, we look at common functionality such as component painting, customization, tooltips, and sizing.

As far as specific `JComponent` descendent classes are concerned, we'll look at the `JLabel`, `JButton`, and `JPanel`, three of the more commonly used Swing component classes. They require understanding of the `Icon` interface for displaying images within components, as well as of the `ImageIcon` class for when using predefined images and the `GrayFilter` class for support. In addition, we'll look at the `AbstractButton` class, which serves as the parent class to the `JButton`. The data model shared by all `AbstractButton` subclasses is the `ButtonModel` interface; we'll look at that and the specific implementation class, the `DefaultButtonModel`.

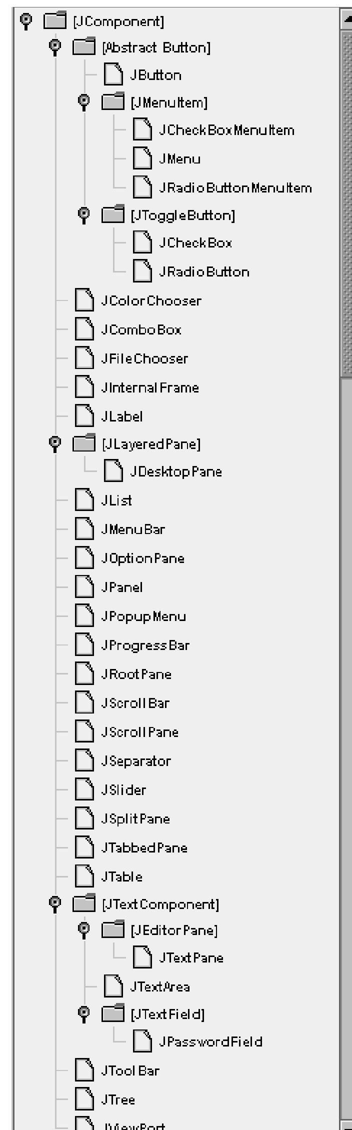



*Figure 4-1: JComponent class hierarchy diagram*

## Class JComponent

The `JComponent` class serves as the abstract root class from which all Swing components descend. The `JComponent` class has 39 descendent subclasses, each of which inherits much of the `JComponent` functionality. Figure 4-1 shows this hierarchy.

Although the `JComponent` class serves as the common root class for all Swing components, many classes in the libraries for the Swing project descend from classes other than `JComponent`. Those include all the high-level container objects such as `JFrame`, `JApplet`, and `JInternalFrame`, as well as `Box`, all the Model-View-Controller (MVC)-related classes, event-handling–related interfaces and classes, and much more. All of these will be discussed in later chapters.

Although all Swing components extend `JComponent`, the `JComponent` class extends the AWT `Container` class, which in turn extends from the AWT `Component` class. This means that many aspects of the `JComponent` are shared with both the AWT `Component` and `Container` classes.

> **NOTE** *`JComponent` extends from the `Container` class, but most of the `JComponent` subclasses aren't themselves containers of other components. To see if a particular Swing component is truly a container, check the bean info for the class to see if the `isContainer` property is set to `true`. To get the `BeanInfo` for a class, ask the `Introspector`.*

## Component Pieces

The `JComponent` class defines many aspects of AWT components that go above and beyond the capabilities of the original AWT component set. This includes customized painting behavior and the several different ways to customize display settings, such as colors, fonts, and any other client-side settings.

### Painting JComponent Objects

Because the Swing `JComponent` class extends from the `Container` class, the basic AWT painting model is followed: All painting is done through the `paint()` method, and the `repaint()` method is used to trigger updates. However, many tasks are done differently. The `JComponent` class optimizes many aspects of painting for improved performance and extensibility. In addition, the `RepaintManager` class is available to customize painting behavior even further.

> **NOTE** *The `public void update(Graphics g)` method, inherited from `Component`, is never invoked on Swing components.*

To improve painting performance and extensibility, the `JComponent` splits the painting operation into three tasks. The `public void paint(Graphics g)` method is subdivided into three separate (`protected`) method calls. In the order called, they are `paintComponent(g)`, `paintBorder(g)`, `paintChildren(g)`, with the `Graphics` argument passed through from the original `paint()` call. The component itself is first painted through `paintComponent(g)`. If you want to customize the painting of a Swing component, you override `paintComponent()` instead of `paint()`. Unless you want to completely replace all the painting, you would call `super.paintComponent()` first, as shown here, to get the default `paintComponent()` behavior.

```
public class MyComponent extends JPanel {
  protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    // customize after calling super.paintComponent(g)
  }
  ...
}
```

> **NOTE** *When running a program that uses Swing components within the Java 2 platform, the* `Graphics` *argument passed to the* `paint()` *method and on to* `paintComponent()` *is technically a* `Graphics2D` *argument. Therefore, after casting the* `Graphics` *argument to a* `Graphics2D` *object, you could use the Java2D capabilities of the Java 2 platform, as you would when defining a drawing* `Stroke`*,* `Shape`*, or* `AffineTransform`*.*

The `paintBorder()` and `paintChildren()` methods tend not to be overridden. The `paintBorder()` method draws a border around the component, a concept described more fully in Chapter 7. The `paintChildren()` method draws the components within the Swing container object, if any are present.

To optimize painting, the `JComponent` class provides three additional painting properties: opacity, optimization, and double buffering.

The opacity setting for a `JComponent` defines whether a component is transparent. When transparent, the container of the `JComponent` must paint the background behind the component. To improve performance, you can leave the `JComponent` opaque and let the `JComponent` draw its own background, instead of relying on the container to draw the covered background.

The optimization setting determines whether immediate children can overlap or not. If children can't overlap, the repaint time is reduced considerably. By default, optimized drawing is enabled for most Swing components, except for `JDesktopPane`, `JLayeredPane`, and `JViewport`.

By default, all Swing components double buffer their drawing operations into a buffer shared by the complete container hierarchy, that is, all the components within a window (or subclass). This greatly improves painting performance, because when double buffering is enabled there is only a single screen update drawn.

> **NOTE** *For synchronous painting, you can call one of the* `public void paintImmediately()` *methods. (Arguments are either a* `Rectangle` *or its parts—position and dimensions.) However, you'll rarely need to call this directly unless your program has real-time painting requirements.*

The public void `revalidate()` method of `JComponent` also offers painting support. When called, the high-level container of the component validates itself. This is unlike the AWT approach requiring a direct call to the `revalidate()` method of that high-level component.

The last aspect of the Swing component painting enhancements is the `RepaintManager`.

### *Class RepaintManager*

The `RepaintManager` is responsible for ensuring the efficiency of repaint requests on the currently displayed Swing components, making sure the smallest "dirty" region of the screen is updated when a region becomes invalid.

Although rarely customized, the `RepaintManager` class is public and provides a static installation routine to use a custom manager: `public static void setCurrentManager(RepaintManager manager)`.

To get the current manager, just ask with `public static void currentManager(JComponent)`. The argument is usually null, unless you've customized the manager to provide component-level support. Once you have the manager, one thing you can do is get the offscreen buffer for a component as an `Image`. Because the buffer is what is eventually shown on the screen, this effectively allows you to do a screen dump of the inside of a window (or any `JComponent`).

```
RepaintManager manager = RepaintManager.currentManager(null);
Image htmlImage = manager.getOffscreenBuffer(comp, comp.getWidth(), comp.getHeight());
```

Table 4-1 shows the two properties of `RepaintManager`. They allow you to disable double buffering for all drawing operations of a component (hierarchy) and to set the maximum double buffer size, which defaults to the end user's screen size.

| PROPERTY NAME | DATA TYPE | ACCESS |
|---|---|---|
| doubleBufferingEnabled | boolean | read-write |
| doubleBufferMaximumSize | Dimension | read-write |

*Table 4-1: RepaintManager properties*

**TIP** *To globally disable double-buffered drawing, call the following:* `RepaintManager.currentManager(aComponent).setDoubleBufferingEnabled(false)`.

Although it's rarely done, providing your own `RepaintManager` subclass does allow you to customize the mechanism of painting dirty regions of the screen, or at least track when they're done. The mechanisms can be customized by overriding any of the following four methods:

```
public synchronized void addDirtyRegion(JComponent component, int x, int y, int
width, int height)
public Rectangle getDirtyRegion(JComponent component)
public void markCompletelyClean(JComponent component)
public void markCompletelyDirty(JComponent component)
```

## *Class UIDefaults*

The `UIDefaults` represents a lookup table containing the display settings installed for the current look and feel, such as which font to use within a `JList`, as well as what color or icon should be displayed within a `JTree` node. The use of `UIDefaults` will be completely described in Chapter 18 with the coverage of Java's pluggable look and feel architecture. Nevertheless, a short description of its usage is needed here.

Whenever you create a component, the component automatically asks the `UIManager` to look in the `UIDefaults` settings for the current settings for that component. Most color- and font-related component settings, as well as some others not related to colors and fonts, are configurable. If you don't like a particular setting, you can simply change it.

**NOTE** *All predefined resource settings in the* `UIDefaults` *table implement the* `UIResource` *interface, which allows the components to monitor which settings have been customized just by looking for those settings that don't implement the interface.*

You can find the listed settings in either one of two places in this book. Appendix A contains a complete alphabetical listing of all known settings for the predefined look-and-feels. In addition, included with the description of each component is a table containing the `UIResource`-related property elements. (To find the specific component section in the book, consult the Index.)

Once you know the name of a setting, you can store a new setting with the `public static void put(Object key, Object value)` method of `UIManager`, where `key` is the key string. For instance, the following code will change the default background color of newly created buttons to black and the foreground color to red:

```
UIManager.put("Button.background", Color.black);
UIManager.put("Button.foreground", Color.red);
```

If you're creating your own components or just need to find out the current value setting, you need only ask the `UIManager`. Although the `public static Object get(Object key)` method is the most generic, it requires you to cast the return value to the appropriate class type. Alternately, you could use one of the more specific `get`*`XXX`*`()` methods, which does the casting for you, to return the appropriate type. Those methods are listed in Table 4-2.

| UIMANAGER GETTER METHODS |
| --- |
| public static Border getBorder(Object key) |
| public static Color getColor(Object key) |
| public static Dimension getDimension(Object key) |
| public static Font getFont(Object key) |
| public static Icon getIcon(Object key) |
| public static Insets getInsets(Object key) |
| public static int getInt(Object key) |
| public static String getString(Object key) |
| public static ComponentUI getUI(JComponent target) |

*Table 4-2: UIManager methods for getting UIResource properties*

> **NOTE** *You can also work with the `UIDefaults` directly, by calling the public static `UIDefaults getDefaults()` method of `UIManager`.*

## *Client Properties*

In addition to the `UIManager` maintaining a table of key-value pair settings, each instance of every component can manage its own set of key-value pairs. This is useful for maintaining aspects of a component that may be specific to a particular look and feel, or for maintaining data associated with a component without requiring the definition of new classes or methods to store such data.

```
public final void putClientProperty(Object key, Object value)
public final Object getClientProperty(Object key)
```

> **NOTE** *Calling `putClientProperty()` with a value of `null` causes the key to be removed from the client property table.*

For instance, the `JTree` class has a property with the Metal look and feel for configuring the line style for connecting or displaying nodes within a `JTree`. Because the setting is specific to one look and feel, it doesn't make sense to add something to the tree API. Instead, the property can be set by calling the following on a particular tree instance:

```
tree.putClientProperty("JTree.lineStyle", "Angled")
```

Then, when the look and feel is the default Metal, lines will connect the nodes of the tree. If another look and feel is installed, the client property will be ignored.

Figure 4-2 shows a tree with and without lines.

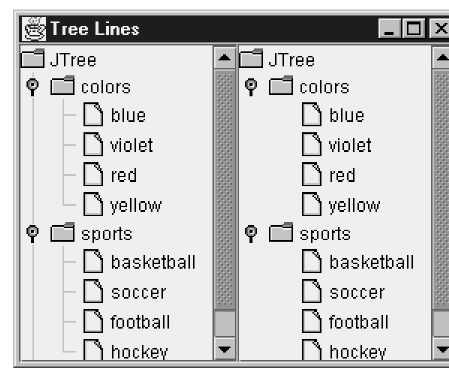



*Figure 4-2: A JTree, with and without angled lines*

> **NOTE** *The list of client properties is probably one of the least documented aspects of Swing. Chapter 18 lists the available properties I was able to determine.*

## *JComponent Properties*

You've seen some of the pieces shared by the different `JComponent` subclasses. Now it's time to look at the JavaBeans properties. Table 4-3 shows the complete list of properties defined by `JComponent`, including those inherited through the AWT `Container` and `Component` classes.

| PROPERTY NAME | DATA TYPE | JCOMPONENT ACCESS | CONTAINER ACCESS | COMPONENT ACCESS |
|---|---|---|---|---|
| accessibleContext | AccessibleContext | read-only | N/A | read-only |
| actionMap | ActionMap | read-write | N/A | N/A |
| alignmentX | float | read-write | read-only | read-only |
| alignmentY | float | read-write | read-only | read-only |
| autoscrolls | boolean | read-write | N/A | N/A |

*(continued)*

*Table 4-3 (continued)*

| PROPERTY NAME | DATA TYPE | JCOMPONENT ACCESS | CONTAINER ACCESS | COMPONENT ACCESS |
|---|---|---|---|---|
| background | Color | write-only bound | N/A | read-write bound |
| border | Border | read-write bound | N/A | N/A |
| bounds | Rectangle | N/A | N/A | read-write |
| colorModel | ColorModel | N/A | N/A | read-only |
| componentCount | int | N/A | read-only | N/A |
| componentOrientation | ComponentOrientation | N/A | N/A | read-write bound |
| components | Component[ ] | N/A | read-only | N/A |
| cursor | Cursor | N/A | N/A | read-write |
| debugGraphicsOption | int | read-write | N/A | N/A |
| displayable | boolean | N/A | N/A | read-only |
| doubleBuffered | boolean | read-write | N/A | read-only |
| dropTarget | DropTarget | N/A | N/A | read-write |
| enabled | boolean | write-only bound | N/A | read-write |
| focusCycleRoot | boolean | read-only | N/A | N/A |
| focusTraversable | boolean | read-only | N/A | read-only |
| font | Font | write-only bound | write-only | read-write bound |
| foreground | Color | write-only bound | N/A | read-write bound |
| graphics | Graphics | read-only | N/A | read-only |
| graphicsConfiguration | GraphicsConfiguration | N/A | N/A | read-only |
| height | int | read-only | N/A | read-only |
| inputContext | InputContext | N/A | N/A | read-only |
| inputMap | InputMap | read-only | N/A | N/A |
| inputMethodRequests | InputMethodRequests | N/A | N/A | read-only |
| inputVerifier | InputVerifier | read-write | N/A | N/A |
| insets | Insets | read-only | read-only | N/A |
| layout | LayoutManager | N/A | read-write | N/A |
| lightweight | boolean | N/A | N/A | read-only |
| locale | Locale | N/A | N/A | read-write |
| location | Point | N/A | N/A | read-write |
| locationOnScreen | Point | N/A | N/A | read-only |
| managingFocus | boolean | read-only | N/A | N/A |
| maximumSize | Dimension | read-write bound | read-only | read-only |
| maximumSizeSet | boolean | read-only | N/A | N/A |

*(continued)*

*Table 4-3 (continued)*

| **PROPERTY NAME** | **DATA TYPE** | **JCOMPONENT ACCESS** | **CONTAINER ACCESS** | **COMPONENT ACCESS** |
|---|---|---|---|---|
| minimumSize | Dimension | read-write bound | read-only | read-only |
| minimumSizeSet | boolean | read-only | N/A | N/A |
| name | String | N/A | N/A | read-write |
| nextFocusableComponent | Component | read-write | N/A | N/A |
| opaque | boolean | read-write bound | N/A | read-only |
| optimizedDrawingEnabled | boolean | read-only | N/A | N/A |
| paintingTile | boolean | read-only | N/A | N/A |
| parent | Container | N/A | N/A | read-only |
| preferredSize | Dimension | read-write bound | read-only | read-only |
| preferredSizeSet | boolean | read-only | N/A | N/A |
| registeredKeyStrokes | KeyStroke[ ] | read-only | N/A | N/A |
| requestFocusEnabled | boolean | read-write | N/A | N/A |
| rootPane | JRootPane | read-only | N/A | N/A |
| showing | boolean | N/A | N/A | read-only |
| size | Dimension | N/A | N/A | read-write |
| toolkit | Toolkit | N/A | N/A | read-only |
| toolTipText | String | read-write | N/A | N/A |
| topLevelAncestor | Container | read-only | N/A | N/A |
| treeLock | Object | N/A | N/A | read-only |
| UIClassID | String | read-only | N/A | N/A |
| valid | boolean | N/A | N/A | read-only |
| validateRoot | boolean | read-only | N/A | N/A |
| verifyInputWhenFocusTarget | boolean | read-write | N/A | N/A |
| visible | boolean | write-only | N/A | read-write |
| visibleRect | Rectangle | read-only | N/A | N/A |
| width | int | read-only | N/A | read-only |
| x | int | read-only | N/A | read-only |
| y | int | read-only | N/A | read-only |

*Table 4-3: JComponent properties*

**NOTE** *Additionally, there's a read-only* `class` *property defined at the* `Object` *level, the parent of the* `Component` *class.*

Including the properties from the parent hierarchy, approximately 65 properties of `JComponent` exist. As that number indicates, the `JComponent` class is extremely well oriented for visual development. There are roughly eight categories of `JComponent` properties, which are summarized as follows:

- Position-oriented properties — The `x` and `y` properties define the `location` of the component relative to its `parent`. The `locationOnScreen` is just another location for `component`, this time relative to the screen's origin (upper-left corner). The `width` and `height` properties define the `size` of the component. The `visibleRect` describes the part of the component visible within the `topLevelAncestor`, whereas the `bounds` property defines the component's area, whether visible or not.

- Component-set oriented properties — The `components` and `componentCount` properties enable you to find out what the children components are of the particular `JComponent`. For each component in the `components` property array, the current component would be its `parent`. In addition to determining a component's `parent`, you can find out its `rootPane` or `topLevelAncestor`.

- Focus-oriented properties—The `managingFocus`, `focusCycleRoot`, `focusTraversable`, `nextFocusableComponent`, `requestFocusEnabled`, `verifyInputWhenFocusTarget`, and `inputVerifier` properties define the set of focus-oriented properties. These properties control the focus behavior of `JComponent` and were discussed in greater depth in Chapter 2.

- Layout-oriented properties—`alignmentX`, `alignmentY`, `componentOrientation`, `layout`, `maximumSize`, `minimumSize`, `preferredSize`, `maximumSizeSet`, `minimumSizeSet`, and `preferredSizeSet` are used to help with layout management.

- Painting support properties — The `background`/`foreground` properties describe the current drawing colors and `font` describes the text style to draw. The `insets` and `border` properties are intermixed to describe the drawing of a border around a component. The `graphics` property permits real-time drawing, although the `paintImmediately()` method might now suffice. To improve performance, there are the `opaque` (`false` is transparent), `doubleBuffered`, and `optimizedDrawingEnabled` properties. The `graphicsConfiguration` adds support for virtual devices. For `debugGraphicsOption`, this allows you to slow down the drawing of your component if you can't figure out why it's not painted properly. The remaining two, `colorModel` and `paintingTile`, store intermediate drawing information.

  The `debugGraphicsOption` property is set to one or more of the settings in Table 4-4. Multiple settings would be combined with the bitwise `OR` ("`|`") operator.

```
JComponent component = new ...();
component.setDebugGraphicsOptions(DebugGraphics.BUFFERED_OPTION |
DebugGraphics.FLASH_OPTION | DebugGraphics.LOG_OPTION);
```

| DEBUGGRAPHICS SETTINGS | DESCRIPTION |
|---|---|
| DebugGraphics.BUFFERED_OPTION | Causes window to pop up, displaying the drawing of the double-buffered image |
| DebugGraphics.FLASH_OPTION | Causes the drawing to be done more slowly, flashing between steps |
| DebugGraphics.LOG_OPTION | Causes a message to be printed to the screen as each step is done |
| DebugGraphics.NONE_OPTION | Disables all options |

*Table 4-4: DebugGraphics settings*

- Internationalization support—The `inputContext`, `inputMethodRequests`, and `locale` properties help when creating multilingual operations.

- State support—To get state information about a component, all you have to do is ask; there's much you can discover. The `autoscrolls` property lets you place a component within a `JViewport` and it automatically scrolls when dragged. The `validateRoot` property is used when `revalidate()` has been called and returns `true` when the current component is at the point it should stop. The remaining seven properties are self-explanatory: `displayable`, `dropTarget`, `enabled`, `lightweight`, `showing`, `valid`, and `visible`.

- The rest—The remaining properties don't seem to have any kind of logical grouping. The `accessibleContext` property is for support with the `javax.accessibility` package. The `registeredKeyStrokes`, `inputMap`, and `actionMap` properties allows you to register keystroke responses with a window. The `cursor` property lets you change the cursor to one of the available cursors. The `toolTipText` property is set to display pop-up support text over a component. The `toolkit` property encapsulates platform-specific behaviors for accessing system resources. The `name` property gives you the means to recognize a particular instance of a class. The `treelock` property is the component tree-synchronization locking resource. The `UIClassID` property is new; it allows subclasses to return the appropriate class ID for their specific instance.

## Handling JComponent Events

Three event-handling capabilities are shared by all JComponent subclasses. We'll look at these shared capabilities as well as review the ones inherited from Component.

### Listening to JComponent Events with a PropertyChangeListener

The JComponent class makes several component properties bound. By binding a PropertyChangeListener to the component, you can listen for particular JComponent property changes and then respond accordingly.

```
public interface PropertyChangeListener extends EventListener {
  public void propertyChange(PropertyChangeEvent propertyChangeEvent);
}
```

To demonstrate, the following PropertyChangeListener was pulled from the Action class definition. The property that changes determines which if-block gets executed.

```
public class ActionChangedListener implements PropertyChangeListener {

  public void propertyChange(PropertyChangeEvent e) {
    String propertyName = e.getPropertyName();
    if (e.getPropertyName().equals(Action.NAME)) {
      String text = (String) e.getNewValue();
      button.setText(text);
      button.repaint();
    } else if (propertyName.equals("enabled")) {
      Boolean enabledState = (Boolean) e.getNewValue();
      button.setEnabled(enabledState.booleanValue());
      button.repaint();
    } else if (e.getPropertyName().equals(Action.SMALL_ICON)) {
      Icon icon = (Icon) e.getNewValue();
      button.setIcon(icon);
      button.invalidate();
      button.repaint();
    }
  }
}
```

For property change support with the `JComponent` class, no class constants exist for the property names. (An instance of a constant existing is `Action.SMALL_ICON` in the `Action` class example just listed.) Instead, the class uses hard-coded `String` constants. These strings are listed in Table 4-5.

| **PROPERTY CHANGE SETTING** |
| --- |
| ancestor |
| background |
| border |
| enabled |
| font |
| foreground |
| maximumSize |
| minimumSize |
| opaque |
| preferredSize |
| UI |

*Table 4-5: JComponent PropertyChangeListener support constants*

> **NOTE** *With the Java 2 platform, some bound properties of* `JComponent` *aren't notified by* `JComponent` *directly. Instead,* `JComponent` *relies on its superclass* `Component` *to do the notification because some properties of* `Component`*, such as foreground color, aren't bound with JDK 1.1 but* are *bound with the Java 2 SDK.*

The bound `UI` property is a protected property overridden by each of the `JComponent` subclasses.

The `ancestor` property name is used when the parent of the component is updated whenever the `addNotify()` / `removeNotify()` methods are called.

> **NOTE** *You can now bind a* `PropertyChangeListener` *to a specific property by adding the listener with* `addPropertyChangeListener(String propertyName, PropertyChangeListener listener)`*. This allows your listener to avoid having to check for the specific property that changed.*

### *Listening to JComponent Events with a VetoableChangeListener*

The `VetoableChangeListener` is another JavaBeans listener that Swing components use. It works with constrained properties, whereas the `PropertyChangeListener` works with only bound properties. A key difference between the two is that the `public void vetoableChange(PropertyChangeEvent propertyChangeEvent)` method can throw a `PropertyVetoException` if the listener doesn't like the requested change.

```
public interface VetoableChangeListener extends EventListener {
  public void vetoableChange(PropertyChangeEvent propertyChangeEvent) throws
PropertyVetoException;
}
```

> **NOTE** *Only one class, `JInternalFrame`, has constrained properties. The listener is meant primarily for programmers to use with their own newly created components.*

### *Listening to JComponent Events with an AncestorListener*

You can use an `AncestorListener` to find out when a component moves, is made visible, or is made invisible. It's useful if you permit your users to customize their screens by moving components around and possibly removing them from the screens. The `AncestorListener` definition is shown below.

```
public interface  AncestorListener extends EventListener {
  public void ancestorAdded(AncestorEvent ancestorEvent);
  public void ancestorMoved(AncestorEvent ancestorEvent);
  public void ancestorRemoved(AncestorEvent ancestorEvent);
}
```

To demonstrate, the following program associates an `AncestorListener` with the root pane of a `JFrame`. You'll see the messages "Removed," "Added," and "Moved" when the program first starts up. In addition, you'll see "Moved" messages when you drag the frame around.

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

```
public class AncestorSampler {
  public static void main (String args[]) {
    JFrame f = new ExitableJFrame("Ancestor Sampler");
    AncestorListener ancestorListener = new AncestorListener() {
      public void ancestorAdded(AncestorEvent ancestorEvent) {
        System.out.println ("Added");
      }
      public void ancestorMoved(AncestorEvent ancestorEvent) {
        System.out.println ("Moved");
      }
      public void ancestorRemoved(AncestorEvent ancestorEvent) {
        System.out.println ("Removed");
      }
    };
    f.getRootPane().addAncestorListener(ancestorListener);
    f.getRootPane().setVisible(false);
    f.getRootPane().setVisible(true);
    f.setSize (300, 200);
    f.setVisible (true);
  }
}
```

### *Listening to Inherited Events of a JComponent*

In addition to the ability to listen for an instance of an `AncestorEvent` or `PropertyChangeEvent` with a `JComponent`, the `JComponent` inherits the ability to listen to many other events from its `Container` and `Component` superclasses.

Table 4-6 lists eight event listeners. You may find yourself using the new listener interfaces quite a bit, but nothing prevents the older ones from working.

| CLASS | EVENT LISTENER | EVENT OBJECT |
|---|---|---|
| Component | ComponentListener | componentHidden(ComponentEvent) |
| | | componentMoved(ComponentEvent) |
| | | componentResized(ComponentEvent) |
| | | componentShown(ComponentEvent) |
| Component | FocusListener | focusGained(FocusEvent) |
| | | focusLost(FocusEvent) |
| Component | HierarchyBoundsListener | ancestorMoved(HierarchyEvent) |
| | | ancestorResized(HierarchyEvent) |
| Component | HierarchyListener | hierarchyChanged(HierarchyEvent) |

*(continued)*

*Table 4-6 (continued)*

| CLASS | EVENT LISTENER | EVENT OBJECT |
|---|---|---|
| Component | InputMethodListener | caretPositionChanged (InputMethodEvent) |
| | | inputMethodTextChanged (InputMethodEvent) |
| Component | KeyListener | keyPressed(KeyEvent) |
| | | keyReleased(KeyEvent) |
| | | keyTyped(KeyEvent) |
| Component | MouseListener | mouseClicked(MouseEvent) |
| | | mouseEntered(MouseEvent) |
| | | mouseExited(MouseEvent) |
| | | mousePressed(MouseEvent) |
| | | mouseReleased(MouseEvent) |
| Component | MouseMotionListener | mouseDragged(MouseEvent) |
| | | mouseMoved(MouseEvent) |
| Component | PropertyChangeListener | propertyChange(PropertyChangeEvent) |
| Container | ContainerListener | componentAdded(ContainerEvent) |
| | | componentRemoved(ContainerEvent) |

*Table 4-6: JComponent inherited event listeners*

## Class JToolTip

The Swing components support the ability to display brief pop-up messages when the cursor rests over them. The class used to display pop-up messages is `JToolTip`.

### *Creating a JToolTip*

Calling the `public void setToolTipText(String text)` method of `JComponent` automatically causes the creation of a `JToolTip` instance when the mouse rests over a component with the installed pop-up message. You don't normally call the `JToolTip` constructor directly. There's only one constructor, and it's of the no-argument variety.

Tooltip text is normally one line long. However, if the text string begins with `<html>` (in any case) then the contents can be any HTML 3.2 formatted text. For instance, the following line causes the pop-up message shown in Figure 4-3.

```
component.setToolTipText("<html>Tooltip<br>Message");
```

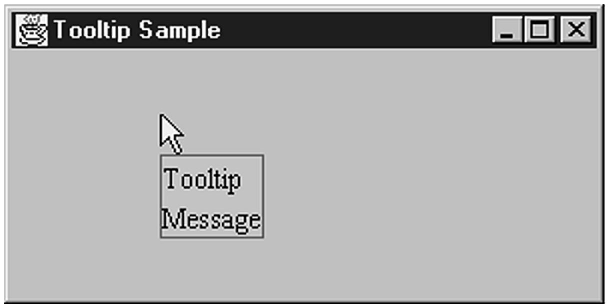



*Figure 4-3: HTML-based tooltip text*

## *Creating Customized JToolTip Objects*

You can easily customize the display characteristics for all pop-up messages by setting `JToolTip` `UIResource` elements, as shown in "Customizing JToolTip Look and Feel" later in this chapter. The `JComponent` class defines an easy way for you to customize the display characteristics of the tooltip when it's placed over a specific component. Simply subclass the component you want to customize and override its inherited `public JToolTip createToolTip()` method. The `createToolTip()` method is called when the `ToolTipManager` has determined that its time to display the pop-up message.

To customize the pop-up tooltip appearance, just override the method and customize the `JToolTip` returned from the inherited method. For instance, the following source demonstrates the setting of a custom coloration for the tooltip for a `JButton`, as shown in Figure 4-4.

```
JButton b = new JButton("Hello World") {
  public JToolTip createToolTip() {
    JToolTip tip = super.createToolTip();
    tip.setBackground(Color.yellow);
    tip.setForeground(Color.green);
    return tip;
  }
};
```


*Figure 4-4: Tooltip text displayed with custom colors*

After the `JToolTip` has been created, you can configure the inherited `JComponent` properties or any of the properties specific to `JToolTip` as shown in Table 4-7.

| PROPERTY NAME | DATA TYPE | ACCESS |
|---|---|---|
| accessibleContext | AccessibleContext | read-only |
| component | JComponent | read-write |
| tipText | String | read-write |
| UI | ToolTipUI | read-only |
| UIClassID | String | read-only |

*Table 4-7: JToolTip properties*

## *Displaying Positional ToolTip Text*

Swing components can even support the display of different tooltip text, depending on where the mouse pointer is located. This requires overriding the `public boolean contains(int x, int y)` method, which originates from the `Component` class.

For instance, after enhancing the customized `JButton` created in the previous section of this chapter, the tooltip text will differ, depending on whether or not the mouse pointer is within 50 pixels from the left edge of the component.

```
JButton button = new JButton("Hello World") {
  public JToolTip createToolTip() {
    JToolTip tip = super.createToolTip();
    tip.setBackground(Color.yellow);
    tip.setForeground(Color.green);
    return tip;
  }
  public boolean contains(int x, int y) {
    if (x < 50) {
      setToolTipText("Got Green Eggs?");
    } else {
      setToolTipText("Got Ham?");
    }
    return super.contains(x, y);
  }
};
```

## *Customizing a JToolTip Look and Feel*

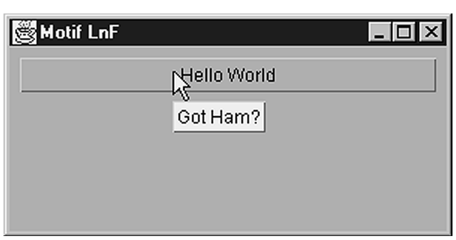
Each installable Swing look and feel provides a different `JToolTip` appearance and a set of default `UIResource` value settings. Figure 4-5 shows the appearance of the `JToolTip` component for the preinstalled set of look and feels: Motif, Windows, and Metal.

The available set of `UIResource`-related properties for a `JToolTip` is shown in Table 4-8. For the `JToolTip` component, there are five different properties.

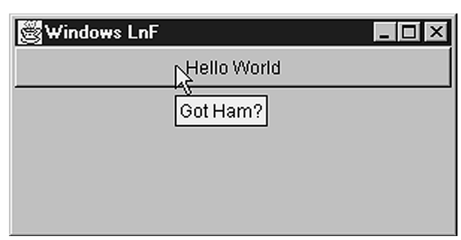
| PROPERTY STRING | OBJECT TYPE |
|---|---|
| ToolTip.background | Color |
| ToolTip.border | Border |
| ToolTip.font | Font |
| ToolTip.foreground | Color |
| ToolTipUI | String |

*Table 4-8: JToolTip UIResource elements*

As noted earlier in this chapter, the `JToolTip` class supports the display of arbitrary HTML content. This permits the display of multi-column/row input. With the original JFC/Swing release, this HTML and multi-line tip support wasn't




*Motif*




*Windows*


*Metal*

*Figure 4-5: JToolTip under different look and feels* available. It was necessary to create and install a new `ToolTipUI` delegate, a concept described more fully in Chapter 18.

## *Class ToolTipManager*

Although the `JToolTip` is something of a passive object, in the sense that the `JComponent` creates and shows the `JToolTip` on its own, there are many more configurable aspects of its usage. However, these configurable aspects are the responsibility of the class that manages tooltips, and not the `JToolTip` itself. The class that manages tooltip usage is aptly named `ToolTipManager`. With the Singleton design pattern, no constructor for `ToolTipManager` exists. Instead, you have access to the current manager through the static `sharedInstance()` method of `ToolTipManager`.

## *ToolTipManager Properties*

Once you have accessed the shared instance of `ToolTipManager`, you can customize when and if tooltip text appears. As Table 4-9 shows, there are five configurable properties.

| PROPERTY NAME | DATA TYPE | ACCESS |
|---|---|---|
| dismissDelay | int | read-write |
| enabled | boolean | read-write |
| initialDelay | int | read-write |
| lightWeightPopupEnabled | boolean | read-write |
| reshowDelay | int | read-only |

*Table 4-9: ToolTipManager properties*

Initially, tooltips are enabled, but you can disable them with `ToolTipManager.sharedInstance().setEnabled(false)`. This allows you to always associate tooltips with components, while letting the end user enable/disable them when desired.

There are three timing-oriented properties: `initialDelay`, `dismissDelay`, and `reshowDelay`. They all measure time in milliseconds. The `initialDelay` property is the number of milliseconds the user must rest the mouse inside the component before the appropriate tooltip text appears. The `dismissDelay` specifies the length of time the text appears while the mouse remains motionless; if the user moves the mouse, it also causes the text to disappear. The `reshowDelay` determines how long a user must remain outside a component before reentry would cause the pop-up text to reappear.

The remaining property `lightWeightPopupEnabled` is used to determine the pop-up window type to hold the tooltip text. If the property is `true` and the pop-up text fits entirely within the bounds of the top-level window, the text appears within a Swing `JPanel`. If this property is `false` and the pop-up text fits entirely within the bounds of the top-level window, the text appears within an AWT `Panel`. If part of the text wouldn't appear within the top-level window no matter what the property setting is, the pop-up text would appear within a `Window`.

Although not properties of `ToolTipManager`, there are two other methods of `ToolTipManager` worth mentioning:

```
public void registerComponent(JComponent component)
public void unregisterComponent(JComponent component)
```

When you call the `setToolTipText()` method of `JComponent`, this causes the component to register itself with the `ToolTipManager`. There are times, however, when you need to register a component directly. This is necessary when the display of part of a component is left to another renderer (see Chapter 16). With `JTree`, for instance, each node of the tree is displayed by a `TreeCellRenderer`. When the renderer displays the tooltip text, you "register" the `JTree` and tell the renderer what text to display.

```
JTree tree = new JTree(...);
ToolTipManager.sharedInstance().registerComponent(tree);
TreeCellRenderer renderer = new ATreeCellRenderer(...);
tree.setCellRenderer(renderer);
...
public class ATreeCellRenderer implements TreeCellRenderer {
...
  public Component getTreeCellRendererComponent(JTree tree, Object value, boolean
selected, boolean expanded, boolean leaf, int row, boolean hasFocus) {
  ...
    renderer.setToolTipText("Some Tip");
    return renderer;
  }
}
```

> **NOTE** *If this sounds confusing, not to worry. We'll revisit the `JTree` in Chapter 16.*

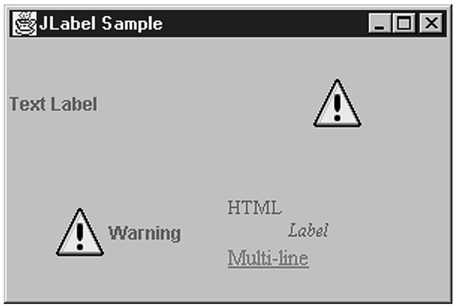



*Figure 4-6: Sample JLabel components*

## Class JLabel

The first Swing component we'll examine closely is the simplest, the `JLabel`. The `JLabel` serves as the replacement component for the AWT `Label` but it can do *much* more. Whereas the AWT `Label` is limited to a single line of text, the Swing `JLabel` can have text, or images, or both. The text can be a single line of text or HTML. In addition `JLabel` can support different enabled and disabled images. Figure 4-6 shows some sample `JLabel` components.

> **NOTE** *A `JLabel` subclass is used as the default renderer for each of the `JList`, `JComboBox`, `JTable`, and `JTree` components.*

### *Creating a JLabel*

With the six constructors for `JLabel`, you can customize any of three properties of the `JLabel`: its `text`, `icon`, or `horizontalAlignment`. By default, the text and icon properties are empty, whereas the initial horizontal alignment depends on the constructor arguments. These settings can be either `JLabel.LEFT`, `JLabel.CENTER`, or `JLabel.RIGHT`. In most cases, not specifying the horizontal alignment setting results in a left-aligned label. However, if only the initial icon is specified, then the default alignment is centered.

1. ```
   public JLabel()
   JLabel label = new JLabel();
   ```

2. ```
   public JLabel(Icon image)
   Icon icon = new ImageIcon("dog.jpg");
   JLabel label = new JLabel(icon);
   ```

3. ```
   public JLabel(Icon image, int horizontalAlignment)
   JLabel label = new JLabel(icon, JLabel.RIGHT);
   ```

4. ```
   public JLabel(String text)
   JLabel label = new JLabel("Dog");
   ```

5. ```
   public JLabel(String text, int horizontalAlignment)
   JLabel label = new JLabel("Dog", JLabel.RIGHT);
   ```
   ```

6. `public JLabel(String text, Icon icon, int horizontalAlignment)`
```
JLabel label = new JLabel("Dog", icon, JLabel.RIGHT);
```

## *JLabel Properties*

Table 4-10 shows the 13 properties of `JLabel`. They allow you to customize the content, position, and (in a limited sense) the behavior of the `JLabel`.

| PROPERTY NAME | DATA TYPE | ACCESS |
|---|---|---|
| accessibleContext | AccessibleContext | read-only |
| disabledIcon | Icon | read-write bound |
| displayedMnemonic | char | read-write bound |
| horizontalAlignment | int | read-write bound |
| horizontalTextPosition | int | read-write bound |
| icon | Icon | read-write bound |
| iconTextGap | int | read-write bound |
| labelFor | Component | read-write bound |
| text | String | read-write bound |
| UI | LabelUI | read-write |
| UIClassID | String | read-only |
| verticalAlignment | int | read-write bound |
| verticalTextPosition | int | read-write bound |

*Table 4-10: JLabel properties*

The content of the `JLabel` is the text and its associated image. Displaying an image within a `JLabel` will be discussed in the section "Interface Icon" later in this chapter. However, different icons can be displayed, depending on whether the `JLabel` is enabled or disabled. By default, the icon is a grayscaled version of the enabled icon, if the enabled icon comes from an `Image` object (`ImageIcon` to be described later in the chapter). If the enabled icon doesn't come from an `Image`, there's no icon when `JLabel` is disabled, unless manually specified.

The position of the contents of the `JLabel` is described by four different properties: `horizontalAlignment`, `horizontalTextPosition`, `verticalAlignment`, and `verticalTextPosition`. The `horizontalAlignment` and `verticalAlignment` properties describe the position of the entire contents of the `JLabel`.

> **TIP** *Alignments have an effect only if there's extra space for the layout manager to position the component. If you're using a layout manager such as `FlowLayout`, which sizes components to their preferred size, these settings will effectively be ignored.*

The horizontal position can be any of the `JLabel` constants `LEFT`, `RIGHT`, or `CENTER`. The vertical position can be `TOP`, `BOTTOM`, or `CENTER`. Figure 4-7 shows various alignment settings, with the label reflecting the alignments.

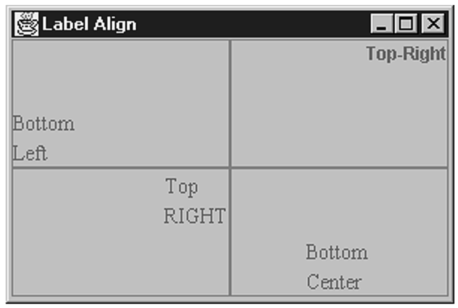



*Figure 4-7: Various JLabel alignments*

The text position properties reflect where the text is positioned relative to the icon when both are present. The properties can be set to the same constants as the alignment constants. Figure 4-8 shows various text position settings, with each label reflecting the setting.

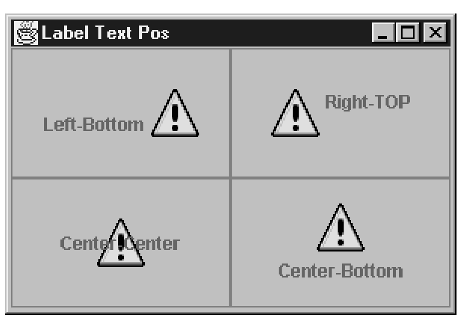



*Figure 4-8: Various JLabel text positions*

**NOTE** *The constants for the different positions come from the* `SwingConstants` *interface that the* `JLabel` *class implements.*

## *Handling JLabel Events*

No event-handling capabilities are specific to the `JLabel`. Besides the event-handling capabilities inherited through `JComponent`, the closest thing there is for event handling with the `JLabel` is the combined usage of the `displayedMnemonic` and `labelFor` properties.

When the `displayedMnemonic` and `labelFor` properties are set, pressing the keystroke specified by the mnemonic, along with the platform-specific hotkey (usually ALT), causes the input focus to shift to the component associated with the `labelFor` property. This can be helpful when a component doesn't have its own manner of displaying a mnemonic setting, such as with all the text input components, as shown in Figure 4-9.

```
JLabel label = new JLabel("Username");
JTextField textField = new JTextField();
label.setDisplayedMnemonic(KeyEvent.VK_U);
label.setLabelFor(textField);
```

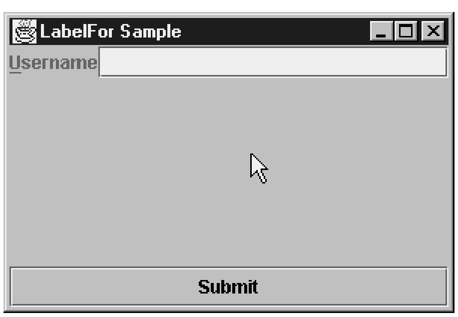



*Figure 4-9: Using a JLabel to display the mnemonic for another component*

> **NOTE** *The component setting of the* `labelFor` *property is stored as a client property of the* `JLabel` *with the* `LABELED_BY_PROPERTY` *key constant. The setting is used for accessibility purposes.*

## *Customizing JLabel Look and Feel*

Each installable Swing look and feel provides a different `JLabel` appearance and set of default `UIResource` value settings. Although appearances differ based on the current look and feel, the differences are minimal within the preinstalled set of look and feels. For the available set of `UIResource`-related properties for a `JLabel`, see Table 4-11. There are eight different properties for the `JLabel` component.

| PROPERTY STRING | OBJECT TYPE |
|---|---|
| Label.actionMap | ActionMap |
| Label.background | Color |
| Label.disabledForeground | Color |
| Label.border | Border |
| Label.disabledShadow | Color |
| Label.font | Font |
| Label.foreground | Color |
| LabelUI | String |

*Table 4-11: JLabel UIResource elements*

## Interface Icon

The `Icon` interface is used to associate glyphs with various components. These *glyphs* (like a symbol on a highway sign that conveys information nonverbally, such as "winding road ahead!") can be simple drawings or GIF images loaded from disk with the `ImageIcon` class. The interface contains two properties describing the size and a method to paint the glyph.

```
public interface Icon {
  // Properties
  public int getIconHeight();
  public int getIconWidth();
  // Other Methods
  public void paintIcon(Component c, Graphics g, int x, int y);
}
```

### *Creating an Icon*

Creating an `Icon` is as simple as implementing the interface. All you have to do is specify the size of the icon and what to draw. The following is one such `Icon` implementation. It will be used throughout the rest of the book. The icon is a diamond-shaped glyph in which the size, color, and filled-status are all configurable.

One tip in implementing the `paintIcon()` method of the `Icon` interface: Translate the drawing coordinates of the graphics context based on the x and y position passed in, and then translate them back when the drawing is done. This greatly simplifies the different drawing operations.

```
import javax.swing.*;
import java.awt.*;
public class DiamondIcon implements Icon {
  private Color color;
  private boolean selected;
  private int width;
  private int height;
  private Polygon poly;
  private static final int DEFAULT_WIDTH = 10;
  private static final int DEFAULT_HEIGHT = 10;

  public DiamondIcon(Color color) {
    this (color, true, DEFAULT_WIDTH, DEFAULT_HEIGHT);
  }

  public DiamondIcon(Color color, boolean selected) {
    this (color, selected, DEFAULT_WIDTH, DEFAULT_HEIGHT);
  }

  public DiamondIcon (Color color, boolean selected, int width, int height) {
    this.color = color;
    this.selected = selected;
    this.width = width;
    this.height = height;
    initPolygon();
  }

  private void initPolygon() {
    poly = new Polygon();
    int halfWidth = width/2;
    int halfHeight = height/2;
    poly.addPoint (0, halfHeight);
    poly.addPoint (halfWidth, 0);
    poly.addPoint (width, halfHeight);
    poly.addPoint (halfWidth, height);
  }

  public int getIconHeight() {
    return height;
  }

  public int getIconWidth() {
    return width;
```

```
  }

  public void paintIcon(Component c, Graphics g, int x, int y) {
    g.setColor (color);
    g.translate (x, y);
    if (selected) {
      g.fillPolygon (poly);
    } else {
      g.drawPolygon (poly);
    }
    g.translate (-x, -y);
  }
}
```

## *Using an Icon*

Once you have your `Icon` implementation, using the `Icon` is as simple as finding a component with an appropriate property. We've already discussed `JLabel`, so we'll use the icon with a `JLabel`.

```
Icon icon = new DiamondIcon(Color.red, true, 25, 25);
JLabel label = new JLabel(icon);
```

Figure 4-10 shows what such a label might look like.


*Figure 4-10: Using an Icon in a JLabel*

## *Class ImageIcon*

The `ImageIcon` class presents an implementation of the `Icon` interface for creating glyphs from AWT `Image` objects, whether from memory (a `byte[ ]`), off a disk (a file name), or over the network (a `URL`). Unlike regular `Image` objects, the loading of an `ImageIcon` is immediately started when the `ImageIcon` is created, though it might not be fully loaded when used. In addition, `ImageIcon` objects are serializable so that they can be easily used by JavaBean components, unlike `Image` objects.

### *Creating an ImageIcon*

There are nine constructors for an `ImageIcon`. The no-argument version creates an uninitialized version (empty). The remaining eight offer the ability to create an `ImageIcon` from an `Image`, `byte` array, file name `String`, or `URL`, with or without a description.

1. 
```
public ImageIcon()
Icon icon = new ImageIcon();
icon.setImage(anImage);
```

2. 
```
public ImageIcon(Image image)
Icon icon = new ImageIcon(anImage);
```

3. 
```
public ImageIcon(String filename)
Icon icon = new ImageIcon(filename);
```

4. 
```
public ImageIcon(URL location)
Icon icon = new ImageIcon(url);
```

5. 
```
public ImageIcon(byte imageData[])
Icon icon = new ImageIcon(aByteArray);
```

6. 
```
public ImageIcon(Image image, String description)
Icon icon = new ImageIcon(anImage, "Duke");
```

7. 
```
public ImageIcon(String filename, String description)
Icon icon = new ImageIcon(filename, filename);
```

8. 
```
public ImageIcon(URL location, String description)
Icon icon = new ImageIcon(url, location.getFile());
```

9. 
```
public ImageIcon(byte imageData[], String description)
Icon icon = new ImageIcon(aByteArray, "Duke");
```

### *Using an ImageIcon*

Using an `ImageIcon` is as simple as using an `Icon`: just create the `ImageIcon` and associate it with a component.

```
Icon icon = new ImageIcon("Warn.gif");
JLabel label3 = new JLabel("Warning", icon, JLabel.CENTER)
```

### *ImageIcon Properties*

Table 4-12 shows the six properties of `ImageIcon`. The height and width of the `ImageIcon` are the height and width of the actual `Image` object. The `imageLoadStatus` property represents the results of the loading of the `ImageIcon` from the hidden `MediaTracker`, either `MediaTracker.ABORTED`, `MediaTracker.ERRORED`, or `MediaTracker.COMPLETE`.

| PROPERTY NAME | DATA TYPE | ACCESS |
| --- | --- | --- |
| description | String | read-write |
| iconHeight | int | read-only |
| iconWidth | int | read-only |
| image | Image | read-write |
| imageLoadStatus | int | read-only |
| imageObserver | ImageObserver | read-write |

*Table 4-12: ImageIcon properties*

Sometimes it's useful to use an `ImageIcon` to load an `Image` and then just ask for the `Image` object from the `Icon`.

```
ImageIcon imageIcon = new ImageIcon(...);
Image image = imageIcon.getImage();
```

There is one major problem with using `ImageIcon` objects: They don't work when the image and class file using the icon are loaded in a JAR (Java archive) file. You can't specify the file name as a `String` and let the `ImageIcon` find the file. You must manually get the image data first and then pass the data along to the `ImageIcon` constructor.

The following `ImageLoader` class provides a `public static Image getImage (Class relativeClass, String filename)` method. You specify both the base class where the image file relative is found and the file name for the image file. Then, you just need to pass the `Image` object returned to the constructor of `ImageIcon`.

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

public final class ImageLoader {

  private ImageLoader() {
  }

  public static Image getImage(Class relativeClass, String filename) {
    Image returnValue = null;
    InputStream is = relativeClass.getResourceAsStream(filename);
    if (is != null) {
      BufferedInputStream bis = new BufferedInputStream(is);
      ByteArrayOutputStream baos = new ByteArrayOutputStream();
      try {
        int ch;
        while ((ch = bis.read()) != -1) {
          baos.write(ch);
        }
        returnValue =
Toolkit.getDefaultToolkit().createImage(baos.toByteArray());
      } catch (IOException exception) {
        System.err.println("Error loading: " + filename);
      }
    }
    return returnValue;
  }
}
```

Here's how you use the helper class:

```
Image warnImage = ImageLoader.getImage(LabelJarSample.class, "Warn.gif");
Icon warnIcon = new ImageIcon(warnImage);
JLabel label2 = new JLabel(warnIcon);
```

> **TIP** *Keep in mind that Java supports GIF89A animated images.*

## *Class GrayFilter*

One additional class worth mentioning here is the `GrayFilter` class. Many of the Swing component classes rely on this class to create a disabled version of an `Image` to be used as an `Icon`. The components use the class automatically, but there might be times when you need an AWT `ImageFilter` that does grayscales. You can convert an Image from normal to grayed out with a call to the one useful method of the class: `public static Image createDisabledImage(Image image)`.

```
Image normalImage = ...
Image grayImage = GrayFilter.createDisabledImage(normalImage)
```

You can now use the grayed-out image as the `Icon` on a component:

```
Icon warningIcon = new ImageIcon(grayImage);
JLabel warningLabel = new JLabel(warningIcon);
```

## Class AbstractButton

The `AbstractButton` class is an important Swing class that works behind the scenes as the parent class of all the Swing button components, as shown in Figure 4-11. The `JButton`, described in the section “Class Button” later in this chapter, is the simplest of the subclasses. The remaining subclasses are described in later chapters.

Each of the `AbstractButton` subclasses uses the `ButtonModel` interface to store their data model. The `DefaultButtonModel` class is the default implementation used. In addition, you can group any set of `AbstractButton` objects into a `ButtonGroup`. Although this grouping is most natural with the `JRadioButton` and `JRadioButtonMenuItem` components, any of the `AbstractButton` subclasses will work. Figure 4-12 shows these UML relationships.

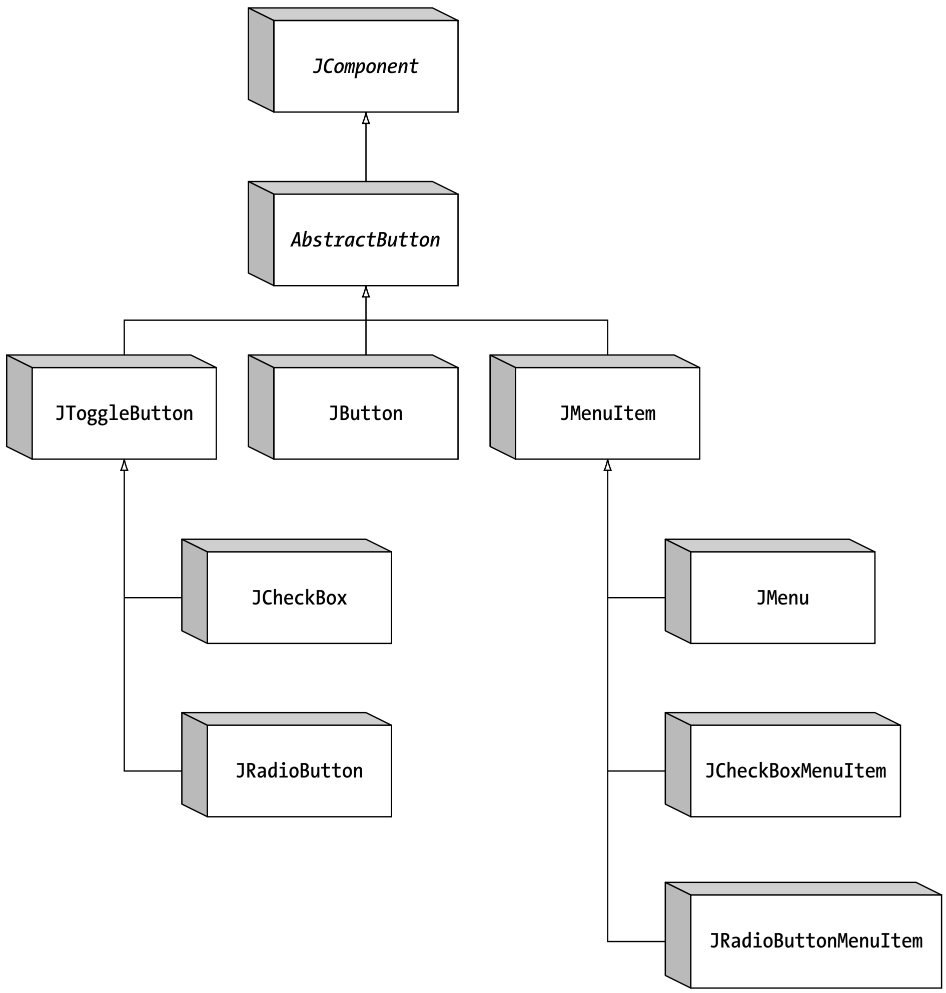



*Figure 4-11: AbstractButton class hierarchy*

## *AbstractButton Properties*

Table 4-13 lists the 26 properties (with mnemonic listed twice) of `AbstractButton` shared by all its subclasses. They allow you to customize the appearance of all the buttons.

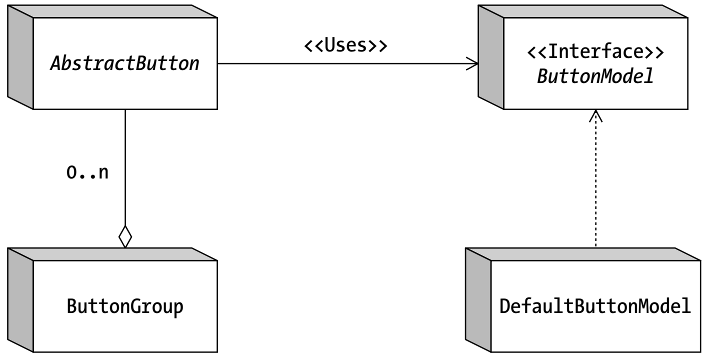



*Figure 4-12: AbstractButton UML relationship diagram*

| PROPERTY NAME | DATA TYPE | ACCESS |
|---|---|---|
| action | Action | read-write bound |
| actionCommand | String | read-write |
| borderPainted | boolean | read-write bound |
| contentAreaFilled | boolean | read-write bound |
| disabledIcon | Icon | read-write bound |
| disabledSelectedIcon | Icon | read-write bound |
| enabled | boolean | write-only |
| focusPainted | boolean | read-write bound |
| focusTraversable | boolean | read-only |
| horizontalAlignment | int | read-write bound |
| horizontalTextPosition | int | read-write bound |
| icon | Icon | read-write bound |
| margin | Insets | read-write bound |
| mnemonic | char | read-write bound |
| mnemonic | int | write-only |
| model | ButtonModel | read-write bound |
| pressedIcon | Icon | read-write bound |
| rolloverEnabled | boolean | read-write bound |
| rolloverIcon | Icon | read-write bound |
| rolloverSelectedIcon | Icon | read-write bound |
| selected | boolean | read-write |
| selectedIcon | Icon | read-write bound |
| selectedObjects | Object[ ] | read-only |
| text | String | read-write bound |
| UI | ButtonUI | read-write |
| verticalAlignment | int | read-write bound |
| verticalTextPosition | int | read-write bound |

*Table 4-13: AbstractButton properties*

**NOTE** *AbstractButton has a deprecated label property. You should use the equivalent text property instead.*

**TIP** *Keep in mind that all AbstractButton children can use HTML with its text property to display HTML content within the label. Just prefix the property setting with the string <html>.*

### *Interface ButtonModel/Class DefaultButtonModel*

The ButtonModel interface is used to describe the current state of the AbstractButton component. In addition, it describes the set of event listeners objects that are supported by all the different AbstractButton children. Its definition follows:

```
public interface ButtonModel extends ItemSelectable {
  // Properties
  public String getActionCommand();
  public void setActionCommand(String newValue);
  public boolean isArmed();
  public void setArmed(boolean newValue);
  public boolean isEnabled();
  public void setEnabled(boolean newValue);
  public void setGroup(ButtonGroup newValue);
  public int getMnemonic();
  public void setMnemonic(int newValue);
  public boolean isPressed();
  public void setPressed(boolean newValue);
  public boolean isRollover();
  public void setRollover(boolean newValue);
  public boolean isSelected();
  public void setSelected(boolean newValue);
  // Listeners
  public void addActionListener(ActionListener listener);
  public void removeActionListener(ActionListener listener);
  public void addChangeListener(ChangeListener listener);
  public void removeChangeListener(ChangeListener listener);
  public void addItemListener(ItemListener listener);
  public void removeItemListener(ItemListener listener);
}
```

The specific implementation of `ButtonModel` you'll use, unless you create your own, is the `DefaultButtonModel` class. The `DefaultButtonModel` class defines all the event registration methods for the different event listeners and manages the button state and grouping within a `ButtonGroup`. Its set of nine properties is shown in Table 4-14. They all come from the `ButtonGroup` interface, except `selectedObjects`, which is new to the `DefaultButtonModel` class, but more useful to the `JToggleButton.ToggleButtonModel`, which is discussed in Chapter 5.

| PROPERTY NAME | DATA TYPE | ACCESS |
| --- | --- | --- |
| actionCommand | String | read-write |
| armed | boolean | read-write |
| enabled | boolean | read-write |
| group | ButtonGroup | read-write |
| mnemonic | int | read-write |
| pressed | boolean | read-write |
| rollover | boolean | read-write |
| selected | boolean | read-write |
| selectedObjects | Object[ ] | read-only |

*Table 4-14: DefaultButtonModel properties*

Most of the time, you don't access the button model directly. Instead, the components that use the `ButtonModel` wrap their property calls to update the button model properties.

**NOTE** *The* `DefaultButtonModel` *also lets you get the listeners for a specific type with* `public EventListener[ ] getListeners(Class listenerType)`.

### *Understanding AbstractButton Mnemonics*

A mnemonic is a special keyboard accelerator that when pressed causes a particular action to happen. In the case of the `JLabel` discussed earlier in the section "Class JLabel," pressing the displayed mnemonic causes the associated component to get the input focus. In the case of an `AbstractButton`, pressing the mnemonic for a button causes its selection.

The actual pressing of the mnemonic requires the pressing of a look-and-feel–specific hotkey (the key tends to be the ALT key). So, if the mnemonic for a button was the "B" key, you'd need to press ALT-B to activate the button with the

B-key mnemonic. When the button is activated, registered listeners will be notified of appropriate state changes. For instance, with the `JButton` all `ActionListener` objects would be notified.

If the mnemonic key is part of the text label for the button, you'll see the character underlined. This does depend on the current look and feel and could be displayed differently. In addition, if the mnemonic isn't part of the text label, there'll be no visual indicator for selecting the particular mnemonic key.

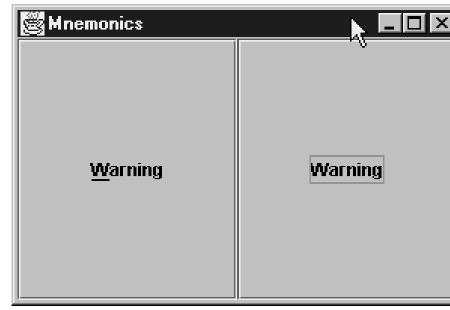



*Figure 4-13: AbstractButton mnemonics*

Figure 4-13 shows two buttons: one with a "W" mnemonic and the other with an "H" mnemonic. The left button has a label with W in its contents, so it shows the first W underlined. The second component doesn't benefit from this behavior.

To assign a mnemonic to an abstract button, you can use either one of the `setMnemonic()` methods. One accepts a `char` argument and the other an `int`. Personally, I prefer the `int` variety, in which the value is one of the many `VK_*` constants from the `KeyEvent` class.

```
AbstractButton button1 = new JButton("Warning");
button1.setMnemonic(KeyEvent.VK_W);
content.add(button1);
```

### *Understanding AbstractButton Icons*

`AbstractButton` has seven specific icon properties. The natural or default icon is the `icon` property. It is used for all cases unless a different icon is specified or there's a default behavior provided by the component. The `selectedIcon` property is the icon used when the button is selected. The `pressedIcon` is used when the button is pressed. Which of these two icons is used depends on the component, because a `JButton` is pressed but not selected, whereas a `JCheckBox` is selected but not pressed.

The `disabledIcon` and `disabledSelectedIcon` properties are used when the button has been disabled [`setEnabled(false)`]. By default, if the icon is an `ImageIcon`, a grayscaled version of the icon will be used.

The remaining two icon properties, `rolloverIcon` and `rolloverSelectedIcon`, allow you to display different icons when the mouse moves over the button (and `rolloverEnabled` is `true`).

### Understanding Internal AbstractButton Positioning

The `horizontalAlignment`, `horizontalTextPosition`, `verticalAlignment`, and `verticalTextPosition` properties share the same settings and behavior as the `JLabel` class. They're listed in Table 4-15.

| POSITION PROPERTY | AVAILABLE SETTINGS |
|---|---|
| horizontalAlignment | LEFT, CENTER, RIGHT |
| horizontalTextPosition | LEFT, CENTER, RIGHT |
| verticalAlignment | TOP, CENTER, BOTTOM |
| verticalTextPosition | TOP, CENTER, BOTTOM |

*Table 4-15: AbstractButton position constants*

## Handling AbstractButton Events

Although you do *not* create `AbstractButton` instances directly, you *do* create subclasses. All of them share a common set of event-handling capabilities. You can register `PropertyChangeListener`, `ActionListener`, `ItemListener`, and `ChangeListener` objects with abstract buttons. The `PropertyChangeListener` object will be discussed next, and the remaining objects I just listed will be discussed in later chapters of this book, with the appropriate components.

### Listening to AbstractButton Events with a PropertyChangeListener

Like the `JComponent` class, the `AbstractButton` component supports the registering of `PropertyChangeListener` objects to detect when bound properties of an instance of the class change.

Unlike the `JComponent` class, the `AbstractButton` component provides a set of class constants to signify the different property changes. These constants are listed in Table 4-16.

| PROPERTY CHANGE CONSTANT |
|---|
| BORDER_PAINTED_CHANGED_PROPERTY |
| CONTENT_AREA_FILLED_CHANGED_PROPERTY |
| DISABLED_ICON_CHANGED_PROPERTY |
| DISABLED_SELECTED_ICON_CHANGED_PROPERTY |
| FOCUS_PAINTED_CHANGED_PROPERTY |

*(continued)*

*Table 4-16 (continued)*

| PROPERTY CHANGE CONSTANT |
| --- |
| HORIZONTAL_ALIGNMENT_CHANGED_PROPERTY |
| HORIZONTAL_TEXT_POSITION_CHANGED_PROPERTY |
| ICON_CHANGED_PROPERTY |
| MARGIN_CHANGED_PROPERTY |
| MNEMONIC_CHANGED_PROPERTY |
| MODEL_CHANGED_PROPERTY |
| PRESSED_ICON_CHANGED_PROPERTY |
| ROLLOVER_ENABLED_CHANGED_PROPERTY |
| ROLLOVER_ICON_CHANGED_PROPERTY |
| ROLLOVER_SELECTED_ICON_CHANGED_PROPERTY |
| SELECTED_ICON_CHANGED_PROPERTY |
| TEXT_CHANGED_PROPERTY |
| VERTICAL_ALIGNMENT_CHANGED_PROPERTY |
| VERTICAL_TEXT_POSITION_CHANGED_PROPERTY |

*Table 4-16: AbstractButton PropertyChangeListener support constants*

Therefore, instead of hard-coding specific text strings, you can create a `PropertyChangeListener` that uses these constants.

```
public class AbstractButtonPropertyChangeListener implements
PropertyChangeListener {

  public void propertyChange(PropertyChangeEvent e) {
    String propertyName = e.getPropertyName();
    if (e.getPropertyName().equals(AbstractButton.TEXT_CHANGED_PROPERTY)) {
      String newText = (String) e.getNewValue();
      String oldText = (String) e.getOldValue();
      System.out.println(oldText + " changed to " + newText);
    } else if (e.getPropertyName().equals(AbstractButton.ICON_CHANGED_PROPERTY))
{
      Icon icon = (Icon) e.getNewValue();
      if (icon instanceof ImageIcon) {
        System.out.println("New icon is an image");
      }
    }
  }
}
```

## Class Button

The JButton component is your basic AbstractButton component that can be selected. It replaces the AWT Button class. Whereas the AWT Button is restricted to a single line of text, the JButton supports text, images, and HTML-based labels, as shown in Figure 4-14.

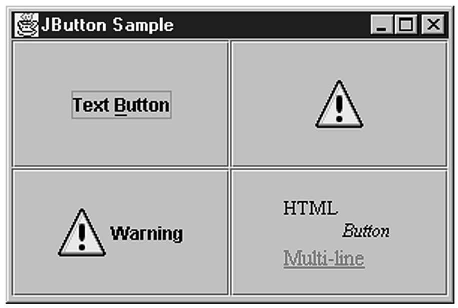



*Figure 4-14: Example JButton components*

### *Creating a JButton*

The JButton class has five constructors. You can create a button with or without a text label or icon. The icon represents the default or selected icon property from AbstractButton.

1. 
```
public JButton()
JButton button = new JButton();
```

2. 
```
public JButton(Icon image)
JButton button = new JButton();
```

3. 
```
public JButton(String text)
JButton button = new JButton();
```

4. 
```
public JButton(String text, Icon icon)
JButton button = new JButton();
```

5. 
```
public JButton(Action action)
Action action = …;
JButton button = new JButton();
```

> **NOTE** *Creating a* `JButton` *from an* `Action` *initializes the text label, icon, enabled status, and tooltip text. In addition, the* `ActionListener` *of the* `Action` *will be notified upon selection.*

## *JButton Properties*

The `JButton` component doesn't add much to the `AbstractButton`. As Table 4-17 shows, of the four properties of `JButton`, the only *new* behavior added is enabling the button to be the default.

| PROPERTY NAME | DATA TYPE | ACCESS |
|---|---|---|
| accessibleContext | AccessibleContext | read-only |
| defaultButton | boolean | read-only |
| defaultCapable | boolean | read-write bound |
| UIClassID | String | read-only |

*Table 4-17: JButton properties*

The default button tends to be drawn with a different and darker border than the remaining buttons. When a button is the default, pressing the ENTER key while in the top-level window causes the button to be selected. This only works as long as the component with the input focus, such as a text component or another button, doesn't consume the ENTER key. Because the `defaultButton` property is read-only, how (you might be asking) do you set a button as the default? All top-level Swing windows contain a `JRootPane`, to be described in Chapter 8. You tell this `JRootPane` which button is the default by setting its `defaultButton` property. Only buttons whose `defaultCapable` property is `true` can be configured to be the default. Figure 4-15 shows the top-right button set as the default.

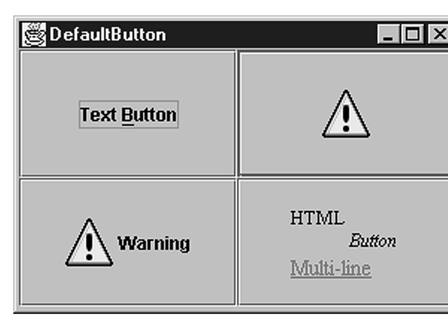



*Figure 4-15: Setting a default button*

The following source code demonstrates the setting of the default button component, as well as the basic `JButton` usage. If the default button appearance doesn't seem that obvious in Figure 4-15, wait until the `JOptionPane` is described in Chapter 9, where the difference in appearance will be more obvious.

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

```
public class DefaultButton {

  public static void main(String args[]) {
    JFrame frame = new ExitableJFrame("DefaultButton");

    Container content = frame.getContentPane();
    content.setLayout(new GridLayout(2, 2));

    JButton button1 = new JButton("Text Button");
    button1.setMnemonic(KeyEvent.VK_B);
    content.add(button1);

    Icon warnIcon = new ImageIcon("Warn.gif");
    JButton button2 = new JButton(warnIcon);
    content.add(button2);

    JButton button3 = new JButton("Warning", warnIcon);
    content.add(button3);

    String htmlButton = "<html><sup>HTML</sup> <sub><em>Button</em></sub><br>" +
      "<font color=\"#FF0080\"><u>Multi-line</u></font>";
    JButton button4 = new JButton(htmlButton);
    content.add(button4);

    JRootPane rootPane = frame.getRootPane();
    rootPane.setDefaultButton(button2);

    frame.setSize(300, 200);
    frame.setVisible(true);
  }
}
```

## *Handling JButton Events*

The `JButton` component itself has no specific event-handling capabilities. They're all inherited from `AbstractButton`. Although you can listen for change events, item events, and property change events, the most helpful listener with the `JButton` is the `ActionListener`.

## *Listening to JButton Events with an ActionListener*

When the `JButton` component is selected, all registered `ActionListener` objects are notified. This behavior is identical to the AWT `Button` component and makes transitioning from the AWT `Button` to the Swing `JButton` that much easier.

When the button is selected, an `ActionEvent` is passed to each listener. This event passes along the `actionCommand` property of the button to help identify which button was selected when a shared listener is used across multiple components. If the `actionCommand` property hasn't been explicitly set, the current `text` property is passed along instead.

Figure 4-15 shows the sample program screen. The following source code adds the event-handling capabilities to the default button example in the previous section of this chapter. Notice that the default button status is ignored, because all the components consume the ENTER key. Another component such as a `JList` or `JComboBox`, is necessary to get the input focus for the ENTER key to work properly.

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

public class ActionButtonSample {

  public static void main(String args[]) {
    JFrame frame = new ExitableJFrame("DefaultButton");

    ActionListener actionListener = new ActionListener() {
      public void actionPerformed(ActionEvent actionEvent) {
        String command = actionEvent.getActionCommand();
        System.out.println ("Selected: " + command);
      }
    };

    Container content = frame.getContentPane();
    content.setLayout(new GridLayout(2, 2));

    JButton button1 = new JButton("Text Button");
    button1.setMnemonic(KeyEvent.VK_B);
    button1.setActionCommand("First");
    button1.addActionListener(actionListener);
    content.add(button1);

    Icon warnIcon = new ImageIcon("Warn.gif");
    JButton button2 = new JButton(warnIcon);
    button2.setActionCommand("Second");
```

```
    button2.addActionListener(actionListener);
    content.add(button2);

    JButton button3 = new JButton("Warning", warnIcon);
    button3.setActionCommand("Third");
    button3.addActionListener(actionListener);
    content.add(button3);

    String htmlButton = "<html><sup>HTML</sup> <sub><em>Button</em></sub><br>" +
      "<font color=\"#FF0080\"><u>Multi-line</u></font>";
    JButton button4 = new JButton(htmlButton);
    button4.setActionCommand("Fourth");
    button4.addActionListener(actionListener);
    content.add(button4);

    JRootPane rootPane = frame.getRootPane();
    rootPane.setDefaultButton(button2);

    frame.setSize(300, 200);
    frame.setVisible(true);
  }
}
```

## *Customizing a JButton Look and Feel*

Each installable Swing look and feel provides a different `JButton` appearance and set of default `UIResource` value settings. Figure 4-16 shows the appearance of the `JButton` component for the preinstalled set of look and feels: Motif, Windows, and Metal.

The available set of `UIResource`-related properties for a `JButton` is shown in Table 4-18. For the `JButton` component, there are 16 different properties.

| PROPERTY STRING | OBJECT TYPE |
|---|---|
| Button.background | Color |
| Button.border | Border |
| Button.dashedRectGapHeight | Integer |
| Button.dashedRectGapWidth | Integer |
| Button.dashedRectGapX | Integer |
| Button.dashedRectGapY | Integer |
| Button.disabledText | Color |
| Button.focus | Color |
| Button.focusInputMap | InputMap |
| Button.font | Font |

*(continued)*

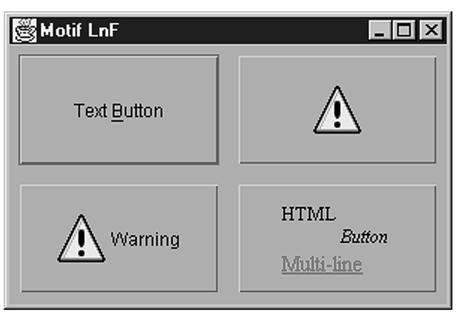
*Table 4-18 (continued)*

| PROPERTY STRING | OBJECT TYPE |
|---|---|
| Button.foreground | Color |
| Button.margin | Insets |
| Button.select | Color |
| Button.textIconGap | Integer |
| Button.textShiftOffset | Integer |
| ButtonUI | String |

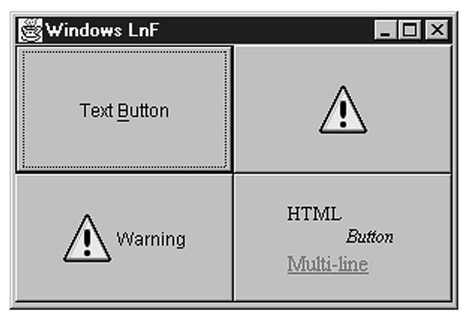
*Table 4-18: JButton UIResource elements*

## Class JPanel

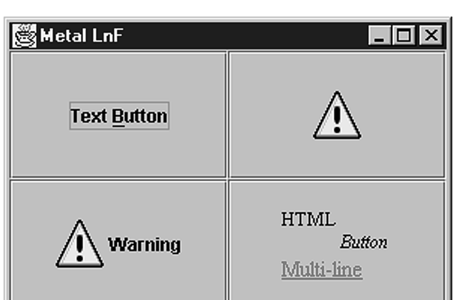
The last of the basic Swing components is the `JPanel` component. The `JPanel` component serves as a replacement for two of the AWT components. It's both a




*Motif*




*Windows*




*Metal*

*Figure 4-16: JButton under different look and feels* general-purpose container object, replacing the `Panel` container, and a replacement for the `Canvas` component, for those times when you need a drawable Swing component area.

## *Creating a JPanel*

There are four constructors for `JPanel`. With the constructors, you can either change the default layout manager from `FlowLayout` or change the default double buffering that's performed from `true` to `false`.

1. `public JPanel()`
   `JPanel label = new JPanel();`

2. `public JPanel(boolean isDoubleBuffered)`
   `JPanel label = new JPanel(false);`

3. `public JPanel(LayoutManager manager)`
   `JPanel label = new JPanel(new GridLayout(2,2));`

4. `public JPanel(LayoutManager manager, boolean isDoubleBuffered)`
   `JPanel label = new JPanel(new GridLayout(2,2), false);`

## *Using a JPanel*

You can use `JPanel` as your general-purpose container or as a base class for a new component. For the general purpose container, the procedure is simple: Just create the panel, set its layout manager if necessary, and add components using the `add()` method.

```
JPanel panel = new JPanel();
JButton okButton = new JButton("OK");
panel.add(okButton);
JButton cancelButton = new JButton("Cancel");
panel.add(CancelButton);
```

When you want to create a new component, subclass `JPanel` and override the `public void paintComponent(Graphics g)` method. Although you can subclass `JComponent` directly, it seems more appropriate to subclass `JPanel`. The following demonstrates a simple component that draws an oval to fit the size of the component; it also includes a test driver. Figure 4-17 shows the test driver program results.

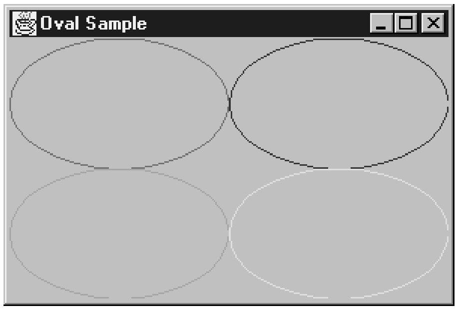



*Figure 4-17: Our new OvalPanel component*

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

public class OvalPanel extends JPanel {

  Color color;

  public OvalPanel() {
    this(Color.black);
  }
  public OvalPanel(Color color) {
    this.color = color;
  }
  public void paintComponent(Graphics g) {
    int width = getWidth();
    int height = getHeight();
    g.setColor(color);
    g.drawOval(0, 0, width, height);
  }

  public static void main(String args[]) {
    JFrame frame = new ExitableJFrame("Oval Sample");

    Container content = frame.getContentPane();
    content.setLayout(new GridLayout(2, 2));

    Color colors[] = {Color.red, Color.blue, Color.green, Color.yellow};
    for (int i=0; i<4; i++) {
```

```
      OvalPanel panel = new OvalPanel(colors[i]);
      content.add(panel);
    }

    frame.setSize(300, 200);
    frame.setVisible(true);
  }
}
```

> **NOTE** *One feature worth noting about the* `JPanel`*: By default,* `JPanel` *components are opaque. This differs from* `JComponent`*, whose opaque setting by default is* `false`*.*

### *Customizing a JPanel Look and Feel*

The available set of `UIResource`-related properties for a `JPanel` is shown in Table 4-19. For the `JPanel` component, there are five different properties. These settings may have an effect on the components within the panel.

| PROPERTY STRING | OBJECT TYPE |
|---|---|
| Panel.background | Color |
| Panel.border | Border |
| Panel.font | Font |
| Panel.foreground | Color |
| PanelUI | String |

*Table 4-19: JPanel UIResource elements*

## Summary

In this chapter, we explored the root of all Swing components: the `JComponent` class. From there, we looked at some of the common elements of all components, such as tooltips, as well as specific components such as `JLabel`. I also discussed how to put glyphs (nonverbal images) on components with the help of the `Icon` interface and the `ImageIcon` class, and the `GrayFilter` image filter for disabled icons.

We also dealt with the AbstractButton component, which serves as the root component for all Swing button objects. We looked at its data model interface, ButtonModel, and the default implementation of this interface, DefaultButtonModel. Next, we looked at the JButton class, which is the simplest of the AbstractButton implementations. And lastly, we looked at the JPanel as the basic Swing container object.

In the Chapter 5, we'll start to dig into some of the more complex AbstractButton implementations: the toggle buttons.